



Cadence C-to-Silicon Compiler User Guide

**Product Version 14.20 s400
June 2016**

Legal Notice

© 2006–2016 Cadence Design Systems, Inc. All rights reserved worldwide.
Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. All other trademarks are the property of their respective holders.

Restricted Permission: This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer;
2. The publication may not be modified in any way;
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement;
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration

Patents: The Cadence C-to-Silicon Compiler, described in this publication, may be protected by U.S. Patents [7,472,361], [7,587,687], [7,673,259] and [8,458,630]: other U.S. Patents Pending.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Contents

1	Release Notes	1-1
1.1	What's New	1-2
1.1.1	Improved Pipeline Support	1-2
1.1.2	Improved Verification	1-4
1.1.3	Improved Ease of Use	1-4
1.1.4	Improved Run Time	1-5
1.2	Deprecated Items	1-5
1.3	Changes to CtoS-Supported OS Platforms and Software Versions	1-6
1.4	Known Limitations	1-6
1.4.1	Pipelined Loops with Reads and Writes to Same Array	1-6
1.4.2	Limited Support for Multiple Clocks	1-8
1.5	Known Problems with Workarounds in 14.20	1-8
1.6	Problems Resolved/Enhancements Made in 14.20	1-13
1.6.1	Problems Resolved in 14.20 s400	1-13
1.6.2	Problems Resolved in 14.20 s302	1-14
1.6.3	Problems Resolved in 14.20 s301	1-14
1.6.4	Problems Resolved in 14.20 s300	1-14
1.6.5	Problems Resolved in 14.20 s201	1-15
1.6.6	Problems Resolved in 14.20 s200	1-15
1.6.7	Problems Resolved in 14.20 s101	1-16
1.6.8	Problems Resolved in 14.20 p100	1-17

2	Why Use CtoS?	2-1
3	Prerequisites for CtoS and for this User Guide	3-1
3.1	CtoS-Supported OS Platforms/Software Versions	3-2
3.2	Audience/Prerequisites for this User Guide	3-3
3.3	How to Find Things in this User Guide	3-3
3.4	Conventions and Syntax in this User Guide	3-3
3.5	CtoS Definition of “Preliminary Feature”	3-4
3.6	Filing a Service Request	3-4
4	CtoS Design Flow	4-1
4.1	CtoS Design Flow (Chart Format)	4-2
4.2	CtoS Design Flow (Text Format with References)	4-3
5	Preparing C Models for High-Level Synthesis	5-1
5.1	Preparing High-Level Synthesis Models	5-2
5.1.1	Assuring Statically Determinable Behavior	5-2
5.1.2	C or C++?	5-2
5.1.3	Partitioning Code into Functional Blocks	5-3
5.1.4	Interprocess Communication	5-4
5.1.5	Creating a Top-Level Module Interface	5-4
5.1.6	Handling Non-Synthesizable Constructs	5-5
5.1.7	Example	5-6
5.2	Preparing Models with Large Container Classes	5-10
5.2.1	Introduction	5-10
5.2.2	Synthesis Requirements	5-11
5.2.3	Modeling Container Classes for Synthesis	5-12
5.2.4	Choosing the Right Modeling Approach	5-16
6	Starting, Setting Up, and Building a Design	6-1
6.1	Starting CtoS	6-2
6.1.1	Starting the CtoS GUI	6-2
6.1.2	CtoS at the Command Line	6-3
6.2	Customizing the CtoS Environment	6-3
6.2.1	Loading Initialization Files at Startup	6-3

6.2.2	Using the Preferences Dialog and File	6-4
6.3	Learning the Basics of the CtoS GUI	6-5
6.3.1	Menu Bar	6-5
6.3.2	Tool Bar	6-7
6.3.3	Quick Launch	6-8
6.3.4	Command Window	6-8
6.3.5	Shortcuts in the CtoS GUI	6-9
6.4	Creating a New Design	6-9
6.4.1	Create New Design Wizard	6-10
6.4.2	Design Property Dialog	6-18
6.5	Building a Design	6-28
6.5.1	Compiling Multi-Source Designs through Single Combined Source	6-29
6.5.2	Using the Interrupt Button	6-32
6.5.3	Resolving Build Internal Errors using Source Context	6-33
6.5.4	Using the -verbose Option of Build for Enhanced Source Context	6-34
6.5.5	Using External Editors on SystemC Files	6-36
6.6	Viewing Input Source	6-38
6.6.1	Syntax-Highlighting	6-39
6.6.2	Context Menus	6-40
6.6.3	Cross-Highlighting with Markers	6-41
6.7	Using the CDFG Viewer	6-41
6.8	Navigating the CtoS Environment	6-43
6.8.1	Task Window	6-43
6.8.2	Hierarchy Window	6-44
6.8.3	Navigating the Virtual Design Directory	6-47
6.8.4	Getting Object IDs	6-47
6.8.5	Setting the Current Design	6-48
6.8.6	Finding Objects in the CtoS Virtual Directory	6-48
6.8.7	Getting Help on Commands and Messages	6-48
6.8.8	Redirecting Output	6-48
6.8.9	Rerunning Previously Run Commands	6-49
6.9	Saving, Opening, and Closing Designs	6-51
6.9.1	Uses for Save/Open	6-51
6.9.2	Saving a Design	6-52
6.9.3	Opening a Design	6-53
6.9.4	Closing a Design	6-53

7	Verifying Designs	7-1
7.1	Writing a Testbench	7-3
7.1.1	Writing Sequencer and Monitor Function Modules	7-3
7.1.2	Understanding Testbench Module Hierarchy	7-4
7.1.3	Using the <code>sc_main</code> Function in a Testbench	7-5
7.2	Generating Models	7-7
7.2.1	Graphical Depiction of all CtoS Models	7-7
7.2.2	Automatic Generation of Models	7-9
7.2.3	Generating Verilog Behavioral Models	7-9
7.2.4	Generating RTL Descriptions (Only after Scheduling)	7-11
7.3	Generating Wrappers	7-11
7.3.1	Automatic Generation of Wrappers	7-11
7.3.2	Generating a SystemC Verification Wrapper	7-11
7.3.3	Generating a Verilog Verification Wrapper	7-13
7.3.4	Generating a TLM Wrapper	7-14
7.3.5	Generating a Top Wrapper for Exported Memories (Only after Scheduling)	7-15
7.4	Modifying a Testbench	7-15
7.5	Defining a Simulation Configuration and Generating Simulation Makefiles	7-17
7.5.1	Defining a Simulation Configuration	7-18
7.5.2	Generating a Simulation Makefile Automatically	7-19
7.5.3	Generating a Simulation Makefile Manually	7-19
7.6	Simulating from the CtoS Environment	7-22
7.6.1	Using the Simulation Monitor	7-22
7.6.2	Launching Simulation	7-24
7.7	Debugging Simulation Mismatches	7-25
7.8	Renaming States in Support of FSM Coverage	7-25
7.8.1	State Identification by Annotation Paths	7-26
7.8.2	Renaming States by Annotation Paths	7-27
7.8.3	RTL Format for FSM Coverage	7-27
7.8.4	Renaming States for Verification	7-28
7.8.5	Renaming States by Script	7-29
8	Specifying Micro-Architecture	8-1
8.1	Resolving Functions	8-3
8.1.1	Inlining Functions	8-3

8.1.2	Pipelining Functions	8-12
8.1.3	Converting a Function to a Table Lookup	8-15
8.1.4	Importing RTL IP into SystemC Designs	8-16
8.2	Resolving Loops	8-16
8.2.1	How Combinational Loops Are Determined in CtoS	8-17
8.2.2	Unrolling Loops	8-17
8.2.3	Breaking Combinational Loops	8-22
8.2.4	Deciding between Unrolling and Breaking Loops	8-24
8.2.5	Pipelining Loops	8-26
8.3	Resolving Arrays (Memories)	8-61
8.3.1	Flattening Arrays	8-61
8.3.2	Merging Arrays	8-63
8.3.3	Splitting Arrays	8-65
8.3.4	Restructuring Arrays	8-78
8.3.5	Floating I/O Accesses	8-91
8.3.6	Floating Array Accesses	8-94
8.4	Specifying Micro-Architecture for a Single Behavior	8-96
8.5	Semi-Automatic Micro-Architecture	8-97
8.5.1	Setting Synthesis Modes	8-98
8.5.2	Overriding Micro-Architecture Actions	8-100
8.5.3	Applying Micro-Architecture Actions	8-102
8.6	Using Synthesis Directive	8-103
8.6.1	What Is Synthesis Directives	8-103
8.6.2	Representation of a Synthesis Directive in the CtoS Database	8-104
8.6.3	Elaboration	8-104
8.6.4	Applying and Removing a Directive	8-105
8.6.5	Use Models	8-105
8.6.6	Commands Supported with Synthesis Directives	8-106

9 Allocating Memory and RTL IP 9-1

9.1	Allocating Memory	9-2
9.1.1	Allocating Built-In RAM	9-3
9.1.2	Allocating Prototype Memory	9-7
9.1.3	Allocating Vendor RAM	9-10
9.1.4	Allocating Vendor ROM	9-14
9.1.5	Allocating Processes to Memory Interfaces	9-17

9.1.6	Specifying Interfaces for Built-In RAM	9-18
9.1.7	Specifying Interfaces for Prototype Memory	9-19
9.1.8	Specifying Memory Clock for Built-In RAM, Prototype Memory, and Vendor RAM	9-19
9.1.9	Exporting Memories	9-20
9.1.10	Registered Memories	9-21
9.1.11	Stallable Memories	9-22
9.1.12	External Arrays	9-22
9.2	Importing RTL IP into SystemC Designs	9-41
9.2.1	RTL IP Requirements for RTL Designs, SystemC Code	9-43
9.2.2	Effect of Linking Behaviors to RTL IP	9-43
9.2.3	Stall Loop Support in RTL IP	9-44
9.2.4	Using RTL IP on User-Defined Types	9-46
10	Analyzing Micro-Architecture	10-1
10.1	Check Design Report	10-2
10.2	Prescheduled Timing Report	10-3
10.3	Prescheduled Area	10-4
10.4	Prescheduled Power Report	10-5
10.5	Summary Report	10-6
11	Advanced Features for Guiding the CtoS Scheduler	11-1
11.1	Creating and Managing Array Dependencies	11-2
11.1.1	Creating Array Dependencies	11-2
11.1.2	Producing an Array Dependency Report	11-2
11.1.3	Breaking Array Dependencies	11-3
11.1.4	Scheduling Sequential Functions with Array Accesses	11-5
11.2	Creating and Managing States	11-6
11.2.1	Limitations When Creating States in Combinational Functions	11-7
11.2.2	Creating Required States	11-8
11.2.3	Creating Individual States	11-9
11.2.4	Specifying External Delay for Process I/O Nets	11-10
11.2.5	Specifying External Delay for Pipelined or Sequential Functions	11-11
11.2.6	Constraining Latency	11-12
11.2.7	Producing a Latency Report	11-13

11.2.8	Creating Protocol Regions	11-14
11.2.9	Constraining ops to Edges	11-17
11.2.10	Constraining ops to Expandable Edges	11-19
11.3	Creating and Managing Resources	11-20
11.3.1	Creating Initial Resources	11-20
11.3.2	Creating Individual Resources	11-21
11.3.3	Controlling addsub Resources	11-23
11.3.4	Controlling Embedded RC Timing Engine	11-23
11.3.5	Constraining ops to Resources	11-23
11.4	Potential Sources of op span Scheduling Failures	11-27
11.4.1	Inconsistent Edge Constraints	11-29
11.4.2	Sequential Distance Violations for Resources with Non-Zero Latency	11-29
11.4.3	Memory Contention	11-31
11.4.4	Pipelining Restrictions	11-31
12.1	Scheduling	12-2
12.1.1	Timing Analysis	12-4
12.1.2	Relaxed Latency Scheduling Mode	12-4
12.1.3	Scheduling Restrictions for Multi-Latency Operations	12-8
12.1.4	Multi-Phase Scheduler	12-11
12.1.5	Controlling Speed Grades	12-14
12.1.6	Scheduling with op Delays Larger than Clock Cycle	12-16
12.1.7	Scheduling for Low Power Design	12-16
12.2	Results of Scheduling	12-17
12.2.1	Scheduling Completes with Positive Slack	12-17
12.2.2	Scheduling completes with negative slack	12-17
12.2.3	Scheduling Fails with Failed Resource Bindings	12-18
12.3	Managing Registers	12-18
12.3.1	Creating Specific Registers	12-18
12.3.2	Binding a Value	12-18
12.3.3	Resetting Registers	12-19
12.3.4	Allocating Registers	12-20
12.3.5	Connecting Terminals to Add Power or Test	12-20
12.3.6	Reporting Registers (Primary and Secondary)	12-21

13 Analyzing and Implementing Designs 13-1

13.1	Using Reports to Analyze Designs	13-2

13.1.1	Starting the CtoS GUI and Setting Up the Design	13-2
13.1.2	Resource Bindings Viewer	13-4
13.1.3	Register Bindings Viewer	13-11
13.1.4	Path Summary Report	13-15
13.1.5	Timing Report	13-16
13.1.6	Cycle Analysis Viewer	13-20
13.1.7	RTL Schematic Viewer	13-23
13.1.8	Reporting Power and Area Using the Tree Map	13-29
13.1.9	Summary Report	13-35
13.2	Generating Models, Wrappers, RTL, SLEC after Scheduling	13-36
13.2.1	Generating a Simulation Model after Scheduling	13-36
13.2.2	Generating a Verification Wrapper after Scheduling	13-36
13.2.3	Generating an RTL Description after Scheduling	13-36
13.2.4	Generating a SLEC Script and XML File	13-38
13.2.5	Generating a Top Wrapper for Exported Memories	13-40
13.3	Generating Gates in CtoS	13-42
13.3.1	Using the Synthesis Monitor	13-42
13.3.2	Generating Gates	13-44
13.3.3	Configuring Logic Synthesis Runs	13-46
13.3.4	Using Foundation Flows	13-47
13.3.5	Understanding RTL Design Configurations	13-47
13.3.6	Configuring Foundation Flows	13-48
13.3.7	Using Plugins to Customize Foundation Flows	13-50
13.3.8	Understanding RC Run Scripts	13-51
13.3.9	Viewing Reports Generated by Default Synthesis Flow	13-52
13.3.10	Generating RC Scripts	13-53
13.3.11	Generating Gates from Logic Synthesis Makefiles	13-53

14 Authoring SystemC for CtoS 14-1

14.1	Modules	14-4
14.1.1	Instantiation of the Top Module	14-4
14.1.2	Module Constructor	14-5
14.2	Ports	14-6
14.2.1	Instantiation	14-7
14.2.2	Binding Array Ports Using a for Loop	14-7
14.2.3	User-Defined Port Binding Function	14-8

14.2.4	Port Promotion	14-8
14.3	Processes	14-9
14.3.1	SC_CTHREAD Processes	14-10
14.3.2	SC_THREAD Processes	14-19
14.3.3	Combinational SC_METHOD Processes	14-19
14.3.4	Clocked SC_METHOD Processes	14-25
14.3.5	Resetting Fields of Modules	14-31
14.3.6	Multiple Resets	14-33
14.3.7	Internally Generated Reset Signals	14-36
14.3.8	Clocks with SC_METHOD and SC_CTHREAD	14-36
14.3.9	dont_initialize() Function Calls	14-36
14.4	Mealy Communication	14-37
14.5	Inputs, Outputs, and Multiple Drivers	14-37
14.6	Variables	14-37
14.6.1	Interprocess Communication through Shared Variables	14-38
14.6.2	Array Variables	14-38
14.6.3	Read-Only Non-Array Shared Data Variables	14-41
14.6.4	Module Member Variables	14-43
14.6.5	Global Variables	14-44
14.6.6	Static Variables	14-44
14.7	Object Models	14-45
14.7.1	Packed Object Model	14-45
14.7.2	Monolithic Object Model	14-47
14.7.3	Field-Fragmented Object Model	14-48
14.7.4	CtoS Assumptions about Data Access	14-49
14.7.5	Using the merge_arrays Command with Object Models	14-51
14.7.6	Choosing the Object Model for a Struct	14-51
14.7.7	Using the build_monolithic_structs Design Attribute	14-52
14.7.8	Known Problems and Limitations with Object Models	14-52
14.8	Data Types	14-53
14.8.1	Primitive C/C++ Data Types	14-53
14.8.2	SystemC Data Types	14-54
14.8.3	User-Defined Data Types	14-57
14.8.4	Pointers	14-66
14.9	Using the CtoS Pragmas	14-71
14.9.1	Skipping Variable Initialization	14-71

14.9.2	Keeping Signals	14-72
14.9.3	Packing Struct	14-75
14.9.4	Keeping Struct Monolithic	14-75
14.9.5	Keeping Instance Boundary	14-76
14.9.6	Annotating Function Calls	14-76
14.10	Specifying Synthesis Directive	14-79
14.10.1	Statements	14-79
14.10.2	Declarations	14-82
14.11	Constructs	14-85
14.11.1	Support for goto Statements	14-85
14.11.2	Unsupported C/C++ Constructs	14-93
14.12	Using C++ Labels	14-94
14.12.1	Labeled Control Statements	14-94
14.12.2	Labeled Block Statements	14-96
14.12.3	Labeled wait Statements	14-96
14.12.4	Labeled Simple Statements	14-97
14.12.5	Control Flow Node Preserve Attribute	14-97
14.13	Coding Tips for High Quality Synthesis	14-98
14.13.1	Controlling the Dynamics of Variables	14-98
14.13.2	Optimizing the Control Flow	14-99
14.13.3	Multi-Dimensional Array Access Management	14-100
14.14	Specifying Synthesis-Specific and Simulation-Specific Code (<code>__CTOS__</code> macro)	14-102
14.15	Specifying Clock Gate Integrated Cells (CGIC)	14-102
14.15.1	Usage of Clock Gating	14-103
14.16	SystemC Standard Language Restrictions	14-106
14.16.1	Language Class Restrictions	14-107
14.16.2	Data Type Restrictions	14-108
14.16.3	Utility Class Restrictions	14-108
14.16.4	Struct Endianness and Byte Alignment Restrictions	14-109
14.16.5	Pointer Restrictions	14-110
14.16.6	Array Restrictions	14-110
14.16.7	Known Discrepancies between OSCI and IEEE	14-110
14.16.8	Restrictions Relating to SystemC 2.2	14-111

15 CtoS Libraries 15-1

15.1	Fixed-Point Library	15-1

15.1.1	Overview of Fixed-Point Data Types	15-1
15.1.2	CtoS Fixed-Point Library	15-5
15.1.3	API of CtoS Fixed-Point Library	15-10
15.2	Flex Channels Library	15-33
15.2.1	Illustrative Example	15-35
15.2.2	Flex Channels Terminology	15-39
15.2.3	Put/Get Channels Protocol	15-40
15.2.4	Parameterizing Structures with Traits	15-41
15.2.5	Initiators and Interfaces	15-44
15.2.6	Pipelined Design with Put/Get Channels	15-50
15.2.7	Hierarchical Initiators	15-54
15.2.8	Custom Traits	15-56
15.2.9	Automatic Configuration of Micro-Architecture	15-61
15.2.10	Simulation and Debugging of Flex Channels	15-62
15.3	TLM Library	15-65
15.3.1	Infrastructure	15-65
15.3.2	Considerations for Synthesis	15-68
15.3.3	Core Interfaces	15-68
15.3.4	Core Channels	15-72
15.3.5	Automatic Configuration of Micro-Architecture	15-74
16	Module Hierarchy	16-1
16.1	Understanding the Module Hierarchy	16-2
16.1.1	Preserved and Collapsed <code>sc_modules</code>	16-2
16.1.2	Factors that Cause an <code>sc_module</code> to Be Collapsed	16-2
16.1.3	Port Bundles	16-3
16.1.4	Uniquified <code>sc_modules</code>	16-3
16.2	Commands, Options, Attributes in Hierarchical Mode	16-3
16.2.1	The <code>build</code> Command and <code>build_flat</code> Attribute	16-3
16.3	Naming Differences between Hierarchical and Flat Modes	16-4
16.4	Integrating Multiple CtoS Designs	16-5
16.4.1	Prefix by SC Module	16-5
16.4.2	Name Module Prefix	16-6

17 Incremental Synthesis 17-1

17.1	Uses for Incremental Synthesis	17-2
17.1.1	Design Space Exploration	17-2
17.1.2	Design Flow Bug Fixing (ECO)	17-3
17.2	Incremental Synthesis Design Flow	17-4
17.2.1	Automated Incremental Synthesis Flow	17-5
17.2.2	Manual Incremental Synthesis Flow	17-6
17.3	Sample Incremental Synthesis Scripts	17-8
17.3.1	Sample Baseline Creation Script	17-8
17.3.2	Sample Incremental Synthesis Script	17-9
17.4	Incremental Synthesis Commands and Options	17-9
17.4.1	set_baseline Command	17-10
17.4.2	unbind_baseline Command	17-11
17.4.3	report_incremental Command	17-11
17.5	Changes Handled, and Not Handled, by Incremental Synthesis	17-11
17.5.1	Changes Handled by Incremental Synthesis	17-12
17.5.2	Changes Not Handled by Incremental Synthesis	17-12

18 Low Power Estimation Using CtoS 18-1

18.1	RTL Power Estimation Flow	18-1
18.2	Commands/Options/Attributes for Power Estimation	18-5
18.2.1	Setting Default Toggle Probability	18-5
18.2.2	Using the analyze Command	18-6
18.2.3	Using the -tcf Option of write_rtl and write_sim Commands	18-6
18.2.4	Using the report_power Command	18-7
18.2.5	Using the read_tcf Command	18-7
18.3	Considerations when Using INCISIV dumptcf	18-9
18.3.1	Scope Used in dumptcf Command	18-9
18.3.2	+access+r and -afile Options of dumptcf	18-9
18.3.3	Module Name Must Correspond to dumptcf Scope	18-9
18.4	Using Clock Gating	18-10
18.4.1	Fine Grain Clock Gating	18-10
18.4.2	Coarse Grain Clock Gating	18-11

A	CtoS Tutorial for ASIC designs	A-1
A.1	Starting, Setting Up and Building the Design	A-2
A.1.1	Starting the CtoS GUI	A-2
A.1.2	Starting and Setting Up the Design	A-4
A.1.3	Building the Design	A-8
A.2	Specifying Micro-Architecture	A-11
A.2.1	Inlining Functions	A-11
A.2.2	Breaking Combinational Loops	A-12
A.2.3	Allocating IP	A-12
A.2.4	Creating Latency Constraints	A-13
A.2.5	Generating a Verilog Simulation Model (_current)	A-14
A.3	Scheduling and Allocating Registers	A-14
A.4	Analyzing the Design	A-16
A.4.1	Understanding Scheduling Actions	A-16
A.4.2	Exploring Registers	A-17
A.4.3	Exploring Resources	A-19
A.5	Implementing the Design	A-20
A.5.1	Generating a Synthesizable RTL Model	A-21
A.5.2	Generating a Verilog Behavioral Model (_final)	A-22
A.5.3	Comparing Simulation Results	A-22
A.5.4	Generating Gates	A-25
A.5.5	Rerunning with Incremental Synthesis	A-26
B	CtoS Tutorial for FPGA designs	B-1
B.1	Starting, Setting Up and Building the Design	B-3
B.1.1	Starting the CtoS GUI	B-3
B.1.2	Starting and Setting Up the Design	B-4
B.1.3	Building the Design	B-9
B.2	Specifying Micro-Architecture	B-10
B.2.1	Inlining Functions	B-10
B.2.2	Breaking Combinational Loops	B-11
B.2.3	Allocating IP	B-11
B.2.4	Creating Latency Constraints	B-14
B.3	Scheduling and Allocating Registers	B-15
B.4	Analyzing and Implementing FPGA Designs	B-16

C Micro-Architectural Exploration Example C-1

C.1	Starting, Setting Up and Building the Design	C-1
C.1.1	Starting the CtoS GUI	C-2
C.1.2	Setting Up the Design	C-3
C.1.3	Building the Design (Flat)	C-4
C.2	Considering Loop Implementation	C-4
C.2.1	Option 1: RAM with 2 Read Ports, No Pipeline, Latency 1	C-7
C.2.2	Option 2: RAM with 2 Read Ports, No Pipeline, Latency 2	C-8
C.2.3	Option 3: RAM with 2 Read Ports, Pipeline	C-8
C.2.4	Option 4: RAM with 1 Read Port, No Pipeline, Latency 2	C-8
C.2.5	Option 5: RAM with 1 Read Port, Pipeline	C-8
C.3	Examining Implementation Options	C-9
C.3.1	Implementing Option 3: RAM with 2 Read Ports, Pipeline	C-9
C.3.2	Implementing Option 4: RAM with 1 Read Port, No Pipeline, Latency 2	C-18
C.3.3	Implementing Option 5: RAM with 1 Read Port, Pipeline	C-21

D CtoS Object Reference D-1

D.1	Design Hierarchy (Graphical Depiction)	D-3
D.2	Design Hierarchy (Tabular Depiction)	D-4
D.3	Object Identifiers (object_id's)	D-5
D.4	Object Attributes	D-5
D.5	Identifying Objects for Commands	D-6
D.6	Design Object Attributes (Designs)	D-9
D.7	Memory Bridge Definition Object Attributes (memory_bridge_defs)	D-29
D.8	Memory Bridge Port Def Object Attributes (memory_bridge_port_defs)	D-29
D.9	Memory Definition Object Attributes (memory_defs)	D-30
D.10	Memory Definition Auxiliary Port Object Attributes (aux_port_defs)	D-33
D.11	Module Object Attributes (Modules)	D-34
D.12	Array Object Attributes (Arrays)	D-36
D.13	External Array Binding Object Attributes (external_array_binding)	D-39
D.14	Pipeline Function Object Attributes (pipeline_functions)	D-40
D.15	Behavior Object Attributes (Behaviors)	D-41
D.16	Array Constraint Object Attributes (array_constraints)	D-46
D.17	Edge Object Attributes (Edges)	D-46
D.18	Floating Constraint Object Attributes (float_constraints)	D-48

D.19	Latency Constraint Object Attributes (latency_constraints)	D-49
D.20	Node Object Attributes (Nodes)	D-50
D.21	Operation Constraint Object Attributes (op_constraints)	D-56
D.22	Operation Object Attributes (Ops)	D-57
D.23	Operation Ports Object Attributes [(op_)ports]	D-60
D.24	Directive Object Attributes	D-61
D.25	Pin Object Attributes (Pins)	D-62
D.26	Protocol Constraint Object Attributes (protocol_constraints)	D-63
D.27	Register Binding Object Attributes (reg_bindings)	D-63
D.28	Reset Condition Object Attributes (reset_conditions)	D-65
D.29	Resource Binding Object Attributes (res_bindings)	D-65
D.30	Tag Object Attributes (Tags)	D-66
D.31	Behavior Terminal Object Attributes [(behavior_)terms]	D-67
D.32	Value Object Attributes (Values)	D-68
D.33	Instance Object Attributes (Insts)	D-69
D.34	Instance Terminal Object Attributes (inst_terms)	D-73
D.35	Instance Terminal Bits Object Attributes [(inst_term_)bits]	D-74
D.36	Net Object Attributes (Nets)	D-76
D.37	Net Bit Object Attributes [(net_)bits]	D-78
D.38	Module Terminal Object Attributes [(module_)terms]	D-79
D.39	Module Terminal Bits Object Attributes [(module_term_)bits]	D-81
D.40	RTL IP Definition Object Attributes (rtl_ip_defs)	D-83
D.41	RTL IP Definition Port Object Attributes [(rtl_ip_def_)ports]	D-84
D.42	Default Simulation Configuration Object Attributes (simulation_configs)	D-85
D.43	Default Synthesis Configuration Object Attributes (synthesis_configs)	D-86

E CtoS Command Reference E-1

E.1	Command Usage Related to CtoS Design Flow	E-2
E.1.1	add_command	E-5
E.1.2	add_message	E-6
E.1.3	allocate_builtin_ram	E-7
E.1.4	allocate_memory	E-9
E.1.5	allocate_memory_interfaces	E-11
E.1.6	allocate_prototype_memory	E-14
E.1.7	allocate_registers	E-18
E.1.8	analyze	E-20

E.1.9	apply_directive	E-22
E.1.10	apply_uarch_action	E-25
E.1.11	bind_value	E-26
E.1.12	break_array_dependency	E-27
E.1.13	break_array_inter_iteration_dependencies	E-28
E.1.14	break_combinational_loop	E-29
E.1.15	build	E-30
E.1.16	cd	E-31
E.1.17	check_design	E-32
E.1.18	close_design	E-33
E.1.19	connect	E-34
E.1.20	constrain_latency	E-36
E.1.21	constrain_op	E-38
E.1.22	convert_to_lookup	E-40
E.1.23	create_array_dependencies	E-41
E.1.24	create_initial_resources	E-42
E.1.25	create_protocol_region	E-43
E.1.26	create_register	E-45
E.1.27	create_required_states	E-46
E.1.28	create_resource	E-47
E.1.29	create_rom_program	E-49
E.1.30	create_state	E-50
E.1.31	ctos (executable)	E-51
E.1.32	ctosgui (executable)	E-52
E.1.33	define_clock	E-53
E.1.34	define_control_error_terminal	E-54
E.1.35	define_sim_config	E-55
E.1.36	define_synth_config	E-56
E.1.37	define_tlm_transactor_pair	E-57
E.1.38	external_delay	E-58
E.1.39	find	E-60
E.1.40	find_by_annotation_path	E-62
E.1.41	find_combinational_loops	E-63
E.1.42	find_from_rtl	E-64
E.1.43	find_in_rtl	E-65
E.1.44	find_source	E-66

E.1.45	flatten_array	E-68
E.1.46	flex_channels_nets	E-69
E.1.47	float_array_accesses	E-70
E.1.48	float_io_accesses	E-71
E.1.49	get_attr	E-73
E.1.50	get_design	E-74
E.1.51	get_install_path	E-75
E.1.52	get_version	E-76
E.1.53	help	E-77
E.1.54	inline	E-78
E.1.55	inline_calls	E-79
E.1.56	launch_sim	E-80
E.1.57	lcd	E-81
E.1.58	list_attr	E-82
E.1.59	lls	E-83
E.1.60	lpwd	E-84
E.1.61	ls	E-85
E.1.62	merge_arrays	E-86
E.1.63	new_design	E-88
E.1.64	open_design	E-89
E.1.65	optimize	E-90
E.1.66	pipeline_function	E-91
E.1.67	pipeline_loop	E-92
E.1.68	print_message	E-95
E.1.69	pwd	E-96
E.1.70	read_ip_defs	E-97
E.1.71	read_tcf	E-98
E.1.72	remove_clock	E-100
E.1.73	remove_directive	E-101
E.1.74	rename_state	E-102
E.1.75	report_area	E-103
E.1.76	report_array_dependencies	E-105
E.1.77	report_behavioral_power	E-106
E.1.78	report_incremental	E-107
E.1.79	report_latency	E-108
E.1.80	report_opspan	E-109

E.1.81	report_paths	E-114
E.1.82	report_power	E-116
E.1.83	report_registers	E-117
E.1.84	report_resources	E-119
E.1.85	report_sccs	E-121
E.1.86	report_schedule	E-125
E.1.87	report_slack	E-126
E.1.88	report_states	E-127
E.1.89	report_summary	E-128
E.1.90	report_timing	E-130
E.1.91	report_tlm_transactor_pairs	E-135
E.1.92	restructure_array	E-136
E.1.93	save_design	E-137
E.1.94	schedule	E-138
E.1.95	set_attr	E-140
E.1.96	set_baseline	E-141
E.1.97	set_design	E-143
E.1.98	set_stall_value	E-144
E.1.99	set_synthesis_mode	E-145
E.1.100	set_uarch_action	E-146
E.1.101	split_array	E-147
E.1.102	split_op	E-148
E.1.103	undefine_control_error_terminal	E-149
E.1.104	undefine_tlm_transactor_pair	E-150
E.1.105	unroll_loop	E-151
E.1.106	unschedule	E-153
E.1.107	use_dsp	E-154
E.1.108	use_ip	E-156
E.1.109	write_gates	E-157
E.1.110	write_rc_run_script	E-158
E.1.111	write_rc_script	E-159
E.1.112	write_rename_states_script	E-160
E.1.113	write_rtl	E-161
E.1.114	write_setup	E-164
E.1.115	write_sim	E-165
E.1.116	write_sim_makefile	E-168

E.1.117	write_synth_makefile	E-169
E.1.118	write_tlm_wrapper	E-170
E.1.119	write_top_wrapper	E-173
E.1.120	write_verilog_wrapper	E-175
E.1.121	write_wrapper	E-176
F	CtoS Example Reference	F-1
G	Structure of the Verification Wrappers	G-1
G.1	Structure of the SystemC Verification Wrapper	G-1
G.1.1	Overview of SystemC Verification Wrappers	G-2
G.1.2	Typical Use Cases for SystemC Verification Wrappers	G-3
G.1.3	Wrapper Module Constructor Arguments	G-3
G.1.4	Other Customizable SystemC Wrapper Member Data	G-5
G.1.5	Converting Inputs and Outputs	G-6
G.1.6	Comparing Outputs	G-6
G.1.7	Comparing Internal Signals	G-6
G.1.8	Requirements for SC_MODULE Ports	G-7
G.1.9	Limitations When Writing Wrappers	G-7
G.2	Structure of the Verilog Verification Wrapper	G-9
G.2.1	Top Level Verilog Verification Wrapper	G-11
G.2.2	Language Adaptor	G-11
G.2.3	Type Wrapper	G-11
H	IP Definition Files for Memories and RTL IP	H-1
H.1	IP Definition (XML) File Description	H-2
H.1.1	RAMdef XML Elements	H-3
H.1.2	ROMdef XML Elements	H-4
H.1.3	rtl_ip_def XML Elements	H-5
H.1.4	memory_bridge_def XML Elements	H-5
H.1.5	cgic_ip_def XML Elements	H-6
H.2	Vendor RAMs	H-6
H.2.1	Using Wrapper Files to Allocate Vendor RAMs	H-7
H.2.2	Using Memory Bridges to Allocate Vendor RAMs	H-9
H.2.3	Specifying Minimum Write Width in Vendor RAMs	H-18

H.2.4	Specifying Reset Behavior in Vendor RAMs	H-18
H.2.5	Specifying Negative Edge Clock in Vendor RAMs	H-18
H.2.6	Specifying Vendor RAM Library File/Simulation Model	H-19
H.2.7	Specifying Auxiliary Ports in Vendor RAMs	H-19
H.2.8	Using Vendor RAMs in FPGA Designs	H-20
H.2.9	Access Protocols (interface_types) in Vendor RAMs	H-21
H.2.10	Verilog Interface Per-Port Terminals for Vendor RAM	H-22
H.2.11	How CtoS Binds Processes to Vendor RAMs	H-23
H.2.12	Limitations Reported during Scheduling and Model Generation	H-24
H.3	Vendor ROMs	H-25
H.3.1	ROM Standard Verilog Ports	H-26
H.3.2	Vendor ROM Library File and Simulation Model	H-26
H.3.3	Vendor ROM Example	H-26
H.4	RTL IP	H-28
H.4.1	Restrictions of RTL IP Pipeline Depth	H-28
H.4.2	RTL IP Example	H-29
I	Debugging Simulation Mismatches	I-1
I.1	Introduction	I-2
I.2	Debug Flow	I-3
I.3	Hints for Adding Observation Points	I-4
I.3.1	Naming with a Specific Prefix	I-4
I.3.2	Observing the Control State of a Process	I-5
I.3.3	Observing Variables in Zero-Delay Loops	I-5
I.4	Scheduling and RTL	I-5
I.5	Caveats	I-6
I.5.1	Some Variables Should Not Be Observed Directly	I-6
I.5.2	Instrumentation Changes the Design	I-6
I.5.3	Instrumentation Deteriorates QoR	I-6
J	Controlling CtoS-Generated Verilog	J-1
J.1	Controlling Case Statement Generation	J-1
J.1.1	Controlling Select Expression, Case Label Generation	J-2
J.1.2	Controlling Default Case	J-2
J.2	Controlling Mismatched Operand Bit Widths	J-3

J.3	Controlling pragma Keyword for Generated RTL	J-4
J.4	Controlling Use of Wire Arrays to Model Lookup Resources	J-4
J.5	Controlling Use of Indexed Part Selects	J-5
J.6	Controlling Use of Delay Control with Non-Blocking Assignments	J-6
J.7	Other Verilog-2001 Constructs Used in Model Generation	J-7
J.7.1	Signed Arithmetic Extensions	J-7
J.7.2	Combinational Logic Sensitivity	J-7
J.7.3	Combined Port and Data Type Declarations	J-8
K	Logic Synthesis Steps	K-1
L	Control Flow Graph (CFG) Viewer	L-1
L.1	Navigating the CFG Viewer	L-2
L.2	Exploring with Annotation in the CFG Viewer	L-3
L.3	“Detail Bubbles” for Objects in the CFG Viewer	L-4
L.4	Canvas, Loops, Function Call Loops Supported	L-7
L.5	Tooltips Provide Helpful Information	L-8
L.6	Pipelined Loops Have Distinct Glyph	L-8
M	Release Notes for 14.10	M-1
M.1	What’s New	M-2
M.1.1	Ease of Use	M-2
M.1.2	QoR Improvements	M-8
M.1.3	Improved Flow Integration	M-9
M.2	Deprecated Items	M-10
M.2.1	schedule -useBirthday Option Renamed	M-10
M.3	Changes to CtoS-Supported OS Platforms and Software Versions	M-10
M.4	Known Limitations	M-10
M.4.1	Pipelined Loops with Reads and Writes to Same Array	M-11
M.4.2	Limited Support for Multiple Clocks	M-12
M.5	Known Problems with Workarounds in 14.10	M-13
M.6	Problems Resolved/Enhancements Made in 14.10	M-20
N	Glossary	N-1

1 Release Notes

The **14.20-S400** release of Cadence C-to-Silicon Compiler (CtoS) is an update release for the **14.20** release of December 2014. For description of problems resolved in this update release, see [“Problems Resolved in 14.20 s400” on page 1-13](#).

This chapter also describes the new features and enhancements of **Cadence C-to-Silicon Compiler (CtoS)**, version **14.20**.

- **What’s New**
 - **Improved Pipeline Support**
 - **Enhanced MicroArch Indicates if a Loop Can be Pipelined**
 - **Stall Propagation**
 - **Improved QOR for Pipelines with Multiple Stalls**
 - **Improved Verification**
 - **Better Handling of State Node Names in Support of FSM Coverage**
 - **Improved Ease of Use**
 - **Shorten Names with Templates**
 - **Scheduler Output Includes QOR for each Level of Scheduling Effort**
 - **Improved Run Time**
 - **Scheduling**
 - **Register Allocation**
 - **Timing Reports**

- **Deprecated Items**
- **Changes to CtoS-Supported OS Platforms and Software Versions**
- **Known Limitations**
- **Known Problems with Workarounds in 14.20**
- **Problems Resolved/Enhancements Made in 14.20**

1.1 What's New

The **Cadence C-to-Silicon Compiler (CtoS)**, production version **14.20**, includes the following new features and enhancements:

- “Improved Pipeline Support” on page 1-2
- “Improved Verification” on page 1-4
- “Improved Ease of Use” on page 1-4
- “Improved Run Time” on page 1-5

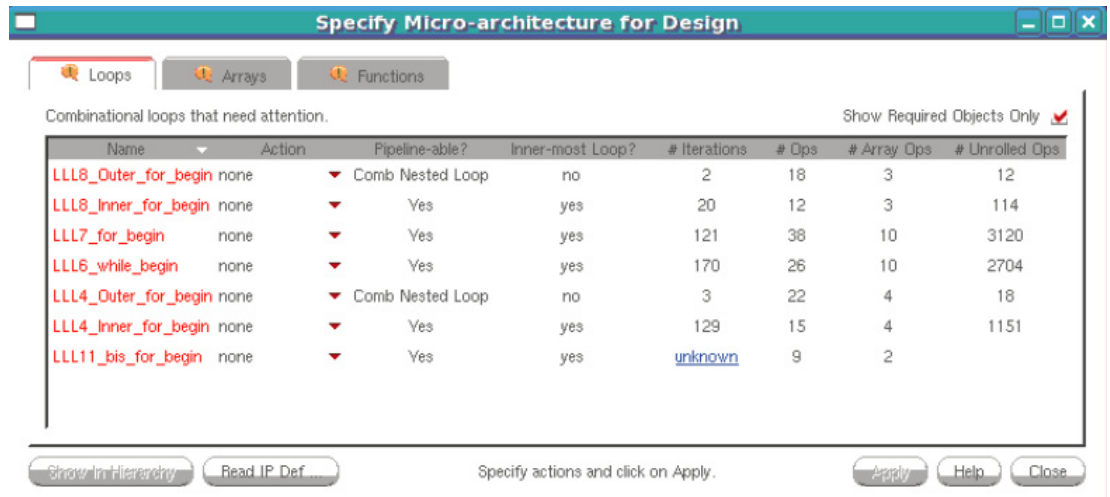
1.1.1 Improved Pipeline Support

- “Enhanced MicroArch Indicates if a Loop Can be Pipelined” on page 1-2
- “Stall Propagation” on page 1-3

1.1.1.1 Enhanced MicroArch Indicates if a Loop Can be Pipelined

The **Specify Micro-architecture Manager** dialog has been enhanced to display the **Pipeline-able?** column in the **Loops** tab. The column displays "Yes" if a loop can be pipelined or lists the reason why a loop cannot be pipelined. Moving the cursor over the message shows a tooltip why the loop cannot be pipelined. For example, in [Figure 1-1](#) the **LLL8_Outer_for_begin** loop cannot be pipelined because it is a combinational nested loop.

Figure 1-1 Specify Micro-architecture - Pipeline Loop



1.1.1.2 Stall Propagation

In the 14.2 release, CtoS introduces stall propagation that enables users to specify a value that a particular net should take, if the pipelined loop writing to the net is stalled. The net may correspond to an **sc_out** or an **sc_signal**. The main benefit is that simple implementations of common communication protocols can be made stallable. Without it, more complicated implementations with side buffers may have been used.

The new command, **set_stall_value** lets you declare the value of a net should be set when the pipelined loop is stalled. For more information on how to make protocols stallable see “[Creating Protocol Regions](#)” on page 11-14 and for more information on the command, see “[set_stall_value](#)” on page E-144.

1.1.1.3 Improved QOR for Pipelines with Multiple Stalls

The stall logic for pipelined loops with multiple stalls has been improved. The new implementation requires fewer registers and less logic thus improving its timing and area.

1.1.2 Improved Verification

1.1.2.1 Better Handling of State Node Names in Support of FSM Coverage

In 14.2 release, the RTL generated by CtoS can be used to analyze FSM Coverage in Incisive Metric Center (IMC). The IMC tool recognizes the states in the generated RTL. CtoS was further enhanced to let you name the states so that they are more recognizable than the state names automatically generated by CtoS. Finally, a new technique (**annotation_paths**) was added to make renaming of states robust even in light of changes to your design.

For more information, see [“Renaming States in Support of FSM Coverage” on page 7-25](#).

1.1.3 Improved Ease of Use

- [“Shorten Names with Templates” on page 1-4](#)
- [“Scheduler Output Includes QOR for each Level of Scheduling Effort” on page 1-4](#)

1.1.3.1 Shorten Names with Templates

If a design uses user-defined template classes and/or template functions, the resulting names for modules, behaviors, and behavior terms will be very long. A new design attribute **name_use_template_parameters** lets you control whether template parameters must be included in the names of modules, behaviors, and behavior terms. The default value is **true**, which results in the naming scheme used in CtoS 14.1.

If the attribute is set to **false** before **build**, template parameters are not included in the names of modules, behaviors, and behavior terms. For more information, see the **Naming** tab of [“Design Property Dialog” on page 6-18](#) and [“name_use_template_parameters” on page D-20](#).

1.1.3.2 Scheduler Output Includes QOR for each Level of Scheduling Effort

For some behaviors, the scheduler will advise you to rerun with a different **scheduling_effort** when QOR is estimated to be better. In this release, the scheduler also outputs QOR estimates for each level of scheduling effort so that you can decide if the QOR improvement is worth the time of rerunning the scheduler.

For more information, [“Recommended Scheduling Effort” on page -13](#).

1.1.4 Improved Run Time

- “Scheduling” on page 1-5
- “Register Allocation” on page 1-5
- “Timing Reports” on page 1-5

1.1.4.1 Scheduling

Scheduling large designs is faster, especially if:

- there are multiple clocks
- The design has a **baseline_dir** (ECO mode)
- there are many registers and states
- the design has deep pipelines

1.1.4.2 Register Allocation

Register allocation is faster on designs that have large arrays that are flattened.

1.1.4.3 Timing Reports

report_timing and **report_slack** are faster on large designs with multiple modules.

1.2 Deprecated Items

The design attribute, **enable_side_by_side_debug** has been removed from this release. The SimVision plug-in, **CtoS Side-by-Side Viewer (CSV)** is no longer supported for debugging designs in the INCISIV/SimVision environment.

1.3 Changes to CtoS-Supported OS Platforms and Software Versions

The following changes have been made, in this release, to the CtoS-supported OS platforms and software versions:

- Cadence's Encounter RTL Compiler (RC) version upgraded from version 14.10-s012 to 14.10-s022.

Note Logic synthesis using RC 14.1 generates better clock gating logic than previous versions. For example, RC 14.1 no longer adds clock gating logic in cases where the enable is rarely de-asserted. In some cases, this may result in a lower percentage of gated flip-flops but a better result overall.

- Cadence's Incisive Enterprise Simulator (INCISIV) upgraded from version 13.1 to 13.2.

1.4 Known Limitations

Here are the known limitations, at present:

- [“Pipelined Loops with Reads and Writes to Same Array” on page 1-6](#)
- [“Limited Support for Multiple Clocks” on page 1-8](#)

1.4.1 Pipelined Loops with Reads and Writes to Same Array

If a pipelined loop contains reads and writes to the same array, the CtoS scheduler will ensure that the order of write operations, with respect to reads or other writes, is not affected by pipelining.

To make sure this is handled correctly, CtoS schedules writes so they are not separated by *more than one II* (*initiation interval*) from related reads and writes. Thus, reads and writes from future iterations are not folded in between reads and writes of this iteration.

The scheduler, however, may be overly conservative in determining which reads and writes are *related*.

For example, consider this loop:

```
while (c) {
    mem[i] = x;
    wait();
    wait();
    y = mem[j];
}
```

Unless the scheduler can ensure that **mem[i]=x** could never write to the same address – in a future iteration – as is read by **y=mem[j]** – in this iteration – the read and write will be scheduled within one **II** (initiation interval) of each other.

You may *know* that two operations are *not related* and find the scheduler too restrictive, that is, if you knew that **i** – in a future iteration – could never equal **j** – in this one – this restriction would be unnecessary.

Consider the following example:

```
LOOP: for (unsigned i = 0; i < n; i++) {
    m[i] = f(m[i]);
    W: wait();
}
```

In this case, CtoS *can* schedule the loop if the memory has a read and write port. There is no dependency between the write of the first iteration (**m[0]**) and the read of the next iteration (**m[1]**). Thus, the CtoS scheduler is able to prove that no two iterations of the loop are dependent on each other and is free to schedule the write at any stage after the read.

However, in more complex cases, manual analysis is required.

Consider the following example:

```
LOOP: for (unsigned i = 0; i < n; i++) {
    m[i+2] = f(m[i]);
    W: wait();
}
```

In this case, it is possible to prove that iteration **i** writes to a location that will be read only two iterations later. Hence, the write at a given iteration can be scheduled *at most* two stages after the read – otherwise, the read two iterations later would read the original value.

You can override this restriction by using the **-type hard** option of the **constrain_op** command to manually schedule read and write operations within the loop. If two memory operations both have fixed constraints, the scheduler will treat them as unrelated.

Here are two examples of using the **-type hard** option of the **constrain_op** command, depending on whether the loop is pipelined:

- If the loop is *not* pipelined, and the memory ops have been constrained using the **constrain_op** command, as shown below, the memory dependency between the write and the next read is ignored:

```
constrain_op -type hard -edge ../edges/LOOP_for_begin ../ops/memread_m_ln99
constrain_op -type hard -edge ../edges/W ../ops/memwrite_m_ln99
```

- If the loop *is* pipelined and the memory ops have been constrained using the **constrain_op** command, as shown below, the memory dependency between the write and the next read is ignored:

```
constrain_op -type hard -stage 1 .../ops/memread_m_ln99
constrain_op -type hard -stage 2 .../ops/memwrite_m_ln99
```

This eliminates the following potential problems, related to scheduling a design with such a loop:

- A memory read is forced to be later, or a memory write to be earlier, thereby reducing the scheduling opportunities for other ops in the loop.
- In extreme cases, the design cannot be scheduled without adding a state.

Note It is the designer's responsibility to ensure that array accesses are never to the same address.

1.4.2 Limited Support for Multiple Clocks

CtoS provides limited support for designs with multiple clocks.

If a design has more than one clock, one must be the *base*, and its frequency must be a multiple of all other frequencies.

Also, CtoS will not create circuitry to ensure clock domains are properly interfaced; defining such circuitry is your responsibility.

Note See also "Create New Design Wizard" on page 6-10.

1.5 Known Problems with Workarounds in 14.20

Here are the known problems, with workarounds, at the time of this release.

CCR Number	Known Problem	Workaround
<i>Problems related to Opening and Saving Designs</i>		
534693	When you open a saved design, the working directory might be different from the directory at the time of the save, and you cannot rebuild or use the CtoS GUI's <i>Show in Source</i> functionality.	Use the lcd command to change the working directory.

CCR Number	Known Problem	Workaround
<i>Problems related to SystemC/C++ Support</i>		
487877	CtoS could issue this error during the build process for designs with internal communication via a user-defined sc_port<IF> whose channel methods and methods that invoke the port method are defined in different compile units (different .cpp files): <pre>ERROR (CTOS-11115): undefined port method.</pre>	For a workaround, see “ Coding Requirements for Multi-Source Designs ” on page 6-30.
<i>Problems related to Specifying Micro-architecture</i>		
814763	CtoS could issue this error if you try to convert the implementation of a combinational function into a <i>table lookup</i>, and that function has a multiply, divide, or modulo op with inputs or outputs with more than 64 bits (63 bits if unsigned): <pre>ERROR (CTOS-17031): Cannot convert function '<FUNCTION>' to a lookup table because CtoS had a problem evaluating the function. Error in processing command convert_to_lookup.</pre>	<i>Workaround 1:</i> Decrease the width of the variable declarations (if it will not affect the precision of your result). <i>Workaround 2:</i> Use the split_op command (“split_op” on page E-148) to split large multiplies into smaller multiplies. <i>Workaround 3:</i> Rewrite divides as multiplies by the reciprocal.
954405	CtoS could issue this error when trying to unroll loops where the starting point for the iterations is a variable: <pre>ERROR (CTOS-17014): Cannot unroll loop 'loop1_for_begin' because it does not terminate after 512 iterations ...</pre>	Rewrite the loop so the initial value of the iterator is a constant.
<i>Problems related to Allocating Memory and RTL IP</i>		
685020	Only one function per thread should access a memory. If multiple functions access arrays stored in the same memory, the memory access order may not be preserved if two functions in a thread access the same memory.	If multiple functions access arrays stored in the same memory, you must inline these functions until only one function accesses the memory. At this point, the memory accesses will be well-ordered.

CCR Number	Known Problem	Workaround
685527	Generated RC scripts (File -> Generate -> RTL or write_rc_script command) may not have input and output delays for ports added to interface to memory external to an RTL module using the Exported Memories feature. The delays should be based on the library specified in the IP definitions file.	Add these delay constraints manually to synthesize the RTL separately from the memory.
<i>Problems related to the CtoS Optimizer</i>		
623328	When two unaligned arrays in the same structure are passed to a function, CtoS does not perform sufficient optimizations. Address calculations are not optimized, which may generate dead code in the function, especially if the size of each array is not a <i>power of two</i>.	Add members to align the arrays to a common <i>power of two</i> boundary to simplify the address calculation.
662227	CtoS sometimes generates larger than expected resources when there are signed array index expressions for a multiple-dimension array.	The problem is subtraction in the array indexing: <pre>int mem[MEM_SIZE_2][MEM_SIZE]; x = mem[i][j-1];</pre> If you make sure the index is positive (assuming <code>MEM_SIZE</code> is a <i>power of two</i>), QoR will improve: <pre>x = mem[i][(j-1) & (MEM_SIZE - 1)];</pre>

CCR Number	Known Problem	Workaround
<i>Problems related to the CtoS Scheduler</i>		
865042	The CtoS scheduler merges two 2-1 muxes to create a 3-1 mux to improve timing (and, to a lesser degree, area). The problem is that it causes CtoS to allocate nine extra registers.	By re-timing the loop, you can prevent CtoS from merging these muxes. The following code is a functionally equivalent form in which CtoS cannot merge the muxes: <pre>// layer 1 void sample::thread0() { rw0 = 0; rd.write(0); while(1){ ////////// WAIT wait(); ////////// register clear if (clear.read()){ rw0 = 0; } ////////// register read if (re.read()){ rd.write(rw0); }else{ rd.write(0); } ////////// register write if (we.read()){ rw0 = wd.read(); } } }</pre>

CCR Number	Known Problem	Workaround
<i>Problems related to Register Allocation/Netlisting</i>		
958054	Extra (unnecessary) registers are allocated to store values computed in an early stage of a pipelined loop, but are used only after the pipelined loop exits.	Rewrite the code for the design with the array as an sc_signal. This change is not simply a matter of changing the declaration, because values written to the array are not visible until the next cycle, so you must ensure there is a wait between a write and a read.
1071739	In some cases, CtoS creates data flow registers instead of using equivalent output register.	If register sharing is required to reduce the number of registers in the design, then following modeling should be used (It may be difficult to achieve in complex code). And, always load and save the value to the output: <pre> for (...) { sc_uint<4> data; wait(); while (...) { data = <expression>; out.write(data); wait(); data = out.read(); } wait(); <another use of data> } </pre> By assigning to 'data' after every wait() CtoS will be able to recognize that 'data' has the same value as 'out' and share the output register.

1.6 Problems Resolved/Enhancements Made in 14.20

In the CtoS 14.20 release, the following problems have been resolved and enhancements have been made:

- “Problems Resolved in 14.20 s400” on page 1-13
- “Problems Resolved in 14.20 s302” on page 1-14
- “Problems Resolved in 14.20 s301” on page 1-14
- “Problems Resolved in 14.20 s300” on page 1-14
- “Problems Resolved in 14.20 s201” on page 1-15
- “Problems Resolved in 14.20 s200” on page 1-15
- “Problems Resolved in 14.20 s101” on page 1-16
- “Problems Resolved in 14.20 p100” on page 1-17

1.6.1 Problems Resolved in 14.20 s400

The following are the problems resolved and enhancements made, at the time of this release:

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Timing Analyzer</i>		
1583176	Multiple reset for Flex Channels.	This issue has been fixed. CtoS now supports multiple reset for Flex Channels designs. For more information, see “Asynchronous and Multiple Resets” on page 15-60.

1.6.2 Problems Resolved in 14.20 s302

The following are the problems resolved and enhancements made, at the time of this release:

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Timing Analyzer</i>		
1559268	CtoS was unable to invoke RC with Stratus License.	This issue has been fixed. The code has been fixed to accept Stratus license type as valid license for invoking RC.

1.6.3 Problems Resolved in 14.20 s301

The following are the problems resolved and enhancements made, at the time of this release:

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Incremental Synthesis (ECO mode)</i>		
1512039	Internal Error in ResMap::validate() in ECO mode.	This issue has been fixed.
<i>Resolved Problems and Enhancements Made, related to Scheduler</i>		
1521478	ERROR (CTOS-13110): Internal error in post-scheduler optimizations.	This issue has been fixed.

1.6.4 Problems Resolved in 14.20 s300

The following are the problems resolved and enhancements made, at the time of this release:

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to RC</i>		
1485689	The create_initial_resources command failed and a Segmentation Fault was reported in RTL Compiler.	Cadence's Encounter RTL Compiler (RC) version has been upgraded from version 14.10-s012 to 14.10-s022.

1.6.5 Problems Resolved in 14.20 s201

The following are the problems resolved and enhancements made, at the time of this release:

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Model Generation</i>		
1441860	Need memory bridge to support multiple resets for the registered memories.	The issue has been fixed. CtoS now supports multiple resets specification for memory bridges.

1.6.6 Problems Resolved in 14.20 s200

The following are the problems resolved and enhancements made, at the time of this release:

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Model Generation</i>		
1422686	CtoS generated incorrect RTL for the assignment to the partial write term of a memory. An incorrect expression is generated which contains a mismatched number of bits.	The issue has been fixed.
1446367	Compilation fails for some header files in the ctos_tlm include directory when the 'pedantic' compiler flag is specified.	The header files have been fixed.
<i>Resolved Problems and Enhancements Made, related to Scheduler</i>		
1447176	When sc_signal is accessed by several processes with different clocks its external delay setting was not set properly.	The issue has been fixed.

1.6.7 Problems Resolved in 14.20 s101

The following are the problems resolved and enhancements made, at the time of this release:

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Optimizer</i>		
1354676	CtoS inadvertantly optimizes out a loop when the while-loop condition does not have any logic operators.	The issue has been fixed. The loop is no longer optimized out.
<i>Resolved Problems and Enhancements Made, related to Register Allocation</i>		
1347228	Synthesis of designs to Altera's Arria V family can result in the following error: <pre>ERROR (CTOS-22028): The Altera resource analyzer could not determine the area for resource 'mux_2_8'.</pre> This may occur during allocate registers.	The issue has been fixed.
<i>Resolved Problems and Enhancements Made, related to Register Allocation (cont'd)</i>		
1356113	Register allocation of design with exported memories may result in the following internal errors: <pre>MyModule.my_mem_A1[8] is unconnected. MyModule.my_mem_A1[8] is unconnected. ... ERROR (CTOS-13110): Internal error while executing allocate_registers: stgModule.cpp:3633: isConnectionComplete(true)</pre> This may occur when distinct modules in the SystemC source have instances of the same sub module, which has exported memories.	The issue has been fixed.

CCR Number	Problem/Enhancement	Resolution
1366254	CtoS may crash during allocate registers with the following error: ERROR (CTOS-13111): Internal error while executing allocate_registers: regTagAlloc.cpp:535: m_availTotal==calcAvailRegs(), 'if_or_and_switch_lnXX_15_0_Z_0'	The issue has been fixed.
<i>Resolved Problems and Enhancements Made, related to Model Generation</i>		
1393974	CtoS may generate RTL that contained unreachable case items in a verilog case statement.	The issue has been fixed. CtoS no longer generates such verilog case statements.
<i>Resolved Problems and Enhancements Made, related to Pipelining</i>		
1407468	Pipeline folding may incorrectly move an IF op before the op that generates the control value for its tag. This results in the internal error: ERROR (CTOS-13112): Internal error while executing schedule: dfgOpValidator.cpp:331: tagSrcOp->dominates(m_op, true)	The issue has been fixed.

1.6.8 Problems Resolved in 14.20 p100

The following are the problems resolved and enhancements made, at the time of this release:

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to CtoS Build</i>		
1008722	If the design uses user-defined template classes and/or template functions very long names for modules, behaviors, and behavior terms may result. This is an enhancement request to reduce the length of names by excluding template_parameters in the name.	The enhancement has been implemented with the addition of design attribute, name_use_template_parameters . For more details, see “Shorten Names with Templates” on page 1-4.

CCR Number	Problem/Enhancement	Resolution
1292616	An internal error occurs in the build command on a design that uses a struct defined in the body of a member function of a template class. For example: <pre>template<T> struct STR { int func() { struct LOCAL_STR {..}; LOCAL_STR str1; .. }; };</pre>	The issue has been fixed.
1297413	The build command fails when the design has an array of flex_channel put_initiator and a process calls a method on an element of this array using a non-constant array index. For example: <pre>SC_MODULE(DUT) { SC_CTOR(DUT) { .. } put_initiator<int> m_dout[2]; void proc() { .. for (int i = 0; i < 2; i++) { m_dout[i].reset_put(); } .. } };</pre>	The issue has been fixed and build now passes.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to CtoS Build (cont'd)</i>		
1302273 1045442	The build command failed to recognize an <code>sc_trace</code> function when it is defined as a friend of a template class (DATA), and DATA is used as the datatype of a <code>sc_in<T></code> or <code>sc_out<T></code> of a template module. This results in rejection of the design in build with a message: <pre>"no instance of overloaded function "sc_core::sc_trace" matches the argument list .."</pre> For example: <pre>template<typename T> struct DATA { T m_val; bool operator==(const DATA &that) {return m_val == that.m_val;} bool operator!=(const DATA &that) {return m_val != that.m_val;} friend void sc_trace(sc_trace_file *, const DATA &, const std::string &) { /* ... */ } friend ostream& operator << (ostream& os, DATA const &) { return os; } }; template<int N> SC_MODULE(SUB) { SC_CTOR(SUB) { /* ... */ sc_in< DATA<int> > m_in; .. }; int sc_main(int argc, char *argv[]) { SUB<1> sub1("sub1"); } };</pre>	The issue has been fixed.
1303975	An internal error happens in the build command on a design that contains multiple instances of a module that has an external array.	The issue has been fixed. CtoS now completes build on the original design.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to CtoS Build (cont'd)</i>		
1310206	The build command may issue the following internal error: <pre>ERROR (CTOS-13110): Internal error while executing build: bstVarTermCorrespImpl.cpp:387: !set.containsKey(var)</pre> This may occur if the design has a function that passes the address of a formal argument into a call to another function that does pointer arithmetic and/or pointer comparison AND pointer dereference on given address. A simplified scenario that exercises the problem is as follows: <pre>int func1(int *v) { return v ? *v : 0; // This function uses both given address and the dereference of that address. } int func2(int v) { return func1(&v); // The address of a formal argument is passed to func1(). } SC_MODULE(DUT) { void proc() { .. func2(..); .. } };</pre>	The issue has been fixed.
1319242	An internal error occurs in build if the design uses an un-named struct that has a non-trivial default constructor. For example: <pre>struct { sc_uint<8> a[100]; sc_uint<8> b[100]; } c;</pre>	The issue has been fixed.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to CtoS Build (cont'd)</i>		
1322235	The build command issues the following internal error: <pre>ERROR (CTOS-13110): Internal error while executing build: sceDFG.cpp:4735: result</pre> This may occur when design has a template function with a formal argument of type 'const T' and the function is called with an array as the actual argument. <pre>template<typename T> int func(const T value) { return value[0] + value[1] + value[2]; } SC_MODULE(DUT) { void proc() { int arr[3]; .. int v = func(arr); .. } };</pre>	The issue has been fixed.
<i>Resolved Problems and Enhancements Made, related to the Save/Open</i>		
1345768	The save_design command may result in a segmentation violation after scheduling a design with pipeline loops	The issue has been fixed.
<i>Resolved Problems and Enhancements Made, related to the Specifying Micro-Architecture</i>		
1155558	If the design uses a multi-dimensional array for which one of the inner dimensions is not a power of 2, multipliers may result from the array accesses. To make the user aware of this situation the flatten_array command should issue a warning. If the functionality of the design is unaffected by increasing the size of that inner dimension to a power of 2, making this change to the design source code may improve QoR.	CtoS now issues warning CTOS-28490 if flatten_array is applied to a multi-dimensional array for which one of the inner dimensions is not a power of 2.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the Optimizer</i>		
1283573	The schedule command may issue the following internal error when scheduling a behavior and there are other behaviors in the same module that are not synthesizable (still have required micro-arch actions): <pre>ERROR (CTOS-13110): Internal error while executing schedule: modOpDepGraph.cpp:4027: m_minArea > 0</pre> This may occur when a design contains a combination function that has writes to sc_signals (nets).	The issue has been fixed.
1305772	The CtoS optimizer might issue the following error: <pre>ERROR (CTOS-13110): Internal error while executing schedule: dfgOp.cpp:3587: getBehavior()->isCombinational()</pre>	The issue has been fixed.
<i>Resolved Problems and Enhancements Made, related to Timing Analyzer</i>		
1323631	CtoS may incorrectly estimate the timing and area for large 'select' and 'replace' resources causing the generated RTL to have non-optimal QOR.	The issue has been fixed.
<i>Resolved Problems and Enhancements Made, related to Pipelining</i>		
1212583	Enhancement request to support stall propagation.	CtoS has been enhanced to support stall propagation with the new set_stall_value command. For more details see “Stall Propagation” on page 1-3.
1278011	During the pipeline_loop command, states may be added to the loop in an unexpected location if there is I/O in a function called from the pipelined loop.	The pipeline_loop will fail and request the user to inline any functions that called from the loop if they include I/O and the pipeline loop command may need to add states to the loop.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Pipelining (cont'd)</i>		
1287930	During pipeline folding (end of the schedule command), CtoS may issue the following internal error: <pre>ERROR (CTOS-13112): Internal error while executing schedule: dfgOpValidator.cpp:320: tagSrcOp->dominates(m_op, true), 'WAIT_GET_while_begin_a2_enable_if', 'write_X_o_line_lnXX'</pre> This may occur when the design has a pipelined loop with enable_multiple_pipeline_stalls=true. Ops may be incorrectly ordered when folding the pipelined loop after scheduling.	The issue has been fixed.
1288730	An internal error with SIGSEGV may occur at end of the schedule command with these entries in the stack trace: <pre>0x13c92f0: _ZNK3dfg3Tag13isPipelineTagEv 0xd04a92: _ZN3mod16LifetimeAnalyzer22tagPipel ineMuxInputUseEv</pre> The problem may occur when there is an <code>sc_signal</code> (<code>stg::Net</code>) that is read, but not written (or driven by an input Term or output InstTerm) and this value is used in an IF statement contained in a pipelined loop.	The issue has been fixed.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Pipelining (cont'd)</i>		
1326175	If the pipeline loop fails to transform a loop to a form that can be pipelined, the pipeline_loop command may fail with the following two errors: <pre> ERROR (CTOS-20439): The 'if-loop' transform cannot be applied to pipelined loop 'main_pipe_while_while_begin'. ERROR (CTOS-13110): Internal error while executing pipeline_loop: stgModule.cpp:3579: !mem->getMemWrites().isEmpty() (getDesign()->getTopWrapper() == this) </pre> The may occur in loop with complicated control structure (includes many calls to 'break') and also contains an array for which all accesses are optimized away.	The issue has been fixed.
<i>Resolved Problems and Enhancements Made, related to the Scheduler</i>		
1209081	Enhancement request for scheduler to provide more details on why a different scheduling effort is recommended.	CtoS has been enhanced to provide more details on why a different scheduling effort is recommended. For more details, see “Scheduler Output Includes QOR for each Level of Scheduling Effort” on page 1-4.
1297256	The create_resource command incorrectly fails when the user creates a resource for a behavior with scheduling effort medium but with default_scheduling_effort set to low . This is incorrect because the behavior scheduling effort should take precedence.	CtoS now correctly honors the precedence of the behavior attribute scheduling_effort .
1302535	The scheduler may finish with larger negative slack when scheduling_effort is medium as compared to when scheduling_effort is high .	The issue has been fixed.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the Scheduler (cont'd)</i>		
1305193	Sometimes the scheduler finishes with negative slack when the behavior contains a pipeline loop. This may happen when timing of a pipeline loop is defined by a strongly-connected component (SCC).	The scheduler was improved and the conditions to move SCC were enhanced, fixing negative slack for this design.
1347831	The schedule command may fail with the following error: <pre>ERROR CTOS-28000 In multi-stall mode the expansion point cannot be after the last stall loop of pipelined loop 'LOOP_TO_PIPELINE_while_begin'.</pre> This may happen if pipelining a loop with enable_multiple_stalls is set to true and the optimizer removes the second stall loop because it determined that the stall loop would never stall. This error is overly restrictive. It should be possible to have a pipelined loop with soft stall that has only one stall loop when no output stall is needed because the output protocol does not require stall, or results are written to memory.	Error message 28000 was converted to a warning message letting the scheduler continue to completion.
<i>Resolved Problems and Enhancements Made, related to Reports</i>		
1284296	In some cases, the report_sccs command shows different pipeline stage information than the pipeline viewer.	The differences have been resolved.
<i>Resolved Problems and Enhancements Made, related to Model Generation</i>		
1073840	CtoS may generate an incorrect model when input port name and array name are the same in SystemC. This may occur when the array name is declared in a function and the port is declared at the calling module. This is valid SystemC because the scopes are different. In Verilog, the arrays/memories are declared at the module scope and thus conflict with other names at the module scope.	The issue has been resolved.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Model Generation (cont'd)</i>		
1278938	This is an enhancement request to reference internal signals in Verilog verification wrapper instead of SystemC DUT ports. For example: <pre>SC_METHOD(ctos_convert_output_1); sensitive << dout_data_cpp;</pre> should be: <pre>SC_METHOD(ctos_convert_output_1); sensitive << m_dut_cpp.dout.data;t</pre>	The Verilog verification wrapper now references internal signals.
1278966	Bit-width mismatch was observed in the default case statement generated for replace module.	The issue has been resolved.
1305787	The RC script generated by the write_rc_script command does not correctly support vendor-RAM wrappers.	The issue has been resolved.
1335495	CtoS may generate redundant case items in designs with non-inlined sequential functions that contain write ops and the caller also contains write ops to the same sc_signals.	The redundant lines are no longer generated.
<i>Resolved Problems and Enhancements Made, related to Incremental Synthesis (ECO mode)</i>		
1314819	In some designs, the ECO difference is larger than expected. Specifically ECO version may show difference even in modules that have not changed.	The issue has been resolved. Modules that are not modified from the baseline will show no differences in the ECO version.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to GUI</i>		
925240	Enhancement request for GUI to explain why loop cannot be pipelined.	The Micro Arch Manager dialog has been enhanced with the Can Pipelined? column on the Loops tab. The reason a loop cannot be pipelined is specified in this column. For more information, see “Enhanced MicroArch Indicates if a Loop Can be Pipelined” on page 1-2.
1279161	When trying to display schematic the following internal error may be issued: <pre>ERROR (CTOS-13110): Internal error while executing gui_schematic: guiSchematicNlView.cpp:1228: ok</pre> This will occur when registers have multiple resets.	Multiple resets are now supported in the schematic viewer.
1322356	The CDFG viewer takes forever to load the data flow for an edge. This may occur when the data flow graph for an edge is very large. This is a request to tell the user that the graph is too large and not try to display this graph.	CDFG Viewer puts limit on size of dataflow graph associated with an edge. The data flow for an edge is considered too large if it has over 500 ops or 2000 arcs. The CDFG viewer will display an edge as too large with the following: • graphical notation displayed in red color, • expand/collapse menu items are not available • tool tip says the edge cannot be expanded with the op and arc count

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to FPGA</i>		
1342488	The use_dsp command does not generate module definitions for multiple instances of same function name.	The issue has been resolved.
<i>Resolved Problems and Enhancements Made, related to Library Models</i>		
1327963	Proper operation of <code>ctos_tlm::tlm_fifo</code> requires that reset_get() and reset_put() have been called before starting normal operation. The fifos of <code>ctos_tlm</code> have been enhanced with a simulation-time check for this requirement. The check is enabled by default, but it can be turned off by defining the macro <code>DISABLE_CTOS_TLM_FIFO_RESET_CHECK</code> before including ctos_tlm.h. For example: <pre>irun -DDISABLE_CTOS_TLM_FIFO_RESET_CHECK <rest of sim args></pre>	The enhancement has been implemented.
<i>Resolved Problems and Enhancements Made, related to User Documentation</i>		
1238873	Section 15.3.2.1 of the CtoS User Guide is not very clear, but the section "16.1 Understanding the Module Hierarchy" actually gives a very good description. Please clean up section 15.3.2.1.	Section 15.3.2.1 of the CtoS User Guide has been cleaned up and references section 16.1 for more details.

2 Why Use CtoS?

This chapter provides answers to the most frequently asked questions (FAQs) about **Cadence C-to-Silicon Compiler (CtoS)** to help you to understand the usage of CtoS.

The chapter covers the followings:

- [“What is High-Level Synthesis \(HLS\)?” on page 2-2](#)
- [“Can HLS Perform all the Tasks Performed by a Design Engineer?” on page 2-2](#)
- [“What Skills are Needed to be an HLS Expert?” on page 2-2](#)
- [“What are the Benefits of HLS over Hand-RTL Flow?” on page 2-2](#)
- [“Can I Use a Software-Style C Model for HLS?” on page 2-3](#)
- [“Can HLS Achieve the same QoR as Hand-Written RTL?” on page 2-3](#)
- [“What Are the Required Inputs to Use CtoS?” on page 2-3](#)
- [“Why not Use SystemVerilog as Input Language?” on page 2-3](#)
- [“Why Use SystemC instead of ANSI-C?” on page 2-3](#)
- [“Does CtoS require any non-standard code?” on page 2-4](#)
- [“What are the rules for synthesizable SystemC?” on page 2-4](#)
- [“Can I use the SystemC Standard Template Library \(STL\)?” on page 2-4](#)
- [“What testbench language should I use with CtoS?” on page 2-4](#)
- [“What verification methodology should I use with CtoS?” on page 2-4](#)
- [“Does simulating SystemC along with the generated-RTL increase my verification effort?” on page 2-4](#)
- [“Do I need to debug machine-generated RTL?” on page 2-5](#)

- “Can I use Synopsys Design Compiler (DC) for logic synthesis?” on page 2-5
- “What if I need a late Engineering Change Order (ECO)?” on page 2-5
- “Where do I get more help?” on page 2-6

What is High-Level Synthesis (HLS)?

HLS uses high-abstraction SystemC to describe design functionality without specifying micro-architecture or implementation details. This makes it easier and faster to write and functionally debug. Additionally, SystemC is verified before and during the synthesis process.

CtoS is used to explore various implementations to compare area, timing, power, pipelining, clocks, tech nodes, and so on. CtoS output is functionally equivalent RTL (or gates), which are pre-verified due to the flow. This RTL can be plugged into the original verification environment to run as regressions.

Can HLS Perform all the Tasks Performed by a Design Engineer?

No, CtoS enables designers to more efficiently design and explore trade-offs. CtoS automates much of the low-level coding needed in traditional RTL flows, but still requires an experienced Design Engineer to drive and perform design analysis of the results. This transition is similar to the transition when Schematic Capture Engineers learned to write RTL.

What Skills are Needed to be an HLS Expert?

Design experience, such as understanding hardware and design trade-offs, muxing, timing, and register cost, is most important. However, now the Design Engineers must also be comfortable with C++, including some object oriented techniques for the best reusable code. Learning SystemC is trivial for anyone comfortable with C++. In addition, Design Engineers must learn the new CtoS views and reports for analyzing results and comparing the micro-architecture implementations.

What are the Benefits of HLS over Hand-RTL Flow?

The following are the benefits of HLS:

- **Design and verification productivity.** Design and verification is done in parallel instead of delaying verification until enough RTL is written. Previously, customers have reported that this can give at least 50% improvement in time to fully verified RTL.
- **Flexibility and reuse.** The HLS flow ensures that the high-level model is always completely in sync with the implementation RTL. This gives flexibility to explore design trade-offs or reuse targets, for future projects, in ways not possible in RTL-flows.

Can I Use a Software-Style C Model for HLS?

No, software models often are written for ease and fast simulation, and do not account for the required hardware structures. HLS does not transform software-style code into efficient hardware. Instead the HLS model needs to be partitioned into large functional blocks showing the intended micro-architecture, which describes the flow and required storage of data. This HLS model is better than traditional RTL because it does not need to model any resource sharing, filling of clock cycles, or explicit FSMs as CtoS automates all this.

Can HLS Achieve the same QoR as Hand-Written RTL?

CtoS has already proven to meet or be better than the QoR (area, timing, power, and latency) of hand-RTL on many large and complex design types.

Also, since CtoS can see the entire state-space of the design, it can achieve much more aggressive resource or register sharing than a Design Engineer who must specifically code it in RTL; for example, an eight-stage pipeline. Since CtoS automates all the low-level RTL generation, the Design Engineer can focus on the larger impact items and perform lot more what-if analysis. The CtoS user interface (CtoS GUI) is specifically designed to give the Design Engineer feedback and control of the resulting implementations.

What Are the Required Inputs to Use CtoS?

The Design Engineer writes SystemC to describe the required design interface and functionality. And, the Design Engineer tells CtoS the design environment such as clock(s) speed and the target tech lib that must be used for logic synthesis. Then, the Design Engineer uses CtoS to explore design tradeoffs (timing, latency vs. area, power). CtoS is an interactive design environment and can also be run with tcl batch commands to automate exploring various implementations.

Why not Use SystemVerilog as Input Language?

SystemVerilog has lots of high-abstraction modeling capabilities; however, most customer feedback showed that the majority of their high-level models usually start in SystemC, C, or C++. Therefore, SystemC was the preferred input language for HLS.

Why Use SystemC instead of ANSI-C?

The previous generation of C-based HLS tools proved that C and C++ do not provide enough hardware organization to achieve competitive Quality-of-Results (QoR). However, SystemC is good to add hierarchy, interfaces, event ordering, and sized data types to wrap around C++ algorithms. Cadence recommends to use SystemC constructs where they add value, while keeping most of the design algorithm in untimed C++ to maximize exploration within CtoS.

Does CtoS require any non-standard code?

No, the input language is completely OSCI-compliant SystemC, and the output RTL is IEEE-compliant Verilog. So, you are free to use any simulator, logic synthesis tools, or other EDA tools in the flow.

What are the rules for synthesizable SystemC?

CtoS requires a top-level SC_Module that describes sized ports and a cycle-accurate description of how they are read or written. These will wrap around the intended design functionality. The biggest synthesizable rule is that all data structures must be statically determinable at compile-time as hardware is persistent. So, do not use *malloc* or *free*; instead of using *new* on an object, you should instantiate it. Pointers can be used as long as their dereferences are known at compile-time. Other rules are relevant for the intended hardware. All signal communications should go through well-defined ports or interfaces instead of global vars. Also, consider the `_required_` data storage, that is, how often does the algorithm need to read/write to an array, which may be mapped to a (single port) RAM. Also, check how much computation is required in loops.

Can I use the SystemC Standard Template Library (STL)?

CtoS supports the use of templated classes and functions, but the STL library is full of dynamic memory allocation, which are not synthesizable. So, the Design Engineer must determine the required storage, and then write their own container class. For more information, see [“Preparing Models with Large Container Classes”](#) on page 5-10.

What testbench language should I use with CtoS?

There are no requirements for the testbench language. CtoS customers have successfully taped out large designs using e, SV, SystemC, and even Verilog-2000 testbenches. The most important thing is to adequately verify that the SystemC is functionally correct.

What verification methodology should I use with CtoS?

Metric-Driven Verification (MDV) methodologies such as UVM are mostly preferred to ensure that you are covering all the required functionality on the input SystemC, and as well as with the CtoS-generated RTL plugged into the original environment. A good verification plan ensures all required functionality is tested and points to where it is most efficient at the higher abstraction level.

Does simulating SystemC along with the generated-RTL increase my verification effort?

No, in fact customer feedback shows that it is much more efficient finding functional bugs in the higher-abstraction SystemC model because there is less code, so it is easier to read and debug. It also simulates faster and hence requires less time to eradicate bugs.

Also, since the block size of HLS models is much larger than traditional hand-RTL flows, there would be lesser bugs from interfaces or handoffs between different designers. Once the RTL is generated, it is simply plugged into the original verification environment and run as regressions. Customer reports that the bugs are significantly fewer at this level since most issues are already been fixed on the higher models.

Do I need to debug machine-generated RTL?

No, feedback from customers shows that the CtoS-generated RTL is usually correct since most functional bugs were found in the faster-simulating SystemC. This generated RTL is simply plugged back into the original verification environment. If an RTL regression hits a mismatch, the Design Engineer can use a CtoS behavioral Verilog Simulation model from any step in the flow to more easily isolate the cause.

For example, immediately after CtoS-compilation, generate a (post-build) sim model to see if there is an initialization problem (SystemC initializes variable to 0, while Verilog inits to X). Note that the CtoS **check_design** utility will also flag these. You can also generate a sim model to see if you broke a protocol requirement, after pipelining a loop. It is always easier to debug these higher abstraction sim models. Additionally, the CtoS GUI has cross links to all models and views for easy correlation.

Can I use Synopsys Design Compiler (DC) for logic synthesis?

Yes, roughly half of the CtoS customers use DC on the CtoS-generated RTL. Hundreds of designs have been taped-out with this flow. Some customers have reported that Cadence RTL Compiler (RC) logic synthesis tool can sometimes further improve area and timing by 2% to 10%.

It is usually a good idea to first run RC on the generated RTL to get a baseline of the expected results. Then, in case DC get worse results, you can isolate the cause (often due to legacy DC scripts which were set up for hand-written RTL coding styles).

What if I need a late Engineering Change Order (ECO)?

CtoS was designed from the ground-up to handle ECOs because this was a major reason why all other HLS tools were not growing. Therefore, CtoS is unique that it is built on a database so you can control the similarity to the previous design more importantly than optimizations. Thus, when in incremental synthesis (ECO) mode, CtoS will output new RTL that is often identical to the previous taped-out design, with just localized changes reflecting the required ECO. So, it is an ideal input for the Cadence Conformal-ECO product to generate a pre or post-mask patched netlist.

Where do I get more help?

For more information, do one of the following:

- Print out the one-page *CtoS Quick Start Guide* that is available in the install doc directory or through the CtoS Guide Help menu.
Tip For a concise overview of the recommended steps, select print options as color, double-sided, and flip on short edge.
- Refer to the *CtoS Users Guide*. To find relevant information, you can perform keyword searches.
- Explore the examples present in the CtoS install. You can copy them to a writeable directory, then run, edit, or rerun to explore.
- Contact your local Cadence AE or Support Engineer. You can obtain the latest CtoS User training material.

3 Prerequisites for CtoS and for this User Guide

The *Cadence C-to-Silicon Compiler User Guide* is a comprehensive, task-oriented user guide that has been designed to help you learn about, and use all of the features of Cadence’s synthesis product, *CtoS*.

This chapter contains *prerequisite-type* information to help you navigate both the CtoS environment and this user guide.

- “CtoS-Supported OS Platforms/Software Versions” on page 3-2
- “Audience/Prerequisites for this User Guide” on page 3-3
- “How to Find Things in this User Guide” on page 3-3
- “Conventions and Syntax in this User Guide” on page 3-3
- “CtoS Definition of “Preliminary Feature”” on page 3-4
- “Filing a Service Request” on page 3-4

3.1 CtoS-Supported OS Platforms/Software Versions

CtoS complies, or is compatible, with the following operating system platforms and software versions:

- Red Hat Enterprise Linux (RHEL) 5 and 6

Notes

- If you are running on a 64-bit RHEL system, be sure you have installed both the **ksh** and 32-bit **elfutils** packages. These are needed to run CtoS in 32-bit mode (the current CtoS default).
- A performance problem with RHEL 5.1 has been fixed in RHEL 5.2. If you are using RHEL5, it is recommended that you *not* use RHEL 5.1 or earlier versions.
- SUSE Linux Enterprise Server 11
- Cadence's Encounter RTL Compiler (RC), version 14.10-s022.

Note RC no longer supports the 32-bit version.

- Cadence's Incisive Enterprise Simulator (INCISIV), version 13.20.

Note INCISIV 11.2 and greater compiles your design with GNU Compiler Collection (GCC), version 4.4.5 (Red Hat 5.5-6.314103).

- Altera Quartus II, version 12.1

Note CtoS does *not* support Quartus II Web Edition Software.

- Xilinx ISE Design Suite, version 14.2

Note *Support for FPGA designs is a preliminary feature.*

- SystemC 2.2

Notes

- See "[SystemC Standard Language Restrictions](#)" on page 14-106.
- The *IEEE Standard 1666-2005 for SystemC Language Reference Manual*, which describes the SystemC language, is referred to as the *LRM*, in this user guide.
- Standard C/C++
Note See <http://www.open-std.org/jtc1/sc22/wg21/> for the official documentation of the C++ Standards Committee)
- *IEEE Standard 1364-1995/2001 for Verilog HDL*
Note See "[Controlling CtoS-Generated Verilog](#)" on page J-1.
- Tcl (Tool Control Language), version 8.4.9
- Adobe Reader (**acroread**) 7 or later

Note This is in order to use the **-doc** option with the **help** command ("[help](#)" on page E-77).

3.2 Audience/Prerequisites for this User Guide

The audience for this user guide is *any level user of CtoS*, from beginner to advanced – all aspects of CtoS are covered in this guide. However, it is assumed that you have prior knowledge of the following topics, and details of these topics will not be covered in this guide:

- using an IEEE-compliant synthesis product
- writing Tcl (Tool Control Language) scripts
- *IEEE Standard 1850 for Property Specification Language (PSL)*

Important It is strongly recommended that you review the “[Glossary](#)” on page N-1 to become familiar with CtoS terminology because SystemC, Verilog, and RTL semantics are merged in the CtoS context.

3.3 How to Find Things in this User Guide

The *CtoS User Guide* is contained in a *single* Adobe PDF (portable document format) file, so you will have *only one place* to look for information. Within this file, there are three different ways to find things:

- Search the PDF file, using the Acrobat search feature.

Note For easy “back and forth” navigation in a PDF file, make sure you have enabled *Previous and Next Views*, as follows: Select **View** -> **Toolbars** -> **More Tools**. Scroll to **Page Navigation Toolbar**, and check **Previous View** and **Next View**. You will now see two small arrows on your toolbar, which essentially work the same as the back and forward arrows in a browser window.

- Check the Table of Contents (“[Contents](#)”). Each page number is hyperlinked, so all you need to do is click on it.
- Check the Index. Again, each page number is hyperlinked, so all you need to do is click on it.

3.4 Conventions and Syntax in this User Guide

This user guide employs the following conventions and syntax, most of which are based on standard Backus-Naur Form (BNF).

- **Green** square brackets (**[]**) within syntax

Action—Items within **green** square brackets (**[]**) are optional. If you *do* enter these items, do *not* type the brackets themselves.

- **Bold** square brackets (**[]**) within syntax

Action—Items within **bold** square brackets (**[]**) are *not* optional. In Tcl, you obtain the results of a command by placing the command in square brackets (**[]**), and the brackets, themselves, *must be typed in* (see the Tcl tutorial at www.tcl.tk/man/tcl8.5/tutorial/Tcl5.html).

- **Bold** curly braces (**{ }**) within syntax

Action—Items within **bold** curly braces (**{ }**) are not optional. In Tcl, you must include a list in curly braces (**{ }**), and the curly braces, themselves, *must be typed in* (see the Tcl tutorial at www.tcl.tk/man/tcl8.5/tutorial/Tcl5.html).

- Separate single line of Courier text

Action—Enter the text at the Linux prompt. You can also add this line to a script, in most cases.

- *Italics* within syntax

Action—This indicates a variable, so substitute your specific name or term for the italicized item.

- Ellipsis (...) within syntax

Action—An ellipsis (three periods in a row, that is, ...) following an item means you can enter as many of this item as you want. Again, do *not* type in the ellipsis itself.

- Pipe (|) within syntax

Action—Choose one of the items separated by the pipe (|). Again, do *not* type in the pipe itself.

- Parentheses () within syntax

Action—Items within parentheses must always be included together.

- Text in **boldface** or *italicized boldface* within a paragraph

Action—This is just used so that commands and arguments will stand out within paragraphs.

3.5 CtoS Definition of “Preliminary Feature”

In CtoS, the term *preliminary* identifies features with the following characteristics and caveats:

- These features have passed all of the CtoS regression tests.
- These features could change in their appearance and in their functionality in later releases.
- These features may be incomplete, but they have enough functionality at this point to warrant their exposure to customers for an early look and possible evaluation.

CtoS welcomes and encourages your feedback on these features, which is the primary reason they have been included here and marked *preliminary*.

3.6 Filing a Service Request

If you wish to request an enhancement, or have a problem you cannot resolve, go to <http://sourcelink.cadence.com>, and file a Service Request for the *Cadence C-to-Silicon Compiler*.

4 CtoS Design Flow

To make your synthesis process as easy and successful as possible, the *CtoS Design Flow* describes the order in which you would usually work through the CtoS process, providing the appropriate commands, reports, and models to use or produce at each step along the way.

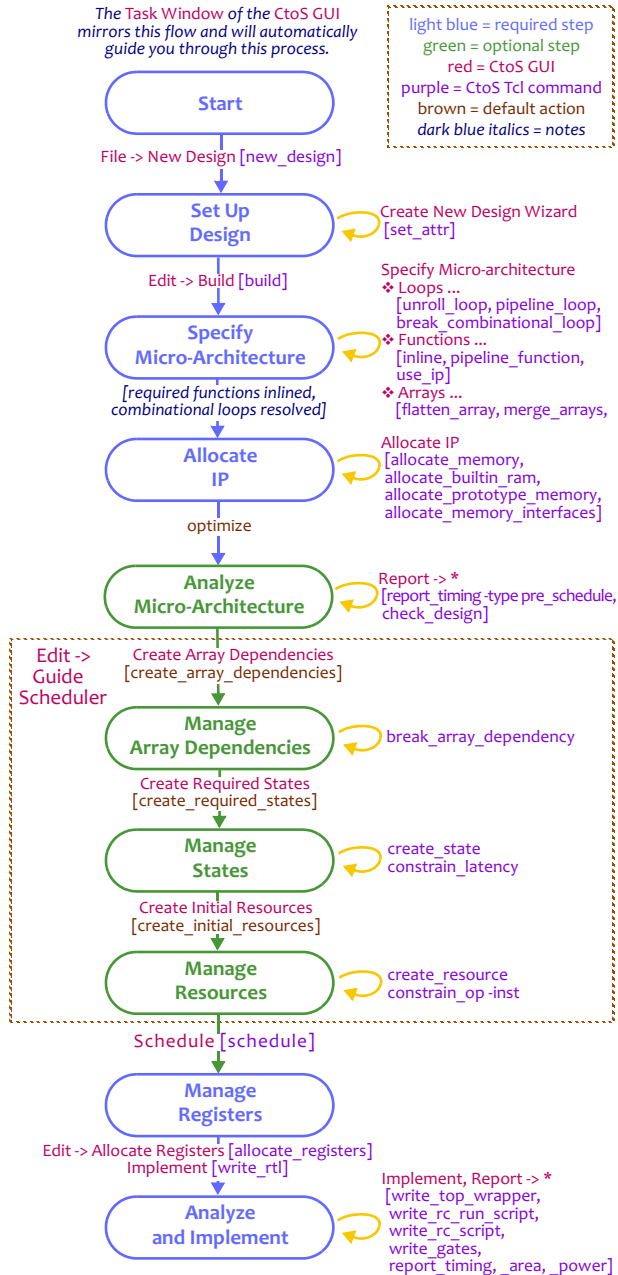
This chapter is an introduction to the CtoS Design Flow.

First, a helpful one-page flowchart is provided, with a very high-level view of the design flow.

Then, a more expanded text version of the design flow is provided, with links to the sections in this user guide that provide more detail.

- [“CtoS Design Flow \(Chart Format\)” on page 4-2](#)
- [“CtoS Design Flow \(Text Format with References\)” on page 4-3](#)

4.1 CtoS Design Flow (Chart Format)



4.2 CtoS Design Flow (Text Format with References)

Here is a brief overview of the recommended CtoS Design Flow. At the end of each topic, there is a link to the section in this user guide that provides more detail.

Start

CtoS uses the concept of *designs* to manage a synthesis environment setup, so your first step will always be to create a new design or to open an existing design.

You can start CtoS in either a graphical user interface (GUI) or a Tcl command-line environment.

The CtoS GUI is highly recommended, as you are provided very helpful menus, dialogs, viewers, and other navigational aids.

The CtoS GUI also lets you type any CtoS or native Tcl command at its command line.

For more detail, see [“Starting CtoS” on page 6-2](#).

Set Up Design

In the Set up Design step, you specify the input source and build the CtoS design.

In the CtoS GUI, the *Create New Design Wizard* takes you through creating designs from SystemC input. By default, the wizard will also **build** the design.

For more detail, see [“Creating a New Design” on page 6-9](#) and [“Building a Design” on page 6-28](#).

In command-line CtoS, you use the **set_attr** command to specify design attributes and the **define_clock** command to add a clock definition.

For example, the **source_files**, **compile_flags**, and **top_module_path** design attributes should be set, followed by a call to **build** the design.

For more detail, see [“CtoS at the Command Line” on page 6-3](#) and [“set_attr” on page E-140](#).

Specify Micro-Architecture

In the Specify Micro-architecture step, you apply micro-architectural transforms to make the design synthesizable or to meet area/timing requirements.

For more detail, see [“Specifying Micro-Architecture” on page 8-1](#).

This includes resolving functions, loops, and arrays.

For more detail, see [“Resolving Functions” on page 8-3](#), [“Resolving Loops” on page 8-16](#), and [“Resolving Arrays \(Memories\)” on page 8-61](#).

Some functions are not synthesizable because different paths through the function require a different number of clock cycles to execute. This can be resolved by inlining the function or by balancing the paths through the function to have the same number of states. Additionally, you can pipeline a combinational function.

For more detail, see [“Inlining Functions” on page 8-3](#), [“Pipelining Functions” on page 8-12](#).

Other functions that are not synthesizable include:

- functions that access a writable array
- combinational functions that contain loops

Considering area/timing requirements, you may optionally inline such functions or convert them to a lookup table. If you already have an RTL implementation of your function, you can link the behavior to the RTL IP instead of letting CtoS synthesize it.

For more detail, see [“Converting a Function to a Table Lookup” on page 8-15](#) and [“Importing RTL IP into SystemC Designs” on page 9-41](#).

Combinational loops must be eliminated because they cannot be implemented in hardware. To eliminate a combinational loop, you can either unroll the loop or insert states to break the loop. Considering area/timing requirements, you may also pipeline the loop.

For more detail, see [“Breaking Combinational Loops” on page 8-22](#) and [“Pipelining Loops” on page 8-26](#).

Also considering area/timing requirements, you may choose to transform arrays by flattening them or merging them. To give the scheduler more flexibility, you can also specify that array accesses can float.

For more detail, see [“Flattening Arrays” on page 8-61](#), [“Merging Arrays” on page 8-63](#), and [“Floating Array Accesses” on page 8-94](#).

Allocate IP

In the Allocate IP step, you allocate arrays to memories (vendor-supplied RAMs or ROMs), or CtoS can synthesize a built-in RAM or create a prototype memory. Vendor RAMs and ROMs are specified as IP definitions, which must be read in.

For more detail, see [“Allocating Memory” on page 9-2](#).

Analyze Micro-Architecture (Optional)

At this point, the design is completely transformed, and CtoS can run final optimizations. You can now analyze the area, timing, and power of your design, and although this is less accurate than analysis *after* scheduling, the results provide comparisons between various micro-architectures.

This is an optional step, and final optimizations are automatically applied if you move directly to the Scheduling step.

For more detail, see [“Analyzing Micro-Architecture” on page 10-1](#).

Scheduling

In the Scheduling step, you apply scheduling operations that bind ops to resources based on constraints. The basic action is to run the CtoS scheduler.

For more detail, see [“Scheduling” on page -2](#).

If this process fails, you may first need to guide the scheduler with the following optional substeps and then try scheduling again:

- *Manage Array Dependencies*: In this substep, you can create array dependencies. The advantage of creating dependencies prior to scheduling is that you can analyze them to determine if you want to break some of them before continuing.

For more detail, see [“Creating and Managing Array Dependencies” on page 11-2](#).

- *Manage States*: In this substep, you can resolve sequential conflicts and resource contention by adding states. You can create the minimal number of states required by the design. If this fails, you may need to create states individually or constrain latency to provide guidance when creating required states.

For more detail, see [“Creating and Managing States” on page 11-6](#).

- *Manage Resources*: In this substep, you can create initial resources to map the ops. To optimize the resources used in the synthesized design, you can create individual resources and manually constrain the ops to these resources. You can then create initial resources again to ensure that all ops have a compatible resource.

For more detail, see [“Creating and Managing Resources” on page 11-20](#).

Manage Registers

In the Manage Registers step, you can allocate registers and bind values to registers.

You can create specific registers, bind a value to create an appropriate value-to-register binding, reset registers, automatically or manually allocate registers, and connect terminals to add power or test.

For more detail, see [“Managing Registers” on page -18](#).

Analyze and Implement

In the Analyzing and Implementing step of the CtoS flow, you can use the many helpful CtoS GUI reports and viewers to analyze your design.

For more detail, see [“Using Reports to Analyze Designs” on page 13-2](#).

You can also generate models and scripts, including RTL descriptions.

For more detail, see [“Generating Models, Wrappers, RTL, SLEC after Scheduling” on page 13-36](#).

5 Preparing C Models for High-Level Synthesis

The first time you prepare a model for CtoS, you will need to make a few adjustments to meet coding constraints; however, the majority of your original timed or untimed code will remain untouched.

After this initial experience, your models will most likely be written in a more synthesis-friendly manner, which will further accelerate your design flow.

Following these simple steps will lead to increased productivity and greater architectural trade-off opportunities. Future high-level models written using these rules will provide an even shorter path to implementation.

This chapter is intended to help you with specific design considerations and is organized as follows:

- [“Preparing High-Level Synthesis Models” on page 5-2](#)
- [“Preparing Models with Large Container Classes” on page 5-10](#)

Note The chapter, [“Authoring SystemC for CtoS” on page 14-1](#), provides extensive information about SystemC, including the synthesizable subset, recommended coding style for CtoS, and more.

5.1 Preparing High-Level Synthesis Models

This section is intended to help you, in general, prepare better models for high-level synthesis:

- “Assuring Statically Determinable Behavior” on page 5-2
- “C or C++?” on page 5-2
- “Partitioning Code into Functional Blocks” on page 5-3
- “Interprocess Communication” on page 5-4
- “Creating a Top-Level Module Interface” on page 5-4
- “Handling Non-Synthesizable Constructs” on page 5-5
- “Example” on page 5-6

5.1.1 Assuring Statically Determinable Behavior

Most of the required changes for a design fall into two areas: creating a well-defined top-level interface and removing non-synthesizable constructs.

In general, the coding style used for CtoS can be thought of as ensuring that all behavior is *statically determinable* at compile-time. Thus, you should avoid dynamically created objects, variable-sized arrays, or pointer aliasing that requires simulation for resolution. Feedback has indicated that system architects and model creators find these constraints reasonable.

5.1.2 C or C++?

Should you use C or C++? C++ is usually more structured than C due to its object-oriented programming approach. For example, in C++, all private variables are known to be local to that class, whereas in C, global variables are often used for the same purpose, which could be accessed potentially by any function in the entire program. Therefore, although not required, starting with C++ offers an easier path to CtoS.

CtoS provides a way to use pure C/C++ as input. It is not recommended for complex designs with crucial interface requirements, but it is intended to ease the introduction to SystemC.

5.1.3 Partitioning Code into Functional Blocks

C and C++ models are usually written with the primary concern of performance; thus, they are often written as one long sequential description without any intent to map to actual hardware. Consequently, they do not consider resource limitations. However, a well-structured and concurrent model with its behaviors partitioned into functional blocks will more easily exploit parallelism.

When CtoS compiles a design, it determines the flow of data dependencies and tries to create parallel logic wherever possible. Improved *quality of results* (QoR) – specifically, smaller area – can occur when CtoS aggressively shares most of the design components, such as adders, multipliers, registers, etc.

Most of the behavior of a design is contained within function calls or classes. CtoS lets you pick and choose the functions to be inlined. This offers an architectural trade-off of flattening some function calls for more aggressive optimizations, or possibly retaining structure (which may be shareable in different parts of the design).

A good QoR advantage may be achieved by using functions for common code patterns, as seen in the following example.

Instead of repeating a common section of code, consider putting it inside a simple function:

```
x1 = (y1 + z1 + 1) / 2;
x2 = (y2 + z2 + 1) / 2;

int average(int y, int z) {
    return (y + z + 1) / 2;
}
x1 = average(y1, z1);
x2 = average(y2, z2);
```

Then, within CtoS, you can decide whether or not to inline this function, depending on implementation goals (more parallelism or less area).

SystemC also provides processes, which are an easy way to specify concurrency by letting you define a particular function as a clock-sensitive thread, called **SC_CTHREAD**, or as a signal-sensitive, untimed method, called **SC_METHOD**. If your design contains one main top-level function, it will typically map into one top-level **SC_CTHREAD**, as you will see in [“Example” on page 5-6](#).

Designs with multiple top-level functions that feed each other are candidates for structural pipelining. This involves declaring each function as an **SC_CTHREAD** and connecting them successively. Thus, they will each be executed in parallel.

5.1.4 Interprocess Communication

Communication between processes (**SC_CTHREADS** or **SC_METHODS**) within the same **sc_module** is supported using shared variables, as follows:

- *Example 1:* Most signals between processes will be of SystemC type, **sc_signal**, and act like Verilog non-blocking assignments, which avoids read/write races. Designs should try not to write a shared variable from two processes during the same clock cycle, and CtoS will generate checkers in the RTL to check for this condition. Module members that are not arrays and not **sc_signals** are always assumed to be unshared, which is probably not what was intended for a design, so CtoS issues a warning.
- *Example 2:* An array accessed by only one process during any single clock cycle can be modeled as an ordinary array (**int my_shared_array[256]**). CtoS implements a shared array as a memory (which can then be mapped to a built-in or user-specified RAM, or to a vector of registers). Although the array can be of **sc_signals**, as in *Example 3*, it will result in a different implementation (as discussed below) as a vector of registers, instead of a memory (and its simulation performance will be slower).
- *Example 3:* If the shared data is an array in which, in a given clock cycle, an element of the array may be written by one process and read from another process, you can model the shared variable using an array of **sc_signal<T>**, that is, **sc_signal<int> my_shared_array[16]** (similar to *Example 1*). Again, each element of the shared array should be written to by only one process during any single clock cycle. CtoS will implement the array as a vector of multi-bit registers (discrete flip-flops).

Communication between different **sc_modules**, that is, across a potential design hierarchy, must use **sc_ports** and **sc_signals**, as is the case for the top-level interface described in the next section.

Note For more detail, see [“Interprocess Communication through Shared Variables” on page 14-38](#).

5.1.5 Creating a Top-Level Module Interface

A top-level module interface consists of two parts: a module port definition, plus a function that defines how the communication protocol supplies data to and from the top-level algorithm function. The module port definition defines the actual signals the module uses to talk to the outside world, which includes all required ports, plus any clocks and resets.

For this purpose, the SystemC **SC_MODULE** provides an easy way to define design hierarchy:

- When starting from a C++ model, the process of creating an **SC_MODULE** involves deriving your top class from the special SystemC provided class, **sc_module**, adding the I/O member variables, and defining the constructor.
- When starting from a C model, it is best to start with the original top-level function call for which the operands can be mapped into input and output signals.

For the communication protocol, a separate function will be created to handle the reading of input signals, driving the top-level function(s) of the design, and writing outputs. This achieves a powerful strategy of separating the interface from the computation, which keeps future changes localized, while allowing concepts such as Transaction-level Modeling (TLM).

5.1.6 Handling Non-Synthesizable Constructs

CtoS can handle most of the expressiveness of C/C++, but to predictably synthesize actual hardware, there are some restrictions.

As previously mentioned, this ensures that all behavior is *statically determinable* at compile-time, so there are no dynamically created objects or variable-sized arrays.

Here is a list of unsupported constructs and, when possible, suggestions for how to replace them:

- *Global variables* must be replaced by module member variables.
Alternatively, they can be replaced by local variables passed as arguments to each accessing function.
- *Recursive calls* must be transformed into loops.
- *Functions in math.h* (sin, cos, power, log, etc.) must be implemented in C (with care for synthesis efficiency).
- *New, delete, malloc* and *free* must be eliminated by making the corresponding objects local variables or class member variables.
- *Pointers and references to sc_signals, sc_modules and sc_ports* are not supported.
Instead, the code should refer to these objects directly using the (variable) name of the object.
- *Multiple inheritance* and *virtual inheritance* are not supported in general.
They are supported only for specifying TLM-style interfaces and channels.

Here is a list of constructs that are supported, but with some caveats:

- *Bounded loops* are preferred over unbounded, non-infinite loops.
CtoS has many micro-architectural options for breaking, constraining the latency of, unrolling, and pipelining loops (see [“Resolving Loops” on page 8-16](#)).
- *Double data types* are supported, but only if CtoS can resolve them to constant values by means of static analysis (see [“CtoS Libraries” on page 15-1](#)).
- For *pointers that are dereferenced*, CtoS must be able to determine the variables whose addresses are assigned to the pointer by static analysis (see [“Pointers” on page 14-66](#)).

Note For complete listing and description of all restrictions and limitations, see [“SystemC Standard Language Restrictions” on page 14-106](#).

5.1.7 Example

In the following example, an implementation of a JPEG IDCT decoder, originally written in C, is transformed into a synthesizable SystemC model. This section contains the following subsections:

- “Interface Protocol” on page 5-6
- “The xbus_hw_idct.h Header File” on page 5-7
- “The xbus_hw_idct.cpp File” on page 5-8
- “Resetting Signals” on page 5-9
- “Large Data Objects” on page 5-9
- “Communication between SC_MODULES” on page 5-10

Note The complete example can be found in your CtoS installation area, in:

`install_directory/share/ctos/examples/flows`

It is also fully explained in “CtoS Tutorial for ASIC designs” on page A-1.

5.1.7.1 Interface Protocol

To create the **SC_MODULE** to define the required design interface, you must first determine the required communication ports and protocol. The original function call includes the data involved:

```
void jpeg_idct_islow_orig (j_decompress_ptr cinfo,
    jpeg_component_info * compptr, JCOEFPTR coef_block,
    JSAMPARRAY output_buf, JDIMENSION output_col )
```

This function call uses **coef_block** and **output_buf** as pointers to 64-entry arrays of 16 bits and 8 bits, respectively, for passing data in and out. Thus, for this example’s hardware, input data is passed at 16 bits at a time into the JPEG block, and output data comes out 8 bits at a time. A simple protocol must be devised, using the signals **read** and **ms** for handshaking and class member variables to hold the data:

```
void xbus_hw_idct::bus_if()
{
    sc_uint<7> out_ind, in_ind;

    wait();
    while(in_ind < DCTSIZE2) {
        while ( ms.read() == true) wait();          // wait until signal ms is false

        if(read.read() == false) {                  // are we in read or write mode?
            data_out.write(output_buf[out_ind]);    // write output_buf to data_out port
            out_ind++;
            wait();
        } else {
            wait();
            coef_block[in_ind] = data_in.read();    // read from data_in port to coef_block
            in_ind++;
        }
        wait();
    }
}
```

The interface protocol waits until the **ms** signal is ready and will then either read the inputs to drive the main function or write to the outputs. This makes use of the SystemC **wait** construct to dictate ordering for the interface by forcing a **wait** until the next clock cycle.

When the communication protocol has been determined, it is easier to see the necessary ports. SystemC provides a macro, **SC_MODULE**, to define ports and hierarchy, which can be used if you prefer a style that looks more like HDLs. However, this example will use the standard C++ style of defining a class that inherits from the **sc_module** class.

5.1.7.2 The xbus_hw_idct.h Header File

In the **xbus_hw_idct.h** header file, the SystemC signal ports **sc_in** and **sc_out** are template classes and are parameterized with the data type being transferred through the port.

```
class xbus_hw_idct : public sc_module
{
public:
    xbus_hw_idct(sc_module_name name); // constructor to initialize module

    SC_HAS_PROCESS( xbus_hw_idct );    // tells SystemC kernel this module
                                      // contains a process

    sc_in_clk clk;                     // ports of this module
    sc_in< bool > reset;
    sc_in< bool > ms;
    sc_in< bool > read;
    sc_in< bool > size;
    sc_in< sc_uint<32> > data_in;
    sc_out< sc_uint<32> > data_out;
};
```

The data type can be almost any primitive C++ data type, such as **bool**, or any SystemC data type, such as **sc_uint<8>** (see “[SystemC Data Types](#)” on page 14-54 for a complete list of supported SystemC data types).

Structures and user-defined data types can also be used if you implement the assignment, equality, and non-equality operators.

At this point, you should consider the sizes of the variables at the interface. Most C programs use generic integer data types with no regard for whether they have significant over-capacity, but in real hardware, overly large data types can lead to a needlessly large area.

By minimizing sizes at the interface, CtoS will optimize most of the data flow. Simulation is often useful to determine the minimum data widths needed to retain the required computation precision.

5.1.7.3 The `xbus_hw_idct.cpp` File

The preceding `xbus_hw_idct.h` header file makes declarations, while the `xbus_hw_idct.cpp` file completes definitions, first of which is the constructor (as you will see in the following description).

The role of the constructor for the `SC_MODULE` is twofold:

- to connect the ports of any other `SC_MODULE`s instantiated within this `SC_MODULE`
- to identify the processes (`SC_CTHREAD` or `SC_METHOD`) of this `SC_MODULE`

In this case, no port connections are required as the example `SC_MODULE` does not instantiate any other hierarchy below. The example `SC_MODULE` constructor specifies one `SC_CTHREAD` process, whose behavior is specified by the function `run()`, which is sensitive to the positive edge of the clock, called `clk`. It also specifies that this process be synchronously reset when the input port `reset` is set low.

```
{ SC_CTHREAD(run, clk.pos());  
    reset_signal_is(reset, false);  
}
```

The behavior of an `SC_CTHREAD` consists of two parts:

- The reset behavior is responsible for resetting any state (variables) owned by the process. It is specified by code in the function associated with the process [`run()`], up until the first call to `wait`.
- Following the first call to `wait` is the behavior describing normal operation of the process.

For this example, it consists of the communication protocol and the top function call of the design.

```
void xbus_hw_idct::run()  
{  
    // optional area for reset logic  
    wait();  
    while(1) {  
        // endless loop since real hardware does not stop  
        bus_if();  
        // function to provide communication protocol  
        jpeg_idct_islow();  
        // top-level function of the design  
    }  
}
```

The first `wait` is special because it separates the reset section from the design functionality, which is inside the infinite loop. All `SC_CTHREAD`s must have an infinite loop because actual hardware does not have a notion of exit. There must be at least one `wait` in the infinite loop (if there is no `wait`, CtoS will insert one automatically).

For this example, the two callable functions also contain `wait`'s. The code written between `wait`'s will be implemented in a single clock cycle (unless the `wait` is specified as *expandable* by a latency constraint).

5.1.7.4 Resetting Signals

SystemC data types, such as `sc_int<8>` and `sc_uint<8>`, provide a convenient way to more accurately size variables. Module member variables of these types are initialized to `0` by the default constructor. However, module member variable initialization using the module constructor is not synthesizable; instead, you should use the previously described reset section to force these variables to a known state.

If you do not specify reset, care should be taken during verification because module member variables will be initialized to `x` (unknown) in the synthesized model, while SystemC will initialize them to `0` at the start of simulation. You can use the CtoS **check_design** command ([“check_design” on page E-32](#)), a linting utility, to report signals that do not get reset.

After defining the top-level interface and the function to dictate communication, you need only to insert the original top-level function definition. You must declare the two arrays, and the other arguments to the top-level call, in the **SC_MODULE**. They can then be passed as arguments to the original top-level function, whose code does not need to be changed. Since these variables are class members, they can also be used directly inside the top-level function, instead of being passed explicitly as pointers.

```
void
xbus_hw_idct::jpeg_idct_islow ()
{
    INT32 z1, z2, z3, z4, z5;
    JCOEFPTR inptr;
    ...
    inptr = coef_block;
    ...
}
```

5.1.7.5 Large Data Objects

Large data objects should be mapped onto RAMs during synthesis. For each data member of an **SC_MODULE** and for each local variable declared in a function, CtoS decides whether the object is to be elaborated as a memory, data flow, or register, based on the type of the object:

- **sc_signal** variables are elaborated as (multi-bit) registers
- arrays of **sc_signal** variables are elaborated as vectors of registers
- arrays of non-**sc_signal** variables are elaborated as memories
- other variables are elaborated as data flow

Note that for sub-objects of a local variable or a member variable of an **SC_MODULE**, the choice is dictated by the choice made for that parent local variable or data member variable. Only variables elaborated as memories can be implemented as RAMs. Therefore, it is important that any large data object of your design be modeled as an array (even of size `1`, if a single large class or struct instance must be implemented as a RAM) that is either a local variable or a module member variable. If a local variable or a module member variable is a struct with a large array as one of its fields, that variable will be elaborated as pure data flow, which cannot be implemented by a RAM.

5.1.7.6 Communication between SC_MODULES

Standard **sc_signals** can be used to connect multiple **SC_MODULES**. CtoS also supports communication by interface method calls via **sc_port<IF>** and **sc_export<IF>** for modules instantiated beneath the top-level module.

CtoS supports user-defined channels, which implement user-defined interfaces.

Other modules can communicate with the user-defined channel by invoking interface methods through an **sc_port<IF>** connected to the user-defined channel. When modeling with user-defined channels, the channel typically does not have any processes; instead, it implements interface methods that can be called from processes outside the channel.

5.2 Preparing Models with Large Container Classes

This section is intended to help you prepare models with *STL* (*standard template library*)-like container classes for CtoS synthesis:

- [“Introduction” on page 5-10](#)
- [“Synthesis Requirements” on page 5-11](#)
- [“Modeling Container Classes for Synthesis” on page 5-12](#)
- [“Choosing the Right Modeling Approach” on page 5-16](#)

5.2.1 Introduction

C++ provides a natural way of abstraction through class definition. As the existence of the STL proves, a set of common principles can be abstracted and implemented efficiently and rigorously tested. A subset of such classes is commonly referred to as *container classes*. In general, container classes provide well-defined interfaces for common tasks, such as content lookup, content access, and iteration methods, and hide implementation details, such as stack behavior, vector implementation, and hash functions.

Naturally, there is a need to utilize container classes during system-level design. Since SystemC is an extension of C++, this can be done through the same means of class abstraction. However, if the SystemC model is intended for high-level synthesis, the goal is to implement the behavior in hardware. This leads to an extension of the problem definition since the most common means of storage for large amounts of data is realized as memory (RAM).

5.2.2 Synthesis Requirements

Since the high-level synthesis process maps behavioral descriptions (SystemC input) onto hardware, it is necessary to have the actual description of each operation present during the synthesis process. This eliminates the use of precompiled libraries. Even if all the source code for a library were available (as in the case of some STL implementations), the use of these libraries is, in general, not advisable as a general purpose implementation. They usually contain too much overhead for an efficient hardware implementation, as well as use dynamic memory allocation schemas.

It is therefore common to have system specifications that utilize container class specifications. These classes are often sized according to data widths and memory requirements and provide only problem-specific functionality.

However, there are additional issues to overcome when you consider synthesis. These container classes are actually more like service-oriented classes, which are used to wrap common functionality around the memory representation, as compared to actual data classes.

This differentiation is important when it comes to synthesis, since a container class is intended to describe memory access, whereas a data class is actually describing data to be stored or manipulated. A data class should result in bus-width implementation or memory layout decisions, while a container class should provide only access protocols of the memory-mapped content.

Note The identification of large container classes and the modeling of them in any of the associated modeling approaches described herein opens up a large selection of synthesis decisions. In contrast to the software memory representation as a single contiguous heap, hardware realizations can be vastly different. It is possible to map the arrays into individual memories and consider memory configurations (data width vs. address space). These are extremely important architectural decisions and have a large impact on performance and QoR of the resulting RTL.

There are several ways to inform CtoS about the intention with respect to a container class definition. The most common ones are explained in the following sections. These modeling techniques are all a result of the modeling guidelines for variables described in [“Variables” on page 14-37](#).

5.2.3 Modeling Container Classes for Synthesis

As aforementioned, container classes are all concerned with the utilization of memory. As a result, each contains an array defining the width and address space of the memory. To better illustrate all of the different modeling approaches, the following simplified SystemC stack class specification will be used as an example:

```
template <class DataType, int Size>
class Stack {
    DataType array[Size];
    int      nextLocation;
public:
    Stack() : nextLocation(0) {};
    void reset(void) {
        nextLocation = 0;
    }
    void push(DataType val) {
        array[nextLocation++] = val;
    }
    DataType pop() {
        return array[--nextLocation];
    }
};
```

This stack class could be introduced to CtoS as a container class in various ways; for example, you could simply rewrite the specification to eliminate instances of this class and make the memory associations explicit. Although in some cases this might provide additional opportunities for optimization, this approach is usually not acceptable because it widely eliminates the intended abstraction of a high-level system description. Thus, only those solutions that continue to provide abstraction, and reduce the amount of code modification necessary based on the utilization of a container class, will be presented.

Before looking at the individual modeling approaches, it is important to clearly state some of the boundary conditions for the given solutions. The following modeling approaches work under the assumption that only a single process is accessing the container class. If the intent is to actually use a container class as part of a communication protocol (for example, a FIFO), the model must be rewritten to a much larger extent to adequately model concurrency, arbitration, and the signal-based synchronization protocol between the different modules. Additionally, the following models assume the container class object is either instantiated as part of the **sc_module** of the *model under test* or as a local variable contained in the actual process definition.

Note If the instance is created as a local variable in the process, the initialization given in the constructor is automatically performed. If the instance is a module member variable, the reset function must be called as part of the process reset definition to ensure adequate initialization.

The following sections describe the various modeling approaches:

- “Modeling Container Classes by Separating Storage/Control” on page 5-13
- “Modeling Container Classes as Local SystemC Modules” on page 5-14
- “Modeling Container Classes as Interface and SystemC Channels” on page 5-15

5.2.3.1 Modeling Container Classes by Separating Storage/Control

The most straightforward approach to modeling container classes is to actually remove the memory representation. This leaves a pure container wrapper that provides access functions to an external reference.

The stack class example could thus be specified as follows:

```
template <class DataType, int Size>
class Stack {
    int      nextLocation;
public:
    DataType *array;

    Stack() : nextLocation(0) {};
    void reset(void) {
        nextLocation = 0;
    }
    void push(DataType val) {
        array[nextLocation++] = val;
    }
    DataType pop() {
        return array[--nextLocation];
    }
};
```

Since only a pointer to the memory is present in the container class, the memory will have to be defined and assigned externally. This could be performed in the module containing the stack, as follows:

```
Stack<int, 20> m_stackWrapper; // wrapper definition
int stackWrapperArray[20];    // actual memory definition
```

During the reset specification of the process accessing the stack, the wrapper array pointer is then assigned to point to the module memory using the following statement:

```
m_stackWrapper.array = stackWrapperArray;
```

Although this is a rather straightforward approach, the code modifications required for the setup of the container class are considerable, and the abstraction provided by the container class is violated.

It should be noted that the separation of memory and control enables the passing of the container objects into functions as pointers. This could potentially be seen as an advantage; however, optimization based on pointer references is extremely difficult, and the QoR might be impacted if the design relies on this capability.

If the container instance can be implemented as a module variable, the need to use indirection – the passing of a pointer reference – becomes unnecessary.

5.2.3.2 Modeling Container Classes as Local SystemC Modules

Since the memory is actually representing a physical component, a more appropriate modeling style is to actually represent the container class as a separate module. This is done in SystemC by deriving the container class from the **sc_module** class. The stack class example could be specified as follows:

```
template <class DataType, int Size>
class Stack : public sc_module {
    DataType array[Size];
    int      nextLocation;
public:
    Stack(sc_module_name n) : sc_module(n), nextLocation(0) {};
    void reset(void) {
        nextLocation = 0;
    }
    void push(DataType val) {
        array[nextLocation++] = val;
    }
    DataType pop() {
        return array[--nextLocation];
    }
};
```

The only other difference in the source code would be the actual call to the container constructor.

The stack constructor call in the parent module should be modified to contain the name, as follows, since modules should be uniquely named in SystemC:

```
class dut_mod : public sc_module
{
    Stack m_stackModule;

    dut_mod(sc_module_name n)
        : sc_module(n),
        ... ,
        m_stackModule("stackModule")
    {
        ...
    }
}
```

Since modules are supposed to be separate components, CtoS requires that all communication between components be purely signal-based. This requirement would result in major rewriting of the container class, which would generally be undesirable. However, for small designs, you can circumvent this by flattening the complete design during elaboration (setting the **build_flat** design attribute during design setup). If CtoS encounters this coding style, you will get an error message suggesting this solution.

Note It is impossible to synthesize a design with references to **sc_modules**. As a result, it is not possible to pass references to a container based on **sc_modules** to a function. In general, this should not be necessary if the container class can be instantiated in the module of the process.

5.2.3.3 Modeling Container Classes as Interface and SystemC Channels

As the complete flattening of the design is not always an option, a container class can also be modeled using the SystemC construct **sc_channel**. SystemC introduced the **sc_channel** construct to specifically model communication behavior. As this is precisely the modeling intent behind the abstraction of a container class, this is a good match as long as no indirection is required to reference the container (see [“Modeling Container Classes as Local SystemC Modules” on page 5-14](#)).

The stack class example could be specified as follows:

```
template <class DataType>
class stack_if : virtual public sc_interface {
public:
    virtual void reset(void) = 0;
    virtual void push(DataType val) = 0;
    virtual DataType pop() = 0;
};

template <class DataType, int Size>
class Stack : public stack_if<DataType>, public sc_channel {
    DataType array[Size];
    int      nextLocation;
public:
    Stack(sc_module_name n) : sc_channel(n), nextLocation(0) {};
    void reset(void) {
        nextLocation = 0;
    }
    void push(DataType val) {
        array[nextLocation++] = val;
    }
    DataType pop() {
        return array[--nextLocation];
    }
};
```

In this case, a virtual interface class will have to be defined, and the actual **sc_channel** representing the stack will have to be derived from it. In addition, as in the case of the **sc_module** implementation, the only other source code change is the named instantiation.

```
class dut_mod : public sc_module
{
    Stack m_stackChannel;

    dut_mod(sc_module_name n)
        : sc_module(n),
        ... ,
        m_stackChannel("stackModule")
    {
        ...
    }
}
```

The amount of SystemC source code modification is higher if the starting point is pure C++; however, this modeling approach has several advantages. For example, the intended abstraction of the container class is completely kept intact, and the CtoS synthesis performance and QoR are not impacted. In addition, if the system-level exploration is completely modeled in SystemC, this way of container class specification can easily be incorporated into modeling guidelines.

5.2.4 Choosing the Right Modeling Approach

All of these modeling styles solve the issues around the implementation of large container classes. From a pure SystemC perspective, they are equivalent, and as stated, each has advantages and disadvantages with respect to the degree of source code modification and actual synthesis results.

As you are trying to decide which style to use, consider the following questions:

- Should the hardware implementation of the container class exhibit parallel behavior (processes)?
If the container class is to provide parallel access to the content, the simple component modeling style previously described is insufficient. The SystemC model will have to be refined to describe the concurrency and provide arbitration based on a signal interface to the implementing memory.
- Is the data contained in the container class to be accessed by multiple processes?

As stated in [“Modeling Container Classes for Synthesis” on page 5-12](#), this approach will also require a SystemC model that will define the behavior in the case of concurrent access. The model will require a signal-based interface and the definition of arbitration rules.

- Does the container class truly require large storage?

The modeling styles presented focus on large container class implementations and enable CtoS to utilize memories. As stated, it is alternatively possible to implement small container classes in registers. If this is the case, the original model will not have to be modified at all, and the underlying storage can be transformed into registers with the help of the CtoS **flatten_array** command.

- Can the container class be seen as static?

This is often encountered in SystemC specifications that represent configurable behavior. Good examples are filters or programmable logic where filter parameters or control parameters are stored in static structures. In hardware, these can often be realized as ROM implementations and lookup tables. CtoS supports these directly, and the appropriate modeling style should be utilized.

Note See “[convert_to_lookup](#)” on page E-40 and “[Read-Only Arrays](#)” on page 14-40.

- Does the RTL need to retain the boundary of the component?

RTL modeling guidelines or downstream tool requirements will sometimes influence the higher level modeling requirements. For example, if separate RTL modules are required, the SystemC modules should be modeled as such.

6 Starting, Setting Up, and Building a Design

CtoS uses the concept of *designs* to manage a synthesis environment setup.

After you start a CtoS session, your first step will always be to *create* a new design, or to *open* an existing design.

Then, after you have set *attributes* for the design, you *build* the design, creating data structures to represent the design in the CtoS database.

This chapter tells you how to perform the following tasks:

- “Starting CtoS” on page 6-2
- “Customizing the CtoS Environment” on page 6-3
- “Learning the Basics of the CtoS GUI” on page 6-5
- “Creating a New Design” on page 6-9
- “Building a Design” on page 6-28
- “Viewing Input Source” on page 6-38
- “Using the CDFG Viewer” on page 6-41
- “Navigating the CtoS Environment” on page 6-43
- “Saving, Opening, and Closing Designs” on page 6-51

6.1 Starting CtoS

You can start CtoS in two different modes – the CtoS GUI is highly recommended:

- “Starting the CtoS GUI” on page 6-2
- “CtoS at the Command Line” on page 6-3

6.1.1 Starting the CtoS GUI

The *CtoS graphical user interface* (CtoS GUI) is the highly recommended way to start CtoS

The CtoS GUI provides a better environment for architectural exploration than standard scripts, by allowing you to see visual representations of your micro-architectural decisions. The CtoS GUI also lets you type any CtoS or native Tcl command at the *Command Input* area of the *Command Window*.

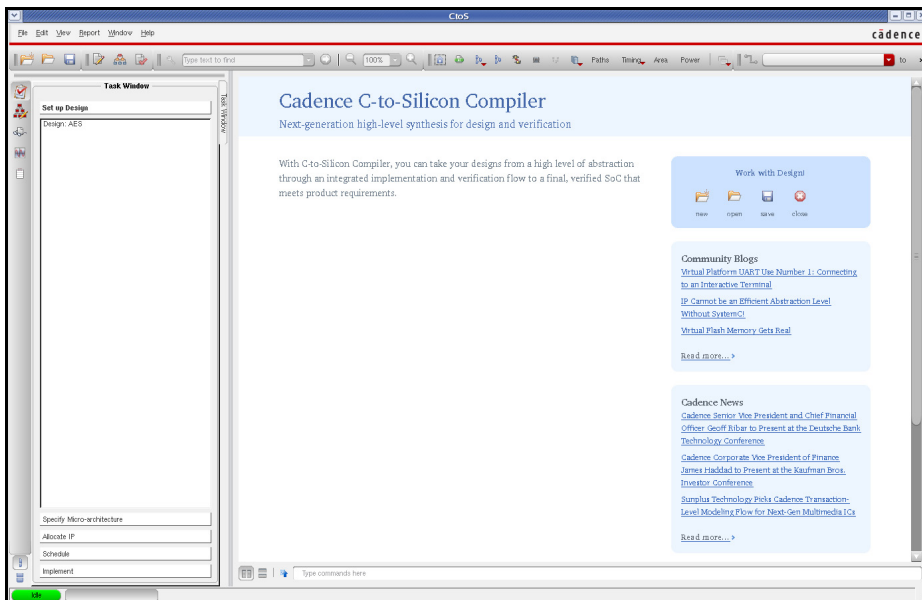
To start the CtoS GUI, type the following, and you will see the *Main View* of the CtoS GUI, as shown in [Figure 6-1 on page 6-2](#):

ctosgui

Notes

- You can include several optional arguments; see “[ctosgui \(executable\)](#)” on page E-52.
- The CtoS executables are located at `install_directory/bin`

Figure 6-1 Main View of the CtoS GUI



6.1.2 CtoS at the Command Line

To run CtoS in a Tcl command-line environment, type:

ctos

Note You can include several optional arguments; see “[ctos \(executable\)](#)” on page E-51.

Version information about the release of CtoS is first displayed. Then, you get a Tcl prompt (the % character). CtoS uses the Tcl programming language as the basis for its command-line interface, and you can use any native Tcl command, in addition to all of the CtoS commands.

To exit from the Tcl environment, simply type:

exit

Note All of the CtoS Tcl commands are described in “[CtoS Command Reference](#)” on page E-1.

6.2 Customizing the CtoS Environment

You can customize your CtoS environment using the following features:

- “[Loading Initialization Files at Startup](#)” on page 6-3
- “[Using the Preferences Dialog and File](#)” on page 6-4

6.2.1 Loading Initialization Files at Startup

CtoS lets you load *initialization files*, written in Tcl, at startup of either the CtoS GUI or CtoS in a Tcl environment. These files are loaded before the scripts specified on the command line are run.

To use this feature, you can create one to three files, each with filename **.ctos_init**, in any of the following directories, and they will be loaded in this search order:

- Site-specific setup: *install_directory*/share/ctos (always loaded)
- User-specific setup: *home_directory*/.cadence/ (loaded unless **-n** specified with **ctosgui/ctos**).
- Design-specific setup: *current_directory* (loaded unless **-n** specified with **ctosgui/ctos**).

For each file loaded, the following is printed to the log:

```
Loading Initialization File: "full_path_to_filename"
```

If the file is not found, or is not readable, it is quietly skipped.

Note See also “[ctosgui \(executable\)](#)” on page E-52 and “[ctos \(executable\)](#)” on page E-51.

6.2.2 Using the Preferences Dialog and File

The **Preferences** dialog (**File -> Preferences**, or shortcut: **Ctrl+Shift+P**), as shown in [Figure 6-2 on page 6-4](#) and [Figure 6-3 on page 6-4](#), lets you specify which icons to display for the *Quick Launch* (“*Quick Launch*” on page 6-8) and window size, log buffer size, background, and text zoom factor for **Command Output** (“*Command Window*” on page 6-8). For future sessions, this data is stored in:

```
home_directory/.cadence/ctos.gui
```

Figure 6-2 Preferences Dialog (Quick Launch)

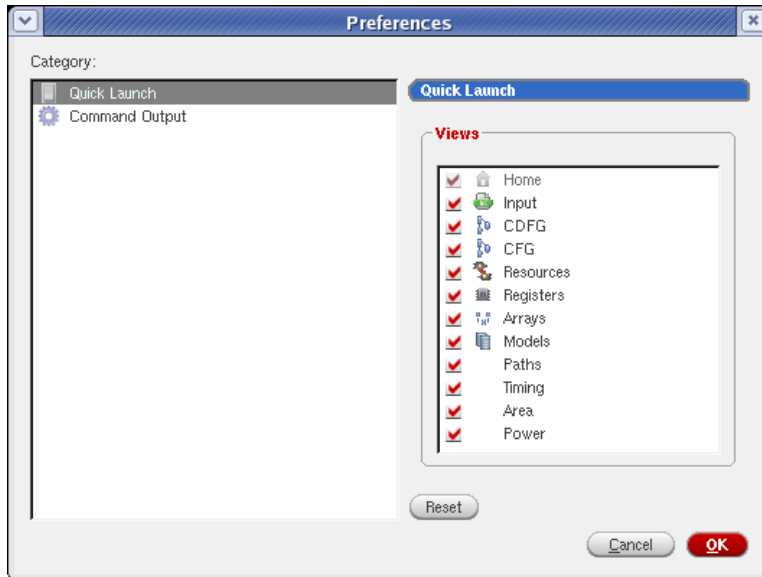
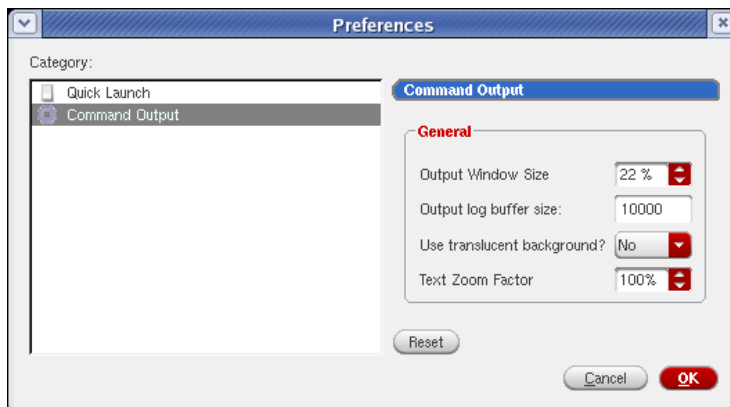


Figure 6-3 Preferences Dialog (Command Output)



6.3 Learning the Basics of the CtoS GUI

The following sections describe some of the basics of the CtoS GUI:

- “Menu Bar” on page 6-5
- “Tool Bar” on page 6-7
- “Quick Launch” on page 6-8
- “Command Window” on page 6-8
- “Shortcuts in the CtoS GUI” on page 6-9

6.3.1 Menu Bar

The *Menu Bar*, as shown in [Figure 6-4 on page 6-5](#), provides commands, using drop-down menus, in the order in which you should usually run them. For added guidance, commands that would not work, or would not provide useful information, at a particular step in the design flow, are grayed out.

Figure 6-4 Menu Bar



Here is a listing of all of the commands in the Menu Bar:

```
File
  New Design
  Open Design
  Close Design
  Command History
  Write Design Setup
  Save Design
  Save Design As
  Generate ->
    RTL
    Verilog Behavioral Model
    Verification Wrapper
    Top Wrapper
    TLM Wrapper
  Preferences
  Exit
Edit
  Undo (Rerun)
  Design Properties
  Build
  Specify Micro-architecture
  Read IP Definitions
  Allocate IP
```

- Use DSP
- Guide Scheduler ->
 - Create Array Dependencies
 - Create Required States
 - Create Initial Resources
- Region ->
 - Constrain Latency
 - Protocol Region
 - Float I/O Accesses
 - Float Array Accesses
- Schedule
- Allocate Registers
- View
 - Home
 - Input
 - CDFG
 - CFG
 - Resources
 - Registers
 - Arrays
 - Generated Models ->
 - RTL
 - Verilog Behavioral Model
- Report
 - Check Design
 - Paths
 - Timing
 - Cycle Analysis
 - Pre-schedule Report
 - Area
 - Power
 - Region Latency
- Window
 - Split Horizontally
 - Split Vertically
 - Task Window
 - Hierarchy Window
 - Call Graph
 - Simulation Monitor
 - Summary Report
 - Toolbars ->
 - Design
 - Edit
 - View
 - Quick Launch
 - Region
 - Close All Views
- Help
 - User Guide
 - Quick Start Guide
 - Commands
 - About

6.3.2 Tool Bar

The *Tool Bar* has icons for four aspects of working with designs:

- *Design*, as shown in [Figure 6-5 on page 6-7](#)
- *Edit*, as shown in [Figure 6-6 on page 6-7](#)
- *View*, as shown in [Figure 6-7 on page 6-7](#), which has two features available only through the *Tool Bar*: *Find* and *Zoom* (shortcut: **Ctrl+F** moves the cursor to the *Find Pattern* box)
- *Region*, as shown in [Figure 6-8 on page 6-7](#)

You can customize the *Tool Bar* by right-clicking on it or by selecting **Window -> Toolbars**.

Figure 6-5 Design Tool Bar (New Design, Open Design, Save Design)



Figure 6-6 Edit Tool Bar (Design Properties, Build, Check Design)



Figure 6-7 View Tool Bar (Find Config, Find Pattern, Find Next, Zoom Out, Zoom, Zoom In)



Figure 6-8 Region Tool Bar (Region Editor, Start Node, End Node)

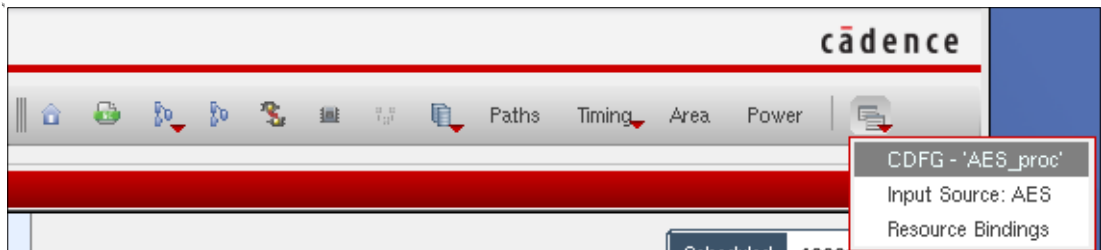


6.3.3 Quick Launch

The *Quick Launch*, as shown in [Figure 6-9 on page 6-8](#), lets you quickly launch the CtoS GUI viewers and models or return to the home page. You can also switch among viewers that you currently have open with the *Opened* views icon (last on the right).

You can customize the *Quick Launch* by right-clicking on it, or by using the **Preferences** dialog (see [“Using the Preferences Dialog and File” on page 6-4](#)).

Figure 6-9 Quick Launch



6.3.4 Command Window

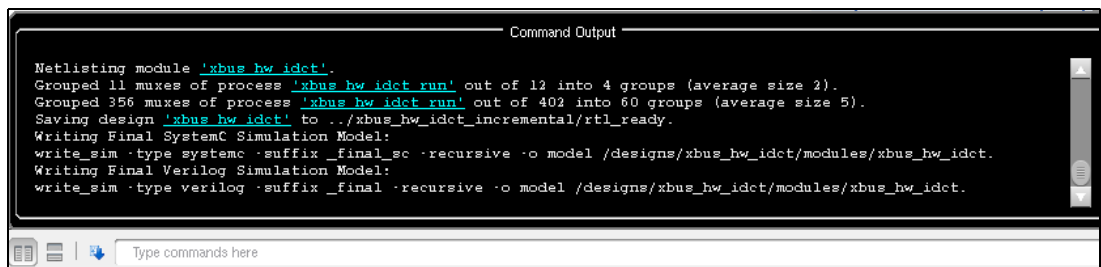
The *Command Window*, as shown in [Figure 6-10 on page 6-8](#), has two components:

- **Command Output**, which displays a running log of each command used in your current session (whether entered by icon, menu, or command line). This log is automatically saved in a log file named **ctosgui.log** – to save it to a different filename, use the optional argument `[-log log_filename]` when you start the CtoS GUI [see [“ctosgui \(executable\)” on page E-52](#)].
- *Command Input* (the blinking >), which lets you enter Tcl-based commands.

You can customize **Command Output** using the **Preferences** dialog (see [“Using the Preferences Dialog and File” on page 6-4](#)).

For shortcuts when using the *Command Window*, see the following section, [“Shortcuts in the CtoS GUI” on page 6-9](#).

Figure 6-10 Command Window



6.3.5 Shortcuts in the CtoS GUI

The CtoS GUI has many shortcuts to help with navigation and other tasks, as follows:

When using the *CtoS GUI*, in general:

- **Ctrl+Shift+P** displays the **Preferences** dialog.
- **Ctrl+F** moves the cursor to the *Find Pattern* box of the *Tool Bar* (assuming you have an appropriate viewer open).
- **Ctrl+space bar**, or the blue arrow to the left of *Command Input*, toggles **Command Output**.
- **Esc** closes **Command Output**.
- *Command Input* has a *completion* feature that lets you type just a defining number of letters of a command, then press the **Tab** key to complete the name. If more than one command starts with the letters typed, they will all be displayed in **Command Output**.

To display the *context menus*:

- **Right-clicking** on an object displays its context menu (the default action is in **boldface**).
- **Double-clicking** on an object performs the default context menu action.

When using the *CDFG*, *Tree Map*, *Cycle Analysis viewer*, and *Pipeline View*:

- **Ctrl with Mouse Wheel**, or **Ctrl with + or - keys**, zooms in and out.

When viewing *web pages from the home page*:

- **Right-clicking** displays a context menu of **Go Back** or **Reload**. The cursor is also moved directly to *Command Input*, so you can start typing immediately.

6.4 Creating a New Design

CtoS provides a step-by-step approach, using two techniques, when you are setting up a new design:

- “[Create New Design Wizard](#)” on [page 6-10](#) steps you through the most commonly required, and some optional, attributes for your design.

You can then continue with the **Design Property** dialog, if you need to set attributes not included in the wizard.

- “[Design Property Dialog](#)” on [page 6-18](#) provides a more comprehensive listing of design attributes, which you can use to set up a design, or to add to a design you have created with the wizard.

6.4.1 Create New Design Wizard

To create a new design, select **File -> New Design** (or the **New Design** icon, as shown in [Figure 6-11 on page 6-10](#)) in the CtoS GUI.

The **Create New Design Wizard**, as shown in [Figure 6-13 on page 6-11](#) through [Figure 6-17 on page 6-18](#), walks you through the setup of your design.

Figure 6-11 New Design Icon



The **Create New Design Wizard** has the following helpful features:

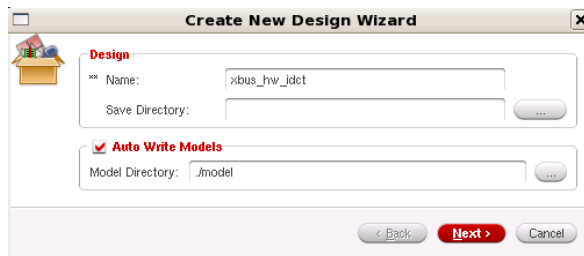
- Required fields are designated with **.
- The **Next** button is disabled until all required fields are specified.

When you complete the **Create New Design Wizard**, you may then build your design or continue setting attributes with the **Design Property** dialog.

Notes

- For a complete listing of design attributes, see “[Design Object Attributes \(Designs\)](#)” on page D-9.
- You could also use the **new_design** command (“[new_design](#)” on page E-88) and set attributes using the **set_attr** command (“[set_attr](#)” on page E-140). The **new_design** command does *not* use the **Create New Design Wizard**.

Figure 6-12 Create New Design Wizard (Page 1)



In this page, you are selecting names and directories for your design:

- **Name:** You can enter a name for the design. The name cannot have any embedded spaces. CtoS will create the design with this name, using the **new_design** command (“[new_design](#)” on page E-88).

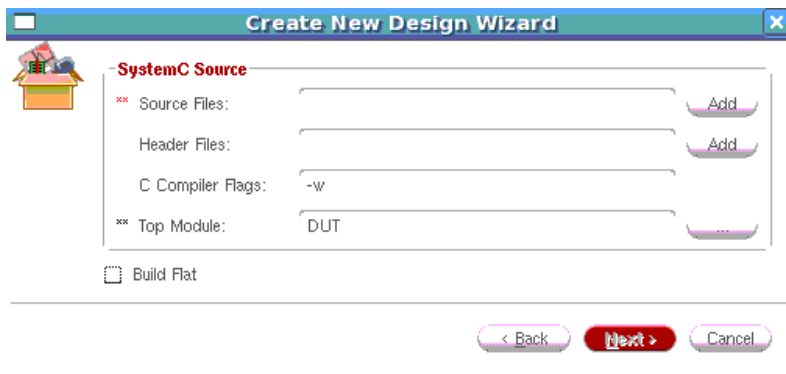
- **Save Directory:** Use the ... button to scroll to the directory in which to save the design. The default is the current directory. This sets the **design_dir** design attribute ("[design_dir](#)" on page D-9).
- **Auto Write Models:** Uncheck this box to *not* have CtoS automatically write simulation models after a successful build (**post_build**) and a successful allocate registers (**final**). If enabled, the models are written to the **Model Directory** specified in the next field. This also writes the verification wrapper and TLM wrapper (if a TLM design) after a successful build. This sets the **auto_write_models** design attribute ("[auto_write_models](#)" on page D-11).

Note See also "Verifying Designs" on page 7-1.

- **Model Directory:** Use the ... button to scroll to the directory in which to store simulation models. This sets the **model_dir** Default Simulation Config object attribute ("[model_dir](#)" on page D-85) by calling the **design_sim_config** command with the **-model_dir directory** option ("[define_sim_config](#)" on page E-55).

Note See also "Verifying Designs" on page 7-1.

Figure 6-13 Create New Design Wizard (Page 2)



In this page, you are identifying your SystemC source and build options:

- **Source Files:** Use the **Add** button to browse to the source file(s) in which the top module of the design is instantiated. You can select each file individually, or use the **Shift** key to select more than one. This sets the **source_files** design attribute ("[source_files](#)" on page D-24).

Note If your design has multiple source files, see "Compiling Multi-Source Designs through Single Combined Source" on page 6-29

- **Header Files:** List the header files needed to instantiate the SystemC DUT in a verilog verification wrapper. This is an ordered list (nested child types before parent type) of file names, and you are responsible for the correct order of these files. The **write_verilog_wrapper** command ("[write_verilog_wrapper](#)" on page E-175) will **#include** these using the relative file path. This sets the **header_files** design attribute ("[header_files](#)" on page D-16).

- **C Compiler Flags:** List any C/C++ compiler flags necessary to parse the source files. This sets the **compile_flags** design attribute (“[compile_flags](#)” on page D-13). CtoS supports the following:

- These flags define the macro **name** as **def**. If the **=def** is omitted, the macro **name** is defined as **1**. You can define function-style macros by appending a macro parameter list to the **name**.

```
--define_macro name [(parm-list)] [=def]  
-Dname [(parm-list)] [=def]
```

- These flags remove any initial definition of the macro **name**. Note that CtoS processes any **--undefine_macro** options after processing all **--define_macro** options in the command line.

```
--undefine_macro name  
-Uname
```

- These flags add **dir** to the list of directories in which to search for **#includes**.

```
--include_directory dir  
-Idir
```

- These flags suppress warnings during parsing, but errors are still issued (**-w** is the default).

```
--no_warnings  
-w
```

- **Top Module:** Use the ... button to parse the SystemC source file to get the hierarchical path to the module instance to be synthesized.

Tip To expand the **Browse Source Modules** window to see long names, resize it by hovering over the corner and dragging the corner to the desired size.

If you are using the **SC_MODULE_EXPORT** macro in the **.cpp** file associated with this module, this is simply the name of the module.

Otherwise, it is the period (.) separated hierarchical path to the instance.

This sets the **top_module_path** design attribute (“[top_module_path](#)” on page D-25).

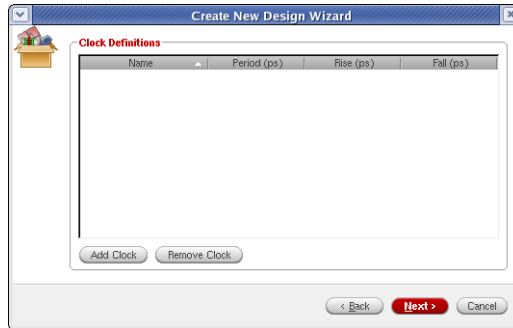
Note See also “[Instantiation of the Top Module](#)” on page 14-4.

- **Build Flat:** Check this box if you do *not* want CtoS to preserve the design hierarchically during build, that is, you want CtoS to *flatten* the **SC_MODULE** hierarchy into a single database module.

This sets the **build_flat** design attribute (“[build_flat](#)” on page D-12).

Note See also “[Module Hierarchy](#)” on page 16-1.

Figure 6-14 Create New Design Wizard (Page 3)



In this page, you are setting up clock definitions, which sets up a clock or clocks, using the **define_clock** command ([“define_clock” on page E-53](#)).

Important CtoS provides limited support for designs with multiple clocks. If a design has more than one clock, one must be the *base*, and its frequency must be a multiple of all other frequencies. Also, CtoS will not create circuitry to ensure clock domains are properly interfaced; this is left to the designer. See [“Limited Support for Multiple Clocks” on page 1-8](#).

- The **Add Clock** button lets you add a clock; default is **clock_0** with **Period 20000**, **Rise 0**, **Fall 10000**.
 - The **Name** specifies the clock name, which is editable with a line editor. The name cannot have any embedded spaces. If a new clock name is not unique, the name is displayed in red (all matches), and the **Next** button is disabled until the name is corrected.

Note The name must be an input port of the top module (specified in **Top Module**); if you do not explicitly set a clock, CtoS will use the *default clock* [using values from the **default_clock** design attribute ([“default_clock” on page D-13](#))] when calling RC.
 - The **Period** specifies the clock period, which is editable with a minimum of 0, a step of 100ps, and a maximum of 2^{24} , which is defined by the Encounter RTL Compiler (RC). When the period changes, the larger of the **Rise** or **Fall** times changes 180 degree in phase.
 - The **Rise/Fall** specifies the rise and fall, which is editable with a minimum of 0 and a maximum of **Period**. If **Rise** = **Fall** (except zero), the value is displayed in red, and the **Next** button is disabled until it is corrected.
- The **Remove Clock** button lets you remove the selected clock.
- You can interactively edit (by double-clicking or typing directly on) the fields in the table.
- You can use copy/paste with the **Ctrl-C** and **Ctrl-V** keys on an individual item or an entire row.
 - For an individual item, select the item, then paste it onto selected item(s), which must be in the same column. Copy will not work if you select the clock name (these names must be unique). Paste will not work if the column is different from the copied cell.
 - For an entire row, select the entire row, then select one or more rows on which to copy (and it must be the complete row).

Figure 6-15 Create New Design Wizard - ASIC (Page 4)

The screenshot shows the 'Create New Design Wizard' dialog box, Page 4, titled 'Create New Design Wizard'. The 'Implementation Aware' checkbox is checked. The 'Implementation Target' is set to 'ASIC' and the 'Default Scheduling Effort' is set to 'high'. Under the 'Setup RC' section, the 'Technology Library' is empty, 'RC Startup Script' is empty, and 'Working Directory' is 'rc_work'. Under the 'Prototype Memory Timing Attributes' section, 'Setup Delay' is '20%' and 'Launch Delay' is '80%'. Under the 'Export and External Memories' section, 'Default Export' is unchecked and 'Top Wrapper Suffix' is '_top_wrapper'. At the bottom, 'Reset Registers' is set to 'Only Required' and 'Maximize Clock Gating' is unchecked. Navigation buttons at the bottom are '< Back', 'Next >', and 'Cancel'.

In this page, you are setting up implementation options. Some apply to either ASIC or FPGA; some are specific to the type of implementation, as follows:

- “[Either ASIC or FPGA \(CNDW, Page 4\):](#)” on page 6-14
- “[ASIC \(CNDW, Page 4\):](#)” on page 6-16
- “[FPGA \(CNDW, Page 4\):](#)” on page 6-17

Either ASIC or FPGA (CNDW, Page 4):

- **Implementation Aware:** Uncheck this box when you do not need to consider timing, which can be useful early in the design cycle when you are simply trying to build your design.
- **Implementation Target:** Choose **ASIC** or **FPGA**. This sets the **implementation_target** design attribute (“[implementation_target](#)” on page D-17).
- **Default Scheduling Effort:** Specifies the number of phases the scheduler should perform. This sets the **default_scheduling_effort** design attribute (“[default_scheduling_effort](#)” on page D-14).
 - **high** performs all phases of the multiple-phase scheduler: Memory, Timing, and Sharing Feasibility and Final Phases.
 - **medium** performs Memory, Timing and Sharing Feasibility.
 - **low** performs Memory Feasibility phase. It also performs the Timing Feasibility phase if technology libraries are specified. See “[tech_lib_names](#)” on page D-24.

Note See also “[Timing Analysis](#)” on page -4.

- **Prototype Memory Timing Attributes:** Enter the setup and launch delays for a prototype memory, if you do not want to use the defaults of 20% and 80%, respectively. This sets both the **prototype_memory_setup_delay** design attribute (“[prototype_memory_setup_delay](#)” on page D-21 and **prototype_memory_launch_delay** design attribute (“[prototype_memory_launch_delay](#)” on page D-21).

Note See also “[Allocating Prototype Memory](#)” on page 9-7 for more detail, including limitations.

- **Default Export:** Check this box to set the default for the *Exported Memory* feature (see “[Exporting Memories](#)” on page 9-20), in which top module ports are created to connect memories instead of memory instances in modules. This separates memories (RAMs or ROMs) from the actual design.

This sets the **default_export_memories** design attribute (“[default_export_memories](#)” on page D-13).

Note Built-in RAMs are *not* exported.

- **Top Wrapper Suffix:** If you have checked the previous box (**Default Export**), you can enter a filename extension for output Verilog top wrapper files, if you do not want to use the default, **_top_wrapper**.

This wrapper connects the memories and the RTL design to your SystemC testbench.

This sets the **verilog_top_wrapper_suffix** design attribute (“[verilog_top_wrapper_suffix](#)” on page D-27). Note that this *cannot* be an empty string because this can result in a name conflict of the top module and the top wrapper module of the design.

Note See also “[Generating a Top Wrapper for Exported Memories \(Only after Scheduling\)](#)” on page 7-15.

- **Reset Registers:** Some test methodologies require *all registers to be reset*. However, since adding reset logic increases the size of a design, CtoS does not, by default, reset registers for a design. If you need to reset registers, check this box, and all registers with a *valid reset path* will be reset, as specified by the SystemC code. All other non-**sc_signals** will be reset to zero; for all **sc_signals** that are *not* reset, you will receive a warning. This does not change the design’s functionality, as the reset state of the registers is observable only in test mode.

Depending on your design requirement, you can select one of the following options:

- **Only Required:** Reset is added only if it is functionally required. By default, this option is selected.
- **Internal:** Reset is added to all internal registers and **sc_signal** registers that are reset in the SystemC. And, a warning is given for **sc_signal** registers that are not reset in the SystemC.
- **Internal + Outputs:** Reset is added to all **internal** and **sc_signal** registers. And, an info message is given for all **sc_signals** that are not reset in the SystemC so the user can review these to see if resetting these registers will cause a simulation mismatch.

Note This sets the **reset_registers** design attribute (“[reset_registers](#)” on page D-23). See also “[Resetting Registers](#)” on page -19 for more detail.

- **Maximize Clock Gating:** Check this box if you want CtoS to optimize this design for clock gating, that is, to choose signals to use as clock enables. This sets the **low_power_clock_gating** design attribute (“[low_power_clock_gating](#)” on page D-17).

Note *Power Estimation is a preliminary feature.* See also “[Using the analyze Command](#)” on page 18-6 for more detail.

ASIC (CNDW, Page 4):

- **Technology Library:** Use the **Add** button to scroll to your technology library. This sets the **tech_lib_names** design attribute (“[tech_lib_names](#)” on page D-24), and there is more information about these libraries in the description for the **rc_startup_script** design attribute (“[rc_startup_script](#)” on page D-22).

Notes

- You *must* include a technology library when using RC; otherwise, you will get an error when CtoS tries to call RC.
- See also “[Specifying Vendor RAM Library File/Simulation Model](#)” on page H-19 for details on the technology libraries used for Vendor RAMs.
- **RC Startup Script:** Use the ... button to scroll to the absolute path of the optional script file to be used as an initialization script for Encounter RTL Compiler (RC).

This script is sourced by RC after it is started, before the tech libraries are set. This sets the **rc_startup_script** design attribute (“[rc_startup_script](#)” on page D-22).

Important In order to reference tech libraries, you must have set them in the RC startup script before they are referenced. CtoS does not close the tech libraries that are opened in the RC startup script if the tech libraries are not set in CtoS or if the tech libraries are identical.

Note A startup script can control certain parameters, for example, parameters like **dont_use** or **dont_touch** in technology library cells can be set to *true* or *false* at run time without changing the library file, itself.

- **Working Directory:** Use the ... button to scroll to your RC working directory, if you do not want to use the default, **rc_work**. This sets the **rc_work_dir** design attribute (“[rc_work_dir](#)” on page D-22).

Figure 6-16 Create New Design Wizard - FPGA (Page 4)

The screenshot shows the 'Create New Design Wizard' dialog box, Page 4, titled 'FPGA'. The dialog has a blue title bar and a light gray background. On the left, there is a small icon of a box with a red 'X' and a green checkmark. The main content area is divided into several sections:

- Implementation Aware:** This section contains two dropdown menus: 'Implementation Target:' set to 'FPGA' and 'Default Scheduling Effort:' set to 'high'.
- Setup FPGA:** This section contains several fields:
 - 'FPGA Vendor:' dropdown menu set to 'Xilinx'.
 - 'Install Path:' text field with a browse button ('...').
 - 'Family Name:' text field.
 - 'Part Name:' text field.
 - 'Working Directory:' text field set to 'fpga_work' with a browse button ('...').
 - 'DSP Cost:' text field set to '100' with a unit '(in terms of LUTs)'.
- Prototype Memory Timing Attributes:** This section contains two dropdown menus: 'Setup Delay:' set to '20%' and 'Launch Delay:' set to '80%'.
- Reset Registers:** A dropdown menu set to 'Only Required'.
- Maximize Clock Gating:** A checkbox that is currently unchecked.

At the bottom of the dialog, there are three buttons: '< Back', 'Next >' (highlighted in red), and 'Cancel'.

FPGA (CNDW, Page 4):

Note Support for FPGA designs is a preliminary feature. See also “CtoS Tutorial for FPGA designs” on page B-1 for an example of the CtoS FPGA Prototype Flow.

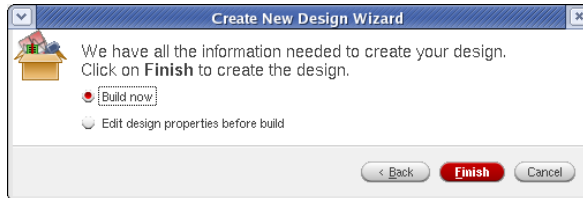
- **FPGA Vendor:** Choose Xilinx or Altera, the currently supported FPGA vendors for CtoS. This sets the first value of the **fpga_target** design attribute (“fpga_target” on page D-16).
- **Install Path:** Use the ... button to scroll to the installation path to the executable for FPGA designs. This sets the **fpga_install_path** design attribute (“fpga_install_path” on page D-16).
- **Family Name:** Enter the device family for this FPGA design. This sets the second value of the **fpga_target** design attribute (“fpga_target” on page D-16).
- **Part Name:** Enter the part name for this FPGA design. This sets the third value of the **fpga_target** design attribute (“fpga_target” on page D-16).

Note By convention, the part name indicates the device name, speed grade and package name. It is important to specify the appropriate part so that CtoS has accurate timing information.

- **Working Directory:** Use the ... button to scroll to your FPGA working directory, if you do not want to use the default, **fpga_work**. This sets the **fpga_work_dir** design attribute (“fpga_work_dir” on page D-16).

- **DSP Cost:** Enter the DSP cost, in terms of LUTs, if you do not want to use the default of **100**. This sets the `fpga_dsp_cost` design attribute ([“fpga_dsp_cost” on page D-16](#)).

Figure 6-17 Create New Design Wizard (Page 6)



In this page, you can choose to build your design, if you have set up everything needed at this point, or alternately, you can continue to set up more attributes and properties for the design with the **Design Property** dialog, as described in the following section ([“Design Property Dialog” on page 6-18](#)) and then build it later, as described the next section ([“Building a Design” on page 6-28](#)).

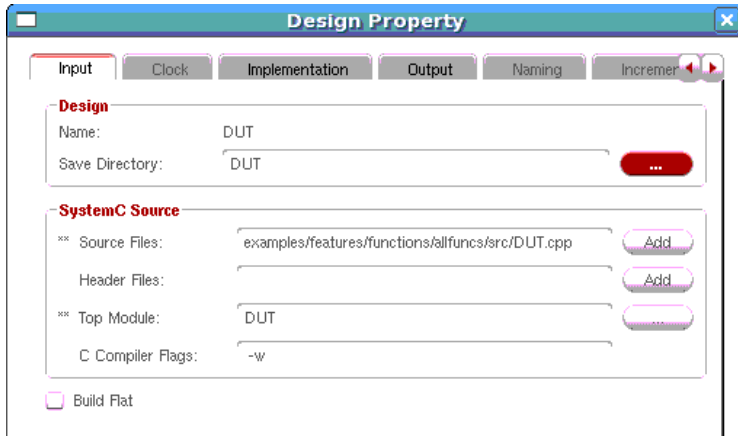
6.4.2 Design Property Dialog

The **Design Property** dialog, as shown in [Figure 6-18 on page 6-19](#) through [Figure 6-23 on page 6-26](#), lets you set, or further edit, the properties for your design. This dialog has the following helpful guides:

- A tab in the **Design Property** dialog is displayed in **red** until data required for a field under that tab is supplied, when it reverts to black.
- Required fields are designated with ******.
- As shown in [Figure A-9 on page A-10](#) and [Figure A-14 on page A-15](#), some of the fields are *locked* after certain commands in the synthesis flow. This is to prevent you from inadvertently changing an attribute that should not be changed after you have built or scheduled your design, in order to preserve the integrity of your synthesis process. However, if you decide you must change something, you can use the **Change Setup** button, which causes the read-only fields to be editable.

Note For a complete list of all design attributes and when you must set them, see [“Design Object Attributes \(Designs\)” on page D-9](#).

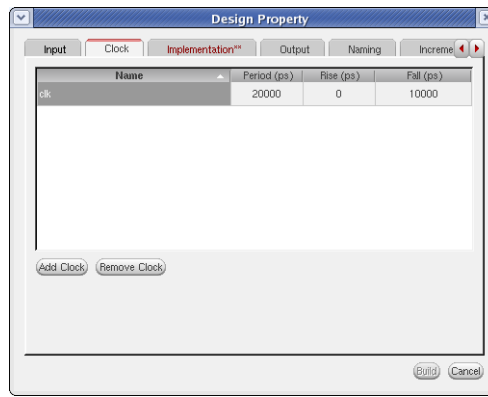
Figure 6-18 Design Property Dialog - Input Tab



In the **Input** tab, you are selecting names, directories, and files for your design. All of these fields are the same as their counterparts in the **Create New Design Wizard**, as follows:

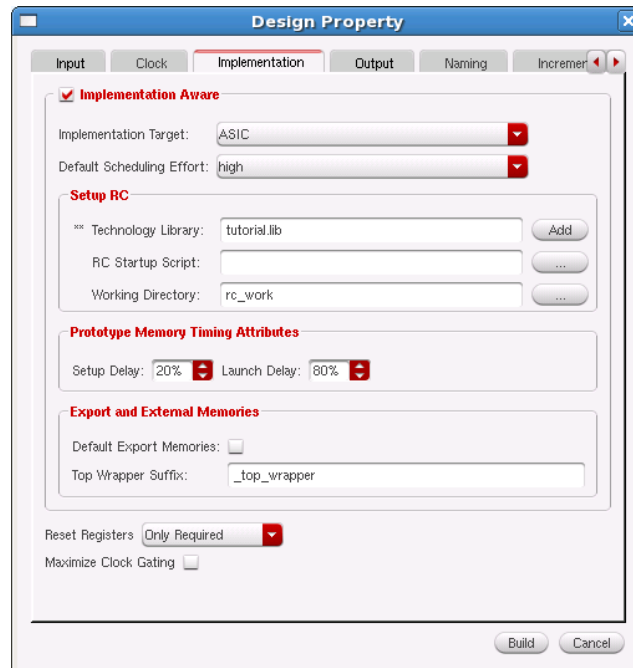
- **Name:** See **Name** in “[Create New Design Wizard \(Page 6\)](#)” on page 6-18.
- **Save Directory:** See **Save Directory** in “[Create New Design Wizard \(Page 6\)](#)” on page 6-18.
- **Source Files:** See **Source Files** in “[Create New Design Wizard \(Page 2\)](#)” on page 6-11.
- **Header Files:** See **Header Files** in “[Create New Design Wizard \(Page 2\)](#)” on page 6-11.
- **Top Module:** See **Top Module** in “[Create New Design Wizard \(Page 2\)](#)” on page 6-11.
- **C Compiler Flags:** See **C Compiler Flags** in “[Create New Design Wizard \(Page 2\)](#)” on page 6-11.
- **Build Flat:** See **Build Flat** in “[Create New Design Wizard \(Page 2\)](#)” on page 6-11.

Figure 6-19 Design Property Dialog - Clock Tab



In the **Clock** tab, you are setting up clock definitions. All of these fields are the same as those on page 3 of the **Create New Design Wizard**, as shown in [“Create New Design Wizard \(Page 3\)”](#) on page 6-13.

Figure 6-20 Design Property Dialog - Implementation Tab



In the **Implementation** tab, you are setting up implementation options. All of these fields are the same as their counterparts in the **Create New Design Wizard**, as follows:

- **Implementation Aware:** See **Implementation Aware** in “[Either ASIC or FPGA \(CNDW, Page 4\):](#)” on page 6-14.
- **Implementation Target:** See **Implementation Target** in “[Either ASIC or FPGA \(CNDW, Page 4\):](#)” on page 6-14.
- **Default Scheduling Effort:** See **Default Scheduling Effort** in “[ASIC \(CNDW, Page 4\):](#)” on page 6-16.

ASIC

- **Technology Library:** See **Technology Library** in “[ASIC \(CNDW, Page 4\):](#)” on page 6-16.
- **RC Startup Script:** See **RC Startup Script** in “[ASIC \(CNDW, Page 4\):](#)” on page 6-16.
- **Working Directory:** See **Working Directory** in “[ASIC \(CNDW, Page 4\):](#)” on page 6-16.

FPGA

- **FPGA Vendor:** See **FPGA Vendor** in “[FPGA \(CNDW, Page 4\):](#)” on page 6-17.
- **Install Path:** See **Install Path** in “[FPGA \(CNDW, Page 4\):](#)” on page 6-17.
- **Family Name:** See **Family Name** in “[FPGA \(CNDW, Page 4\):](#)” on page 6-17.
- **Part Name:** See **Part Name** in “[FPGA \(CNDW, Page 4\):](#)” on page 6-17.
- **Working Directory:** See **Working Directory** in “[FPGA \(CNDW, Page 4\):](#)” on page 6-17.
- **DSP Cost:** See **DSP Cost** in “[FPGA \(CNDW, Page 4\):](#)” on page 6-17.

ASIC or FPGA

- **Prototype Memory Timing Attributes:** See **Prototype Memory Timing Attributes** in “[Either ASIC or FPGA \(CNDW, Page 4\):](#)” on page 6-14.
- **Default Export Memories:** See **Default Export** in “[ASIC \(CNDW, Page 4\):](#)” on page 6-16.
- **Top Wrapper Suffix:** See **Top Wrapper Suffix** in “[ASIC \(CNDW, Page 4\):](#)” on page 6-16.
- **Reset Registers:** See **Reset Registers** in “[Either ASIC or FPGA \(CNDW, Page 4\):](#)” on page 6-14.
- **Maximize Clock Gating:** See **Maximize Clock Gating** in “[Either ASIC or FPGA \(CNDW, Page 4\):](#)” on page 6-14.

Figure 6-21 Design Property Dialog - Output Tab



In the **Output** tab, you are selecting names and directories for your design:

- **Auto Write Models:** See **Auto Write Models** in “[Create New Design Wizard \(Page 6\)](#)” on page 6-18.
- **Model Directory:** See **Model Directory** in “[Create New Design Wizard \(Page 6\)](#)” on page 6-18.
- **Maximum number of characters per line in model output:** Specifies the maximum length of a line of an output simulation and RTL file, both for SystemC and Verilog (the default is 80). This should be set to a high value in incremental synthesis, to minimize differences due to line breaking. This sets the **max_output_line_length** design attribute (“[max_auto_flatten_array_size](#)” on page D-18).

Note See also “[Incremental Synthesis](#)” on page 17-1.

- **SystemC Source:** Specifies the extension for CtoS-output **SystemC** source files (the default is **.cpp**). This sets the **systemc_out_source_ext** design attribute (“[systemc_out_source_ext](#)” on page D-24).
- **SystemC Header:** Specifies the extension for CtoS-output **SystemC** header files (the default is **.h**). This sets the **systemc_out_header_ext** design attribute (“[systemc_out_header_ext](#)” on page D-24).

- **Verilog:** Specifies the extension to be used for output **Verilog** files produced by CtoS (default is **.v**). This sets the **verilog_out_file_ext** design attribute (“[verilog_out_file_ext](#)” on page D-26).

Note See also “[Generating Verilog Behavioral Models](#)” on page 7-9.

- **Mux Style:** Specifies the RTL coding style for select expression and case labels. The valid values are:
 - **Reverse One Hot** (Default). Produces *case* with constant select expression and variable case labels.
 - **Use Verilog State Names (for One Hot Mux Style)**. Specifies to use the name provided by the user for the wait states.
 - **Case Z**. Produces *casez* with variable select expression and constant case labels.
 - **One Hot**. Produces *case* with variable select expression and constant case labels.

This sets the **verilog_mux_style** design attribute (“[verilog_mux_style](#)” on page D-26).

Note See also “[Controlling Select Expression, Case Label Generation](#)” on page J-2.

- **Use Aligned Widths:** Ensures that inputs to certain arithmetic and all comparison/equality operations are padded to the same matching bit width in the generated RTL. In addition, results of shift operations will be performed with matching bit widths. This sets the **verilog_use_aligned_widths** design attribute (“[verilog_use_aligned_widths](#)” on page D-27).

Note See also “[Controlling Mismatched Operand Bit Widths](#)” on page J-3.

- **Use Case Default X or O:** Controls how the default case will be generated. This sets the **verilog_use_case_default_x** design attribute (“[verilog_use_case_default_x](#)” on page D-27).

Note See also “[Controlling Default Case](#)” on page J-2.

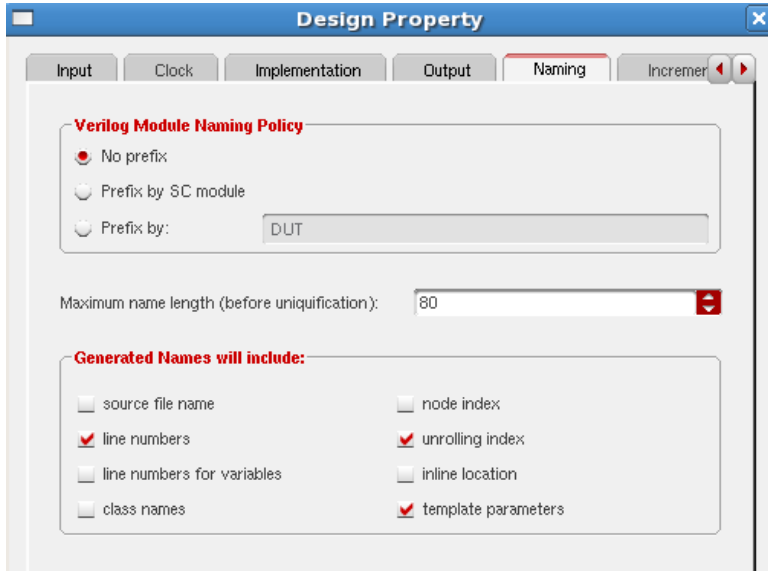
- **Logic Synthesis Pragma Keyword:** Lets you specify a pragma keyword, other than the default of **pragma** (for ASIC) or **synthesis** (for FPGA). This sets the **verilog_pragma_keyword** design attribute (“[verilog_pragma_keyword](#)” on page D-26).

Note See also “[Controlling pragma Keyword for Generated RTL](#)” on page J-4.

- **RTL Model Suffix:** Specifies the extension to be used for output Verilog RTL model files produced by CtoS (the default is **_rtl**). This sets the **verilog_rtl_model_suffix** design attribute (“[verilog_rtl_model_suffix](#)” on page D-26).

Note See also “[Generating an RTL Description after Scheduling](#)” on page 13-36.

Figure 6-22 Design Property Dialog - Naming Tab



In the **Naming** tab, you can control the way CtoS names design objects.

- **Verilog Module Naming Policy:** If you synthesize parts of a design separately, and later use the resulting Verilog files together in simulation and logic synthesis, you could possibly have name conflicts of module-level names, including global functions. To avoid name conflicts of module-level names when sub-modules are synthesized independently, select **Prefix by SC module** (recommended) or **Prefix by** user-specified string:
 - **No prefix:** The names of RTL Verilog modules remain unchanged.
 - **Prefix by SC module:** This option affects names in the resulting RTL in the following ways:
 - The names of RTL Verilog modules for SystemC modules have the same name as the SystemC module.
 - The names of RTL Verilog modules that are generated by CtoS and that do not correspond to SystemC modules are prefixed with the name of the SystemC module that references these modules.
 - The names of RTL Verilog modules provided by user remain unchanged.
 - **Prefix by:** The value of the prefix should be the name of the module that is being synthesized, followed by an underscore '_'. This prefix affects the names of the following modules:
 - The RTL Verilog modules for SystemC modules
 - The RTL Verilog modules that are generated by CtoS and that do not correspond to SystemC modules.

For a detailed listing of the modules included and excluded in each naming options, see [“Integrating Multiple CtoS Designs” on page 16-5](#).

- **Maximum name length (before uniquification):** Specifies the maximum length of any name used in a design, after which truncation occurs, but before uniquification, which means the actual name length may be 2-5 characters longer (the minimum is 16, and the default is 80).

This sets the **name_max_length** design attribute ([“name_max_length” on page D-18](#)).

- **Generated Names will include:** Specifies what to include with the generated names:

Note See also [“Using C++ Labels” on page 14-94](#).

- **source file name:** Check if you want CDFG objects to include source filename(s). This sets the **name_use_file_name** design attribute ([“name_use_file_name” on page D-19](#)).
- **line numbers** (checked by default): Uncheck if you do *not* want the string **_ln**, followed by the line number, to be used for names of nodes and operations in the CDFG derived from statements and expressions, not from variables or fields.

This sets the **name_use_line_number** design attribute ([“name_use_line_number” on page D-19](#)).

- **line numbers for variables:** Check if you want the string **_ln**, followed by the line number, to be used for names of CDFG nodes derived from local method variables. The filename is also used if **name_use_file_name** is also *true* (because variable names can also generate global objects, such as arrays, and hence knowing the source files is useful for multi-file designs).

This option is useful if the SystemC coding style uses several variables with the same name (for example, for loop indices) within the same method. This sets the **name_use_var_line_number** design attribute ([“name_use_var_line_number” on page D-20](#)).

- **class names:** Check if you want the class name to be prepended to behavior names derived from class/field (including **sc_module**) members. This sets the **name_use_class_name** design attribute ([“name_use_class_name” on page D-19](#)).
- **node index:** Check if you want the string **_i**, followed by the CDFG node index, to be part of the name. (This is for debugging only.) This sets the **name_use_node_index** design attribute ([“name_use_node_index” on page D-20](#)).
- **unrolling index** (checked by default): Uncheck if you do *not* want the string **_unr**, followed by the current iteration index, to be added when unrolling a loop. This sets the **name_use_unrolling_index** design attribute ([“name_use_unrolling_index” on page D-20](#)).
- **inline location:** Check if you want the string **_inl**, followed by the inlined call location (including the file and line number, as specified), to be added when inlining a function call. This sets the **name_use_inline_location** design attribute ([“name_use_inline_location” on page D-19](#)).

- **template parameters:** Check if you want the template parameters to be included in the names of modules, behaviors, and behavior terms. This is applicable when design uses user-defined template classes and/or template functions. This sets the **name_use_template_parameters** design attribute (“[name_use_template_parameters](#)” on page D-20).

Figure 6-23 Design Property Dialog - Incremental Tab



In the **Incremental** tab, you are setting up your design up for Incremental Synthesis:

Note See also “[Incremental Synthesis](#)” on page 17-1.

- **Auto Save Baseline:** Specifies the directory into which CtoS will save the baseline design at two places in the Incremental Synthesis flow: after the **build** command (in a subdirectory **elaborated**) and after registers have been allocated for every behavior in the design (in a subdirectory **rtl_ready**).

This sets the **auto_save_dir** design attribute (“[auto_save_dir](#)” on page D-11).

- **Compare to Baseline:** Specifies the name of the directory in which a design is to be saved during the baseline run in the Incremental Synthesis flow.

This sets the **baseline_dir** design attribute (“[baseline_dir](#)” on page D-12).

- **Match Threshold:** Specifies the minimum amount of matching between the baseline and a new design when in Incremental Synthesis mode. A low value usually signals an error, because an incremental design must be similar to the baseline. The default is 90, that is, 90%.

This sets the **required_match_percentage** design attribute (“[required_match_percentage](#)” on page D-22). If the new design does not meet the required match percentage, as compared with the baseline, CtoS will issue an error.

- **Sharing Policy for New Ops:** Specifies the sharing policy for new ops that are not in the baseline design. If set to **Reuse Existing Resources**, new ops can be shared on baseline resources, minimizing area. If set to **Create New Resources**, new resources are created for new ops and baseline resources are not shared, minimizing the eco difference. This policy applies to new ops only.

6.5 Building a Design

After your design has been set up, you can build it by selecting the *Build* icon (as shown in [Figure 6-24 on page 6-28](#)) or **Edit -> Build**.

During the build process, CtoS reads input files, elaborates them, and creates data structures representing the many modules and behaviors that make up a design in the CtoS database.

Figure 6-24 Build Icon



The following sections describe additional information about building a design:

- If your design has multiple source files, see [“Compiling Multi-Source Designs through Single Combined Source” on page 6-29](#)
- If you need to interrupt the **build** process for any reason, see [“Using the Interrupt Button” on page 6-32](#).
- If you get an internal error during the **build** process, see [“Resolving Build Internal Errors using Source Context” on page 6-33](#)
- If you get an error where more information about the source context would be helpful, see [“Using the -verbose Option of Build for Enhanced Source Context” on page 6-34](#)
- If you get a warning or error during the **build** process, you can start an external editor on the specified file, which will be scrolled to the appropriate line and column; see [“Using External Editors on SystemC Files” on page 6-36](#).

Notes

- CtoS preserves the **SC_MODULE** hierarchy by default; for more detail on hierarchical design, see [“Module Hierarchy” on page 16-1](#). You can also choose to flatten the design by setting the **build_flat** design attribute ([“build_flat” on page D-12](#)).
- If you are planning to perform incremental synthesis, see [“Incremental Synthesis” on page 17-1](#)
- You can also use the **build** command ([“build” on page E-30](#)).

6.5.1 Compiling Multi-Source Designs through Single Combined Source

Note *As of the 8.2.0 release, the processing mode corresponding to the **-combine_sources** option of the **build** command, introduced in the 8.1.2 release, is the only and fixed behavior. It is not possible to turn it off or to control the name or location of the combined source file.*

Designs consisting of multiple source files impose certain penalties and limitations on the efficiency, and even correctness, of the **build** process. In previous versions of CtoS, each source file was first parsed independently – with shared headers, most notably *systemc.h*, processed multiple times, and without cross-source type/function referencing. Some type/function cross-source referencing and error detection occurred later, but the elaborator was less precise, less efficient, and less exhaustive than parsing the front end. In comparing a CtoS **build** of a design consisting of several source files against a **build** of another version of the same design, but with only one source file, the following differences were noted:

- The version with multiple source files took significantly longer to elaborate.
- The version with multiple source files resulted in a significantly larger array footprint of CtoS.
- Link-type errors in the version with multiple source files were detected later in the **build** process than similar errors in the version with a single source file, resulting in messages harder to decipher.

For designs following standard C++ good coding practices, it should be possible to obtain a version of the design with a single source file from a version with multiple source files by simply including (using the **#include** directive) all source files in a single new source file.

This process has been automated and turned on permanently as of the 8.2.0 release and is described in the following sections:

- [“How Multi-Source Designs are Built” on page 6-29](#)
- [“Coding Requirements for Multi-Source Designs” on page 6-30](#)

6.5.1.1 How Multi-Source Designs are Built

When a design contains multiple source files, CtoS automatically creates a combined source file consisting of a series of lines of the following pattern:

```
#include "original_source_file.cpp"
```

The file has the following characteristics:

- The file is named **ctos_combined_source_PID.cpp**, where **PID** is the current process ID prefixed with a unifying lowercase letter, and the name is not user-modifiable.
- The file is created in the current directory.
- If a file with such a name already exists in the current directory, it is truncated and overwritten.

Then, CtoS elaborates the design from the new file **ctos_combined_source_PID.cpp** instead of elaborating each file specified in the **source_files** attribute of the design.

Important If the **source_files** attribute of the design consists of only one file, the **ctos_combined_source_PID.cpp** file is not created, and the design is elaborated from the original file.

If no errors are found during parsing, the file is deleted immediately after parsing is finished.

If, however, there are compilation errors, the file is preserved, and the front-end sign-off message points to the file as to a possible source of these errors.

If, after error analysis, it appears that the combined source file has caused the errors, you should carefully revise your design for possible bad coding practices, such as abusing preprocessor, defining a function with the same name in two source files, etc.

There is further discussion about this in the following section, [“Coding Requirements for Multi-Source Designs” on page 6-30](#).

This mechanism of combining sources does not affect location information reported in diagnostic messages or required by some CtoS commands; they will still correctly refer to and work with the original source files.

6.5.1.2 Coding Requirements for Multi-Source Designs

CtoS handles designs consisting of multiple source files differently from most C++ compilers, as described in the previous sections.

As a result, CtoS requires that designs consisting of multiple source files meet the following requirements:

- Any directly or indirectly included header files shared among several source files should almost certainly have header guards in them; otherwise, they may produce duplicate definitions, macro redefinitions, etc., resulting in compilation warnings and/or errors.
- There should be no conflicting function/class/enum/global variable/etc. definitions in the source files. Previous versions of CtoS allowed multiple function or method definitions in multiple source files.

For each source file with a definition, the definition was used whenever the function was called in the code.

For source files without a definition, one master definition was selected out of the available ones. This scheme will no longer work.

Function and method definitions must be unique across the source files; otherwise, a duplicate function definition error will be reported and elaboration will fail.

- There should be no directly conflicting preprocessor directives in the combined source: whenever there is a **#define SOME_MACRO** directive in a source, it either should not have been defined in any of the preceding sources or should have been properly undefined using **#undef** at some point.
- It is strongly recommended that you not alter contents of a header file shared among several source files by changing the source-to-source preprocessing directives that affect it. The source files, in effect, would certainly differ from what they would have been if compiled separately. This problem is non-detectable and will not be reported.

For example, in the following code, when compiled separately, **source1.cpp** and **source2.cpp** will effectively implement **xxx()** as **doThis(3)** and **yyy()** as **doThat(6)**. When compiled through a combined source, the second inclusion of **header.h** in **source2.cpp** will effectively not happen because of the header guards, so the implementation of **yyy()** will coincide with that of **xxx()**.

```
// header.h
#ifndef header_h
#define header_h

inline void zzz(int x)    // would produce 2 different definitions under
                        // CtoS multiple function definition resolution
{
    #if SWITCH == 1
        doThis(x);
    #else
        doThat(x);
    #endif
}

const int Const = GET_CONST(2);

#endif

-----

// source1.cpp
#define SWITCH 1
#define GET_CONST(v) v+1
#include "header.h"
void xxx()
{
    zzz(Const);
}

-----

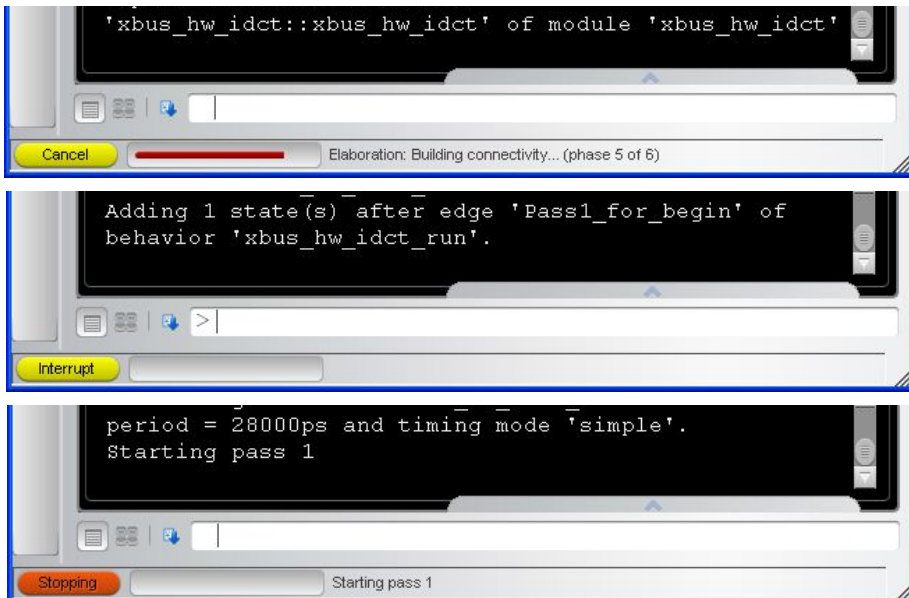
// source2.cpp
#define SWITCH 0
#define GET_CONST(v) v*3
#include "header.h"
void yyy()
{
    zzz(Const);
}
```

6.5.2 Using the Interrupt Button

The *Interrupt* button, as shown in [Figure 6-25 on page 6-32](#), lets you interrupt the long-running commands – **build**, **create_required_states**, **create_initial_resources**, and **schedule** – as well as the sourcing of a Tel script. It changes to *Cancel* for **build** and *Interrupt* for the **schedule**-related commands and Tel scripts. If you click this button while the command is still running, the button will change to *Stopping*, and the command will stop as soon as it can, as follows:

- For **build**, it stops fairly quickly and resets the design back to before **build**. You can rerun the **build** again (from the beginning) after interrupting it.
- For **schedule**, it stops at the end of the current pass. The design state is *Failed Schedule*, and you can view the failed ops in the CDFG viewer.
- For **create_initial_resources**, it stops at the end of processing the current behavior. This leaves the design in the state of having some behaviors with initial resources and others without.
- For **create_required_states**, it stops fairly quickly, leaving the states it has already created. You can run the **create_required_states** command again, basically continuing where it left off.
- For Tel scripts, CtoS checks before the start of each CtoS command in the Tel script to see if the interrupt button has been selected. If it has, a message is issued that interrupt has occurred, and processing of the rest of the script is stopped. The state of the design is valid because it is only stopping between commands, and you can continue to work with the design.

Figure 6-25 Interrupt Button



6.5.3 Resolving Build Internal Errors using Source Context

CtoS reports the *source context* for each failure of the **build** command, due to an internal error. This takes the form of a stack, where the topmost item is the one closest to the location of the problem. Each item consists of the following:

```
SOURCE_LOCATION: ELABORATION_PHASE CONSTRUCT_KIND.
```

where:

- **SOURCE_LOCATION** identifies the location of the construct in the source code.
- **ELABORATION_PHASE** tells what the elaborator was doing, for example, *“Building data flow.”*
- **CONSTRUCT_KIND** identifies the C++/SystemC construct, for example, *module p19, function void p19::p19().*

For example:

```
ERROR (CTOS-11112): [SystemC Elaborator] Internal error during elaboration of 'p19':
Abort in file sceTypeAnalyzer.cpp at line 1127
0x8b88160: _ZN9CallStack4initEv
0x8b88313: _ZN9CallStackC1Ev
0x8b50eca: _ZN5Error6formatERPc
0x8b514b7: _ZN5ErrorC1Ejz
...
0x8bd53ce: Tcl_Main
0x8053da5: main
0x55ede3: .
The source context in which this error occurred is as follows:
include/p19.h:21:14: Analyzing type for construct.
include/p19.h:21:14: Analyzing type for construct.
include/p19.h:23:13: Building declarations for construct.
include/p19.h:16:1: Building declarations for module p19.
p19_synth_wrapper.h:18:1: Building declarations for module p19_synth.
p19_synth_wrapper.cpp:16:15: Building declarations for module p19_synth.
Error in processing command build
```

The reason for not simply reporting the topmost item is that sometimes the source location information of this item is not useful.

For example, if the construct is part of a compiler-generated copy-constructor, looking down the stack may provide you with more relevant source locations for the problem.

6.5.4 Using the -verbose Option of Build for Enhanced Source Context

CtoS provides the **-verbose** option to get enhanced source context for **build** errors.

Note Because the additional source context may make it more difficult to spot the actual error message in the log, this is the optional, rather than the default, behavior.

In the following example, you can see how this option can be very useful:

Here is the default (non-verbose) output of **build**, for this example:

```
[SystemC Elaborator] Elaborating design in flat mode ...
/home/user/dev2/Install/share/ctos/include/systemc/IUS08.10-s003/lnx86
/all_systemc.h:16058:32: ERROR (CTOS-11289): [SystemC Elaborator]
    Unsupported construct of type 'pointer to function'
[SystemC Elaborator] Errors during elaboration of 'sc_main.rip_dut_main'
Error in processing command build
```

In the source reference (above in red), which is in the SystemC headers shipped with CtoS:

```
15962 class ios_base
15962 {
...
16058 ios_base::event_callback _M_fn;
```

This suggests that the design is using **ios_base**, but where? To answer this question, you can run the **build** command with the **-verbose** option, which will give you a much more detailed report, as follows:

```
[SystemC Elaborator] Elaborating design in flat mode ...
/home/user/dev2/Install/share/ctos/include/systemc/IUS08.10-s003/lnx86
/all_systemc.h:16058:32: ERROR (CTOS-11289): [SystemC Elaborator]
    Unsupported construct of type 'pointer to function'
The source context in which this error occurred is as follows:
/home/user/dev2/Install/share/ctos/include/systemc/IUS08.10-s003/lnx86/al
    l_systemc.h:16058:32: Analyzing type for variable _M_fn.
/home/user/dev2/Install/share/ctos/include/systemc/IUS08.10-s003/lnx86/al
    l_systemc.h:16058:32: Analyzing type for variable _M_fn.
/home/user/dev2/Install/share/ctos/include/systemc/IUS08.10-s003/lnx86/al
    l_systemc.h:16074:22: Analyzing type for variable _M_callbacks.
/home/user/dev2/Install/share/ctos/include/systemc/IUS08.10-s003/lnx86/al
    l_systemc.h:16074:22: Analyzing type for variable _M_callbacks.
```

```
/home/user/dev2/Install/share/ctos/include/systemc/IUS08.10-s003/lnx86/al
l_systemc.h:16181:5: Analyzing type for variable this.
/home/user/dev2/Install/share/ctos/include/systemc/IUS08.10-s003/lnx86/al
l_systemc.h:16181:5: Analyzing type for variable this.
/home/user/dev2/Install/share/ctos/include/systemc/IUS08.10-s003/lnx86/al
l_systemc.h:16181:5: Building declarations for variable this.
src/rip_dut_proc_alg.h:1407:10: Building declarations for construct.
src/rip_dut_proc_alg.h:871:4: Building declarations for construct.
src/rip_dut_proc_alg.h:393:3: Building declarations for process
    rip_proc_alg<8, 512, 7, 15, 4, 1088, 32, 16, 5, 11, 4, 3, 98, 17, 2, 64,
    16, 12, 4, 7, 64, 7, 12, matrix_log_template::matrix_log<1088, 6,
    sc_uint<32> >, 0, 230, 8, 5, 3, 5>::void rip_proc_alg<8, 512, 7, 15, 4,
    1088, 32, 16, 5, 11, 4, 3, 98, 17, 2, 64, 16, 12, 4, 7, 64, 7, 12,
    matrix_log_template::matrix_log<1088, 6, sc_uint<32> >, 0, 230, 8, 5,
    3, 5>::pa_main_alg_vert().
src/rip_dut_proc_alg.h:68:9: Building declarations for module
    rip_proc_alg<8, 512, 7, 15, 4, 1088, 32, 16, 5, 11, 4, 3, 98, 17, 2, 64,
    16, 12, 4, 7, 64, 7, 12, matrix_log_template::matrix_log<1088, 6,
    sc_uint<32> >, 0, 230, 8, 5, 3, 5>.
src/rip_dut_main.h:110:1: Building declarations for module rip_dut_main.
src/rip_dut_main.cc:279:1: Building declarations for module rip_dut_main.
[SystemC Elaborator] Errors during elaboration of 'sc_main.rip_dut_main'
Error in processing command build
```

You still get the same error, but with a much more useful reference to the design source:

```
1406 for (int c = 0; c<PA_RAM_PIX_WIDTH/2; c++){
1407     cout.width(3);
1408     cout<<c;
```

You can now see that the problem is CtoS was unable to ignore some debug code. The workaround would be to exclude this code when seen by CtoS, as follows:

```
1406 for (int c = 0; c<PA_RAM_PIX_WIDTH/2; c++){
1407     #ifndef __CTOS__
1408         cout.width(3);
1409         cout<<c;
1410     #endif
```

6.5.5 Using External Editors on SystemC Files

CtoS lets you edit your SystemC source files using an external editor. When you get a warning or error during **build**, you can start an external editor on the specified file, which will be scrolled to the appropriate line and column.

- [“Starting an External Editor” on page 6-36](#)
- [“Notes about Individual Editors” on page 6-36](#)
- [“Troubleshooting External Editors” on page 6-37](#)

6.5.5.1 Starting an External Editor

To use an external editor on a SystemC file, simply hover over the warning or error message to see a description of it, then right-click and select **Edit File**.

- If the build failed, **Edit File** is the default, so you can double-click the message to start the editor.
- If the build was successful, **Go to Location** in the **Input Source** viewer is the default.

In the **Input Source** viewer, itself, the pop-up menu item, **Edit File**, starts the editor on the current file.

CtoS decides which editor to start by looking at:

- \$VISUAL
- \$EDITOR environment variable
- (if neither is applicable) defaults to **vim**.

Since each of the editors has slightly different capabilities, CtoS has implemented a set of them with the appropriate command line arguments.

Your editor can be specified as full path to the editor executable or simply the base name, and the \$PATH will be used to find it. [See the note in [“Notes about Individual Editors” on page 6-36](#) on running the Nirvana Editor client (**nc**).]

6.5.5.2 Notes about Individual Editors

Here are a few notes about some specific editors:

- **emacsclient**

emacs is the default server editor, but can be overwritten by setting the environment variable, **ALTERNATIVE_EDITOR**. So **emacs** is seen as the server, add this to the **.emacs** file:

```
(server-start)
```

- **gnuclient**

You must start **xemacs** manually before invoking the editor from CtoS: otherwise, you will get the following messages in an xterm:

```
gnuclient: Connection refused
gnuclient: unable to connect to local
```

In order for **xemacs** to be seen as the server, add the following to the **.xemacs** file:

```
(load "gnuserv") (gnuserv-start)
```

In order to reuse the same frame, add the following to the **.xemacs** file:

```
(setq gnuserv-frame (selected-frame))
```

A **gnuclient** is active until you finish editing the buffer. Press C-x # to finish editing the buffer. If you don't finish the buffer, then when quitting **xemacs**, you will get the following message:

```
Gnuserv buffers still have clients: exit anyway?
```

It is safe to say **Yes**. CtoS is not listening to the client.

- **gedit**

gedit reuses the existing editor, but doesn't recognize when a file is already open in buffer and opens another buffer.

- **nc** (Client for Nirvana Editor)

nc reuses the existing editor, but opens another window frame in version 5.3. Version 5.5 supports tabs.

If you start **nedit** manually and want it to be reused by CtoS, then run it as:

```
: nedit -server
```

If you get the following error, make sure **/usr/X11R6/bin** is before **/usr/bin** in **\$PATH** because there is another program named **nc** (**netcat**):

```
invalid interval time ne
```

6.5.5.3 Troubleshooting External Editors

If you are having trouble starting an external editor, here are some things to try:

- Review “[Notes about Individual Editors](#)” on page 6-36 to see if there could be a problem with the editor.
- Look at the xterm in which **ctosgui** was started, and you might see more error messages.
- Make sure your editor can be started from another xterm. If there are problems, try setting the variable to the full path to where the executable resides.

```
Invoke $VISUAL
Invoke $EDITOR
```

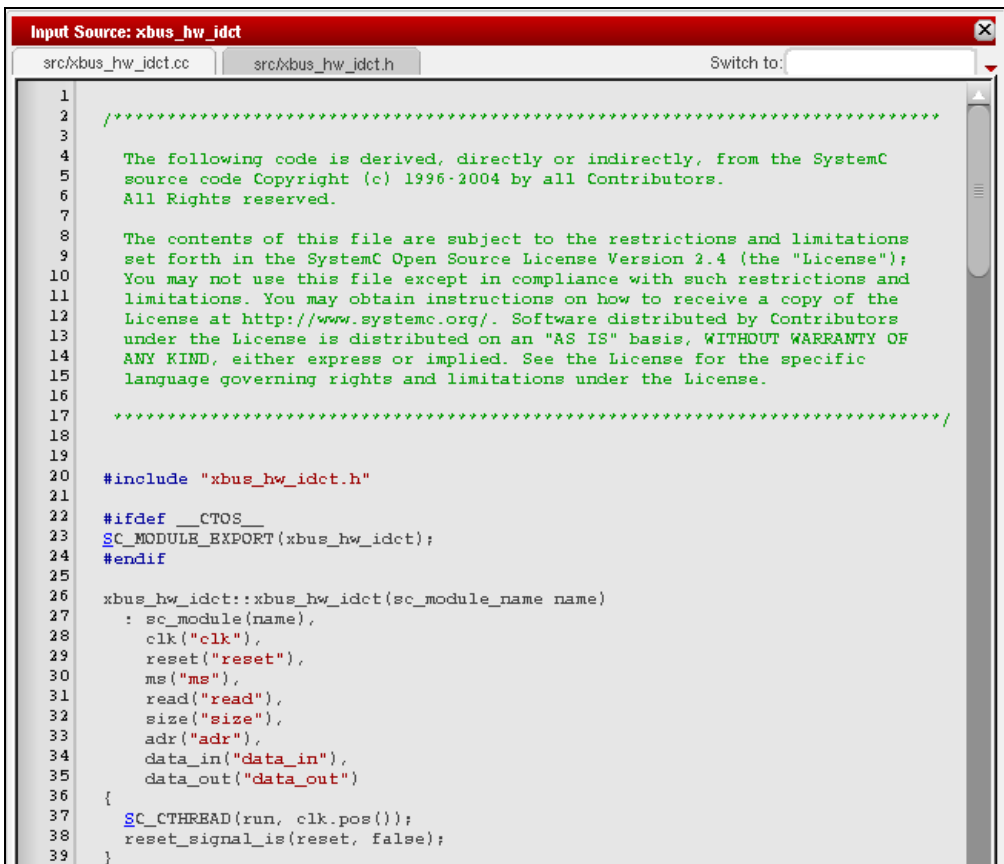
- Make sure the command line is properly constructed by CtoS by looking at the info messages in the CtoS output window, which is the actual command line being invoked.

6.6 Viewing Input Source

After the build process is complete for a design, CtoS will automatically launch the **Input Source** viewer, as shown in [Figure 6-26 on page 6-38](#). This viewer displays multiple text files according to each one's purpose. For example, multiple files may be used as input sources, and you may group them.

You can also launch the **Input Source** viewer by right clicking on an object and selecting **Show Input Source**, or by selecting **View->Input Source**.

Figure 6-26 Input Source viewer



The screenshot shows a window titled "Input Source: xbus_hw_idct". It has a tab bar with two tabs: "src/xbus_hw_idct.cc" and "src/xbus_hw_idct.h", with the second tab selected. A "Switch to:" dropdown menu is visible. The main area displays the C++ source code for "xbus_hw_idct.h" with syntax highlighting. The code includes a license notice and a module definition. Line numbers 1 through 39 are visible on the left margin.

```
1
2  /*****
3
4      The following code is derived, directly or indirectly, from the SystemC
5      source code Copyright (c) 1996-2004 by all Contributors.
6      All Rights reserved.
7
8      The contents of this file are subject to the restrictions and limitations
9      set forth in the SystemC Open Source License Version 2.4 (the "License");
10     You may not use this file except in compliance with such restrictions and
11     limitations. You may obtain instructions on how to receive a copy of the
12     License at http://www.systemc.org/. Software distributed by Contributors
13     under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF
14     ANY KIND, either express or implied. See the License for the specific
15     language governing rights and limitations under the License.
16
17     *****/
18
19
20     #include "xbus_hw_idct.h"
21
22     #ifdef __CTOS__
23     SC_MODULE_EXPORT(xbus_hw_idct);
24     #endif
25
26     xbus_hw_idct::xbus_hw_idct(sc_module_name name)
27     : sc_module(name),
28       clk("clk"),
29       reset("reset"),
30       ms("ms"),
31       read("read"),
32       size("size"),
33       adr("adr"),
34       data_in("data_in"),
35       data_out("data_out")
36     {
37         SC_CTHREAD(run, clk.pos());
38         reset_signal_is(reset, false);
39     }
```

Here are some of the many features of this viewer:

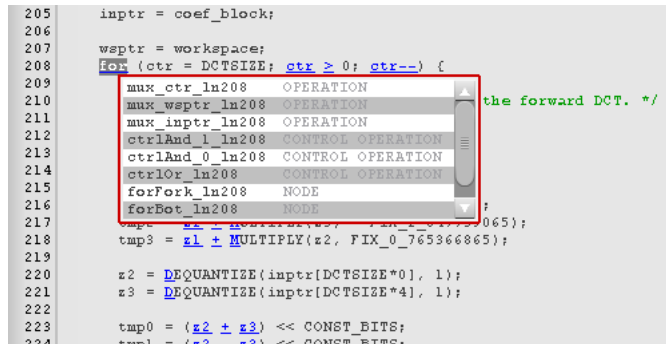
- "Syntax-Highlighting" on page 6-39
- "Context Menus" on page 6-40
- "Cross-Highlighting with Markers" on page 6-41

6.6.1 Syntax-Highlighting

The **Input Source** viewer is fully syntax-highlighted (in C++). All source links are highlighted in the source, with the following color-coding:

- **dark blue** – C++ preprocessor directives (for example, **#include**, **#ifdef**, **#define**, and so on)
- **red** – string literals (strings in quotes, such as names of instances)
- **green** – comments
- **black** – C++ keywords (for example, **while**, **for**, **if**, **class**, and so on)
- **blue** – links
- **dark grey background with white letters** – selected objects
- **magenta background with white letters** – markers

To explore the links at a given location, simply hover over the text of interest. If it is underlined, a bubble appears with a list of all links available. The bubble gives you not only the name, but the type of object represented by the link, as well.

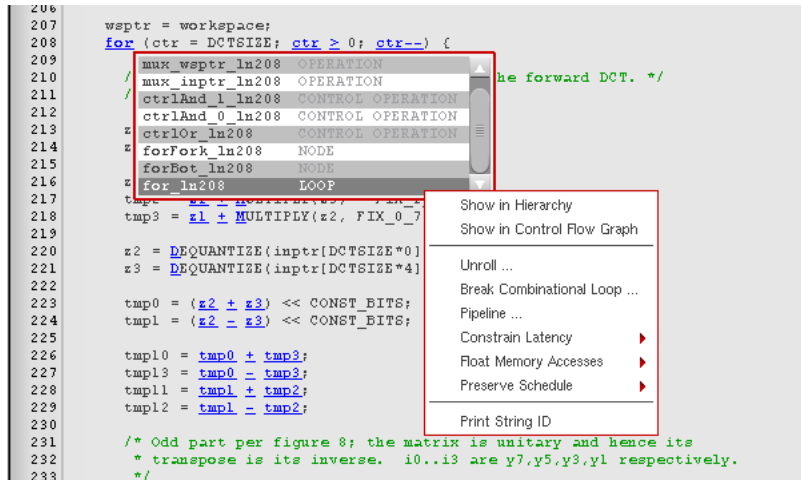


If you hover over a block of code that was previously selected, the bubble also lists all links associated with the selected code.

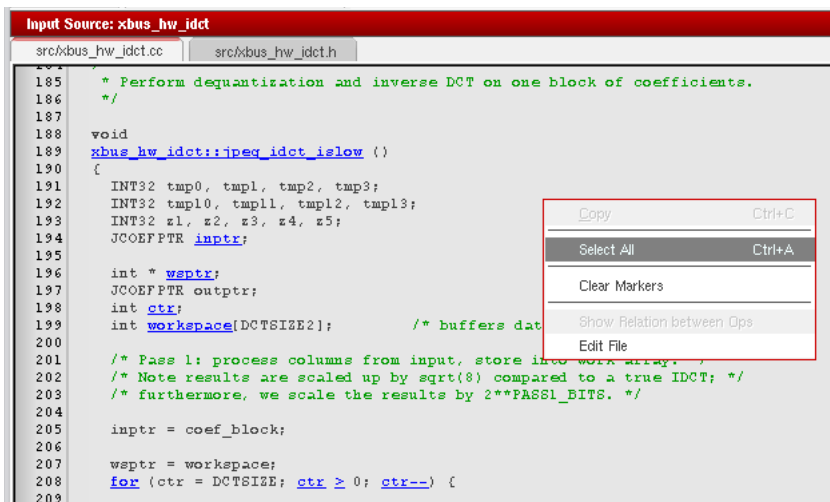
In addition to simple hovering, you can also drag-select a block of code. In this case, every link within the bounds of the selection will be listed in the bubble.

6.6.2 Context Menus

Right-clicking on any entry in the link list brings up the context menu for that object, just like all other areas of the CtoS GUI. A disabled entry means no context menu is available.



If you right-click on the **Input Source** viewer directly, you get a context menu for the viewer, itself.



6.6.3 Cross-Highlighting with Markers

The source is cross-highlighted in the **Input Source** viewer. From the **Hierarchy Window**, if you try to show, for example, **for_In208** in the input source, the matched object will be highlighted with a *marker* in the **Input Source** viewer.

Note For more about the **Hierarchy Window**, see “[Hierarchy Window](#)” on page 6-44.

```
201  /* Pass 1: process columns from input, store into work array. */
202  /* Note results are scaled up by sqrt(8) compared to a true IDCT; */
203  /* furthermore, we scale the results by 2**PASS1_BITS. */
204
205  inptr = coef_block;
206
207  wsptr = workspace;
208  for (ctr = DCTSIZE; ctr > 0; ctr--) {
209
210      /* Even part: reverse the even part of the forward DCT. */
211      /* The rotator is sqrt(2)*c(-6). */
212
213      z2 = DEQUANTIZE(inptr[DCTSIZE*2], 1);
214      z3 = DEQUANTIZE(inptr[DCTSIZE*6], 1);
```

A marker does not automatically go away when you click somewhere else in the viewer; the advantage of this, over simple selection, is it can be used to highlight non-contiguous results (for example, showing all array accesses in the source).

To clear markers, right-click anywhere in the viewer, and select **Clear Markers** from the context menu.

6.7 Using the CDFG Viewer

The CDFG (control and data flow graph) viewer shows both the control flow and the data flow – nodes, loops, edges, ops, and data dependencies – for your design. This viewer is displayed by selecting **View -> CDFG**, or by selecting **Show in CDFG** when you right-click on an object in the design.

Edges with ops are indicated with a bluish green triangle glyph and can be expanded to reveal the data flow among the ops, as shown in [Figure 6-27 on page 6-42](#).

To reduce visual clutter (and for performance), data flows that go across edge boundaries are not shown, but can be explored using the Predecessors, Successors, Fan-in, and Fan-out context menus, as shown in [Figure 6-28 on page 6-42](#).

Figure 6-27 CDFG Viewer - Expanding/Collapsing Loops

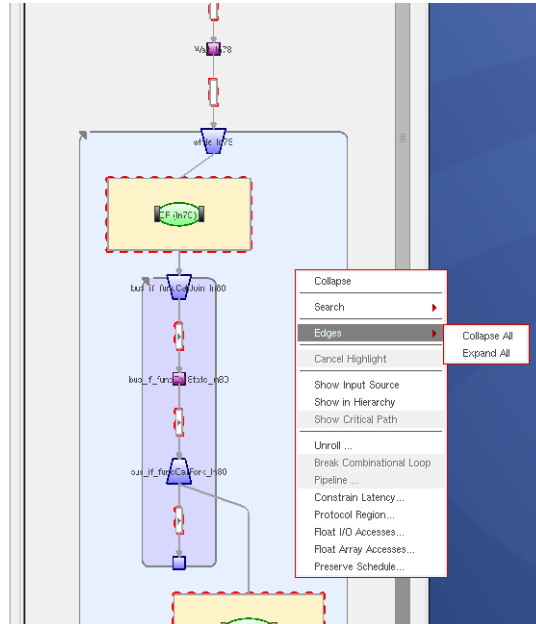
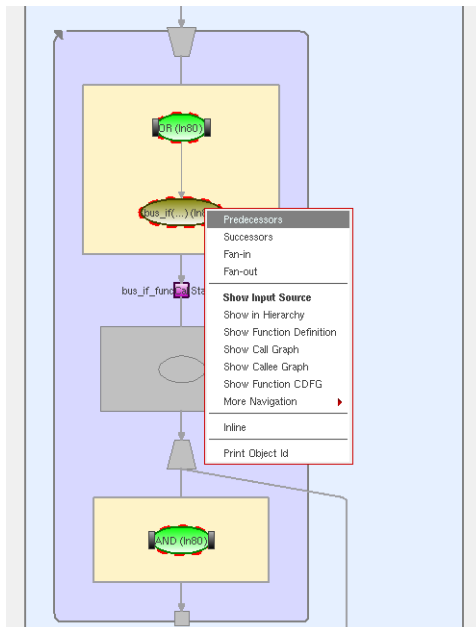


Figure 6-28 CDFG Viewer - Exploring Data Flows



6.8 Navigating the CtoS Environment

The following features have been designed to help you navigate in the CtoS environment:

- “Task Window” on page 6-43
- “Hierarchy Window” on page 6-44
- “Navigating the Virtual Design Directory” on page 6-47
- “Getting Object IDs” on page 6-47
- “Setting the Current Design” on page 6-48
- “Getting Help on Commands and Messages” on page 6-48
- “Redirecting Output” on page 6-48
- “Rerunning Previously Run Commands”

6.8.1 Task Window

The **Task Window**, as shown in [Figure 6-29 on page 6-44](#), pares down your CtoS design flow to the essential steps. Additionally, the steps that *must* be addressed before you can proceed to scheduling are indicated by an exclamation mark in a yellow call-out.

This is designed to be a straightforward way to get a design through the CtoS process, especially a first design. Therefore, advanced users may find the **Hierarchy Window** (described in the next section, [“Hierarchy Window” on page 6-44](#)) more helpful, as it displays *all* of your design objects, while the **Task Window** displays *only* those you *must* modify.

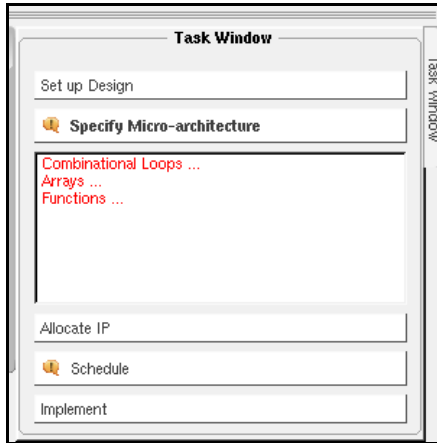
The steps in the **Task Window** contain the following potential tasks:

Tip You do not always have to follow a specific order within these steps; in fact, you can achieve vastly different results by varying the order of inlining of functions and loop unrolling.

- **Set up Design**
 - **Designs**
- **Specify Micro-architecture**
 - **Combinational Loops**
 - **Arrays**
 - **Functions**
- **Allocate IP**
 - **Arrays**
- **Schedule**
- **Implement**

- Write RTL
- Write Verification Wrapper

Figure 6-29 Task Window



6.8.2 Hierarchy Window

The **Hierarchy Window**, as shown in [Figure 6-30 on page 6-45](#) and [Figure 6-31 on page 6-46](#), displays the content of your design's input source as design data, after you have built the design. This data is abstracted from the internal CtoS representation to user-recognizable items, such as loops, function calls, states, labels, and arrays.

The **Hierarchy Window** displays *all* of your design objects; the **Task Window** (described in the preceding section, "[Task Window](#)" on page 6-43), displays *only* those you *must* modify.

In the **Hierarchy Window**, as in the **Task Window**, if an object is shown in **red**, you *must* set constraints for that object before you can perform scheduling. For example, combinational loops must be eliminated, because they cannot be implemented in hardware; similarly, you must specify how arrays should be implemented.

There are two modes for the **Hierarchy Window** (you can switch between them using the yellow arrow at the top right of the window):

- **Traversal Mode**, as shown in [Figure 6-30 on page 6-45](#), displays *all* of your design objects, in a traditional, hierarchical fashion.
- **Identification Mode**, as shown in [Figure 6-31 on page 6-46](#), displays only those items you have selected using **Show in Hierarchy**. This better describes how *particular* objects were transformed (unrolled or inlined). It is in tree format, where each level in the tree represents a transform. The **X** button at the top right clears all items from the list (this button is disabled in **Traversal Mode**).

Figure 6-30 Hierarchy Window - Traversal Mode

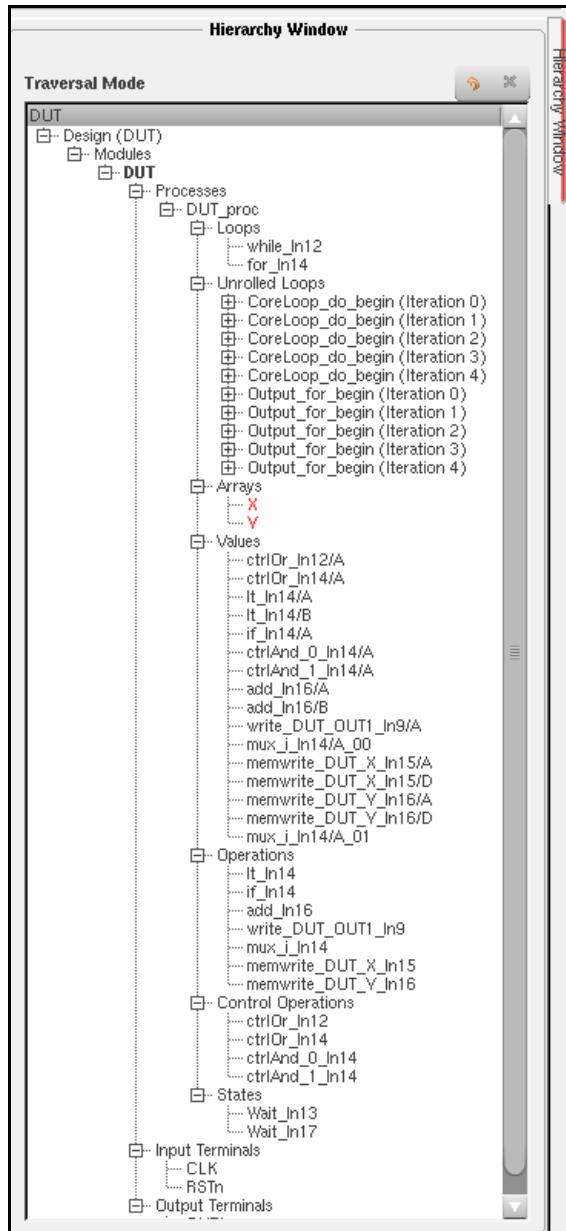
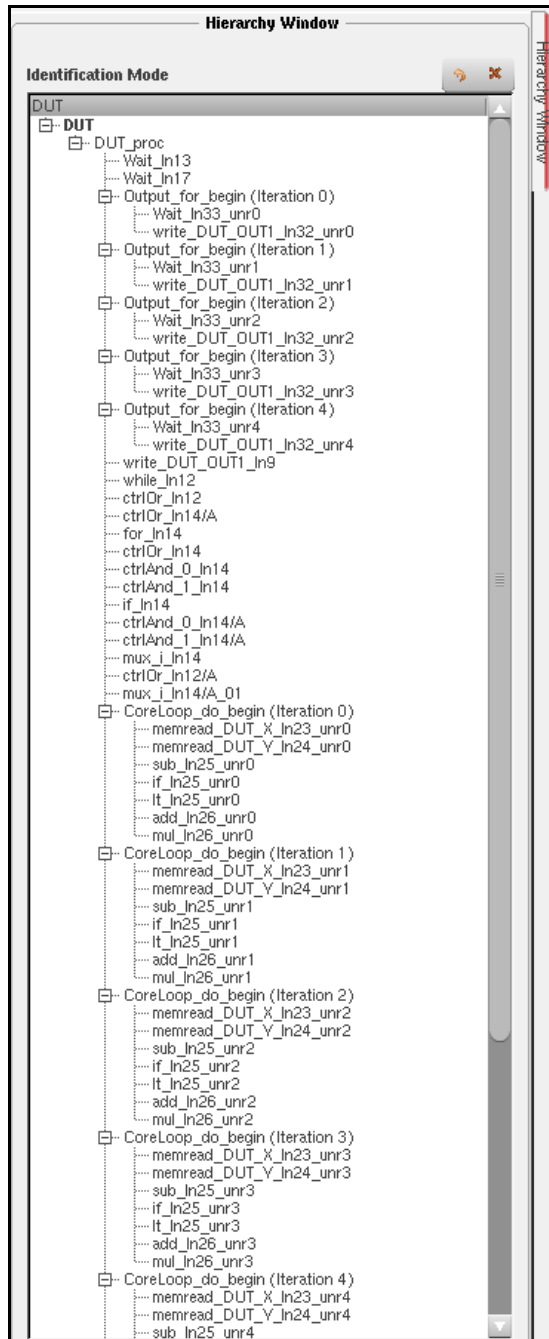


Figure 6-31 Hierarchy Window - Identification Mode



6.8.3 Navigating the Virtual Design Directory

After a design is built, CtoS provides a virtual directory of the design and supports three navigation commands: **ls** (“ls” on page E-85), **cd** (“cd” on page E-31), and **pwd** (“pwd” on page E-96).

- To set the current scope for identifying objects by a relative object ID:

```
cd object_path
```

The *object_path* is the object path to be used as the current scope.

The returned value is the current scope.

- To list the subscopes of the specified scope:

```
ls [object_path]
```

The optional *object_path* is the scope whose subscopes are listed.

If you do not specify an *object_path*, the default is the current scope.

The returned value is a list of subscopes of the specified scope.

- To show the current scope:

```
pwd
```

Notes and Limitations

- For objects with *multiple path-RegBinding and ResBinding?*, the native object path is returned.
- To navigate in the Linux directory structure from within CtoS, use these same commands, preceded by an **I**, that is, **Ils** (“Ils” on page E-83), **Icd** (“Icd” on page E-81), and **Ipwd** (“Ipwd” on page E-84).
- See “CtoS Object Reference” on page D-1 for a complete tabular listing of all of the objects in the design database.

6.8.4 Getting Object IDs

To get the object IDs for objects in your design, in the CtoS GUI, you can simply right-click on the object, in most of the CtoS GUI windows and viewers, and select **Print Object Id**.

The Identification Mode of the **Hierarchy Window** (“Hierarchy Window” on page 6-44) can be very helpful to identify the correct op from a whole set of ops, all associated with the same source code line.

Notes

- You can also use the **get_design** command (“get_design” on page E-74) to get the object ID of the current design.
- For more about all aspects of the CtoS object hierarchy, see “CtoS Object Reference” on page D-1.

6.8.5 Setting the Current Design

The **set_design** command (“[set_design](#)” on page E-143) lets you set the *current* design to any of your open designs. If you have a current design when you use this command, that design will remain present in the session, but will no longer be the current design.

6.8.6 Finding Objects in the CtoS Virtual Directory

To find objects in the CtoS virtual directory, you can use the many search and find tools in the CtoS GUI (see “[Learning the Basics of the CtoS GUI](#)” on page 6-5. You can also use the **find** command (“[find](#)” on page E-60) to find objects that match the arguments you supply.

6.8.7 Getting Help on Commands and Messages

You can use the **help** command (“[help](#)” on page E-77) to get information about CtoS commands and error messages and to be directed to specific parts of this *CtoS User Guide* and *CtoS Quick Start Guide*.

To view the *CtoS Quick Start Guide*, select **Help > Quick Start Guide** in the CtoS GUI or specify the **-quick_start** option to the **help** command. For example **help -quick_start**.

To view the *CtoS User Guide*, select **Help > User Guide** in the CtoS GUI.

To get a list of all CtoS commands, use the **help** command with no arguments. Additionally, to get complete documentation of all commands, select **Help > Commands** in the CtoS GUI. This displays the [Appendix E “CtoS Command Reference”](#) of this *CtoS User Guide*.

To know the syntax of a specific command with a brief description of each argument, type **help <commandName>**. For example **help schedule** prints the information about the **schedule** command. You can get the same information by typing the CtoS command with the **-h** option; for example, **schedule -h**. For the complete documentation on any command, specify the **-doc** option to the **help** command. For example typing **help -doc schedule** displays the description of the *schedule* command in the CtoS User Guide.

To get extended help on an error message, type **help <msg #>**; for example, type **help 20500**. The extended help provides more details of the error and suggestions of the possible corrective actions.

To see your particular version of CtoS, select **Help > About** in the CtoS GUI. Three numbers are displayed: the major release number, the minor release number, and the bug fix release number. You can also use the **get_version** command (“[get_version](#)” on page E-76).

6.8.8 Redirecting Output

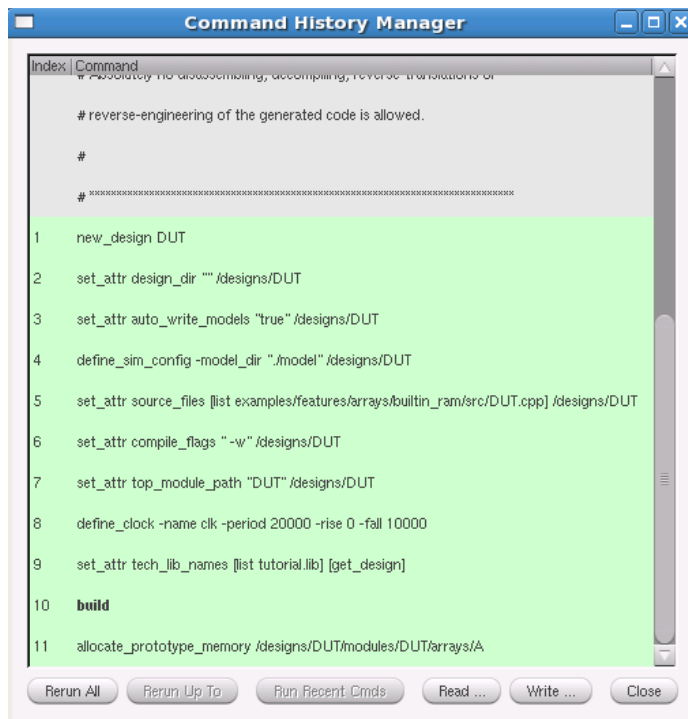
You can *redirect* the output of any command using **>** for informational output and **>&** for all output, including errors and warning flags. The redirect operator must be followed by a filename.

6.8.9 Rerunning Previously Run Commands

CtoS lets users to undo the last command that was performed, by automatically recording and rerunning the top-level TCL commands applied to a design. CtoS reruns all the TCL commands that you have performed until before the last command, which is the same as undoing the previous command that was performed. To undo the last run command, click **Edit > Undo (Rerun)**. The Task window will be refreshed and the CtoS design flow will show until the last command that you have performed.

Additionally, the **Command History Manager** dialog lists all the commands that you have performed, and lets you to rerun all the commands, rerun to selected row, run commands from file, and write history to file. To view the **Command History Manager** dialog, as shown in [Figure 6-32 on page 6-49](#), click **File > Command History**.

Figure 6-32 Command History Manager Dialog



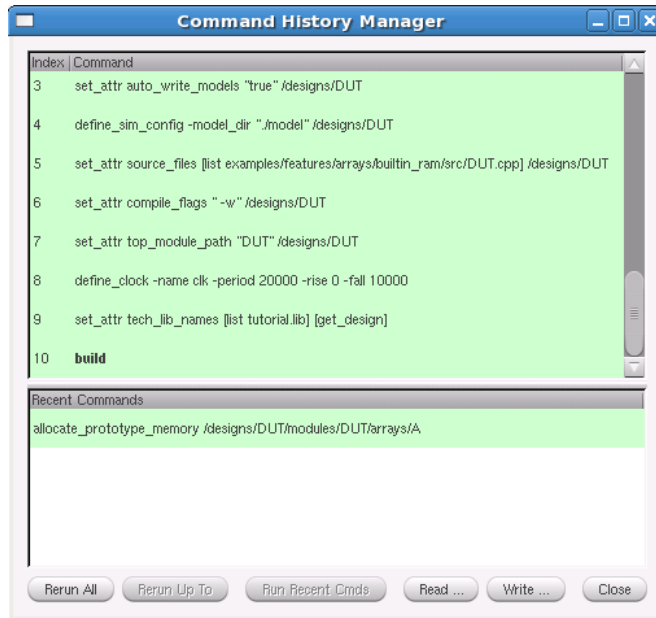
In the **Command History Manager** dialog, you can perform the following:

- **Rerun All:** CtoS reruns all the commands that are listed in the **Command** section.

- **Rerun Up To:** To rerun up to a selected command, select the command up to which you want rerun the commands. This enables the **Rerun Up To** button. Then, click **Rerun Up To**. The commands are rerun, and the commands that are were omitted in the rerun are moved down to the **Recent Commands** section, as shown in [Figure 6-33 on page 6-50](#).

For example, if you select the **Build** command, CtoS reruns all the commands up to the **Build** command, and the commands after the **Build** Command, are moved down to the **Recent Commands** section.

Figure 6-33 Command History Manager - Rerun Up To



- **Run Recent Cmds:** To rerun the commands that are listed in the **Recent Commands** section, select a command in the **Recent Commands** section. This enables the **Rerun Recent Cmds** button. Then, click **Run Recent Cmds**. CtoS reruns the specified command, and this command is moved up in to the **Command** section, and the other commands, if any, in the **Recent Commands** section remains as is.

Note If CtoS is unable to rerun any command that you selected or unable to read from file, then that command is displayed in red. And, the user is prompted with dialog explaining the reason for failure.

- **Read:** To run commands from a TCL file, click **Read**. In the **Browse TCL Command Files** dialog, select the TCL file that you want to use, and click **Open**. CtoS reruns the commands that are listed in the TCL file, and the **Command** section is refreshed with list of commands that were rerun from the TCL file.

- **Write:** To save/write the commands listed in the **Commands** section to a TCL file, click Write. In the **Browse TCL Command Files** dialog, browse to the location where you want to save the file, and click **Open**.

6.9 Saving, Opening, and Closing Designs

CtoS lets you *save* and *open* designs any time during the design flow. This is a convenient feature that lets you effectively explore the design space, as well as perform an Incremental Synthesis design flow.

Important When a design is saved, CtoS *does not* save the source code, but *does* save the parse tree from which you could derive the source code. It *does not* save the **.lib** files, but *does* save the *path* to the **.lib** files (however, this is not very useful without the **.lib** files). It *does not* save the RC cache, but *does* save timing data. Additionally, CtoS *does not* save the Tcl interpreter (and thus *does not* save Tcl variables, which are in the interpreter) – it saves only the CtoS design database.

The following subsections are included in this section:

- “Uses for Save/Open” on page 6-51
- “Saving a Design” on page 6-52
- “Opening a Design” on page 6-53
- “Closing a Design” on page 6-53

6.9.1 Uses for Save/Open

There are two primary uses for the save/open feature:

- “Incremental Synthesis Flow” on page 6-51
- “Design Space Exploration” on page 6-52

6.9.1.1 Incremental Synthesis Flow

Another typical use of the save and open features is in an *Incremental Synthesis flow*.

Suppose you need to change part of the SystemC description because of an ECO (engineering change order) *after* CtoS has synthesized a design and produced an implementation. If you had saved the original design at the end of scheduling, you could run CtoS in Incremental Synthesis mode.

In this mode, CtoS compares the original design with the new design and synthesizes the new design to reduce, as much as possible, the differences between the new implementation and the original implementation.

In this way, it becomes easier to analyze what has changed in the resulting implementation because of the changes introduced in the input description, which facilitates the scheduling and analysis in CtoS for the new design and also simplifies the verification tasks of the resulting implementation.

Note See “[Incremental Synthesis](#)” on page 17-1 for more information on this design flow.

6.9.1.2 Design Space Exploration

The SystemC description of the behavior of your design contains many objects that can be implemented in multiple ways. For example, an array in the description may be implemented as a set of individual registers, as a register file, or as a RAM provided by a RAM vendor. A loop may be implemented as a pipeline or with various timing constraints on its latency.

Each of these choices results in an implementation with differing quality, and it is not always obvious at the beginning of scheduling which choices (or which combinations of choices over all such objects) may result in a desired implementation quality.

By using the *save* and *open* features, you can save a design just before you set a constraint (when you are specifying micro-architecture) to indicate a particular implementation choice. You can then continue with scheduling and analyze the resulting implementation.

If you want to try another implementation choice on some of the objects, you can then open the saved design. This lets you continue scheduling from the point where the design was saved; therefore, you can set another constraint to indicate a different implementation choice you want to explore and again continue to scheduling.

Note See “[Micro-Architectural Exploration Example](#)” on page C-1 for more about this topic.

6.9.2 Saving a Design

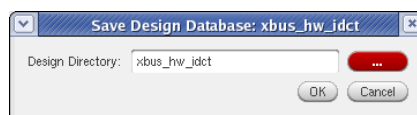
CtoS lets you save a design, using **File -> Save Design** or **File -> Save Design As**, at any point during a CtoS session. All necessary design information is saved in such a way that if you later open the saved design, you can continue with your CtoS design flow from this saved point.

For **Save Design**, CtoS saves the design information in the directory specified by **Save Directory** in the “[Create New Design Wizard \(Page 6\)](#)” on page 6-18.

For **Save Design As**, you can specify a different directory in the **Save Design Database** dialog, as shown in “[Save Design Database Dialog](#)” on page 6-52. For either, if a design with the same name exists in the specified directory, it will be overwritten.

Note You could also use the `save_design` command (“[save_design](#)” on page E-137).

Figure 6-34 Save Design Database Dialog

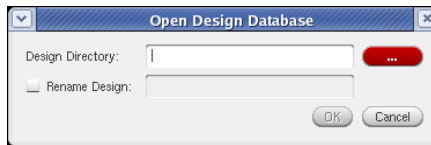


6.9.3 Opening a Design

To open a design, you simply select **File -> Open Design**, specifying the name of the directory in which you saved the design you want to open in the “[Open Design Database Dialog](#)” on page 6-53. You can optionally rename the design in this dialog, as well.

Note You could also use the **open_design** command (“[open_design](#)” on page E-89).

Figure 6-35 Open Design Database Dialog



6.9.4 Closing a Design

To close a design, you simply select **File -> Close Design**, which displays the **Close Design Database** dialog, as shown in “[Close Design Database Dialog](#)” on page 6-53.

Note You could also use the **close_design** command (“[close_design](#)” on page E-33).

Figure 6-36 Close Design Database Dialog



7 Verifying Designs

CtoS has been specifically designed to actively support the design verification process.

The individual verification features within CtoS are presented in this chapter, along with an introduction to the recommended flow.

Simulate and Exercise Designs as much as Possible

The most basic approach to design verification is to simulate and exercise a design as extensively as possible. Applying this to the space of high-level synthesis implies that you should simulate as much as possible at the highest level of abstraction (simulation speed) and then extend and reuse your simulation setup toward lower levels of abstraction.

You should always verify, before synthesis, that your SystemC design operates according to specifications, to avoid debugging more complicated lower-level models.

To debug and investigate potential issues, CtoS allows the creation of different *simulation models* throughout the synthesis process.

CtoS Simulation Models

In addition to the *Verilog RTL model*, which is the eventual output of the synthesis process, you can generate a *Verilog behavioral simulation model* at any time. By default, CtoS automatically creates simulation models after the build process and after scheduling a design.

CtoS Wrappers and Makefiles

To facilitate the verification process for all of these different models, CtoS provides *verification wrappers* and the automatic generation of a *Makefile*, which can be used to exercise and validate the models (including the original design). Following this flow lets you reuse the same testbench across all models generated by CtoS.

What's Included in this Chapter

This chapter discusses how to generate the models, wrappers, and Makefiles *after a successful build*.

It also provides a general guideline for writing a testbench.

In addition, customizations to the testbench and the process of debugging simulation mismatches are also detailed in this chapter.

Note For more information about the generation of models, wrappers, etc., *after the successful generation of a schedule and the allocation of registers*, see [“Generating Models, Wrappers, RTL, SLEC after Scheduling” on page 13-36](#).

The following topics are discussed in this chapter:

- [“Writing a Testbench” on page 7-3](#)
- [“Generating Models” on page 7-7](#)
- [“Generating Wrappers” on page 7-11](#)
- [“Modifying a Testbench” on page 7-15](#)
- [“Defining a Simulation Configuration and Generating Simulation Makefiles” on page 7-17](#)
- [“Simulating from the CtoS Environment” on page 7-22](#)
- [“Debugging Simulation Mismatches” on page 7-25](#)
- [“Renaming States in Support of FSM Coverage” on page 7-25](#)

7.1 Writing a Testbench

CtoS does not impose restrictions on the structure of a testbench. In practice, many styles of testbench are used in the CtoS flow.

Universal Verification Methodology (UVM), a methodology standardized in the industry, enables thorough verification with constrained-random test pattern generation with reusable testbench structures. This methodology is often applied to designs synthesized in the CtoS flow.

It is equally popular to use *directed test patterns* to verify individual design units synthesized by CtoS. In this method, the testbench is often simpler than the UVM testbench and is written in SystemC.

Regardless of the method used, however, you can make a testbench more reusable across different levels of abstraction and across different design projects if you consider certain aspects when writing the testbench. In the following sections, guidelines for writing good SystemC testbenches are provided:

- “Writing Sequencer and Monitor Function Modules” on page 7-3
- “Understanding Testbench Module Hierarchy” on page 7-4
- “Using the `sc_main` Function in a Testbench” on page 7-5

An example of a testbench developed based on these guidelines can be found in:

```
install_directory/share/ctos/examples/libraries/flex_channels/may_block
```

Note Before running any CtoS examples, first review “Setup for Examples” on page F-2.

7.1.1 Writing Sequencer and Monitor Function Modules

A testbench implements two primary functions:

- the *sequencer* function for producing test patterns to the design model, and
- the *monitor* function for observing the behavior of the design to check the correctness.

It then involves interfaces with the design model, as well as among internal components within the testbench.

It is beneficial to write the two primary functions in separate modules. It enhances the reusability of the code written for each function, because unnecessary modifications in the code for one function can be avoided when changes must be introduced in the other function. It also allows these functions to be written concurrently, so each module interacts with the design model independently and synchronizes with each other only as needed.

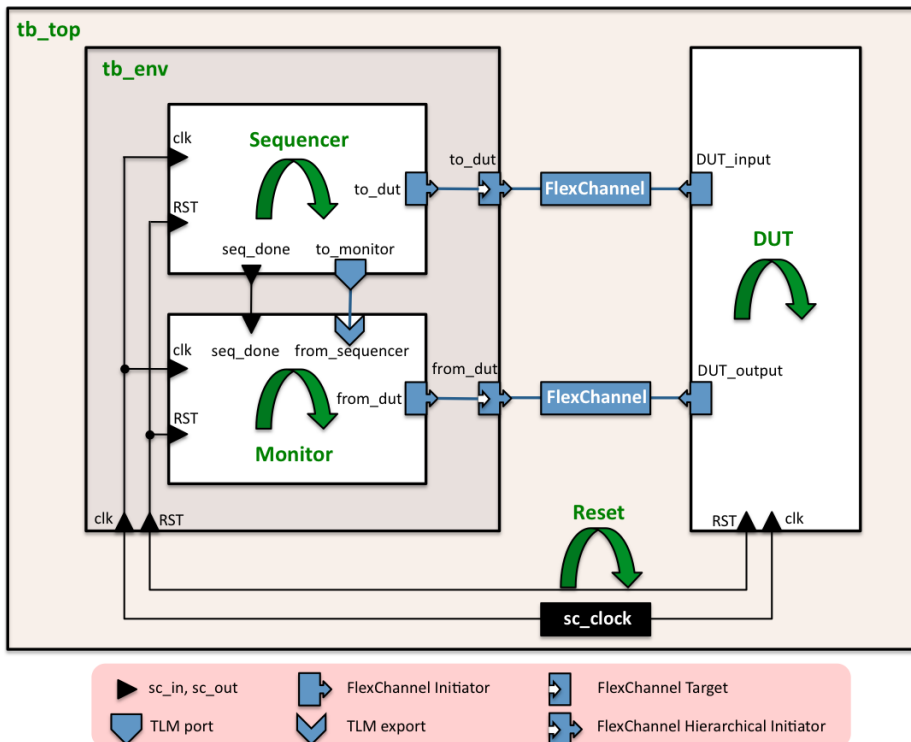
The sequencer and the monitor are sometimes decomposed further, in terms of the types of interfaces of the design model. For example, in image processing applications, the sequencer for providing the image might be separated from the sequencer for providing the headers. The monitor might also be decomposed so the function of checking correctness is separated from the function of collecting information from the design model. The decomposition often helps reusability, but the modeling of interactions among the modules could be complicated or additional overhead could be involved. This trade-off is important to evaluate in defining the structure of a testbench.

The interfaces with the design model might be also separated from the sequencer or monitor modules. This is especially advantageous when the interfaces must be changed when the design model is refined from SystemC to Verilog RTL through CtoS synthesis.

7.1.2 Understanding Testbench Module Hierarchy

The module hierarchy in a testbench depends on the decomposition made in the individual functions of the testbench. However, it usually involves at least three levels of, as shown in Figure 7-1 on page 7-4, which describes the module hierarchy of the example mentioned in “Writing a Testbench” on page 7-3.

Figure 7-1 Module Hierarchy of Testbench in flex_channels/pipelining Example



Here is a description of the three levels of hierarchy shown in [Figure 7-1 on page 7-4](#):

- The *lowest level* of the hierarchy consists of the individual modules for the sequencer and monitor functions, possibly with separate modules for interfaces. In [Figure 7-1 on page 7-4](#), the modules shown in white and marked **Sequencer** and **Monitor** belong to this level.
- The next level in the hierarchy defines a *single module* so it encapsulates all, and only, the modules that belong to the testbench. This level is the top level for the testbench, and the interfaces of this module consist only of those that must be connected to the design model, together with any other signals that must be provided externally to execute the testbench, such as the clock and reset signals. In [Figure 7-1 on page 7-4](#), the module **tb_env** corresponds to this module.

Note The hierarchy of this figure is the simplest case, where there is no intermediate level between **tb_env** and the individual modules.

- The *highest level* of the hierarchy is a module (**tb_top** in [Figure 7-1 on page 7-4](#)) that instantiates the top-level module of the testbench (**tb_env** in the example) and the top-level module of the design, and then connects all the external interfaces. You define the clock and reset signals in this module, and the module itself does not have any external interfaces. The clock signals can be defined by using the **sc_clock** class. For the reset signals, you use simple SystemC processes sensitive to the clock edges. In the example, the **tb_top** module shows an example of the reset processes.

7.1.3 Using the sc_main Function in a Testbench

In SystemC, **sc_main** is the entry point function defined to initiate and control the simulation of SystemC models.

The module hierarchy described in the previous section defines at the highest level a single module that instantiates both the testbench and the design model, generates the clock and reset signals, and establishes all connections among them. This single module is then instantiated in the **sc_main** function, and this is the only SystemC object created in this function.

An alternate approach would be to use the **sc_main** function as the top module. In this case, the module hierarchy would not define the global top-level module (**tb_top** in the example), but instead **sc_main** would instantiate the design model and the testbench, define the clock and resets, and connect them. While this alternate approach is used in practice, it is recommended that you use the previous approach – in which the global top-level module is explicitly defined.

When the global top-level module is explicitly defined, the **sc_main** function is focused on specifics related to the simulation of the target SystemC model, while the model itself is fully defined under a single top-level module. Furthermore, the reset signals and their behaviors are defined by using SystemC processes, which lets you easily change the reset behaviors, by modifying only the code inside the reset processes. In the alternate approach, if you used the **sc_main** function to define the reset signals, the code describing the reset behavior could be scattered within the function, and modifying the reset behavior could result in changes introduced in many places in the **sc_main** function.

The **sc_main** function typically has three sections:

- The first is to parse the command-level options and the arguments.
- The second is to instantiate the SystemC model to be simulated.
- The third is to start the simulation and apply post-simulation procedures as needed.

Note The *LRM* provides more information about these sections.

It is important to note that when instantiating the SystemC model, it is better to use the **new** operator to create the object, rather than statically instantiating the object.

For example, it is better to do the instantiation as:

```
tb_top *top_module = new tb_top("top_module");
```

rather than:

```
tb_top tb_top("top_module");
```

The **new** operator method creates the SystemC object using the heap, while the static instantiation method uses the stack to allocate the object. The size of the stack is statically defined and may not have enough space to store the whole SystemC model that you are trying to simulate.

Unlike the main function of the general C++ language, **sc_main** is just a regular function from the point of view of C++ compilers, and therefore when the target design with the testbench is big, the static instantiation method might result in an error in the elaboration phase. Although it is often possible to specify an option to control the stack size in the simulator, the **new** operator method works better, in general, for practical designs.

7.2 Generating Models

This section provides a graphical depiction of CtoS simulation models, then describes how CtoS automatically generates models and how you can also manually generate models:

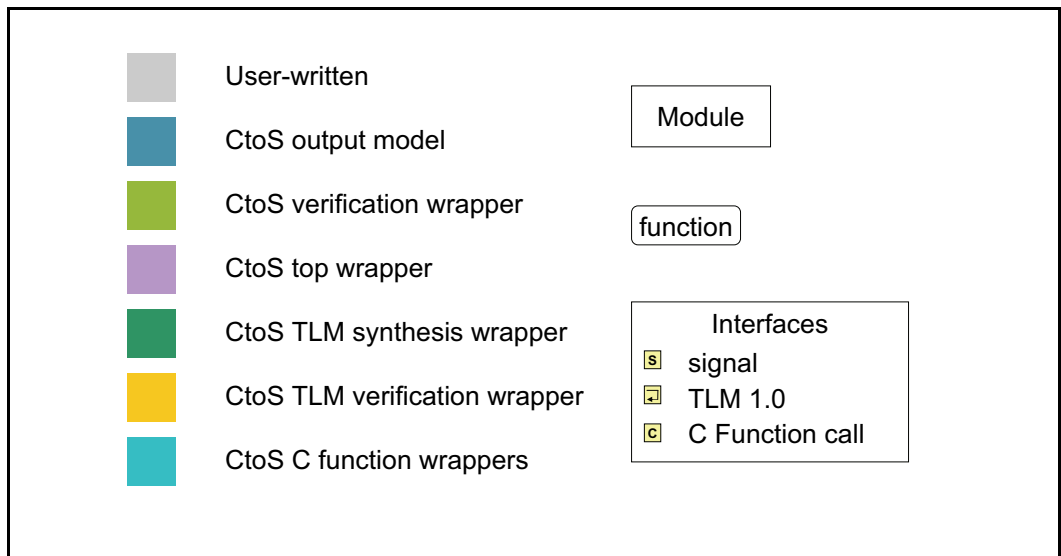
- “Graphical Depiction of all CtoS Models” on page 7-7
- “Automatic Generation of Models” on page 7-9
- “Generating Verilog Behavioral Models” on page 7-9
- “Generating RTL Descriptions (Only after Scheduling)” on page 7-11
- “Generating RTL Descriptions (Only after Scheduling)” on page 7-11

7.2.1 Graphical Depiction of all CtoS Models

Figure 7-3 on page 7-8 provides a graphical depiction of all of the CtoS models.

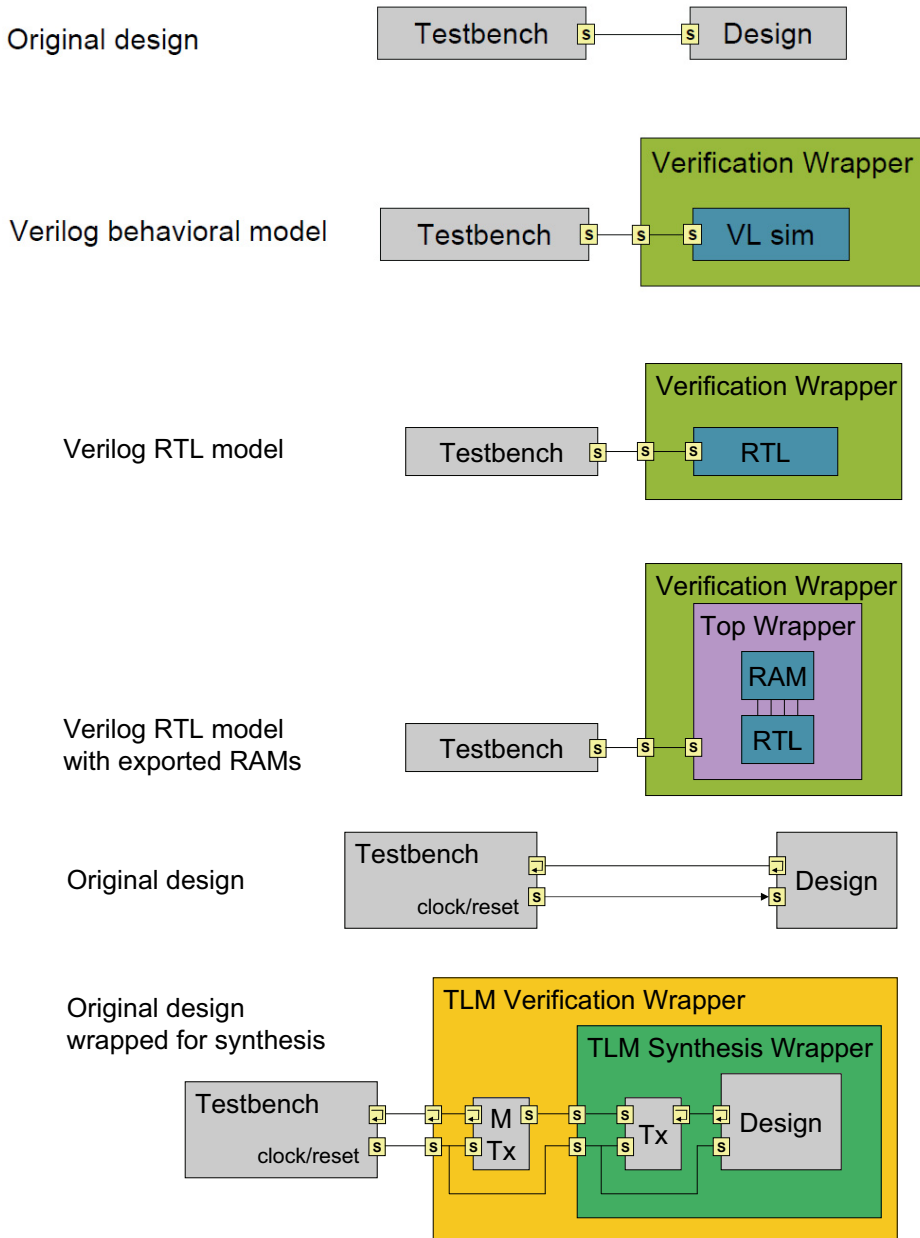
In this drawing, the following legend is used:

Figure 7-2 Legend for Graphical Depiction of CtoS Models



These models are further described in the remainder of this chapter.

Figure 7-3 Graphical Depiction of all CtoS Models



7.2.2 Automatic Generation of Models

CtoS automatically generates simulation models after a successful build, as well as after the successful generation of a schedule and the allocation of registers.

These models are written to the **model_dir** specified in the simulation configuration (see “[Defining a Simulation Configuration and Generating Simulation Makefiles](#)” on page 7-17).

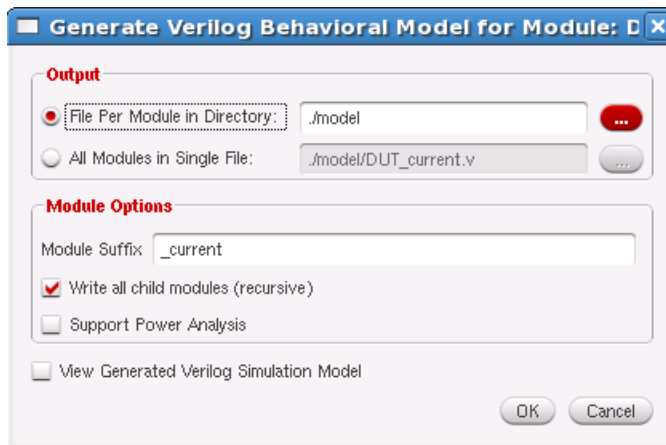
To prevent models from being generated, you can disable the **auto_write_models** design attribute (“[auto_write_models](#)” on page D-11) by unchecking the **Auto Write Models** box in the **Create New Design Wizard** (“[Create New Design Wizard](#)” on page 6-10).

7.2.3 Generating Verilog Behavioral Models

As described in the previous section, CtoS automatically generates a simulation model after a successful build. The filename for an automatically generated Verilog behavioral simulation model is *module_post_build.v*.

You can also *manually* generate a Verilog behavioral simulation model anytime after a design is built by selecting **File -> Generate -> Verilog Behavioral Model** and completing the **Generate Verilog Behavioral Model** dialog, as shown in [Figure 7-4 on page 7-9](#).

Figure 7-4 Generate Verilog Behavioral Model Dialog



In the **Generate Verilog Behavioral Model** dialog, complete the fields, as follows:

- **Output:**
 - **File Per Module in Directory:** Select this option if you want CtoS to generate separate files for each sub-modules transitively initiated from the given module. Specify the directory name in which to generate models.
 - **All Modules in a Single File:** Select this option if you want CtoS to generate a single output file for all modules. Specify the filename with path to generate models.
- **Module Suffix:** Specifies an optional string to be used as a suffix to every module name. The default is **_current**.

The **_current** suffix is used by the auto-generate model flow, and the **_sim** suffix is the standard suffix for the **write_sim** command.

- **Write all child modules (-recursive):** Specifies that a model be generated for the given database module, and every module transitively instantiated from that module.

By default, it is set to recursive. Unchecking this option will make the behavior non-recursive. CtoS will not descend into module hierarchy and a model is generated for the specified module only. For more information, see [“write_rtl” on page E-161](#).

- **Support Power Analysis:** Generates a simulation model compatible with TCF (toggle count format) power analysis.

TCF is the Cadence standard format to describe switching activity information in a design. The switching activity information contained in the file is required for accurate power analysis or power optimization of a design. This instructs model generation to promote all process-local registers up to the module scope. In addition, it causes the creation of extra registers, which are assigned a value of 1 in every clock cycle for which an edge or tag of the CDFG is active.

- **View Generated Verilog Simulation Model:** Specifies whether you want CtoS to display the new Verilog model in the **Verilog Behavioral Model** window.

Notes

- See also [“Controlling CtoS-Generated Verilog” on page J-1](#).
- You could also use **verilog** for the **-type** option to the **write_sim** command ([“write_sim” on page E-165](#)).

7.2.4 Generating RTL Descriptions (Only after Scheduling)

After the successful generation of a schedule, including the allocation of registers and netlist generation, for a module or all modules in a design, you can create a synthesizable RTL description for the specified module or all modules in a design. This is described in detail in [“Generating an RTL Description after Scheduling” on page 13-36](#).

7.3 Generating Wrappers

In order to verify your design using CtoS-generated models, CtoS also lets you generate different kinds of wrappers. These wrappers are necessary to connect CtoS-generated models to a SystemC testbench and enable verification via simulation.

This section first defines how CtoS automatically generates wrappers and then how you can also *manually* generate the various wrappers:

- [“Automatic Generation of Wrappers” on page 7-11](#)
- [“Generating a SystemC Verification Wrapper” on page 7-11](#)
- [“Generating a Verilog Verification Wrapper” on page 7-13](#)
- [“Generating a TLM Wrapper” on page 7-14](#)
- [“Generating a Top Wrapper for Exported Memories \(Only after Scheduling\)” on page 7-15](#)

7.3.1 Automatic Generation of Wrappers

CtoS automatically generates verification wrappers after a successful build, as well as after the successful generation of a schedule and the allocation of registers. These wrappers are written to the **model_dir** specified in the simulation configuration (see [“Defining a Simulation Configuration and Generating Simulation Makefiles” on page 7-17](#)).

To prevent wrappers from being generated, you can disable the **auto_write_models** design attribute ([“auto_write_models” on page D-11](#)) by unchecking the **Auto Write Models** box in the **Create New Design Wizard** ([“Create New Design Wizard” on page 6-10](#)).

7.3.2 Generating a SystemC Verification Wrapper

A *SystemC verification wrapper* facilitates the connection of a Verilog model with a SystemC testbench and is described in following subsections:

- [“How SystemC Verification Wrappers Connect Models with a Testbench” on page 7-12](#)
- [“How to Manually Generate a SystemC Verification Wrapper” on page 7-12](#)

Note For more details about the structure of a verification wrapper, including an example, see [“Structure of the SystemC Verification Wrapper” on page G-1](#).

7.3.2.1 How SystemC Verification Wrappers Connect Models with a Testbench

A SystemC verification wrapper is a SystemC model with a parameterized constructor that configures the design to be simulated.

In addition to the *model under test*, the original SystemC model can also be instantiated as a *reference model* inside a wrapper for cycle-by-cycle comparisons of the *model under test* output and the original SystemC model output.

Verification wrappers can be reused with models generated throughout the design flow.

Here are some of the details of how a SystemC verification wrapper connects generated models with a testbench:

- To support simulating models during the entire design flow, the wrapper defines the **CTOS_TARGET_SUFFIX** macro, which automatically selects the appropriate model, based on the makefile target.

For example, **CTOS_MODEL=final** would be converted to the string needed in the wrapper instantiation, “**_final**”

- To simplify the declaration of a model in a SystemC testbench, the wrapper declares a **typedef** for *module_name_ctos*, which can be used to declare the wrapper module, based on macros supported by the generated makefile.

Here is an example of **typedef** for an original SystemC model named **DUT**:

```
typedef DUT_ctos_wrapper DUT_ctos;
```

For compatibility with older versions, the **typedef** for *module_ctos_wrapper* is also declared.

```
typedef DUT_ctos_wrapper DUT_ctos_wrapper;
```

- To simplify instantiating a model in a SystemC testbench or comparing it with another model, the wrapper file declares the macros **CTOS_INSTANCE** and **CTOS_COMPARE_INSTANCE**.

Here is an example of macros for an original SystemC model named **DUT**:

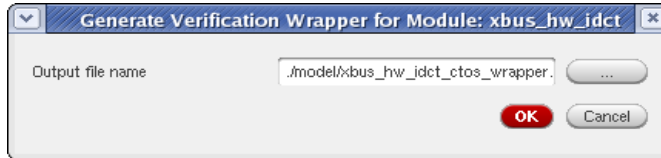
```
#define DUT_CTOS_INSTANCE(INSTNAME) DUT_ctos(INSTNAME,  
CTOS_TARGET_SUFFIX(CTOS_MODEL), NULL, false)  
  
#define DUT_CTOS_COMPARE_INSTANCE(INSTNAME, REFNAME)  
DUT_ctos(INSTNAME, CTOS_TARGET_SUFFIX(CTOS_MODEL), REFNAME, true)
```

7.3.2.2 How to Manually Generate a SystemC Verification Wrapper

CtoS automatically generates a SystemC verification wrapper after a successful build. The filename for an automatically generated verification wrapper is *module_ctos_wrapper.h*.

You can also *manually* generate a SystemC verification wrapper anytime after a design is built by selecting **File -> Generate -> Verification Wrapper** and completing the **Generate Verification Wrapper** dialog, as shown in [Figure 7-5 on page 7-13](#).

Figure 7-5 Generate Verification Wrapper Dialog



In the **Generate Verification Wrapper** dialog, there is one field to complete, as follows:

Output file name: Specifies the name for the generated wrapper file. The wrapper module constructor has parameters that are used to name the *model under test* and *reference model* instances. This is done indirectly by providing the suffixes of the module names via the constructor. The default is **module_ctos_wrapper.h**.

Notes

- If a name has been specified in the **sim_wrapper_filename** design attribute ("[sim_wrapper_filename](#)" on [page D-23](#)), and you specify a different name, you will get a warning.
- You could also use the **write_wrapper** command ("[write_wrapper](#)" on [page E-176](#)).

7.3.3 Generating a Verilog Verification Wrapper

This is a Preliminary feature.

The verilog verification wrapper facilitates the simulation of SystemC input model in your verilog testbench. You can substitute the instance of the RTL model in your verilog testbench with this wrapper because it has the same I/O ports as the RTL model.

More details are described in the following subsections:

- Generating a verilog verification wrapper with the **write_verilog_wrapper** command, see "[write_verilog_wrapper](#)" on [page E-175](#).
- The structure of a verilog verification wrapper, including an example, is described in "[Structure of the Verilog Verification Wrapper](#)" on [page G-9](#).

Notes

- Verilog verification wrapper is not generated automatically.
- Users must specify the header files, where top verilog module definition resides, using the **define_header_files** design attribute. See "[header_files](#)" on [page D-16](#).

7.3.4 Generating a TLM Wrapper

If you have imported a TLM design, you need an additional *TLM wrapper* to connect the TLM interface of your SystemC testbench, as shown in [Figure 7-3 on page 7-8](#).

This section is divided into the following subsections:

- “Description of a TLM Wrapper” on page 7-14
- “How to Manually Generate a TLM Wrapper” on page 7-15

7.3.4.1 Description of a TLM Wrapper

The TLM wrapper described here encapsulates the verification wrapper and can be reused with models generated throughout the design flow.

The constructor of the TLM wrapper takes the following arguments:

- module name (string)
- directive to use the verification wrapper (Boolean type)
- specification of the *model under test* (same as the verification wrapper)
- specification of the *reference model* (same as the verification wrapper)
- directive to compare the *model under test* and the *reference model* (Boolean type)

If you specify *false* for the directive to use the wrapper, the TLM wrapper instantiates the TLM model with its transactors. Otherwise, the TLM wrapper instantiates the Verilog verification wrapper, and the values for the *model under test*, *reference model*, and comparison are passed directly to the underlying verification wrapper.

The C++ syntax for these typical usage scenarios is as follows:

- Instantiation of the TLM design with transactor in a SystemC-only simulation:

```
DUT_ctos_tlm_wrapper dut("dut", false);
```

- Instantiation of a Verilog module with no *reference model* to *model under test* comparison:

```
DUT_ctos_tlm_wrapper dut("dut", true, CTOS_TARGET_SUFFIX(CTOS_MODEL));
```

In both cases, the TLM wrapper will expose a TLM interface to its testbench environment. See “[TLM Library](#)” on page 15-65 for more illustrative TLM examples.

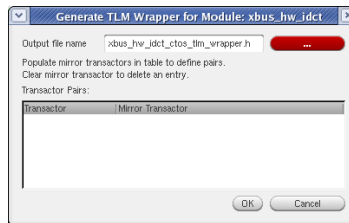
Note The `CTOS_TARGET_SUFFIX` macro is used to automatically select the appropriate model, based on the makefile target.

7.3.4.2 How to Manually Generate a TLM Wrapper

CtoS automatically generates a TLM wrapper after a successful build. The filename for an automatically generated TLM wrapper is `module_ctos_tlm_wrapper.h`.

You can also *manually* generate a TLM wrapper anytime after a design is built by selecting **File -> Generate -> TLM Wrapper** and completing the **Generate TLM Wrapper** dialog, as shown in [Figure 7-6 on page 7-15](#).

Figure 7-6 Generate TLM Wrapper Dialog



Note You could also use the `write_tlm_wrapper` command (“[write_tlm_wrapper](#)” on page E-170).

7.3.5 Generating a Top Wrapper for Exported Memories (Only after Scheduling)

If the memories in your design are exported outside the top model (the `default_export_memories` design attribute is set to `true`; see “[default_export_memories](#)” on page D-13), then you need an additional *top wrapper* to connect the memories and the RTL design to your SystemC testbench. You can generate a top wrapper once, after you have generated RTL.

Notes

- Built-in RAMs are *not* exported.
- See also “[Generating a Top Wrapper for Exported Memories](#)” on page 13-40.

7.4 Modifying a Testbench

You must modify your testbench to instantiate the wrapper with the appropriate arguments. Instead of instantiating the *model under test*, the testbench should instantiate the verification wrapper. To keep the original testbench running, these modifications should be guarded by the definition of `CTOS_MODEL`.

1. **#include** the appropriate wrappers at the top of the testbench (after the original model header file or module declaration).
2. In the declaration of the testbench module, declare the wrapper.

3. In the constructor, instantiate the wrapper with the appropriate constructor arguments. You can use the **CTOS_TARGET_SUFFIX** macro to convert the **CTOS_MODEL** string to the suffix string expected by the verification wrapper.
4. (optional) Print the message that simulation was successful. This can then be checked for success when simulation is invoked, using the **-success_msg** option to the **define_sim_config** command ([“define_sim_config” on page E-55](#)).

Although this step is optional, it is highly recommended, since it provides positive confirmation that the simulation ran as expected (which ensures that the makefile detects failed simulations and that correct simulation pass/fail status is displayed in the CtoS GUI simulation summary pane).

The SystemC verification wrapper generates error events, and an error monitor can be added to the testbench to monitor these events.

All of the examples in the examples hierarchy follow this style.

Note Before running any CtoS examples, first review [“Setup for Examples” on page F-2](#).

It is *not* recommended that you build testbenches driven by absolute time, rather than the clock. The communication between the testbench and the *model under test* should be done only by using signals at the interface of the latter.

Example

```
#include "dut.h"

#ifdef CTOS_MODEL
    #include "model/dut_ctos_wrapper.h"
#endif

// Declaration of Testbench Module

#ifdef CTOS_MODEL
    dut_ctos *m_dut;
#else
    dut *m_dut;
#endif

// Constructor of Testbench Module

#ifdef CTOS_MODEL
    m_dut = new DUT_CTOS_INSTANCE("wrapper");
#else
    m_dut = new dut("dut");
#endif
```


Note The recommendation to instantiate the top module of your design via **new** in a testbench does *not* apply to modules nested within the top module of your design. A module nested underneath the top module of your design should be instantiated via a field of the top module of the appropriate module type.

If your modeling style precludes you from instantiating the top module of your design via **new**, then you will not be able to use the **DUT_CTOS_INSTANCE/DUT_CTOS_COMPARE_INSTANCE** macros. Instead, you will need to do the following:

Example

```
#include "dut.h"
#ifdef CTOS_MODEL
#include "model/dut_ctos_wrapper.h"
#endif
// Declaration and constructor of Testbench Module
#if defined CTOS_MODEL
    cout << "Instantiating wrapper with DUT" << CTOS_TARGET_SUFFIX(CTOS_MODEL) << endl;
    dut_ctos      m_dut("dut", CTOS_TARGET_SUFFIX(CTOS_MODEL), "", true);
#else
    cout << "Instantiating DUT" << endl;
    dut           m_dut("dut");
#endif
```

Note The SystemC verification wrapper is created and initialized by specifying the constructor arguments. In this example, the verification wrapper is set up for a side-by-side comparison of the output values of the CtoS-generated model (as specified by the **CTOS_MODEL** macro) and the original SystemC module (**dut**).

7.5 Defining a Simulation Configuration and Generating Simulation Makefiles

CtoS lets you generate *makefiles* to simulate and verify CtoS-generated models. As part of this process, you must describe your simulation environment to CtoS by defining a *simulation configuration*.

Important For makefile generation, it is recommended that you set **install_directory/bin** in your path for the **ctos** and **ctosgui** executables. This ensures that the header files will be found for simulation.

The following topics are described in this section:

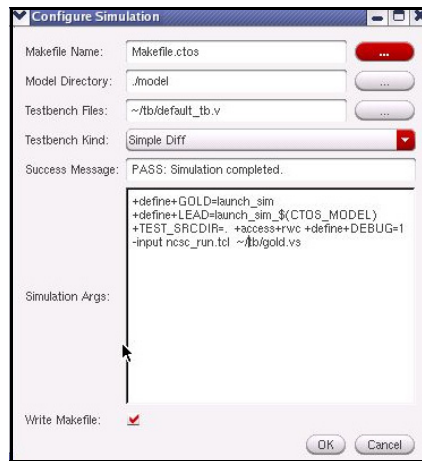
- [“Defining a Simulation Configuration” on page 7-18](#)
- [“Generating a Simulation Makefile Automatically” on page 7-19](#)
- [“Generating a Simulation Makefile Manually” on page 7-19](#)

7.5.1 Defining a Simulation Configuration

A design's *simulation configuration*, a *child scope* of the design, contains objects that must be consistent across the various steps in the simulation flow. A single simulation configuration object, named **default_sim_config**, is automatically created for each new design.

To define a simulation configuration, select the **Configure** button (second from the left) in the **Simulation Monitor**, to display the **Configure Simulation** dialog, as shown in [Figure 7-7 on page 7-18](#).

Figure 7-7 Configure Simulation Dialog



The fields are preset with values from the design's simulation configuration [see "[Default Simulation Configuration Object Attributes \(simulation_configs\)](#)" on page D-85]. If you change any value and press OK, the **define_sim_config** command ("[define_sim_config](#)" on page E-55) is called to change the modified values.

Although you may modify these settings at any time during the design flow, it is recommended that before you build the design, if you do not want to use the default of **/model**, you set the model directory using:

- the **Model Directory** field in **Configure Simulation** dialog, or
- the **Create New Design Wizard** ("[Create New Design Wizard \(Page 6\)](#)" on page 6-18), or
- the **-model_dir** option of the **default_sim_config** command.

Also, before launching simulation, you must specify your testbench files using:

- the **Testbench Files** field in the **Configure Simulation** dialog or
- the **-testbench_files** option of the **default_sim_config** command.

7.5.2 Generating a Simulation Makefile Automatically

The **Configure Simulation** dialog, as shown in [Figure 7-7 on page 7-18](#), also has a checkbox for **Write Makefile** (whose default is *on*). If you change any options on this dialog, a new makefile is generated with the `write_sim_makefile` command ("[write_sim_makefile](#)" on [page E-168](#)).

Important The **work** and **run** directories used by the IES simulator are created in the same directory as the makefile. Thus, if you specify a path, the **work** and **run** directories will also be in this path.

7.5.3 Generating a Simulation Makefile Manually

CtoS supports simulation with rules defined in a makefile, which you can manually generate with the `write_sim_makefile` command ("[write_sim_makefile](#)" on [page E-168](#)).

The following topics are described in this section:

- "[Targets of CtoS-Generated Simulation Makefiles](#)" on [page 7-19](#)
- "[Simulation Outputs of CtoS-Generated Simulation Makefiles](#)" on [page 7-20](#)
- "[Using Variables to Specify Work/Run/Model Directories](#)" on [page 7-21](#)
- "[Overriding Certain Variables in CtoS-Generated Simulation Makefiles](#)" on [page 7-21](#)
- "[Using Wildcards with CtoS-Generated Simulation Makefiles](#)" on [page 7-22](#)

7.5.3.1 Targets of CtoS-Generated Simulation Makefiles

A CtoS-generated simulation makefile has the following *targets*:

- **orig_sim** – simulates the original SystemC model and runs the testbench without defining **CTOS_MODEL**. This is the synthesizable model provided to CtoS as input.
- **post_build_sim** – simulates the Verilog behavioral model with suffix **_post_build** in the model directory (specified in **model_dir** in the simulation configuration) and runs the testbench with **CTOS_MODEL="post_build"**

Note CtoS automatically generates these source files after a successful build.

- **current_sim** – simulates the Verilog behavioral model with suffix **_current** in the model directory (specified in **model_dir** in the simulation configuration) and runs the testbench with **CTOS_MODEL="current"**

Note You can generate models any time during the design flow with **File -> Generate -> Verilog Behavioral Model** (or the **write_sim** command). The **Generate Verilog Behavioral Model** dialog prompts you to write models to the **model_dir** with suffix **_current**. See also [“Generating Verilog Behavioral Models” on page 7-9](#)

- **final_sim** – simulates the Verilog behavioral model with suffix **_final** in the model directory (specified in **model_dir** in the simulation configuration) and runs the testbench with **CTOS_MODEL="final"**

Note CtoS automatically generates these files after a successful schedule and register allocation.

- **rtl_sim** – simulates the Verilog RTL with suffix **_rtl** in the model directory (specified in **model_dir** in the simulation configuration) and runs the testbench with **CTOS_MODEL="rtl"**

Note You can generate RTL after scheduling and register allocation with **File -> Generate -> RTL** (or the **write_rtl** command). The **Generate RTL** dialog prompts you to write models to the **model_dir** with suffix **_rtl**. See also [“Generating an RTL Description after Scheduling” on page 13-36](#).

All makefile targets start the IES front-end with their respective source files.

The design’s simulation configuration specifies other data needed for the simulator, including a list of testbench files.

Any other simulation arguments (for input stimulus, output waveform database, etc.) can also be specified there.

Note These arguments should apply to all targets since they are written to the makefile. Arguments that are not target-specific, or more dynamic in nature, should be specified when launching simulation. See also [“Default Simulation Configuration Object Attributes \(simulation_configs\)” on page D-85](#).

7.5.3.2 Simulation Outputs of CtoS-Generated Simulation Makefiles

Simulation outputs are written to each **run_target** directory. The simulation log is written to the **log** directory. To determine if simulation is successful, search for **success_msg** (as specified in the **simulation_config**) in the simulation **log** file. Generated files are written to each **work_target** directory.

The makefile also supports self-checking testbenches or simple **diff** comparisons (performed on the **run_target** directory and **run_orig** directory) against the original simulation (as specified in **testbench_kind** in the simulation configuration). The makefile returns **0** if a success message is found, and **diff** checking returns **true** (if used).

Important The **diff** mechanism relies on the generated files being in the **run_*** directories. However, the files will be in the **work_*** directories if you generate the files with the following:

```
ofstream ofile("data.txt");
```

To get the files in the right location, thus enabling the **diff**, use the following:

```
#ifndef CTOS_MODEL
    std::string filepath("../run_orig_sim/data.txt");
    ofstream ofile(filepath.c_str());
#else
    std::string filepath("../run");
    filepath += CTOS_TARGET_SUFFIX(CTOS_MODEL);
    filepath += "_sim/data.txt";
    ofstream ofile(filepath.c_str());
#endif
```

7.5.3.3 Using Variables to Specify Work/Run/Model Directories

If a simulator needs the **work** or **run** directory name at run-time, the CtoS-generated makefile exports both the **CTOS_MODEL** and **CTOS_TARGET_NAME** variables to the shell running the simulation.

You can specify the **work/run** directory or target-specific arguments by embedding these variables in the arguments to the **-simulator_args** option to the **define_sim_config** command as **\\$(variable)**. For example, to specify a **run** directory for writing simulation output using the **CTOS_TARGET_NAME**:

```
-simulator_args "+systemc_args -i ../inputfile.ext -o run_\$(CTOS_TARGET_NAME)/outfile.ext"
```

Similarly, if you need to dynamically select a CtoS-generated model, you can embed **model_\\$(CTOS_MODEL)** in the simulator arguments. CtoS looks for these variables and replaces them with the correct model or target name before writing these arguments out to the makefile.

You can also use variables in a simulator Tcl script (often used to probe signals in a waveform database) using the **-input** option. In this script, you can set the **work** directory to write into the waveform database using the **"\$::env()"** Tcl directive on the **CTOS_TARGET_NAME** variable, for example:

```
File: input.tcl
set targetName "../work_${::env(CTOS_TARGET_NAME)}"
database -open -shm -into $targetName/waves.shm waves -default
```

7.5.3.4 Overriding Certain Variables in CtoS-Generated Simulation Makefiles

You can override the following variables in CtoS-generated simulation makefiles at the command line:

```
CTOS_EXE    - Specify path to CtoS executable (if not in your path, or you
               want to use a version different from the one in your path)
USER_ARGS   - Specify additional arguments to the simulator
SIM_GUI     - Bring up SimVision
```

For example:

```
make CTOS_EXE=/cad/ctos/Install/bin/ctos USER_ARGS="-gui"
```

7.5.3.5 Using Wildcards with CtoS-Generated Simulation Makefiles

You can use makefile wildcards in the arguments to the **-testbench_files** option of the **define_sim_config** command to avoid having to list all the files, as follows:

```
-testbench files "<dir path>/tb/default_tb.cpp \$(wildcard <dir path>/tb/monitor_*.cpp) "
```

The wildcard will be detected by CtoS and transformed in the generated makefile as follows:

```
File: Makefile
TB_WILDCARD_0      := $(wildcard <dir path>/tb/monitor_*.cpp)
DESIGN_TB_FILES    := <dir path>/tb/default_tb.cpp $(TB_WILDCARD_0)
```

You can also embed shell wildcards in the simulator or testbench arguments.

7.6 Simulating from the CtoS Environment

CtoS supports an *Integrated Environment*, in which you can start IES within CtoS, as follows:

- “Using the Simulation Monitor” on page 7-22
- “Launching Simulation” on page 7-24

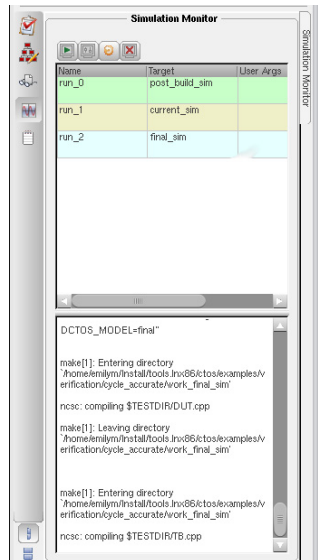
7.6.1 Using the Simulation Monitor

The CtoS GUI provides a **Simulation Monitor**, as shown in [Figure 7-8 on page 7-23](#).

You start the monitor by selecting the icon in the shape of an oscilloscope located on the left, between the icons for the **Call Graph** viewer and the **Summary Report**.

On the top left edge of the monitor is a set of action buttons – **Launch**, **Configure**, **Re-run**, and **Terminate**, which are described on the following page.

Figure 7-8 Simulation Monitor



- The **Launch** button display depends on whether you have specified testbench files and a makefile:
 - If testbench files have not been specified, you will be prompted to go to the **Configure Simulation** dialog, which is described in “[Defining a Simulation Configuration](#)” on page 7-18.
 - If testbench files have been specified but the makefile does not exist, you will be prompted to create a makefile, and if you click **Yes**, the **write_sim_makefile** command (“[write_sim_makefile](#)” on page E-168) will be called.
 - If both testbench files and a makefile have been specified, you will see the Launch Simulation dialog, which is described in “[Launching Simulation](#)” on page 7-24.
- The **Configure** button displays the **Configure Simulation** dialog, which is described in “[Defining a Simulation Configuration](#)” on page 7-18.
- The **Re-run** button relaunches the simulation on the selected row.
- The **Terminate** button removes the entry on the selected row and terminates the process if running.

Under these buttons is a table of simulation runs that have been launched in this session. Each row represents a simulation run, with column headers **Name**, **Target** and **User Args**. Below this window is the output of the selected run (or if no run is selected, the most recent run is shown).

The background color of the row specifies the state of the run:

- **Blue** : Run has started
- **Green** : Run has successfully finished.
- **Yellow** : Run has finished, but the makefile returns a non-zero status (diff).
- **Red** : Run has crashed.

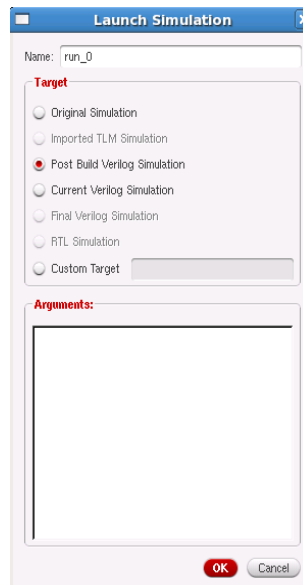
7.6.2 Launching Simulation

When you have specified testbench files and a makefile, you can launch simulation (through the Simulation Monitor, described in the previous section) using the **Launch Simulation** dialog, as shown in [Figure 7-9 on page 7-24](#).

You will be prompted for a unique run name (displayed in the docking window); the default is generated from the **run_id** target name and runtime makefile arguments. When you click **OK**, the simulation is launched, and an entry is added to the simulation runs in the table.

The state is running, so it is colored **blue** and will remain so until the run is completed. This dialog explicitly displays the predefined set of targets, along with the custom target, which is the text field.

Figure 7-9 Launch Simulation Dialog



You can also use the **launch_sim** command. This command changes the working directory to the directory of the makefile and then invokes the makefile with the specified target and user arguments.

The **launch_sim** command (“[launch_sim](#)” on page E-80) also specifies the location of the CtoS installation with:

```
-DCTOS_EXE=install_directory/bin/ctos
```

CtoS will print output messages generated by the makefile and simulator and blocks until complete.

Note Simulation requires that **testbench_files** be specified in the design’s simulation configuration (see “[Defining a Simulation Configuration and Generating Simulation Makefiles](#)” on page 7-17).

7.7 Debugging Simulation Mismatches

CtoS has developed a technique for speeding up the debugging of simulation mismatches.

This technique consists of adding **sc_signals** to a design and writing variables of interest to those signals.

CtoS can preserve the **sc_signals** defined in the design, throughout the synthesis process, so they can then be compared automatically in a side-by-side simulation using a verification wrapper.

This technique is described in detail in [“Debugging Simulation Mismatches” on page I-1](#).

7.8 Renaming States in Support of FSM Coverage

You can specify and maintain names of the FSM state nodes, so that you can refer to the state nodes using names that are more meaningful. CtoS supports friendly and robust naming of states (even in light of design changes). CtoS recommended flow is as follows:

1. Construct CtoS annotation paths that are recognizable by you (see [“State Identification by Annotation Paths” on page 7-26](#))
 - Add labels to wait states of interesting nodes
 - Add pragmas to function calls of sequential functions that will be inlined
2. Rename states based on the annotation paths. The **State Report Viewer** supports interactive editing of state names. (see [“Renaming States for Verification” on page 7-28](#))
3. Enable CtoS to use states names when modeling FSMs in the generated RTL. The **Design Property Dialog** has checkbox to "Use Verilog State Names" (see [“RTL Format for FSM Coverage” on page 7-27](#).)
4. Facilitate renaming states after modifications to design (see [“Renaming States by Script” on page 7-29](#)):
 - Use the **Write Rename State** button in the **State Report Viewer** in the first session.
 - Use the **Rename From File** button in the **State Report Viewer** in subsequent sessions.

The FSM states are represented in the generated RTL, which is understood by FSM coverage tool in Incisive Metrics Center (IMC). For more information, see the *Incisive Metrics Center User Guide*.

7.8.1 State Identification by Annotation Paths

Renaming states after every design change can be a lot of work because designs can have many states. Renaming states through a script with **find** and **set_attr** commands provides some automation, but state names may change even after a small design change, which would require changes in that script after every design change.

CtoS supports **annotation_paths** on state nodes that improve the robustness of identifying FSM states.

An **annotation_path** is an alternative name for a state node. The initial value of the **annotation_path** is the user-specified label on the wait statement in SystemC source code. During micro-architecture and optimization, **annotation_paths** are prefixed appropriately to reduce the chance of name clashes.

Each state node has TCL attribute **annotation_paths**, which lists the **annotation_paths** of this state:

```
get_attr annotation_paths <state_node_id>
```

The **find_by_annotation_path** command returns state nodes that match the specified **annotation_path**:

```
find_by_annotation_path -behavior $beh path1 => state_nodes
```

For more information, see [“Annotating Function Calls” on page 14-76](#).

7.8.1.1 Understanding Annotation Paths

Users assign labels to interesting wait statements in the SystemC code. For example:

```
STATE1: wait();
```

The user should add labels to states in the SystemC source code for those states of interest to verification. After successful **build** command, each state node preceded with label is given an **annotation_path** the same as the label. For example, the **annotation_path** associated with above-mentioned state node is **STATE1**.

To unifiy the annotation due to inlining, user can optionally add annotation pragmas before calls to sequential functions in the SystemC source code. For example:

```
#pragma cto annotation FUNC_CALL_1  
func(a, b);
```

After inlining sequential functions, the value of the pragma (if provided) is prefixed to the **annotation_path** of each state node in the function. For example, the **annotation_path** associated with the state "STATE1" after inlining this function call is:

```
FUNC_CALL_1/STATE1
```

After unrolling loops that contain states, state nodes with annotation `_paths` will be prefixed with the unrolling index. For example the annotation_path associated with the above state node in loop iteration 2 will be:

```
unr2/FUNC_CALL_1/STATE1
```

Notes

- A state node can have more than one annotation path if two states are merged together through CtoS optimizations.
- Multiple states in a given behavior can have the same set of annotation paths, if
 - the state node is not labeled OR
 - the inlined function calls are not annotated
- To avoid two states ending up with same set of annotation paths, it is recommended to annotate all involved inlined function calls and label all the wait(s).

7.8.2 Renaming States by Annotation Paths

The combination of the **find_by_annotation_path** command and setting the name on a state node through the **set_attr** command is a more robust way to assign user-friendly names to states. For more information see [“find_by_annotation_path” on page E-62](#) and [“set_attr” on page E-140](#).

Furthermore a state may have more than annotation_path caused by state merging during optimization. To find and rename a state that has set of annotation_paths, CtoS supports the **rename_state** command. ([“rename_state” on page E-102](#)).

7.8.3 RTL Format for FSM Coverage

You can analyze FSM coverage of CtoS-generated RTL using the Incisive Metrics Center (IMC). This section describes how to improve the correspondence between FSM states shown in IMC and wait statements in the SystemC source code. This should enable users to relate missing coverage to their SystemC source code and more readily determine what additional tests are needed to improve FSM coverage. The FSM coverage can be analyzed through IMC after SimVision simulation of RTL with coverage arguments.

To enable IMC to display FSM states with names that are recognizable, the RTL generated by CtoS should have the name of the state that corresponds to the wait statement for the state variable of the case statement. CtoS will generate this form of the case statement if the following steps are followed:

- Set the design attribute **verilog_mux_style** to **one_hot** (see [“verilog_mux_style” on page D-26](#))

- Set the design attribute **verilog_use_state_names** to true (see “[verilog_use_state_names](#)” on page D-28)

You can also set these attributes using the **Output** tab of the **Design Properties Dialog**. For more information, see “[Design Property Dialog](#)” on page 6-18.

Additionally, you can also set these attributes in TCL:

- `set_attr verilog_mux_style one_hot [get_design]`
- `set_attr verilog_use_state_names true [get_design]`

With the settings of these attributes, the **write_rtl** command generates RTL in the following style:

- Local parameter definitions (localparam) for each state, with the name of the state that corresponds to the wait statement.
- Each item for the case statement of the state variable for the behavior is the name of a state in the behavior.
- The assignments to the next state variable are parameter names instead of hexadecimal values.

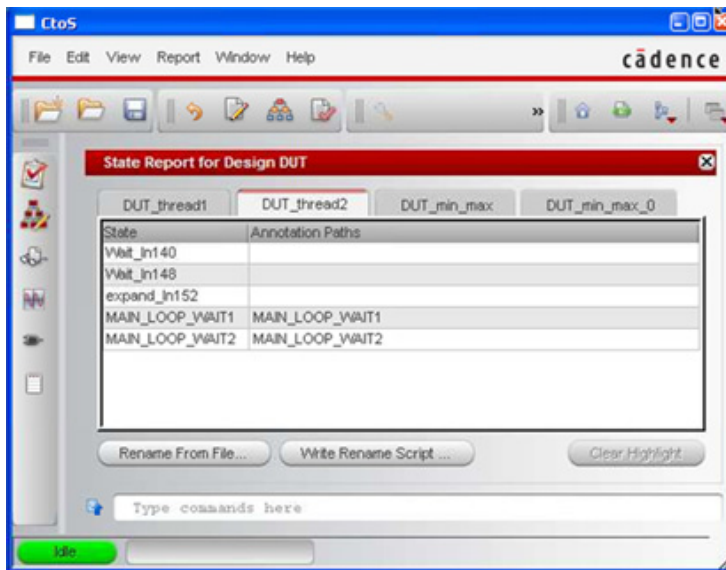
For more information, see the *Incisive Metrics Center User Guide*.

7.8.4 Renaming States for Verification

Mapping names of states shown in IMC tool back to wait statements in SystemC code is not easy. CtoS supports renaming states to user-friendly names after register allocation. These user-friendly names are then referenced in the generated RTL and displayed in the IMC tool.

The state names can also be viewed in the state report. Click **Report > States** menu item on the design to view **State Report Viewer** in the CtoS GUI, as shown in . Each tab corresponds to a behavior in the design.

Figure 7-10 State Report Viewer



If you want to display states for single behavior, then use the context menu on a behavior in the Hierarchy Viewer.

The name of a state can be changed by double clicking the name of a state in the first column. For more information on how to rename the states using script, see the following section .[“Renaming States by Script” on page 7-29](#).

Additionally in TCL, the **report_states** command generates the states report ([“report_states” on page E-127](#)). The name of the state can be changed by setting the name attribute on the state node using the **set_attr** command ([“set_attr” on page E-140](#)).

7.8.5 Renaming States by Script

CtoS supports writing a script that renames all annotated states in each behavior based on their annotation paths. This script can then be applied to subsequent version of the design and be robust in light of SystemC code changes.

The recommended flow is to use the **State Report Viewer** to rename all the interesting states. The second column in the report lists the annotation paths to help you recognize the states. After renaming all your states in the GUI, write the rename script that will specify all the states with the new names. For subsequent runs of CtoS on modified versions of the design, you must apply the rename script to get all the same states renamed in one command.

The **State Report Viewer** has the **Write Rename Script** button that prompts you for a file name ([Figure 7-10 on page 7-29](#)). On subsequent run the same script can be loaded with the **Rename From File** button. Upon successful renaming of a state, the state name is highlighted in green in the table. This gives immediate feedback on states that were not mentioned in the script and thus may be new states that need more attention from the designer to understand the impact on FSM coverage. After analyzing the state name changes, you can clear the green highlight by clicking on the **Clear Highlight** button.

Note that when script is applied, messages are issued when a state is not found for the annotation paths. This indicates that the states in design have changed and require further analysis by designer to understand the impact on FSM coverage.

The script can also be generated from Tcl. After register allocation, the **write_rename_states_script** command ([“write_rename_states_script” on page E-160](#)) can be used to write a script to rename the existing states to their current names, using annotation paths. The names in that script can be edited in a text editor, and the script can then be sourced to rename states accordingly. The same script can be used in subsequent CtoS runs.

8 Specifying Micro-Architecture

For examples of the Specifying Micro-architecture step of the CtoS flow, look in the following directory:

```
install_directory/share/ctos/examples/features
```

Note Before running any CtoS examples, first review [“Setup for Examples”](#) on page F-2

In the Specifying Micro-architecture step of the CtoS flow, you provide additional information to help CtoS implement your design as closely as possible to your design goals.

In other words, CtoS is trying to understand the answers to the following questions:

- Do you want a very small, but possibly slower, implementation?
- Do you need the fastest possible implementation, and size is not really a concern?
- Is a low power implementation of most importance to you?

The situation is probably more complex than these simple questions imply – you would like certain functions that are important, or occur frequently, to be performed rapidly, and you are willing to dedicate space to get speed for this.

Conversely, you would like other functions that are unimportant, or occur infrequently, to be implemented in a way that minimizes space.

In the Specifying Micro-architecture step, you will address the following tasks, which will transform the design to be synthesizable and ultimately guide the CtoS scheduler to find the best possible implementation for your design:

- Resolving Functions
- Resolving Loops
- Resolving Arrays

CtoS provides powerful commands to let you explicitly make all micro-architecture choices giving you complete control over the quality of results.

In the GUI, the Task Manager and Micro-Architecture Manager helps guide you through this process providing lists of required tasks as well as estimations of the impact of making each choice. The Micro-Architecture Manager can also be invoked on a single process letting you focus on resolving the loops, arrays and called functions of a single process.

Alternatively, CtoS can semi-automatically resolve micro-architecture choices in one of several prescribed ways called synthesis modes, that correspond to typical design scenarios. Synthesis modes supports designs that follows the following patterns:

- your SystemC design is cycle-accurate, that is you want the generated RTL to be cycle-by-cycle equivalent to your SystemC
- your SystemC design consists of one top-level loop that you want to pipeline
- your design has no latency or throughput constraints

In addition, synthesis modes is useful for performing a trial synthesis run where RTL is generated quickly without concern for optimality of performance or area. You may also specify synthesis mode and then fine-tune it to improve quality of results.

For more information, see the following sections:

- [“Resolving Functions” on page 8-3](#)
- [“Resolving Loops” on page 8-16](#)
- [“Resolving Arrays \(Memories\)” on page 8-61](#)
- [“Specifying Micro-Architecture for a Single Behavior” on page 8-96](#)
- [“Semi-Automatic Micro-Architecture” on page 8-97](#)

8.1 Resolving Functions

You have several choices when resolving functions:

- [“Inlining Functions” on page 8-3](#)
- [“Pipelining Functions” on page 8-12](#)
- [“Converting a Function to a Table Lookup” on page 8-15](#)
- [“Importing RTL IP into SystemC Designs” on page 8-16](#)

8.1.1 Inlining Functions

For examples of some of the steps in inlining functions, look in the following directory:

```
install_directory/share/ctos/examples/features/functions
```

Note Before running any CtoS examples, first review [“Setup for Examples” on page F-2](#).

Inlining is a technique in which the *complete body* of a function is inserted at the places where the function is called.

Important By default, CtoS inlines only functions that you specify. However, the optimization process sometimes automatically inlines a function.

This typically occurs when a function has only assignments, and the CtoS optimizer can propagate the assignments to the place where the function is called (thus, the function is no longer needed). This can happen when a reset member function assigns member variables to a constant.

The following sections provide detailed information about inlining:

- [“Simple Inlining Example” on page 8-4](#)
- [“Functions that Must be Inlined before Scheduling” on page 8-4](#)
- [“Deciding Whether to Inline other Functions” on page 8-6](#)
- [“Using the Call Graph Viewer for Inlining Decisions” on page 8-8](#)
- [“How to Inline a Function” on page 8-11](#)

8.1.1.1 Simple Inlining Example

Throughout the discussion on inlining, the following example will be used:

```
int subfunc(int a, int b, int c) {
    return (a - b) * c;
}

void DUT::proc() {
    int qnt;
    ...
    if(IN1.read() == true) {
        qnt = subfunc(D1.read(), D2.read(), C.read());
    }
    else {
        qnt = subfunc(D1.read(), D2.read(), 8);
    }
    ...
}
```

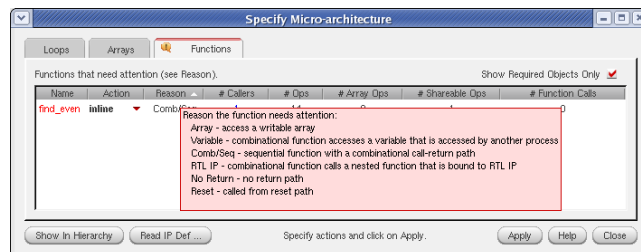
If inlining is applied to the function **subfunc**, the resulting program would look like this:

```
void DUT::proc() {
    int qnt;
    ...
    if(IN1.read() == true) {
        qnt = (D1.read() - D2.read()) * C.read();
    }
    else {
        qnt = (D1.read() - D2.read()) * 8;
    }
    ...
}
```

8.1.1.2 Functions that Must be Inlined before Scheduling

CtoS requires that the some functions *always* be inlined, or otherwise resolved, before scheduling. In the **Specify Micro-architecture** dialog, the **Reason** column gives you a shorthand description of the reason why each function listed must be inlined, or otherwise resolved. You can see a context menu for these shorthand descriptions if you hover over **Reason**, as shown in [Figure 8-1 on page 8-4](#). A more detailed description of each of these reasons is given on the next page.

Figure 8-1 Specify Micro-Architecture Dialog - Reasons for Inlining a Function



The following functions *must* be inlined before scheduling:

- A function that accesses a writable array (**Reason: Array**).

Alternatively, you could flatten the array by clicking on **Array** (in the CtoS GUI), which is a link to that specific array (see [“Flattening Arrays” on page 8-61](#)).

- A combinational function with I/O ops that access a net or a variable accessed by another process (**Reason: Variable**).
- A sequential function with a combinational call-return path (**Reason: Comb/Seq**).

This type of function *conditionally* requires more than one clock cycle to complete execution, that is, it has a path of execution that takes *more than one* clock cycle, as well as a path that can be executed within a *single* clock cycle.

For example, consider the following function:

```
int subfunc(int max_length, int *C, bool *valid) {
    int i;
    for(i = 0, *valid = false; i < max_length; i++) {
        if(C[i] % 2 == 0){ *valid = true; break; }
        wait();
    }
    return i;
}
```

This function may complete its execution in a *single* clock cycle – without executing the **wait** statement – if **max_length** is not greater than 0 or if **C[0] % 2** is equal to 0. Otherwise, it will take *at least one* clock cycle. A function of this type must always be inlined.

However, if for some reason you do not want to inline the function, and if it is acceptable to always spend at least one clock cycle to implement it, you could insert a **wait** statement so any executable path of the function includes at least one **wait** statement. In this example, a **wait** statement could be inserted just before the **return** statement at the end of the body, as follows:

```
int subfunc(int max_length, int *C, bool *valid) {
    int i;
    for(i = 0, *valid = false; i < max_length; i++) {
        if(C[i] % 2 == 0){ *valid = true; break; }
        wait();
    }
    wait();
    return i;
}
```

- A combinational function that calls a nested function bound to RTL IP (**Reason: RTL IP**).

See [“Importing RTL IP into SystemC Designs” on page 9-41](#).

- A function that does not have a return path, that is, has an infinite loop (**Reason: No Return**).
- A function called from the reset path (**Reason: Reset**).

All behavior that takes place on the reset path is implemented by the reset functionality of the flip-flops; therefore, no function calls are allowed on the reset path (that is, in a thread process, before the first **wait** statement). To address this, function calls on the reset path must be inlined or moved after the first **wait** statement in the source code.

8.1.1.3 Deciding Whether to Inline other Functions

This section provides some guidance for when, or when *not*, to inline functions not actually *required* to be inlined.

Note Functions *required* to be inlined were described in the previous section, “[Functions that Must be Inlined before Scheduling](#)” on page 8-4.

- “[Advantages of Inlining](#)” on page 8-6
- “[Disadvantages of Inlining](#)” on page 8-6
- “[Deciding whether to Inline Small Functions or Functions Called Only Once](#)” on page 8-7
- “[Deciding whether to Inline Complex Combinational Functions](#)” on page 8-7
- “[Deciding whether to Inline Complex Sequential Functions](#)” on page 8-8
- “[Considering Timing when Making Inlining Decisions](#)” on page 8-8

Advantages of Inlining

There are several advantages to inlining a function:

- The overhead of implementing the function calls can be avoided.
- The body of the function may be further optimized in the context of the caller.

In the “[Simple Inlining Example](#)” on page 8-4, the third argument of the function **subfunc** is used as an operand of a multiplication, but is set to **8** when the function is called inside the **else** clause.

Thus, CtoS could replace the multiplication with a bit-shift operation to realize a more efficient implementation. Such an optimization would not be possible if the function were not inlined.

Disadvantages of Inlining

There are several disadvantages to inlining:

- The complexity of the behavior could increase.

- The implementation could be more expensive.

For example, if a function has a large body and is called at many places, inlining the function could significantly increase the total number of operations or control states to be implemented.

Since the body of the function may be optimized in the individual contexts where it is called, CtoS may not be able to use common hardware resources to implement those operations after inlining.

Deciding whether to Inline Small Functions or Functions Called Only Once

If a function is called *only once*, in general, it is always a good idea to inline it.

Even if a function is called in multiple places, if it is a *small function* and does not require expensive hardware resources to implement, then inlining is usually beneficial.

Deciding whether to Inline Complex Combinational Functions

If a function is called *multiple times*, you should evaluate the potential increased complexity from inlining the function.

If the function is combinational, that is, its execution can always be completed in a single clock cycle, you should look primarily at the metrics for the operations and types of operations.

If you look at the **Summary Report** before and after the inlining of the function, and if the number of operations increases significantly, it is probably *not* a good idea to inline the function.

Similarly, compare the types and number of hardware resources in the **Op Constraint Viewer** to evaluate how inlining would affect the resources used in the implementation.

During RTL generation, inlining the combinational function causes it to be merged into the main module, while not inlining the combination function causes it to be represented as a separate module. One or more instances of this new module are then instantiated inside the main module.

Notes

- For the **Summary Report**, select **Window -> Report Summary**, or use the **report_summary** command ("[report_summary](#)" on page E-128).
- For the **Op Constraint Viewer**, select **View -> Resources**, or use the **report_resources** command ("[report_resources](#)" on page E-119).
- Additionally, you can pipeline a complex combinational functions. For more information on pipeline function, see "[Pipelining Functions](#)" on page 8-12.

Deciding whether to Inline Complex Sequential Functions

If a function is called *multiple times*, but is not combinational, then in addition to operations, you should consider the number of control states and the timing in the resulting implementation. If such a function is not inlined, CtoS will implement it with a separate process.

Therefore, if function **f** calls another function **g**, and if **g** is not inlined, the number of states required to implement **f** and **g** would be the sum of the states required for each. Conversely, if **g** is inlined at all the places in which **f** calls **g**, the number of states in the resulting implementation is proportional to the product of the number of states of **g** and the number of states of **f** at which **g** is called.

Thus, the inlining of sequential functions could rapidly increase the number of states, which is particularly true when a function transitively calls many sub-functions, and the sub-functions are inlined successively.

Again, you can use the **Summary Report** to see the number of states of the current design. By checking this before and after inlining a function, you can see if the number of states is increased by inlining.

During RTL generation, inlining the sequential function causes it to be merged into the main module, while not inlining the sequential function causes it to be represented as a separate always block in the main module.

Considering Timing when Making Inlining Decisions

Another aspect to consider with inlining is timing.

If a function is *sequential* and is *not* inlined, CtoS inserts logic into the calling function to transfer control to the called function and to wait for the called function to complete. The calling function resumes its execution after the called function completes its execution.

However, if the function *is* inlined, the body of the function is implemented in the context of the calling function. Thus, if some of the operations of the inlined function can be executed in earlier states, or if the execution of the others can be deferred to later states, CtoS can explore this flexibility to distribute these operations during scheduling. Therefore, inlining may be beneficial if this kind of exploration is important to achieve a desired implementation.

For a *combinational* function, even if it is *not* inlined, CtoS can schedule it without being subject to the original states at which it is called in the input SystemC program. Since it is not inlined, CtoS will not distribute the operations of the called function in the context of the caller, but CtoS *will* analyze a set of states of the caller at which the function can be executed and explore this flexibility during scheduling.

8.1.1.4 Using the Call Graph Viewer for Inlining Decisions

The **Call Graph** viewer, as shown in [Figure 8-2 on page 8-10](#), provides a better overview of source code than simply looking at source files. It can be very useful in deciding which functions should be inlined.

To display the **Call Graph** viewer, select the icon with *eyeglasses* at the left border of the CtoS GUI, or select **Window -> Call Graph**.

You can optionally display the viewer by right-clicking on a function call, function definition, or inlined function in **Function Calls** (in the **Micro-architecture** area), **Input Source**, or **Hierarchy Window**.

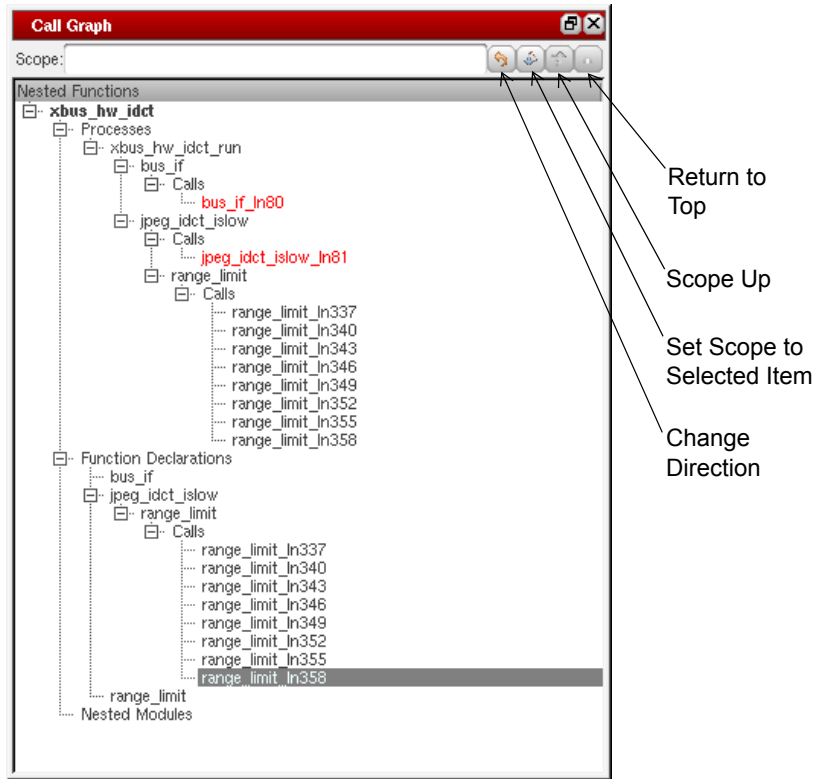
You can change the focus of the tree using the icons on the **Call Graph** viewer.

You can select an item in the tree and click **Set Scope to Selected Item**, which will display the selected item as the top left node in the tree. The **Scope** field shows the current scope. The **Scope Up** icon moves one level up, and the **Return to Top Scope** icon resets back to the top-level module.

The **Change Direction** icon toggles between the *Caller Graph* and the *Callee Graph*. You can change to *Callee Graph* only when the selected item is either a function declaration or a function call.

- The *Callee Graph* displays the functions called by the specified function, This is useful if you want to see where function **A** is called and decide whether to inline any or all of its calls.
- The *Caller Graph* displays the hierarchical modules of the current design, starting with the top module. For each module, the tree displays the behavior **Processes**, as well as the **Function Declarations** in this module.

Figure 8-2 Call Graph Viewer



In the *Caller Graph*, to examine a particular method, simply look for the method name in **Function Declarations** and expand the item to see the calls made inside this function. To see the functions called within the various processes, simply expand **Processes** to explore the call graph structure of the various processes. Function calls are aggregated by function declaration.

For example, if function **A** calls function **B** twice and function **C** multiple times, the tree looks like:

```
+ A
  + B
  + C
```

You can click on **B** to see the actual calls of **B** within **A**:

```
+ A
  + B
    + Calls
      B1
      B2
```


+ C

8.1.1.5 How to Inline a Function

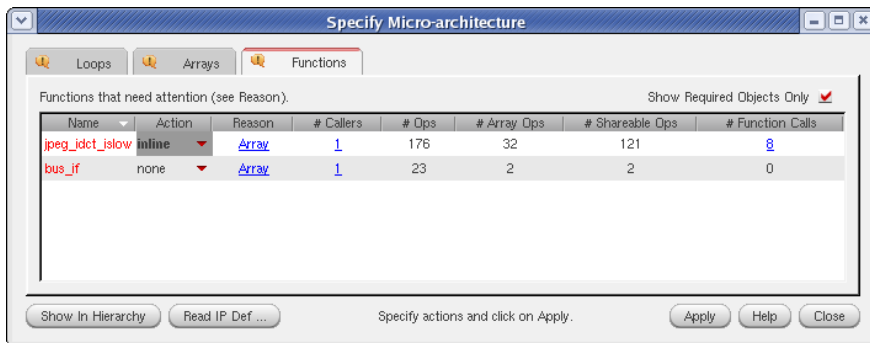
After you have built a design, if any functions *must* be resolved, the term **Functions** will be listed in red in the **Specify Micro-architecture** pane of the **Task Window**.

Double-click **Functions**, and the **Specify Micro-architecture** dialog, as shown in [Figure 8-3 on page 8-11](#), will be displayed.

Notes

- If you want to inline a function that is not *required* to be resolved (but could still benefit from being inlined), simply select **Edit -> Specify Micro-architecture** to display this dialog.
- You can also use the **inline** command (“[inline](#)” on page E-78) or **inline_calls** command (“[inline_calls](#)” on page E-79).

Figure 8-3 Specify Micro-architecture Dialog - Inlining a Function



In the dialog, select the function or operation you want to inline, select **Inline** from the **Action** column, continue with other actions, and, when you are done, click **Apply**.

- If the argument is a *function*, all calls to that function are inlined.
- If the argument is an *operation*, the operation must be for calling a function, and, in that case, the function is inlined only for that operation.

If you want to inline all function calls *inside* a behavior, module, or design to sequential functions contained in the specified behavior, module, or design, right-click the object, and select **Inline Calls to this Behavior**.

8.1.2 Pipelining Functions

Pipeline function is a CtoS built-in technique where a combinational function which executes in more than one clock cycle can have a pipelined implementation (similar to RTL-IP).

If the user determines that the combinational function executes in more than one clock cycle, the user can decide to map the function to a pipeline function implementation.

The user can specify a fix latency, or a min/max latency for the pipelined implementation. If no latency is specified, default values are assumed. Specifying a range of min/max latency gives the CtoS scheduler freedom to insert states to resolve memory contention and avoid scheduling with negative slack.

Note Specifying a range of latency may not find the minimum latency within that range that closes timing. You can instruct the scheduler to perform an exhaustive search for the minimal latency that closes timing within the user-specified min and max latency intervals by checking the **Extra Timing Effort** checkbox, as shown in [Figure 8-5 on page 8-15](#). This option may increase run time of the scheduler. The recommendation is to enable this option on a second run of the scheduler with a smaller latency interval narrowed down around latency value by the first run. For example, if latency *lat* is found, try +/- 5 states around *lat*.

The pipelined implementation of the function runs concurrent with its callers. Therefore, the user is required to verify that in the caller functions, the specified latency is available for the pipelined implementation to be feasible. In other words, the number of clock cycles between the pipelined function call and its use must equal the specified maximum latency.

The following sections provide detailed information about pipelining:

- [“Requirements for Pipelining a Function” on page 8-12](#)
- [“Simple Pipeline Example” on page 8-13](#)
- [“How to Pipeline a Function” on page 8-14](#)

8.1.2.1 Requirements for Pipelining a Function

A function can be pipelined if the function meets the following requirements:

- Is a combinational function.
- Is not bound to RTL-IP (that is, the **use_ip** command is not already executed).
- Is not marked as protocol constraint region.
- Should have a simple CFG with no fork or join nodes.
- Does not contain any multi-cycle ops.

- Does not have array accesses except read only. The read-only array should not be an external array or bound to vendor ram.
- Does not have IO accesses of type write.
- Does not contain loops. Any function internal loops must be unrolled to ensure fixed latency of the function and to enable pipelining.
- Does not contain labels.
- Is not called from a reset-less process or other combinational functions.

In addition to the above mentioned synthesizability check of the function itself, the following conditions must also be met:

- All functions called by this function (directly or indirectly):
 - Must not be a pipeline function.
 - Must not be bound to multi-cycle RTL-IP.
- The top caller process must be a thread. The pipeline function itself can be called from:
 - a thread process
 - a sequential function
 - a pipeline loop (with no stall, single, or multiple stalls)

8.1.2.2 Simple Pipeline Example

The following is an example of when to pipeline a combinational function:

```
main()
{
    ...
    int var = foo(inputA, inputB, inputC);

    wait(3);
    ...
    out = var;
    ...
}

int foo(int a, int b, int c)
{
    return (a + b * c);
}
```

In this example, the combinational function **foo()** can have a pipeline function implementation. The latency of the function is between **1** and **3** given that there are 3 **wait()** statements between the function call and the use of its result. Therefore, you can pipeline the function using the following command:

```
pipeline_function -latency {1 3} [find -behavior foo]
```

8.1.2.3 How to Pipeline a Function

After you have built a design, if any functions *must* be resolved, the term **Functions** will be listed in red in the **Specify Micro-architecture** pane of the **Task Window**.

To pipeline a function:

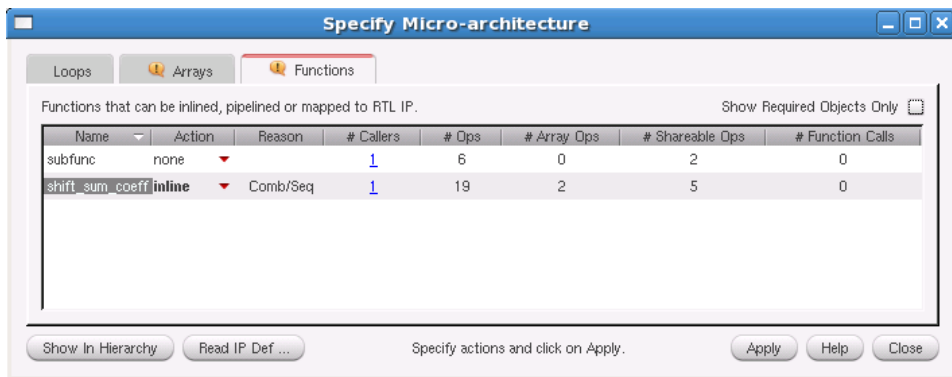
1. Double-click the **Functions** tab that is listed in red in the **Specify Micro-architecture** pane of the **Task Window**.

The **Specify Micro-architecture** dialog is displayed, as shown in [Figure 8-4](#).

Note You can also Pipeline a function by one of the following ways:

- Select **View > CDFG > Combination Functions > *Function name***. Then, right-click anywhere in the **CDFG** pane, and click **Pipeline**. The **Specify Micro-architecture** dialog is displayed, as shown in [Figure 8-4](#).
- Use the **pipeline_function** command. For more information, see “[pipeline_function](#)” on page E-91.

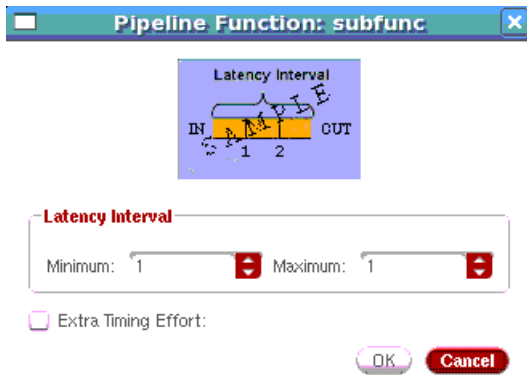
Figure 8-4 Specify Micro-architecture Dialog - Pipelining a Function



2. In the **Specify Micro-architecture** dialog, select the function you want to pipeline, select pipeline from the **Action** column.

The **Pipeline Function** dialog is displayed, as shown in [Figure 8-5](#).

Figure 8-5 Pipeline Function Dialog



3. In the **Pipeline Function** dialog:
 1. Specify the minimum and maximum latency.
 2. Select the **Extra Timing Effort** checkbox, if you want the scheduler to perform an exhaustive search for the minimal latency that closes timing within the user-specified min and max latency intervals.
 3. Click **OK**.
4. In the **Specify Micro-architecture** dialog, click **Apply**.

The combinational function is converted into a pipeline implementation, which is represented by a thread process. The process is reset-less. The specification behavior DFG is encapsulated inside a while true loop. The while true loop is pipelined using **II** set to **1** and **LI** set to the user specified latency.

8.1.3 Converting a Function to a Table Lookup

You can convert the implementation of a combinational function into a *table lookup*. The function's inputs drive the address of the array, and its outputs are driven by the values read out of the array.

This conversion may provide a more efficient implementation and may enable additional optimizations around the function call by propagating additional constants through the function.

For example, this conversion may be useful for functions where timing is not met, because it can improve the timing (and area) for arithmetic optimizations.

The function to be converted must meet the following requirements:

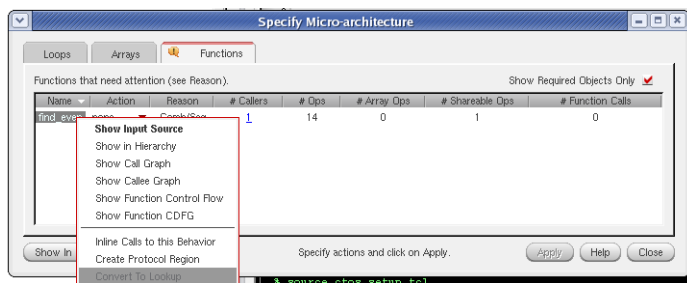
- It must be a function, not a process.
- It must be combinational, that is, no states.
- It must not access writable arrays.

- It must not access any global variables.
- The sum of the widths of its inputs must not be greater than 16.

Calls to the converted function are not inlined; the function remains, containing only the array lookup. You may inline all calls to the function, if desired, after the conversion is complete.

To perform this task, after you have built a design, if there are any functions that must be resolved, the term **Functions** will be listed in red in the **Specify Micro-architecture** pane of the **Task Window**. Double-click **Functions**, and the **Specify Micro-architecture** dialog, as shown in [Figure 8-6 on page 8-16](#), will be displayed. Right-click the function, and select **Convert to Lookup**.

Figure 8-6 Specify Micro-Architecture Dialog - Convert to Lookup



Notes

- If you want to convert a function that is not *required* to be resolved (but could still benefit from being converted), simply select **Edit -> Specify Micro-architecture** to display this dialog.
- You can also use the `convert_to_lookup` command ([“convert_to_lookup” on page E-40](#)).

8.1.4 Importing RTL IP into SystemC Designs

CtoS provides an *RTL IP* feature, in which combinational functions in the SystemC specification of a design are implemented using an existing RTL design. For a complete, detailed description of this feature, including the steps for how to use it, see [“Importing RTL IP into SystemC Designs” on page 9-41](#).

8.2 Resolving Loops

For examples of some of the steps in dealing with loops, look in the following directory:

```
install_directory/share/ctos/examples/features/loops
```

Note Before running any CtoS examples, first review [“Setup for Examples” on page F-2](#).

You could also review the example in [“Micro-Architectural Exploration Example” on page C-1](#).

The following sections are intended to help you resolve loops in your design:

- [“How Combinational Loops Are Determined in CtoS” on page 8-17](#)

- [“Unrolling Loops” on page 8-17](#)
- [“Breaking Combinational Loops” on page 8-22](#)
- [“Deciding between Unrolling and Breaking Loops” on page 8-24](#)
- [“Pipelining Loops” on page 8-26](#)

8.2.1 How Combinational Loops Are Determined in CtoS

Combinational loops must be eliminated before synthesis because they cannot be implemented in hardware.

A loop is said to be *combinational* if at least one program execution from the top to the bottom of the loop does not include waiting for a clock edge.

CtoS considers a loop to be combinational if it cannot prove at compile time that any execution from the top to the bottom of the loop takes more than one clock cycle.

For example, consider the following program:

```
i = 0;
do {
    for( j=0; j< i+1 && exit_condition ; j++) {
        wait();
    }
    i++;
} while(i < 5);
```

If CtoS is unable to prove that the execution from the outer loop to the inner loop will always go inside the inner loop, CtoS will report the outer loop to be combinational (even though the inner loop always takes at least one clock cycle in any execution).

Note After you have built your design, if the design contains any combinational loops, they are displayed in the CtoS GUI **Task Window** in the **Specify Micro-architecture** pane ([“Task Window” on page 6-44](#)). You can also use the **find_combinational_loops** command ([“find_combinational_loops” on page E-63](#)).

8.2.2 Unrolling Loops

Unrolling a loop is the process by which the statements in a loop are copied for as many times as they would have been executed in a design.

CtoS provides three options for unrolling loops:

- [“Complete or Full Unrolling” on page 8-18](#)
- [“Partial Unrolling” on page 8-19](#)
- [“Body Unrolling” on page 8-21](#)

8.2.2.1 Complete or Full Unrolling

In *complete* or *full* unrolling, the loop is replaced with the complete number of copies of the loop.

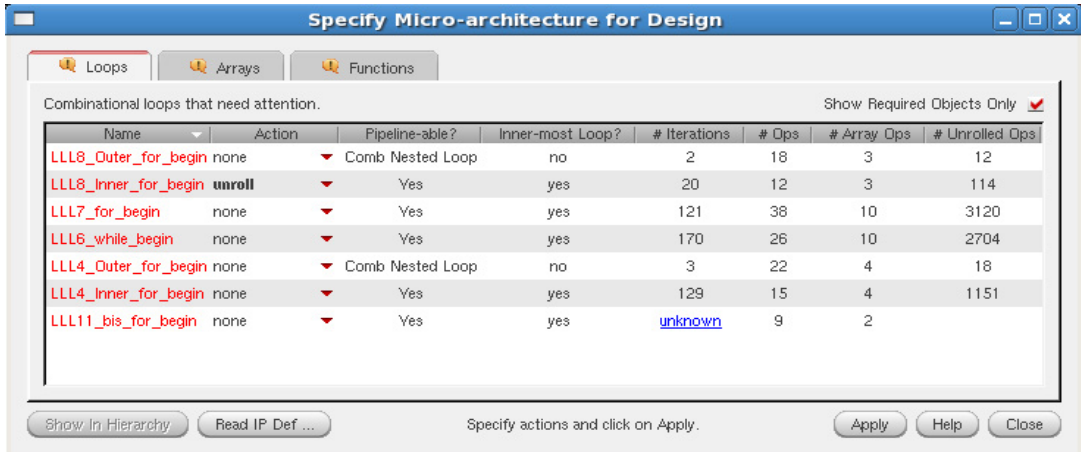
For example, consider the following program:

```
i = 0;
do {
    x1 = X[i];
    x2 = Y[i];
    diff = (x1 < x2) ? x2 - x1 : x1 - x2;
    A[i] = x1 * diff + x2;
    i++;
} while(i < 5);
```

If the loop in this program were *unrolled*, you would get a program made of five copies of the body of the loop, as follows:

```
// The first copy
x1 = X[0];
x2 = Y[0];
diff = (x1 < x2) ? x2 - x1 : x1 - x2;
A[0] = x1 * diff + x2;
// The second copy
x1 = X[1];
x2 = Y[1];
diff = (x1 < x2) ? x2 - x1 : x1 - x2;
A[1] = x1 * diff + x2;
...
// The fifth copy
x1 = X[4];
x2 = Y[4];
diff = (x1 < x2) ? x2 - x1 : x1 - x2;
A[4] = x1 * diff + x2;
```


Figure 8-7 Specify Micro-Architecture Dialog - Unrolling a Loop



To unroll a loop, you use the **Specify Micro-architecture** dialog, as shown in [Figure 8-7 on page 8-19](#).

After you have built a design, if any loops *must* be resolved, the term **Combinational Loops** will be listed in red in the **Specify Micro-architecture** pane of the **Task Window**.

Double-click **Combinational Loops**, and the **Specify Micro-architecture** dialog is displayed. In this dialog, click the loop, and select **unroll** as the **Action** to unroll the loop with the default values. (To use values other than the defaults, see the following section, [“Partial Unrolling” on page 8-19](#).)

To resolve a loop that is not *required* to be resolved (but could still benefit from being changed), simply select **Edit -> Specify Micro-architecture**, from the menu bar, to display this dialog.

8.2.2.2 Partial Unrolling

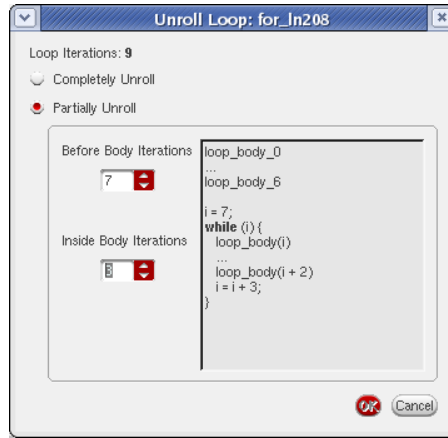
In *partial unrolling*, a loop is unrolled only partially, while the remaining executions are kept in a loop.

For the example in the previous section ([“Complete or Full Unrolling” on page 8-18](#)), if the loop were unrolled only once, the resulting program would have one copy of the body of the loop, followed by a loop with four iterations starting from the loop index equal to 1, as follows:

```
// The first copy
x1 = X[0];
x2 = Y[0];
diff = (x1 < x2) ? x2 - x1 : x1 - x2;
A[0] = x1 * diff + x2;
// The original loop, with four iterations
i = 1;
do {
    x1 = X[i];
```

```
x2 = Y[i];  
diff = (x1 < x2) ? x2 - x1 : x1 - x2;  
A[i] = x1 * diff + x2;  
    i++;  
} while(i < 5);
```

Figure 8-8 Unroll Loop Dialog



When unrolling a loop, the default is to **fully** unroll the loop, with **0** iterations unrolled before the loop and **1** original loop body in the loop after unrolling.

To *partially* unroll a loop, right-click the loop (either from the **Specify Micro-architecture** dialog or directly in the **Hierarchy Window**) and select **Unroll**.

In the **Unroll Loop** dialog, as shown in [Figure 8-8 on page 8-20](#), select **Partially Unroll** and specific *before* and *inside* iteration numbers.

In this dialog, you will see one of the following scenarios:

- If CtoS can identify the number of iterations (fewer than 100), that number is specified as a label, and the default option is **Completely Unroll**.
- If CtoS identifies the loop to have no exit edges, the iteration count is specified as **Infinite**, and the **Completely Unroll** option is disabled.
- If CtoS determines the loop is either unbounded or bounded with more than 100 iterations, you can specify the number of iterations to unroll the loop completely. If you click **Completely Unroll**, you are prompted to specify the number of iterations, and if this number is valid, the **OK** button is enabled. You may also select the **Cancel** button, which then lets you **Partially Unroll** the loop.

Notes

- See also “[Advantages of Partial Unrolling Combined with Breaking](#)” on page 8-25.
- You can also use the **unroll_loop** command (“[unroll_loop](#)” on page E-151).

8.2.2.3 Body Unrolling

In *body unrolling*, the body of a loop is copied for a specified number of times while the execution is still kept in a loop.

For the previous example, after you applied partial unrolling, if you then applied body unrolling to the remaining loop, with the number of copies equal to two, the resulting program would look like the following:

```
// The first copy
x1 = X[0];
x2 = Y[0];
diff = (x1 < x2) ? x2 - x1 : x1 - x2;
A[0] = x1 * diff + x2;
// The original loop, with four iterations
i = 1;
do {
// The first copy of the body unrolling
x1 = X[i];
x2 = Y[i];
diff = (x1 < x2) ? x2 - x1 : x1 - x2;
A[i] = x1 * diff + x2;
i++;
// The second copy of the body unrolling
if(i >= 5) break;
x1 = X[i];
x2 = Y[i];
diff = (x1 < x2) ? x2 - x1 : x1 - x2;
A[i] = x1 * diff + x2;
i++;
} while(i < 5);
```

After applying body unrolling, CtoS automatically analyzes the unrolled body of the loop to simplify the operations

For example, in the previous code, the statement **if(i >= 5) break;** would be eliminated, as CtoS would determine that the value of **i** would be either 2 or 4, and the condition would always be false.

Notes

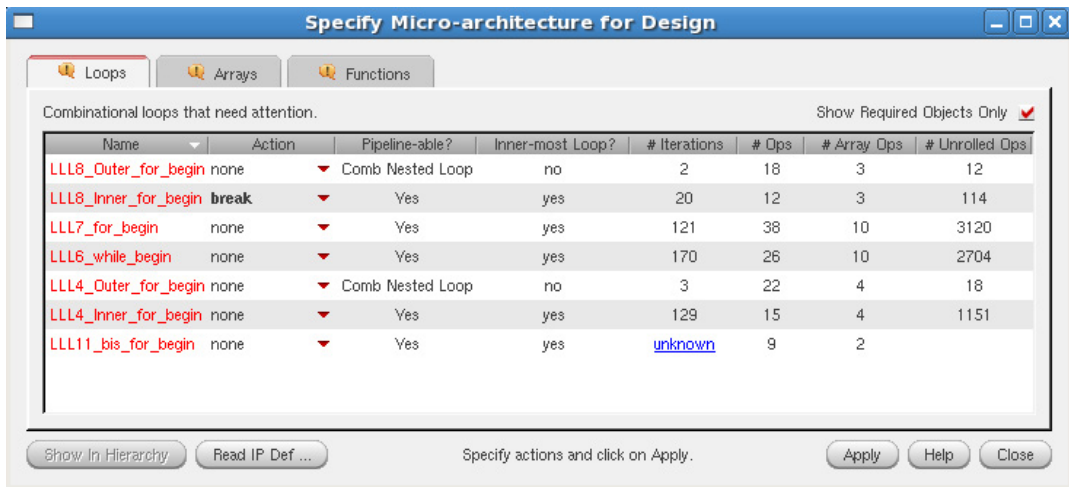
- See also “[Advantages of Body Unrolling Combined with Breaking](#)” on page 8-25.
- You can also use the **unroll_loop** command (“[unroll_loop](#)” on page E-151).

8.2.3 Breaking Combinational Loops

To break a combinational loop, CtoS inserts states so that each iteration of the loop will take at least one clock cycle. For example, for the loop shown in “[Complete or Full Unrolling](#)” on page 8-18, to insert one state, CtoS may replace the loop with the program shown below:

```
i = 0;
do {
    wait();
    x1 = X[i];
    x2 = Y[i];
    diff = (x1 < x2) ? x2 - x1 : x1 - x2;
    A[i] = x1 * diff + x2;
} while(i < 5);
```

Figure 8-9 Specify Micro-Architecture Dialog - Breaking a Loop



To break a loop, you use the **Specify Micro-architecture** dialog, as shown in [Figure 8-9 on page 8-22](#).

After you have built a design, if any loops *must* be resolved, the term **Combinational Loops** will be listed in red in the **Specify Micro-architecture** pane of the **Task Window**.

Double-click **Combinational Loops**, and the **Specify Micro-architecture** dialog is displayed. In this dialog, click on the loop, and select **break** as the **Action** to break the loop.

To resolve a loop that is not *required* to be resolved (but could still benefit from being changed), simply select **Edit -> Specify Micro-architecture**, from the menu bar, to display this dialog.

Important

- Review the restrictions for the specified behavior in “[Limitations When Creating States in Combinational Functions](#)” on page 11-7.
- It is recommended that you create a state, instead of breaking a loop, if you want to specify the *exact* place to break the loop; see “[Creating and Managing States](#)” on page 11-6 for more detail.
- You can also use the **break_combinational_loop** command (“[break_combinational_loop](#)” on page E-29).

8.2.4 Deciding between Unrolling and Breaking Loops

The following sections provide some guidance for deciding how to resolve combinational loops in a design:

- [“Advantages of Unrolling Loops” on page 8-24](#)
- [“Disadvantages of Unrolling Loops” on page 8-24](#)
- [“Advantages of Breaking Loops” on page 8-25](#)
- [“Disadvantages of Breaking Loops” on page 8-25](#)
- [“Advantages of Body Unrolling Combined with Breaking” on page 8-25](#)
- [“Advantages of Partial Unrolling Combined with Breaking” on page 8-25](#)

8.2.4.1 Advantages of Unrolling Loops

The advantage of completely unrolling a loop is it eliminates the loop; therefore, the operations inside the loop in the original program can be synthesized together with the rest of the program.

In general, this facilitates further optimization of those operations.

It also allows the operations to be scheduled in the set of states defined in the rest of the program.

8.2.4.2 Disadvantages of Unrolling Loops

The disadvantage of unrolling a loop is it can increase the number of operations, because operations inside the loop are copied many times.

If there are too many operations, it may be difficult to schedule all operations in the set of states available for the behavior.

Also, although the operations originate in the same loop, since they are optimized in the context of the program after unrolling, it may be difficult to share hardware resources to implement them.

The area of the resulting implementation might thus be larger than if the loop were not unrolled.

Another consideration is that CtoS can unroll a loop only if it can determine an upper bound of the number of iterations of the loop at compile time.

If it cannot prove the bound, you will get an error indicating that the specified loop cannot be unrolled.

8.2.4.3 Advantages of Breaking Loops

If a loop cannot be unrolled, breaking the loop is another option.

Also, if the increased number of operations created by unrolling would cause a problem, breaking the loop should be considered.

In breaking a loop, CtoS simply inserts states in the loop.

Since the operations inside the loop are implemented using the states inserted in the loop, this option can always be applied, and there is no problem about increasing the number of operations.

8.2.4.4 Disadvantages of Breaking Loops

The disadvantage of breaking a loop is more states are introduced, increasing the latency required to implement the behavior.

Also, since states are inserted in the body of a loop, a corresponding number of clock cycles is consumed for executing each iteration of the loop.

This is problematic if there are only a small number of operations, which can all be implemented with a fraction of the clock period, in the body of the loop.

The rest of the clock period would best be used to implement other operations, but this cannot be exploited, as operations in the body of a loop must be implemented within the loop boundary.

8.2.4.5 Advantages of Body Unrolling Combined with Breaking

Body unrolling can be an effective option, and it can be used together with the breaking of a loop.

For example, if the given clock period is adequate to implement all of the operations from two consecutive iterations of a loop, you can apply body unrolling to copy the operations of the body of the loop twice.

If you then break the resulting loop by inserting a state, the resulting implementation will execute the operations from two iterations of the original loop within a single clock cycle.

8.2.4.6 Advantages of Partial Unrolling Combined with Breaking

Partial unrolling can also be used with the breaking of a loop.

Unlike complete unrolling, the number of operations introduced in the resulting implementation is controlled by adjusting the amount of unrolling applied to the loop.

Of course, since both body unrolling and partial unrolling are related to complete unrolling, they inherit the same advantages and disadvantages as complete unrolling.

8.2.5 Pipelining Loops

For an example of some of the steps in pipelining loops, look in the following directory:

`install_directory/share/ctos/examples/features/loops/pipelining`

Note Before running any CtoS examples, first review “[Setup for Examples](#)” on page F-2.

For a known limitation, see “[Pipelined Loops with Reads and Writes to Same Array](#)” on page 1-6.

The following sections describe loop pipelining in detail:

- “[Introduction to Loop Pipelining](#)” on page 8-26
- “[Requirements for Loops to be Pipelined](#)” on page 8-29
- “[Pipelining Trade-Offs and Caveats](#)” on page 8-30
- “[How to Pipeline a Loop](#)” on page 8-32
- “[Using the Pipeline View](#)” on page 8-36
- “[Resolving Pipelining Scheduling Failures](#)” on page 8-42
- “[Using the report_schedule Command with Pipelining](#)” on page 8-49
- “[Stalling/Flushing a Pipelined Loop](#)” on page 8-49
- “[Stalling in Loop that Terminates](#)” on page 8-54
- “[Behavior of Inputs and Outputs of a Pipelined Loop](#)” on page 8-57
- “[Pipeline Object Names](#)” on page 8-60

8.2.5.1 Introduction to Loop Pipelining

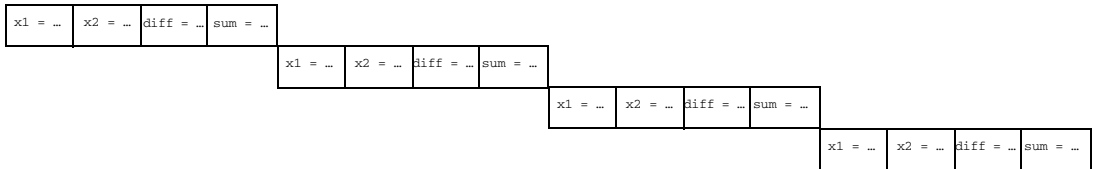
Loop pipelining is a technique in which the executions of the operations in a loop are *overlapped* in the resulting implementation. This means that an iteration of a loop can be started before the previous iteration has completed. This technique is a way to improve *throughput* – the rate at which inputs are sampled and outputs are produced.

For example, consider the following program:

```
void DUT::proc() {
    ...
    for(i=0; i<32; i++){
        x1 = IN.read();
        wait();
        x2 = IN.read();
        wait();
        diff = (x1 < x2) ? x2 - x1 : x1 - x2;
        sum = sum + x1 * diff + x2;
    }
    OUT.write(sum)
```


}

Figure 8-10 Loop without Pipelining

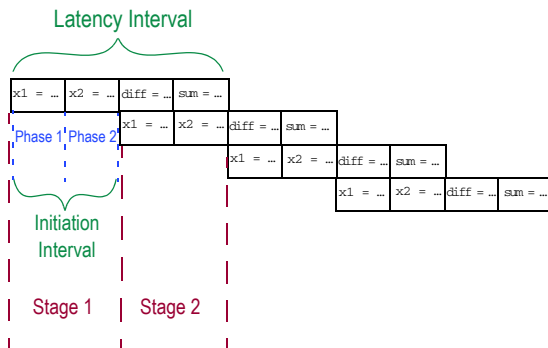


The implementation of the loop *without pipelining*, reads data from port **IN** *only* in the first two clock cycles for each iteration, while the other two clock cycles are used *only* for executing the last two statements of the loop, as shown in [Figure 8-10](#) on [page 8-27](#).

Loop pipelining would allow the implementation to access the port in *every* clock cycle.

As shown in [Figure 8-11](#) on [page 8-27](#), if loop pipelining were applied – using four clock cycles to execute each iteration of the loop – the resulting implementation would overlap the executions of two successive iterations.

Figure 8-11 Loop with Pipelining



Each row corresponds to the execution of a single iteration of the loop, and each column corresponds to a single clock cycle. The executions of the operations in the loop are *overlapped* – the operations of two successive iterations of the loop are executed in a single clock cycle.

Therefore, while the execution of a single iteration of the loop takes four clock cycles in either case, the pipelined implementation will *double* the rate at which port **IN** is accessed.

Pipelining is often not only *advantageous* to improve the performance of an implementation, it is sometimes *necessary* to maintain the protocol of the input and output operations defined in the rest of the system.

In the previous example, the module that sends data to port **IN** may have a requirement that data be sent every two clock cycles.

If the implementation of the loop takes more than two clock cycles to execute the operations of the last two statements, pipelining would then be *required* to satisfy this requirement.

Initiation Interval (II) and Latency Interval (LI)

Figure 8-11 on page 8-27 also illustrates the concepts of *initiation interval* (II) and *latency interval* (LI) for a pipelined loop.

The **II** is the number of clock cycles that elapse between two executions of the same operation – in this example, the **II** is two clock cycles, because the same operation (**x1=IN.read();**) is executed every two cycles.

The **LI** is the number of clock cycles required to execute the operations of a single iteration of the loop – in this example, the **LI** is four clock cycles.

Using II and LI to Compute Number of Stages

A single **II** is called a *pipeline stage*, and the number of stages is the *ceiling* of **LI/II**:

$$\text{number\_of\_stages} = \lceil \text{LI/II} \rceil$$

Using II and LI to Compute Number of Cycles

The number of cycles taken by a pipelined loop can be computed using the following formula:

$$\text{number\_of\_cycles} = (\text{number\_loop\_iterations} - 1) * \text{II} + \lceil \text{LI/II} \rceil * \text{II}$$

where:

$$(\text{number\_loop\_iterations} - 1) * \text{II}$$

shows the number of cycles
to initiate the first N - 1
pipelined executions.

$$+ \lceil LI/II \rceil * II$$

shows the number of cycles to complete the last execution, including possibly some empty phases at the end that are required to complete the last stage.

8.2.5.2 Requirements for Loops to be Pipelined

In order for a loop to be pipelined, it must meet the following criteria:

Important Ctos will check some of these conditions immediately when you pipeline a loop, but some will be checked only by the CtoS scheduler.

- The loop cannot be part of the reset path. This means there always must be at least one wait() before the loop.
- The loop must be in a thread or a sequential function called from a thread. (Loops in SC_METHODs cannot be pipelined).
- The loop must be an inner loop, that is, it must not contain another loop, other than a stall loop (multiple stall loops are allowed if **enable_multiple_stalls** is **true**); see [“Stalling/Flushing a Pipelined Loop” on page 8-49](#).
- If the loop calls a function, the function must either be purely combinational (no wait() statements and no I/O) or have been inlined. Pipelined functions are allowed.
- The loop must not intersect regions that have been preserved or for which a protocol region or latency constraint region has been created. A protocol region is allowed in the loop, if it is fully contained in the pipelined loop.

Note See [“Constraining ops to Edges” on page 11-17](#), [“Creating Protocol Regions” on page 11-14](#), and [“Constraining Latency” on page 11-12](#).

- If the loop contains a **break** statement, the following restrictions apply:
 - It cannot have any complex conditional breaks; for example, the following is *not* supported:

```
if (a) { wait(); if (b) break; }
```

However, the following *is* supported:

```
if (a) { if (b) break; }
```

- It cannot have any **wait** statements in conditional breaks if there is more than one loop exit. For example, the following is *not* supported:

```
if (a) { wait(); break; }
if (b) { break; }
```

The loop condition counts as an exit, so this is also *not* supported:

```
while (c) {
    ...
}
```

```
        if (a) { wait(); break; }  
    }
```

The following form *is* supported:

```
while (1) {  
    ...  
    if (a) { wait(); break; }  
}
```

- Labels may be used in a loop to be pipelined. However, due to the nature of pipelining, such labels will be removed during the **pipeline_loop** command..

The **Loops** tab in the **Specify Micro-architecture Manager** dialog lists all the loops in a design. The **Pipeline-able?** column lists "Yes" if a loop can be pipelined or a message indicating why a loop cannot be pipelined. Moving the cursor over the message shows a tooltip why the loop cannot be pipelined. For a complete list of reasons why a loop cannot be pipelined, see [Table 8-1](#) in section “How to Pipeline a Loop” on page 8-32. See also “[Pipelining Functions](#)” on page 8-12.

8.2.5.3 Pipelining Trade-Offs and Caveats

You should consider the following trade-offs and caveats when pipelining a loop.

Parallelism and Increased Area

While loop pipelining can improve performance, it may require more area if pipelining allows more operations to be executed in parallel; therefore, more hardware resources may be used simultaneously in the same clock cycles.

Relaxed Constraints and Reduced Area

For a given throughput constraint, a pipelined loop may require less area than a given non-pipelined loop, because it reduces performance constraints, thus allowing smaller implementations for resources.

Branching Structure and Fixed Latency

The improvement of performance is also subject to the control structure of the operations in a loop. If a loop has many conditional branches, and if the number of clock cycles significantly varies depending on the branches to be taken, the performance gain may not be significant with respect to the non-pipelined implementation.

This is due to the fact that CtoS implements the pipeline with the same number of clock cycles for all branches. This is particularly true if the branches with fewer numbers of clock cycles are more likely to be executed than those with larger numbers of clock cycles. Therefore, when loop pipelining is applied, you must analyze the resulting implementation to confirm any benefit.

In the following three examples, note that this affects latency only, not throughput, because throughput depends only on the II, and not on the control structure.

Example 1

In this example, assuming that neither **-expand_before** nor **-expand_after** has been specified (that is, no states have been inserted between **in** and **out**), the latency of the common case is not changed, because the common case output is produced in stage 2.

```
pipeline: while (1) {
    ... = in;
    if (common_case) {
        // simple computation that fits in 1 stage
        wait();
        out = ...;
    } else {
        // complex computation that fits in 10 stages
        wait(10);
        out = ;
    }
}
```

Example 2

In this example, assuming the expansion point is set between **in** and **out** (using either **-expand_before out** or **-expand_after in**), the latency of the common case *does* change, because the states are always inserted before *both* output operations:

```
pipeline: while (1) {
    ... = in;
    if (common_case) {
        // simple computation that fits in 1 stage
        wait();
        out = ...;
    } else {
        // complex computation that fits in 10 stages
        wait();
        out = ;
    }
}
```

Example 3

In this example, the latency of the common case is *always* increased, whether or not the expansion point is specified, because the paths are equalized in latency, always taking the longest one between **in** and **out**:

```
pipeline: while (1) {
    ... = in;
    if (common_case) {
```

```

        // simple computation that fits in 1 stage
        wait();
    } else {
        // complex computation that fits in 10 stages
        wait(10);
    }
    out = ;
}

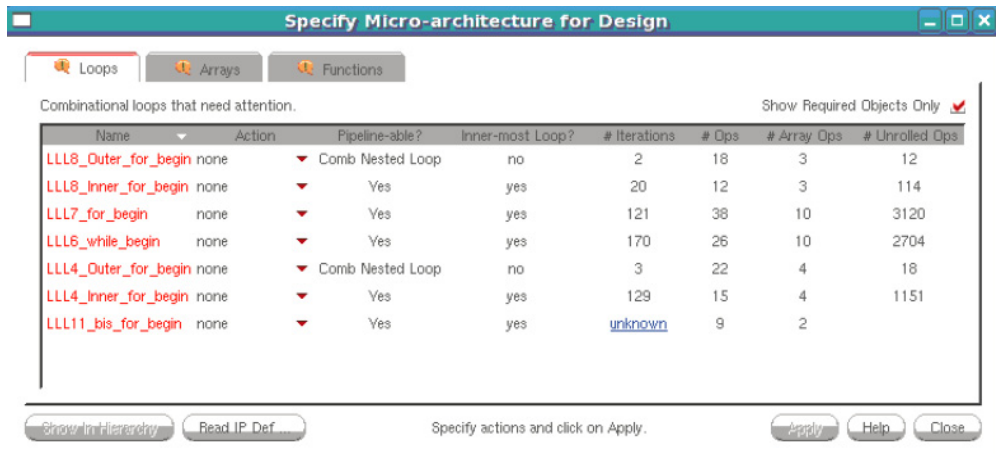
```

8.2.5.4 How to Pipeline a Loop

If you know that a particular loop should be pipelined, you can navigate to that loop in the **Hierarchy Window** or node in the **CDFG** and click **Pipeline ...** in the context menu. The **Pipeline Loop** dialog is displayed and you can specify the options for pipelining.

To list the loops that can be pipelined, click the **Edit > Specify Micro-architecture ...** menu item. The **Specify Micro-architecture** dialog is displayed, as shown in [Figure 8-12](#).

Figure 8-12 Specify Micro-architecture Dialog - Pipelining a Loop

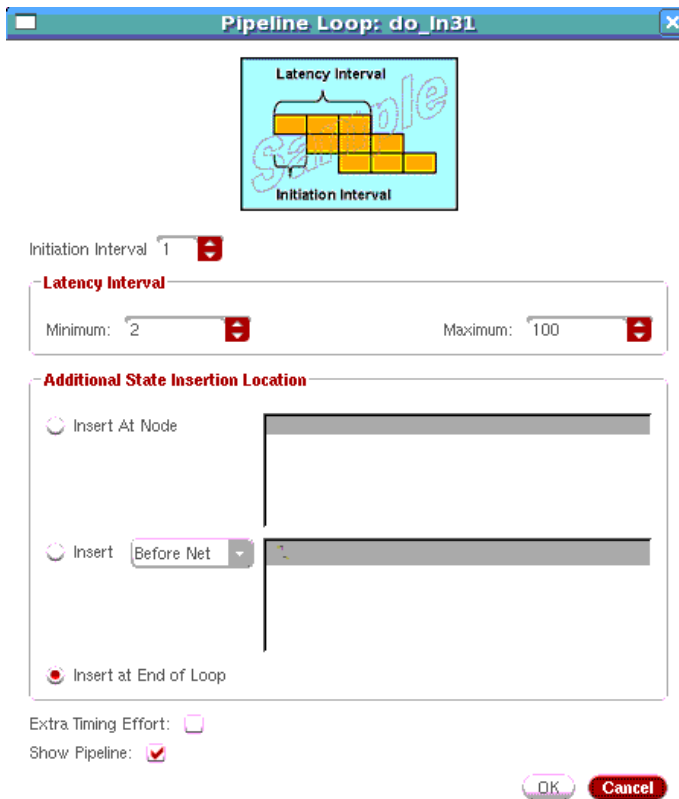


The **Loops** tab on the **Specify Micro-architecture** dialog has a column named **Pipeline-able?**, which for each loop specifies whether it can be pipelined or the reason it cannot be pipelined. The entry is “Yes” if it can be pipelined and you can select the specific pipeline action from the **Action** column. If a loop is not pipeline-able, the entry will be a short hand description of the reason it cannot be pipelined. If you hover your mouse over the reason, the tooltip will show a more detailed description. In the example, **LLL8_Inner_for_begin** can be pipelined while **LLL8_Outer_for_begin** cannot be pipelined because it contains a nested loop.

For a complete list of reasons why a loop cannot be pipelined, see [Table 8-1](#). Note that these restrictions are further explained in “Requirements for Loops to be Pipelined” on [page 8-29](#).

The **Pipeline Loop** dialog, as shown in [Figure 8-13](#), lets you specify options for the pipeline.

Figure 8-13 Pipeline Loop Dialog



The **Pipeline Loop** dialog sets up the specified loop for pipeline scheduling and lets you select the number of states to be used as an **II** and for the minimum and maximum **LI**. Specifying a range of minimum to maximum **LI** gives the CtoS scheduler freedom to insert states to resolve memory contention and avoid scheduling with negative slack.

Specifying the range of latency may not find the minimum latency within that range that closes timing. You can instruct the scheduler to perform an exhaustive search for the minimal latency that closes timing within the user-specified minimum and maximum latency intervals by checking the **Extra**

Timing Effort checkbox. This option may increase run time of the scheduler. The recommendation is to enable this option on a second run of the scheduler with a smaller latency interval narrowed down around latency value by the first run. For example, if latency *lat* is found, try +/- 5 states around *lat*.

You can also choose **Additional State Insertion Location**, as follows:

- **Insert at Node:** The label node at which CtoS is allowed to insert additional states, either as required by the specified minimum latency interval, or as determined by the scheduler in order to satisfy timing constraints. The expansion point is set between the last DFG op before the label and the first DFG op after the label. The label node must not be inside a stall loop or inside a conditional statement.
- **Insert Before Net:** The first access to any of the specified nets will be the expansion point at which CtoS will insert additional states in the loop to increase latency. If the access is inside the stall loop, the insertion point will be before the stall loop.
- **Insert After Net:** The last access to any of the specified nets will be the expansion point at which CtoS will insert additional states in the loop to increase latency. If the access is inside the stall loop, the insertion point will be after the stall loop.
- **Insert at End of Loop:** For pipelines that do not write results to I/O signals but only compute values to be used by other operations within the same behavior, CtoS may insert states at the end of the pipelined loop.

When you click the **OK** button, if you have selected the **Show Pipeline** checkbox, you will see the results of pipelining in the **Pipeline View**, as described in the following section, [“Using the Pipeline View” on page 8-36](#).

Notes

- For a pipelining example, see [“Specifying the Pipelining Constraint for the Loop \(Option 3\)” on page C-10](#).
- You can also use the **pipeline_loop** command ([“pipeline_loop” on page E-92](#)).

Table 8-1 Reasons Why a Loop Cannot be Pipelined

Reason	Description
Yes	Can be pipelined
rtl ip	Loop is within behavior bound to an RTL IP def
ctos_dont_touch	Loop is within behavior that is marked with pragma ctos_dont_touch
edge constraint	Loop contains custom op which is constrained to an op or an edge
	Loop contains edges with constraints

Table 8-1 Reasons Why a Loop Cannot be Pipelined

Reason	Description
nested loop	Loop has custom op of behavior which is already pipelined
	Loop has a nested pipelined loop
	Loop has deeply nested stall loops
	Loop contains combinational loops
	Loop contains stall loop that is not a while loop
	Loop contains stall loop that has loop variables (counters) or compute values in the loop body
	Loop contains stall loop that has memory op
	Loop contains stall loop that has op with latency > 0
	Loop has stall loop with multiple states
nested seq func	Loop calls a sequential function
in pipeline	Loop is in a pipelined loop
SC method	Loop cannot be inside an SC_METHOD
reset path	Loop intersects with the reset path
resetless func	Loop is contained in sequential function called by resetless process
resetless stall	Loop is within resetless process with stall loop
exit	Loop has multiple exits and wait on exit edge
	Loop has mismatched exits. All exits must go directly to the same join node.
	Loop has stall loop with multiple exits
region constraint	Loop contains edge which is part of a preserve region
	Loop overlaps with latency constraint
	Loop overlaps with protocol region
	Loop writes to net that is floating region
	Loop overlaps with protocol region

Table 8-1 Reasons Why a Loop Cannot be Pipelined

Reason	Description
protocol	Loop contains non-stallable protocol region that contains state

8.2.5.5 Using the Pipeline View

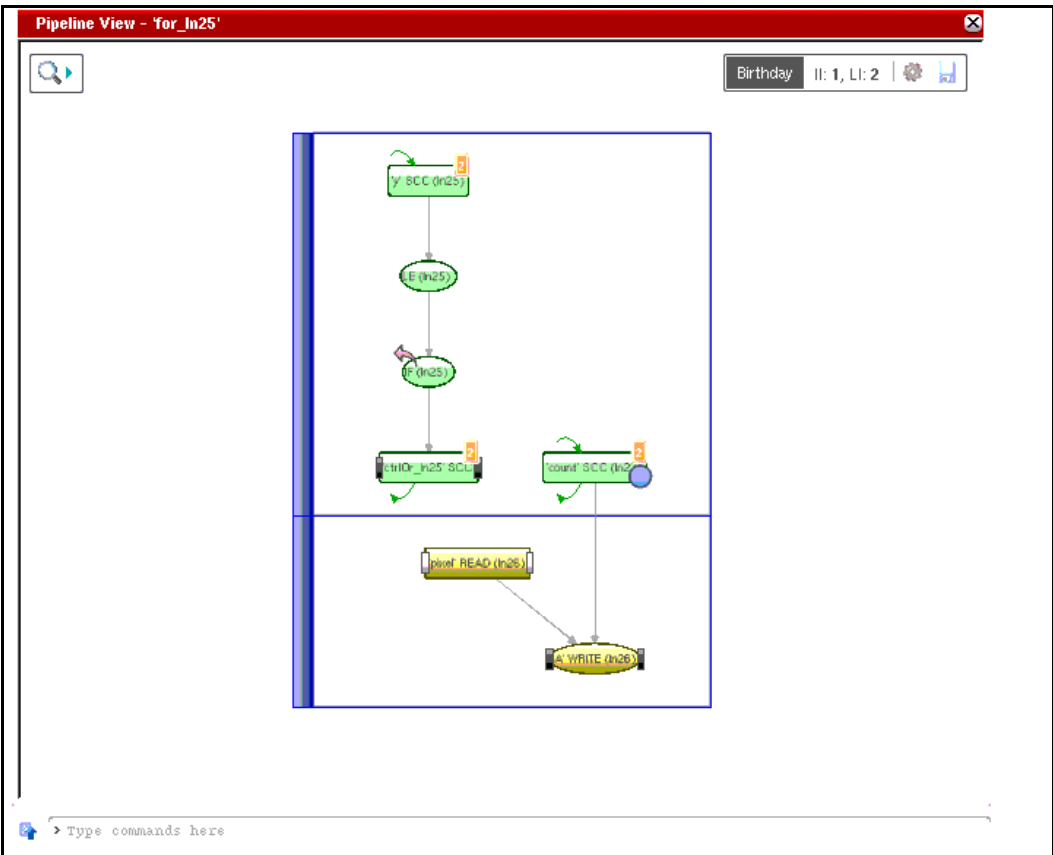
The **Pipeline View**, as shown in [Figure 8-14 on page 8-37](#), visualizes the dependency of ops and SCCs (*strongly connected components*; see “[SCC](#)” on [page N-14](#)) of a loop in the context of pipeline stages and phases.

Note If the loop is in a scheduled behavior, the **Pipeline View** will come up in read-only mode, although you may still change filtering settings.

The following sections describe the **Pipeline View** in more detail:

- “[Pipeline View Conventions](#)” on [page 8-37](#)
- “[Note about Control ops in the Pipeline View](#)” on [page 8-38](#)
- “[Moving an op/SCC to a New Stage/Phase](#)” on [page 8-38](#)
- “[Changing Filtering/Pipelining Settings in the Pipeline View](#)” on [page 8-40](#)

Figure 8-14 Pipeline View



Pipeline View Conventions

The **Pipeline View** has the following graphical notation:

- Rounded rectangles indicate SCCs (*strongly connected components*; see “SCC” on page N-14)
- Ovals indicate *Individual Ops*.
- Nodes with a dark bar on either side are *Fixed* and cannot be moved.
- I/O ops with a lighter bar on either side can be moved, but the CtoS scheduler will *not* move them.

The **Pipeline View** has the following color-coding conventions:

- Level 1 ops are **mustard**.
- Other levels of ops are **green**.

After a failed schedule:

- Failed ops are **red**.
- Ops that are not scheduled are white with a dotted border.
- Ops that were scheduled are their normal color (mustard or green, as specified above).

The **Pipeline View** categorizes ops by level, as follows:

- Level 1: Custom, Divide, Modulo, Multiply, Read, Write, Array Read, Array Write, Masked Array Write
- Level 2: Mux, Select, Replace
- Level 3: Add, Subtract, Negate, Increment, Decrement
- Level 4: Left Shift, Right Shift, Left Rotate, Right Rotate
- Level 5: Greater Than, Less Than, Greater (than or) Equal, Less (than or) Equal
- Level 6: Equal, Not equal, And, Or, Xor, Nand, Nor, Xnor, Unary Not, Unary And, Unary Or, Unary Xor, Unary Nand, Unary Nor, Unary Xnor, Case, Case1, Switch, If

Note about Control ops in the Pipeline View

In CtoS, a *control op* is an operation that is part of the control state machine generated during synthesis.

Control ops in pipelining are not associated with a particular stage, because in pipelining several pipeline stages are executed simultaneously.

To distinguish which op takes place in which stage, an additional control mechanism is implemented using the *pipeline stage vector* [[“PSV \(Pipeline Stage Vector\)” on page 8-60](#)]. The PSV takes values from control ops as input and combines them with a control value showing which stage is active (multiple stages could be active simultaneously).

The output values of the PSV are used to tag ops in pipeline stages. Therefore, ops from different stages consume values from control ops (in the form of PSV-produced tags), and control ops cannot be assigned to a particular stage. They will thus show up in timing reports for different stages.

Moving an op/SCC to a New Stage/Phase

To move an op or an SCC (*strongly connected component*; see [“SCC” on page N-14](#)) to a new stage or phase, you simply drag it to the desired stage or phase.

Any other ops or SCCs with a dependency on that op will be automatically pushed out to the correct place, as shown in [Figure 8-15 on page 8-39](#).

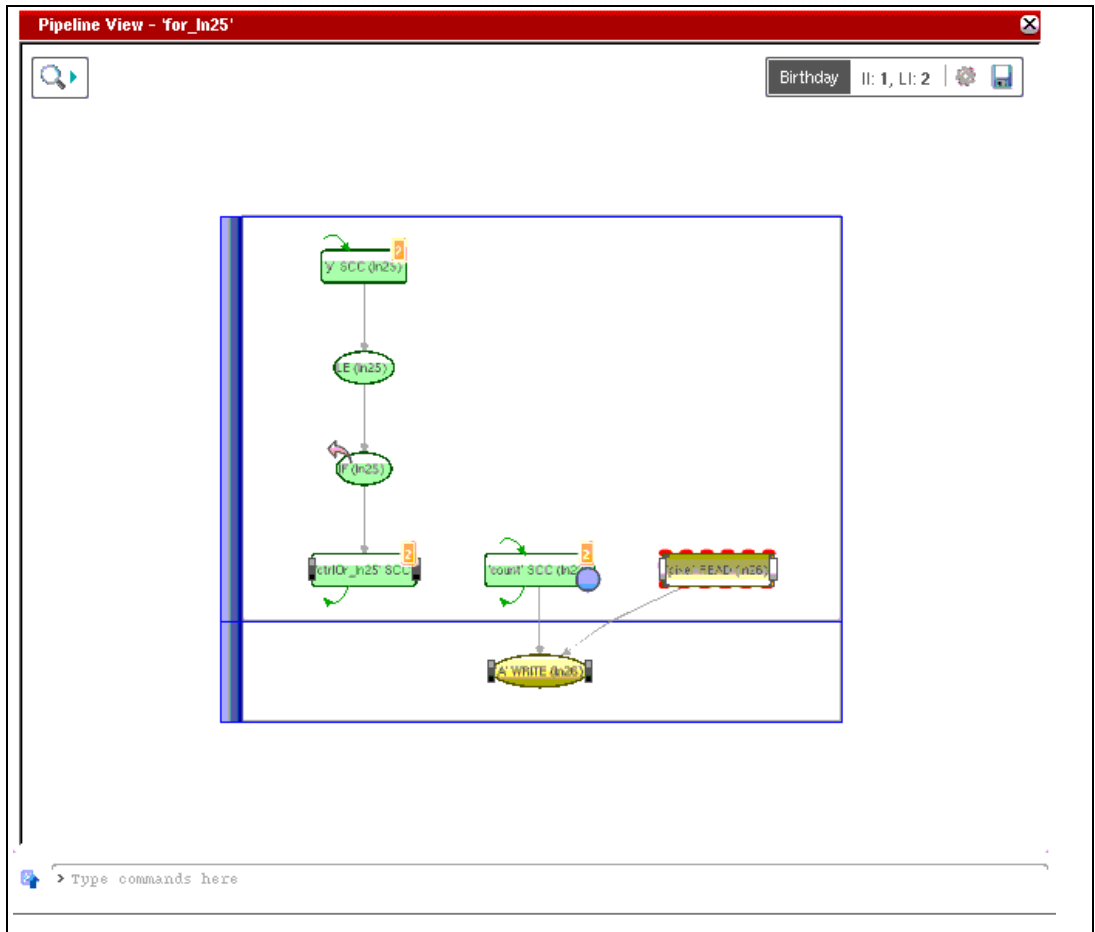
You can specify the type of constraint by right-clicking on the op, and selecting **Soft** or **Hard**.

You must commit any changes you make by selecting the  icon, which becomes enabled when there are unsaved pipeline changes.

When you constrain ops or SCCs to the new stage/phase, you will see a number of **constrain_op** commands in the **Command Output** window.

CtoS will also prompt you if you make changes and exit the viewer before applying them.

Figure 8-15 Moving an op/SCC to New Stage/Phase



Changing Filtering/Pipelining Settings in the Pipeline View

The **Pipeline View** lets you change several settings for filtering and pipelining. Filtering provides a way to deal with loops with large numbers of ops, as many ops make it difficult to understand the structure of a loop, and the time required to lay out dependencies is also significantly increased.

To change these settings, select the  button to bring up the **Pipelining and Filtering** dialog, as shown in “[Pipelining and Filtering Dialog](#)” on page 8-41.

- The **Pipelining** checkbox lets you change the **II** or **LI** of a loop.
- The **Op Filtering** checkbox lets you disable or enable the level of ops displayed (see “[Pipeline View Conventions](#)” on page 8-37 for categories). To enable understanding of the impact of filtering, the total number of ops matched for each level is displayed. By default, CtoS does no filtering for a small number of ops. As the number increases, however, CtoS progressively increases filtering, so the viewer can be displayed quickly, but with minimal sacrifice to your ability to understand the loop.
- The **Edge Filtering** checkbox lets you choose the following:
 - *Transitive Dependency Only*. For example, if **N1** is connected to **N2**, and **N2** is connected to **N3**, and **N1** is connected to **N3**, you do not need to display the edge connecting **N1** to **N3**, so with *Transitive Dependency Only* unchecked, you would see something like the following:

```
N1 -- N2 --- N3
      \_____/
```

With *Transitive Dependency Only* checked (the default), you would simply see:

```
N1 -- N2 --- N3
```

- *Exclude Tag Dependencies*. By default, tag dependencies are excluded, but you can enable them.

Figure 8-16 Pipelining and Filtering Dialog

Pipelining

It: 1 Lt: 2 Apply

Op Filtering

By Types:

Op	Not Op
☒ LEVEL 1	8
☒ LEVEL 2	2
☒ LEVEL 3	2
☒ LEVEL 4	
☒ LEVEL 5	
☒ LEVEL 6	4

Select All Clear All

Name Pattern:
x

Edge Filtering

☒ Transitive Dependency Only
☒ Exclude Tag Dependencies

Apply

8.2.5.6 Resolving Pipelining Scheduling Failures

Pipelined loops have more constraints than non-pipelined ones; hence, the pipelining of a loop may fail during scheduling even if the corresponding non-pipelined version could be scheduled. You will usually get an error message [ERROR (CTOS-20320) or ERROR (CTOS-20322), among others] listing the ops involved in the overconstrained cycle or path.

In this section, some of the most common causes for failed scheduling of a pipelined loop are presented, along with suggested corrective actions.

- “Ops of SCCs Must Be Scheduled within a Stage” on page 8-42
- “Ops with same Memory (Implicit SCC) Must Be within Single Stage” on page 8-43
- “Ops Controlling Loop Exit Must Be in First Stage of Pipeline” on page 8-48
- “Last Statement of Loop Body Must Be wait()” on page 8-48

Ops of SCCs Must Be Scheduled within a Stage

An SCC (*strongly connected component*; see “SCC” on page N-14) is a set of ops that can all be reached from each other or that have *cyclic dependencies* (also known as *loop-carried dependencies*).

For example, a *loop-carried dependency* would occur when a variable is written in an iteration of a loop, and read in a subsequent iteration – any op affecting the new value and depending on the old value of that variable is in the SCC.

Consider the following example:

```
for (i = 0; i < 16; i++) {  
    a = sum * ...;  
    sum = a - 12;  
    ...  
}
```

- The addition that increments **i** and the *join mux* bringing back the value of **i** into the loop comprise an SCC.
- The multiplication, subtraction, and multiplexing that brings **sum** back into the loop comprise another SCC. Conversely, variable **a** is not alive *across* loop iterations – only *within* an iteration – and hence does not require such a join mux. However, this variable and the multiplication that creates its value are part of the SCC, since it is part of the cycle that uses the old value of **sum** from the past iteration and computes the new value of **sum** for the next iteration.

All operations of a given SCC must be *scheduled within a stage*, since their result is needed for the next iteration of the loop.

The CtoS scheduler can choose any stage for those operations, as long as they are all within a stage. For example, with the typical case of **II=1**, each of the previously described SCCs must complete its execution within a clock cycle.

The **Pipeline View** (described in “Using the Pipeline View” on page 8-36) therefore displays an SCC as a single op, because it must be scheduled as a unit.

This constraint (scheduling an SCC as a unit) can cause two kinds of problems:

1. **Tighter timing constraints on the ops in the SCC**, because they cannot use the whole **LI** of the loop, only the **II**, which may result in negative slack at the end of the schedule.
2. **Scheduler failures characterized by the following errors:**

ERROR (CTOS-20233) – An op cannot be constrained to an edge.

ERROR (CTOS-20051) – An op cannot be scheduled to any edge.

ERROR (CTOS-20080) – Scheduling cannot be performed because some ops have no span.

ERROR (CTOS-20085) – An op cannot be scheduled due to span.

ERROR (CTOS-20087) – Cannot schedule sequential op because it would leave an empty span.

This can occur due to ops with sequential latency, such as synchronous memory reads or user IPs.

An SCC with a synchronous memory read cannot be scheduled by CtoS with **II=1** – it requires at least two cycles to execute (one to set up the address and one to use the data), for example:

```
a = M[a];
```

where **M** is a memory with synchronous reads.

For more information about these problems, you can observe the failed loop with the CtoS GUI **Pipeline View** (see [“Using the Pipeline View” on page 8-36](#)). Each SCC is displayed as a rounded rectangle with label that indicates the SCC variable causing it to be strongly connected. The number of ops in the SCC is also annotated to give hint of the size of the SCC. Double clicking on an SCC node displays dataflow graph of the ops in the SCC.

In addition, the **report_sccs** command provides textual information about SCCs in behavior, loop or for op within an SCC. The detailed report lists the SCC variable, its source link location and the list of all ops in the SCC. For more information, see [“report_sccs” on page E-121](#).

Ops with same Memory (Implicit SCC) Must Be within Single Stage

CtoS constrains all operations involving a given memory in a pipeline so they occur within a single stage, that is, all ops referring to that memory belong to an *implicit SCC*.

This is due to the fact that CtoS is unable to analyze memory addresses to infer that two memory operations *could* be moved into different iterations.

In the following example, in the non-pipelined form, the location updated by the write is not used by the following read. However, it would be unsafe to move the read, for example, three stages later, because it would then read the new, updated value.

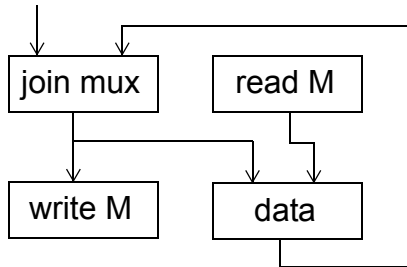
```
for (i = 0; i < N; i++) {
    ...
    M[i] = ...;
```

```
... = M[i+2];
```

Furthermore, *single-ported memories* (memories with one shared port for reading and writing) can be scheduled in a pipelined loop only if their read and write operations are mutually exclusive. This can be satisfied only if they are accessed under mutually exclusive conditions, as in the following example:

```
if (a) M[i] = ...;  
else ... = M[i+2];
```

Figure 8-17 SCC of Muxes



Additionally, the two operations must be scheduled in the same **II**, because only then would the two operations, potentially executed in the same clock cycle, belong to the same iteration, and thus be mutually exclusive.

For example, if the memory read in the previous example was scheduled in stage 1, and the memory write was scheduled in stage 2 with **II=1**, then a write from iteration **j** would be executed in the same cycle as a read in iteration **j+1**.

However, there is no guarantee that these two operations would still be mutually exclusive (**a** could be true in iteration **j** and false in iteration **j+1**). Hence, the CtoS scheduler would reject such a design as incorrectly constrained.

If these two operations were also constrained by an SCC, as shown in [Figure 8-17 on page 8-44](#), you could not legally implement such operations with a single-ported memory with synchronous read – they could not be executed in a single cycle for the aforementioned reason (the SCC between the two memory operations) – and they could not be executed in different cycles (the single-port limitation).

You can manually override such a constraint using the **constrain_op** command (“[constrain_op](#)” on [page E-38](#)) to force an op to be in a specific stage and phase if, after performing the previously mentioned address analysis by hand, you are sure that the accesses would not conflict.

For example, in the previous case, it would be both legal and correct, if the memory **M** has separate read and write ports, to constrain the ops as follows:

```
constrain_op -stage 1 memwrite_M_ln...  
constrain_op -stage 2 memread_M_ln...
```

In this case, being one stage apart would still assure that the original value of **M[i+2]** is read. However, manual constraints can cause scheduler failures, with the same error messages as those reported previously [and also ERROR (CTOS-20289)].

In the following example, issues related to SCCs and issues related to sequential memory reads interact and require manual constraints in order for scheduling to succeed. In the case of a dual-ported memory, manual constraints are possible and lead to successful scheduling. In the case of a single-ported memory, scheduling will never be possible, even with manual constraints.

```
if(c) {  
    M[index] = prev + ...;  
} else {  
    prev = M[index+1];  
}
```

This example could not be scheduled without constraints if the memory has synchronous reads and $II=1$, because:

- Variable **prev** induces an SCC of muxes, as shown in [Figure 8-17 on page 8-44](#), since its value after the **if** may come from a past iteration (which creates a CDFG cycle) or from the memory read.
- The synchronous memory read that drives the SCC has the result available one cycle later. Hence, the read and the write cannot be scheduled in the same stage, which contradicts the memory constraint.

You can manually constrain (if it is safe to do so, given the functionality of the behavior) the memory write to occur at least one cycle after the memory read, and in that case, the schedule would succeed.

Additional Examples of Memory Constraint Considerations with Pipelined Loops

The CtoS scheduler requires that all ops accessing a memory be scheduled within II (initiation interval) states, because CtoS has limited inter-iteration memory address analysis. Namely, it can identify when two consecutive iterations of a loop do not access the same address in an array.

For example, CtoS is able to detect that the following loop does *not* have any inter-iteration dependency, and can thus schedule the write at any stage after the read:

```
LOOP: for (unsigned i = 0; i < n; i++) {  
    m[i] = f(m[i]);  
    W: wait();  
}
```

However, in more complex cases, manual analysis is needed, as shown in the following example:

```
LOOP: for (unsigned i = 0; i < n; i++) {  
    m[i+2] = f(m[i]);  
    W: wait();  
}
```

In this case, it is possible to prove that iteration **i** writes to a location that will be read only two iterations later. Hence, the write at a given iteration can be scheduled *at most* two stages after the read – otherwise, the read two iterations later would read the original value.

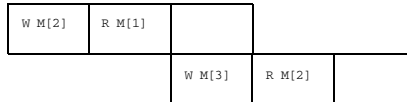
The following sections describe three more examples of dependencies that must be manually resolved:

- “[Example 1: Read After Write \(RAW\) hazard with \$II=2\$.” on page 8-47](#)
- “[Example 2: Write After Read \(WAR\) hazard with \$II=1\$.” on page 8-47](#)
- “[Example 3: Write After Read \(WAR\) hazard with \$II=1\$.” on page 8-48](#)

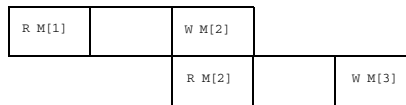
Example 1: Read After Write (RAW) hazard with II=2:

```
for (i=0; ...; i++) {
    M[i+1] = v1;
    v2 = M[i]; // M[i] should read the value written by the i-1 iteration
```

A *legal* schedule within **II** would be as follows – read **M[2]** *correctly* gets the updated value:



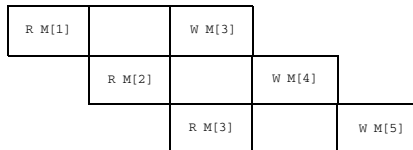
An *illegal* schedule outside **II** would be as follows – read **M[2]** *incorrectly* gets the original value, if write takes effect at the next cycle:



Example 2: Write After Read (WAR) hazard with II=1:

```
for (i=0; ...; i++) {
    v1 = M[i]; // M[i] should read the value written by the i-2 iteration
    M[i+2] = v1 + ...;
```

An *illegal* schedule would be as follows – read **M[3]** *incorrectly* gets the original value:



For the reasons exemplified by these two examples, “[Example 1: Read After Write \(RAW\) hazard with II=2:](#)” and “[Example 2: Write After Read \(WAR\) hazard with II=1:](#)”, and due to the fact that CtoS does not currently perform inter-iteration loop address analysis, the default CtoS behavior is that all ops on a given memory must occur within an **II**.

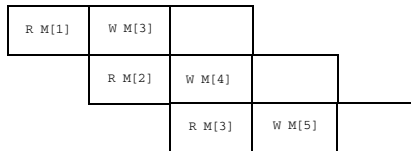
However, as previously mentioned, this could be overridden with manual constraints when the designer is sure two or more operations that refer to the same memory can be safely spread over a broader interval due to the properties of the address computation patterns, as shown in the following example, “[Example 3: Write After Read \(WAR\) hazard with II=1:](#)”.

Example 3: Write After Read (WAR) hazard with $\Pi=1$:

```
for (i=0; ...; i++) {
    v1 = M[i]; // M[i] should read the value written by the i-2 iteration
    M[i+2] = v1 + ...; // Sequential read: the write must be scheduled
                      // at least one cycle later than the read
}
```

CtoS cannot automatically schedule the write one cycle after the read (due to the *potential* WAR hazard with $\Pi=1$).

A manually constrained *legal* schedule would be as follows – read M[3] *correctly* gets the original value:



Ops Controlling Loop Exit Must Be in First Stage of Pipeline

Operations controlling the *exit* of a loop must be scheduled in the first stage of the pipeline.

This means that the schedule of any op before them must fit within Π clock cycles.

This is similar to the effects of an SCC on slack and scheduling failures, but is stronger, in that an SCC can float to any stage, while the loop exit is constrained to the first stage.

Last Statement of Loop Body Must Be wait()

The last statement of the loop body must be a **wait()**, either inserted explicitly, or inserted implicitly by the **pipeline_loop** command (“[pipeline_loop](#)” on page E-92).

This means CtoS automatically inserts a **wait()** at the *end* of the loop, even though the Tcl command would direct it to insert states *before* that point, if **-expand_before out2** or **-expand_after out1** were used with **pipeline_loop** for the following loop, because the loop could not be pipelined otherwise:

```
while (1) {
    ...
    out1.write(...);
    out2.write(...);
}
```

8.2.5.7 Using the report_schedule Command with Pipelining

Setting up the pipelining of a loop is similar to applying latency constraints to a loop. When the behavior is scheduled, the CtoS scheduler implements the loop with the pipelining you have specified.

If you run the **report_schedule** command ("[report_schedule](#)" on page E-125) after scheduling, it reports on the operations of the loop that have been scheduled in the stages of the pipeline in the column *Stage*. This information can be useful when you want to explore the number of pipeline stages required.

The phase information, if **II** is greater than 1, is not reported as an integer, but as an edge name, since the number of phases is exactly the same as the number of edges in the pipelined loop body, after scheduling is complete. For example, if there are stages in which no operations are scheduled, it may be possible to reduce the **LI**.

8.2.5.8 Stalling/Flushing a Pipelined Loop

Stalling/flushing pipelined loops is a preliminary feature.

During normal processing, a pipelined loop takes **LI** clock cycles to complete each iteration and starts a new iteration every **II** clock cycles. If and when the loop terminates, it also takes **LI** cycles from when the exit condition becomes true to finish the last iteration. This is because the exit condition is evaluated in the first stage of the pipeline.

However, occasionally a pipeline must temporarily suspend its execution because input data is not available from other processes, or output data cannot be accepted by other processes. This is known as *stalling* the pipeline.

Here is an example of a pipeline with **II**=1 and **LI**=5 that may need to stop if an input to stage 1 or another input to stage 3 are not available, or if an output cannot be accepted in stage 5.

```
while (...) {
  while(stall1) {wait();} // stage 1 stall
  ... = in1;
  wait();                // stage 1
  ...;
  wait();                // stage 2
  while(stall2) {wait();} // stage 3 stall
  ... = in2;
  wait();                // stage 3
  ...;
  wait();                // stage 4
  while (stall3) {wait();} // stage 5 stall
  out1 = ...;
  wait();                // stage 5
};
```

When a pipeline is stalled, data already in the pipeline from previous iterations (i.e. in later stages) could be stalled or continue its execution. Continuing to finish execution is known as *pipeline flushing*. For example, if stage 3 is stalled then data in stage 4 and 5 continue execution. While flushing, stages after

the stalling stage no longer have valid data and are said to contain a *pipeline bubble*. Once the stall is set to false, execution resumes filling the *pipeline bubbles* with valid data, which is also known as *bubble squashing*.

The following picture illustrates flushing and bubble squashing for the above example with a stream a alphabetic characters as input data. In this picture, cycle time progresses top to bottom. For this test, stall activity is shown on the left and shows stall2 becoming active in cycle 5 for 2 cycles (impacting stage 3). Then stall3 becomes active in cycle 7 for 2 cycles (impacting stage 5).

Cycle\Stage	1	2	3	4	5
1	a				
2	b	a			
3	c	b	a		
4	d	c	b	a	
stall2 active	5	d	c	**	b
stall2 active	6	d	c	**	**
stall3 active	7	e	d	c	**
stall3 active	8	f	e	d	c

Along the top are the stages in the pipelining and as time progresses, data ‘a’ moves from stage 1 to 5. Data ‘b’ comes into stage 1 at cycle 2. Data progresses through the pipeline stages until cycle time 5, stage 2 is stalled (keeping ‘c’ from moving to stage 3 as shown in red). Even though stage 3 is stalled, stages 4 and 5 are still executing processing data items ‘a’ and ‘b’ resulting in a bubble in stage 3 (as shown in green). Stall3 is activated in cycle 6 causing ‘b’ to be suspended in stage 5 (as shown in red). Since stall2 is no longer set, data ‘c’ moves to stage 3 and squashes the bubble (as shown in purple).

CtoS support for pipeline stalls:

- Stalling the entire pipeline without flushing and bubble squashing, limited to a single stall loop as described in [“Using a While Loop as a Stall Loop to Stall all Stages” on page 8-52](#) and subject to limitations listed in [“Limitations of using a While Loop as a Stall Loop” on page 8-52](#).
- Stalling with flushing and bubble squashing, limited to II=1 as described in [“Pipelined Loops with Multiple Stall Loops” on page 8-53](#) and subject to limitations listed in [“Limitations of Pipelined Loops with Multiple Stall Loops” on page 8-53](#)
- Stalling with flushing and not bubble squashing, limiting to the first stall being in the first stage and it being expressed as an if-statement as described by: [“Using an if Statement to Insert Bubbles in the Pipeline” on page 8-54](#).

Specifying Hand-Shaking Protocols

The *handshaking protocol* must use *transition-based outputs*, because when a downstream stall is activated, all preceding stages are stalled; hence, they cannot be producing frozen *valid* outputs. Note that this is automatically satisfied by FIFO interfaces that directly increment or decrement pointers in the pipeline code itself.

Here is an example of output handshaking protocol that will cause undesired results during stalling because it produces frozen *valid* outputs:

```
out_valid = false;
while (1) { // loop to pipeline
    while (!in_ready) wait(); // stall loop
    ... = in_data;
    ...
    while (!out_ready) wait(); // stall loop
    out_data = ...;
    out_valid = true; // frozen output
    wait();
    out_valid = false;
    wait();
}
```

The expectation is that *out_valid* is **true** in any clock cycle in which a new value is assigned to *out_data*. This behaves correctly when the loop is not pipelined or it never stalls. However, when the loop stalls (*!in_ready* or *!out_ready*) then the last assigned value to *out_valid* in the previous clock cycle (from a previous iteration of the loop) remains assigned for the duration of the stall. If *out_valid* happened to be **true**, then the other side of the protocol would incorrectly assume that *out_data* has a new value in every stalling clock cycle.

Alternatively, the following output handshaking protocol would behave as expected during stalling because it uses *transition-based outputs*:

```
out_valid = false;
while (1) { // loop to pipeline
    while (!in_ready) wait(); // stall loop
    ... = in_data;
    ...
    while (!out_ready) wait(); // stall loop
    out_data = ...;
    out_valid = !out_valid; // transition-based output
    wait();
}
```

In this case, the expectation is that a new value is assigned to *out_data* in every clock cycle in which *out_valid* changes value (from high to low or from low to high). This protocol works correctly both for a non-pipelined and a pipelined version of the loop. Even when the loop stalls, it works correctly because *out_valid* is not updated and no new value on *out_data* is thus expected.

Note that the example considers two stall loops, but the very same problem occurs also with a single stall loop.

Using a While Loop as a Stall Loop to Stall all Stages

In the following example, if the stall signal *is not* asserted (**m_stall = false**), the pipelined loop proceeds as usual.

If the stall signal *is* asserted (**m_stall = true**), all pipeline stages are stalled, and the values of all variables are kept intact.

When the stall signal is de-asserted, the loop continues.

```
do {                                     // Pipelined loop

    while (m_stall) { wait(); }          // Stall loop

    sum = sum + m_j.read();

    out1.write(i);

    wait()

} while (m_i.read().range(2, 0) > 0);
```

Limitations of using a While Loop as a Stall Loop

There are a few limitations if you are using a **while** loop as a stall loop:

- The stall loop *must* be of type **while**.
- The stall loop must have a *single state node* and may not have nested loops.
- Operations of the pipelined loop must have *no dependencies* on operations of the stall loop.

This also means the stall loop and the pipelined loop *cannot use the same array*.

- The stall loop must have a *single exit*, which must exit into the pipelined loop.

In other words, the stall loop must not contain a **return** statement, which would also exit the pipeline loop.

Pipelined Loops with Multiple Stall Loops

Pipelined loops may have multiple stall loops.

To enable this feature, before you build your design, set the **enable_multiple_pipeline_stalls** design attribute (see “[enable_multiple_pipeline_stalls](#)” on page D-15), using the **set_attr** command (see “[set_attr](#)” on page E-140).

Limitations of Pipelined Loops with Multiple Stall Loops

When using multiple stall loops, there are some requirements and restrictions on the pipelined and stall loops, as follows:

- The **II** (initiation interval) *must* be **1**, or else an error message is issued.
- There can be no *loop join mux* inside the stall loop, that is, there can be no computation involving the update of a variable [SCC (*strongly connected component*)], or even the communication of a variable value across the back edge of the loop, or else an error message is issued. Here are some examples of this type of unsupported pipelined loop:

```
PIPE: while(1) {
    sc_signal<int> sig1, sig2;
    int x = 0;

    STALL1: while (stall) {
        x++;
        wait();
    }
    ...
    STALL2: while (stall) {
        sig2.write(x);
        wait();
        x = sig1.read();
    }
    ...
}
```

- There can be no *memory ops* within a stall loop.
- Every stall loop must be in a *separate stage*.
- *User RTL IP* and *registered memories* for latency > **1** are not supported.
- *Output assignment operations* can occur only before the stall **wait**, or else an error message is issued. This is because the ops within the stall loop are all mapped to the entering stage of the stall.

Here is an example of this type of unsupported pipelined loop:

```
PIPE: while(1) {
    out.write(false);
    STALL1: while (stall) {
        wait();
        out.write(true);
    }
    ...
}
```

Using an if Statement to Insert Bubbles in the Pipeline

Using an **if** statement lets you insert bubbles in the first stage of the pipeline even if multiple stall support is not enabled. With a stall loop, when all inputs are consumed, the loop could stall before output is written. Adding an **if** statement as the first statement of the loop body lets you insert bubbles in the pipeline, so the outputs are flushed out, that is, written out, as shown in the following example:

```
do {
    // Pipelined loop
    if (m_valid) {
        sum = sum + m_j.read();
        out1.write(i);
    }
    wait();
} while (m_i.read().range(2, 0) > 0);
```

Additionally, if a stall loop occurs before any operation with an externally observable effect (for example, value update of a loop-carried variable, memory write, output operation), an **if** statement may be implemented more efficiently than a **while** loop, as it may directly translate to a clock enable signal.

In the previous loop, the only operation that can occur outside the **if** statement is the read of **m_j**, since it is the only one without side effects (the increment of **sum** and the assignment of **out1** are observable).

Stalling in Loop that Terminates

The stalling condition for a pipelined loop is checked in the stage in which the stalling loop occurs in the original specification. For example, consider the following code:

```
PIPE: for(int i=0; i<10; i++){
    while (STALL) wait();    // stall in stage 1
    int _temp = MEM[i];
    wait();
    wait();
    OUT.write(_temp);        // out in stage 3
}
```

When the pipeline enters the 10th iteration (in the 10th clock cycle since its beginning), it checks the stall loop for the last time, in stage 1. However, the pipeline still needs to execute two more clock cycles before the final output is generated (in the third stage).

The stall signal is not checked in those two additional clock cycles, since it is assumed to guard the reading of the memory, in stage 1, and not the writing of the output, in stage 3. However, if you want to stall in these two additional clock cycles, you must modify the original specification, as follows:

```
PIPE: for(int i=0; i<10; i++){
    int _temp = MEM[i];
    wait();
    wait();
    OUT.write(_temp);          // out in stage 3
    wait();
    while (STALL) wait();     // stall in stage 4
    wait();
}
```

This means that:

- The stall loop must occur after the *last* operation performed by the pipeline, if you want to stall until the end of the epilogue.
- The total number of stages of the pipeline (which affects only the control, that is, requires only two additional flip-flops to hold the pipeline state with respect to the original specification) must be extended to **LI+1** states if the last output is produced *after* stage **LI**, as in the original specification.

Remember that outputs are registered; hence the first output is produced in the third clock cycle from the beginning of the pipeline execution, even though the pipeline has only two stages.

This ensures pipeline control remains active even when output is updated, which was one cycle after the pipeline terminates in the original specification.

8.2.5.9 Pipelined while and for Loops Transformed to do/while Loops

CtoS transforms *while* and *for* loops *that are always taken* into *do/while* loops during optimization. CtoS also transforms *all pipelined while* and *for* loops (even if they are *not* always taken) into *do/while* loops.

For example, the following loop:

```
while (!a && b) {  
    wait();  
    a = b + 15;  
    m[i] = a;  
    wait();  
}
```

... is transformed to:

```
do () {  
    bool do_loop = (!a && b);      <- computation of exit condition  
    wait();  
    a = b + 15;  
    if (do_loop) { <- significant operations applied only if not exiting  
        m[i] = a;  
    }  
    wait();  
} while (do_loop)                  <- execution of exit condition
```

In this example, the exit condition is applied at the end of an iteration, instead of at the beginning, implying the loop will take one additional iteration to exit. The last execution of the loop, without actions with side effects, is required to allow the previous iterations, before the exit condition became true, to drain.

To prevent the extra iteration from changing values computed by the loop, the exit condition is computed at the beginning of the iteration and is used to skip significant operations, so **bool do_loop** is computed at the beginning of the iteration, and the memory write to **m** is not executed if exiting.

However, there are some differences in how many clock cycles it takes before you reach the loop exit:

- Transformed pipelined loops are always executed at least once. With a **while** loop, the loop could be skipped entirely, so a **while** loop whose condition is initially false takes **0** clock cycles, while a transformed **do** loop takes the **LI**.
- When an exit condition becomes false during an iteration, a **while** loop exits before starting the next iteration. The transformed **do** loop, however, waits until the beginning of the next iteration to compute the next exit condition – then executes one pipeline stage (not *live*, so no significant operations will happen) before exiting. This means it takes one extra stage (the **II**) to exit the loop.

Note This latency penalty occurs only for loops that *cannot be proved to be always taken* during optimization.

8.2.5.10 Behavior of Inputs and Outputs of a Pipelined Loop

This section describes the behavior of the inputs and outputs of a pipelined loop, as follows:

- “Value of Pipeline Output after Reset” on page 8-57
- “Multiple Assignments to a Pipeline Output” on page 8-58
- “Allowing I/O Reordering during Pipelining” on page 8-59

Value of Pipeline Output after Reset

Outputs in a pipeline are not reset immediately. Outputs are reset when the stage that writes to the output is activated.

For example, consider the following loop, pipelined with **II=1**, where **sc_out<bool> o** is not written until stage **3** of the pipeline:

```
void thread() {
    int cnt = 0;
    while(1) {
        wait();
        wait();
        wait();
        o.write(cnt++);
    }
}
```

In this scenario, the output **o** will not be reset until **3** cycles after the reset signal. In addition, the value of output **o** will be frozen at its previous value for the first **3** cycles after the reset signal. This is shown in the following output trace, where the repeated values of **3** are highlighted in yellow, followed by the reset of **cnt** to **0** in the next cycle:

reset	1	1	0	0	0	0	0	0	1	0	0	0	0	0	0
o	X	X	X	X	X	0	1	2	3	3	3	3	0	1	2

Multiple Assignments to a Pipeline Output

In general, it does not make sense to have more than **II** assignments of values to a given output signal in the body of a pipelined loop, because only **II** such assignments can be executed in **II** clock cycles after scheduling (of course, several assignments under mutually exclusive conditions count as one).

However, it is common to use *valid* signals to notify the external world when a pipeline is producing an output.

For example, the following case – modeling a very simple memory access protocol – has **Re** and **We** as handshaking outputs; **Radr**, **Wadr**, and **Wdat** as address/data outputs; and **Rdat** as a data input:

```
for( unsigned int i=0 ; i<16 ; i++) {  
    Re=1;  
    Radr=i;  
    wait();  
    Re=0;  
    wait();  
    wait();  
    ...=Rdat;  
    wait();  
    We=1;  
    Wadr=i;  
    Wdat=...;  
    wait();  
    We=0;  
    wait();  
}
```

If the previous example were pipelined with **II=1**, only one write to **We** and **Re** could occur in any given clock cycle.

The semantics CtoS uses is that the assignment from the iteration in the earlier stage *wins*.

For example, after two clock cycles from the loop's beginning, both **Re=0** (from the first clock cycle of the second iteration) and **Re=1** (from the second clock cycle of the first iteration) are executed.

The intended semantics is that signal **Re** is high, to signal that the corresponding assignment to **Wadr** is valid:

```
Re=1; Re=0;  
    Re=1; Re=0;  
        Re=1; Re=0;  
            Re=1; ...
```

The corresponding waveform in a non-pipelined execution of the loop would show pulsed values for **Re** and **We**, while the pipelined execution would show a *burst* of cycles in which those signals are continuously high for **16** cycles.

The *burst* at value **1** starts from the first cycle for **Re** and from the fifth cycle for **We**, while **Re** goes to **0** six cycles before exiting the loop, and **We** goes to **0** one cycle before.

Allowing I/O Reordering during Pipelining

CtoS tries to avoid breaking SystemC design intentions while executing code transforms.

However, pipelining a loop initiates a very powerful transform.

In a pipelined loop, the sequential loop body is *folded*, and the original multi-state loop is executed by smaller sequences in parallel. In some cases, this can lead to unintended consequences.

Consider the following source code:

```
for(int count=0; count<SIZE; count++) {  
    int tmp = val.read();  
    wait();  
    // process other values and update tmp  
    val.write(tmp);  
    wait();  
}
```

If this loop were pipelined with an **II** of **1**, the signal **val** would be read and written in two different pipeline stages.

In a sequential execution, the second iteration would be reading the value from the previous iteration, but this might not be true after pipeline folding.

To avoid this potential problem, CtoS creates an error message whenever it encounters this situation in your source code. More precisely, CtoS creates an error message if it encounters a signal read/write operation that is *at least as many states apart as the II of the pipeline*.

However, also consider the following example in which a whole array (**A**) of signals is processed:

```
for(int count=0; count<SIZE; count++) {  
    int tmp = A[count].read();  
    wait();  
    // process other values and update tmp  
    A[count].write(tmp);  
    wait();  
}
```

If this loop is pipelined with an **II** of **1**, the same signal would also be read and written in two different pipeline stages. However, it is clear that the same signal is never read in any of the later iterations.

Because CtoS would flag this issue incorrectly as an error, an overwrite mechanism is provided for the **pipeline_loop** command ("**pipeline_loop**" on page E-92): the **-allow_io_reordering** option. By using this option, the potential error discovered during loop analysis will be transformed into a warning.

In the CtoS GUI, you will be prompted with a message box, asking if it is okay to reorder I/Os. If you click **OK**, then the loop will be pipelined with I/Os reordered.

8.2.5.11 Pipeline Object Names

This section describes how objects are named during pipelining:

- “Pipeline mux” on page 8-60
- “PSV (Pipeline Stage Vector)” on page 8-60
- “Stall Condition” on page 8-60

Note See “Using C++ Labels” on page 14-94 for how using labels affects these naming conventions.

Pipeline mux

A *pipeline mux* is a pipeline folding artifact.

It represents the fact that a value computed in one stage of a pipeline can be used in a later stage, and after folding, it can appear to be used by an op that comes before the op that produces the value.

After register allocation, it generally becomes a pipeline register.

The naming convention for a pipeline mux is:

delay_op-name_stage

For example, **delay_add_ln48_stage3** is both the name of an op that represents a value that is *folded back* in the pipeline, as well as the name of a pipeline register after register allocation, which most likely contains the result of operation **add_ln48** computed in stage 3 and used in stage 4.

Note Names are not always a precise indication of the functionality of an object.

PSV (Pipeline Stage Vector)

A *PSV (pipeline stage vector)* keeps track of active stages and consists of several ops (**mux**, **and**, **or**, **if**).

The naming convention for a PSV is:

mux_loop-name_psv and_loop-name_psv or_loop-name_psv if_loop-name_psv

Stall Condition

The naming convention for a stall condition is:

and_stall-loop-name_stall if_stall-loop-name_stall

8.3 Resolving Arrays (Memories)

For examples of some of the steps in resolving arrays, look in the following directory:

```
install_directory/share/ctos/examples/features/arrays
```

Note Before running any CtoS examples, first review [“Setup for Examples” on page F-2](#).

Some arrays in source code are built into memories (see [“Array Variables” on page 14-38](#)). You can implement such arrays either as a set of *flat* registers or with a RAM. You may choose to use CtoS-generated or built-in RAM or RAM from a vendor library.

Before resolving arrays (memories), it is recommended that you review [Table 9-1 on page 9-2](#) for a comparison of all of the related options for arrays (memories).

CtoS provides the following options for resolving arrays (memories):

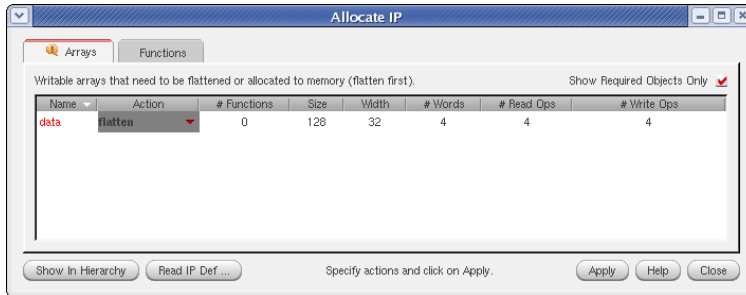
- [“Flattening Arrays” on page 8-61](#)
- [“Merging Arrays” on page 8-63](#)
- [“Splitting Arrays” on page 8-65](#)
- [“Restructuring Arrays” on page 8-78](#)
- [“Floating I/O Accesses” on page 8-91](#)
- [“Floating Array Accesses” on page 8-94](#)
- [“Allocating Memory and RTL IP” on page 9-1](#)

8.3.1 Flattening Arrays

After a module has been built, you can *flatten* an array (or arrays) to remove it and replace all of its reads and writes with equivalent reading and writing variables representing each word in the array.

Flattening arrays that are small and heavily used, or whose index is typically a constant, may produce smaller, faster designs; however, flattening large arrays produces very large designs that may be impractical.

Figure 8-18 Allocate IP Dialog - Flattening an Array



Before flattening an array, make sure the array meets these requirements:

- The array must be used by only one behavior or function. However, if both a behavior and a function use the array, you could inline the function and then flatten the array.
- The array must not be bound to a Vendor or Built-in RAM.
- The array must not be a constant array.
- The array must not be accessed from two different threads – in this case, it must be mapped to a RAM.
- The array must not be a read-only array [in this case, you will get ERROR (CTOS-17019)]. However, you can use the following workaround to flatten such an array:

```
if {[catch {set memories [find /designs/$test/modules/$test/memories/*]}]} then {  
    foreach memory $memories {  
        set readOnly [get_attr read_only $memory]  
        if { $readOnly == 0 } {  
            flatten_array $memory  
        }  
    }  
}
```

To flatten an array, you use the **Allocate IP** dialog, as shown in [Figure 8-18 on page 8-62](#).

After you have built a design, if any arrays *must* be resolved, the term **Arrays** will be listed in red in the **Allocate IP** pane of the **Task Window**. Double-click **Arrays**, and the **Allocate IP** dialog is displayed. Select the array you want to flatten, select **Flatten** from the **Action** column, and click **Apply**.

Notes

- To flatten an array that is *not required* to be resolved (but could still benefit from being flattened), simply select **Edit -> Allocate IP** to display this dialog.
- You can also use the `flatten_array` command ("[flex_channels_nets](#)" on page E-69).

8.3.2 Merging Arrays

CtoS lets you merge two or more – there is no limit – arrays to form a single array, using the **merge_arrays** command (“[merge_arrays](#)” on page E-86).

One of the main benefits of merging arrays is that a single array is smaller than multiple individual arrays.

A typical scenario is shown in [Figure 8-19 on page 8-64](#). In this example, CtoS creates, by default, multiple arrays for arrays of POD-structs. In this case, it is recommended that you merge these arrays.

Merging arrays must be done before any of the arrays has been mapped to hardware, that is, before RAMs have been allocated.

You have two implementation choices – you can merge the *addresses* or the *data*, as follows:

- When merging *addresses*, the arrays are mapped to *different addresses* in the resulting array, so accesses to the arrays must share the same read and write port, which may create contention.

In the resulting array, the number of words is at least the sum of the words in the arrays being merged, and the data width is the maximum between the *data's* of the arrays being merged.

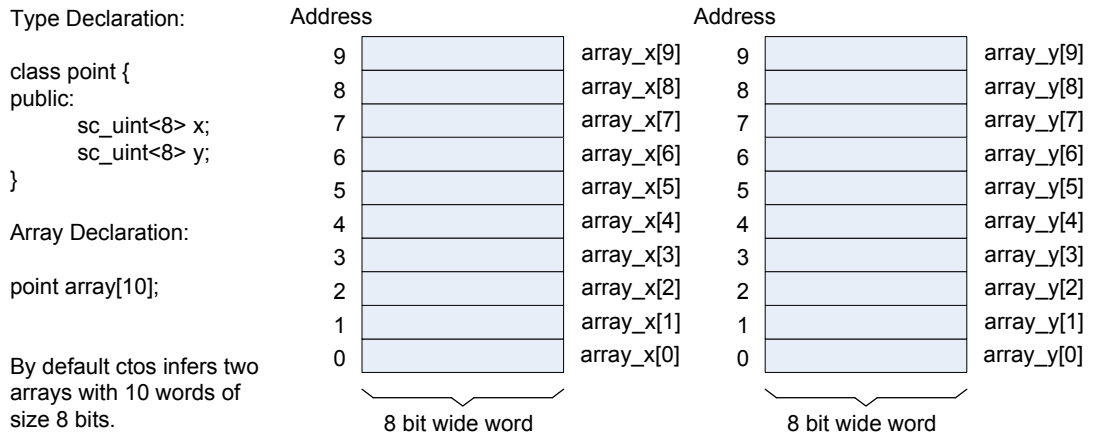
- When merging *data*, the arrays are mapped to *different bits of the word*.

In the resulting array, the number of words is the maximum of the words in the arrays being merged, and the data width is at least the sum of the data widths of the arrays being merged.

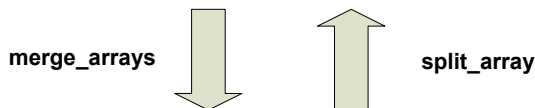
Additionally, when merging arrays accessed by multiple processes, you must ensure the following:

- Merging of arrays accessed by multiple processes can lead to memory contention. Users must ensure that processes do not access the merged array simultaneously in the same clock cycle.
- Merging of arrays could also lead to a situation where merged array is accessed simultaneously by a single process multiple times in the same cycle and can lead to memory contention. Users must manually resolve such memory contentions or use the **float_array_access** command to resolve such memory contentions.

Figure 8-19 Example Showing CtoS Creating Multiple Arrays for Arrays of POD-Structs



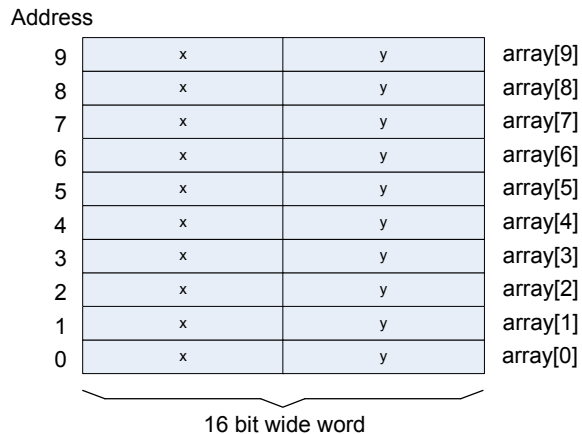
Use the `split_array` and `merge_array` commands if the default layout does not suit your design.



Another way to affect the layout of C++ objects is via the **ctos packed pragma** (Section 14.6.5). In the example below, only 1 array of 10 words of 16 bits is inferred. However, the directive to use the packed object model applies to all objects of type 'point' in the design.

Type Declaration:

```
#pragma ctoS packed
class point {
public:
    sc_uint<8> x;
    sc_uint<8> y;
};
```



8.3.3 Splitting Arrays

CtoS lets you *split an array* – based on either the *address* or *data* bits of the array elements – into two smaller arrays, using the **split_array** command (“[split_array](#)” on page E-147), as described in:

- “Requirements for Original Array, Bit Index, Resulting Array” on page 8-65
- “Handling Previous Micro-Architectural Choices, other Concerns” on page 8-65
- “Splitting an Array, based on Address Bits” on page 8-66
- “Splitting an Array, based on Data Bits” on page 8-73

8.3.3.1 Requirements for Original Array, Bit Index, Resulting Array

Before splitting an array, make sure the *original* array and the bit index *meet*, and the *resulting* arrays *will meet*, these requirements:

- The *original* array must not be an external array.
- The *original* array must not be marked with **pragma ctos dont_touch**.
- Earlier optimization phases in CtoS may remove certain array elements or trim the number of data bits, so be sure to specify the **bit_index** with respect to the newly trimmed array, not your *original* array.
- If you are splitting based on data bits (**-data** option), the **bit_index** must be greater than 0 and less than number of data bits in each element of the *original* array.
- If you are splitting based on address bits (**-addr** option), the **bit_index** must be less than the number of address bits of the *original* array.
- If you are splitting based on address bits (**-addr** option), the *original* array must have more than two elements.
- The data width of both of the *resulting* arrays must be a multiple of the minimum write width of the *original* array.

8.3.3.2 Handling Previous Micro-Architectural Choices, other Concerns

During the splitting of an array, CtoS will respect some previous micro-architectural decisions, as well as other aspects of the array, as follows:

- For each memory op accessing the array, an additional memory op is created on the same birthday edge.
- If the array accesses for the original array have been floated (see “[Floating Array Accesses](#)” on page 8-94), the new array accesses are also floated.

- If the original array is constrained to an edge or a stage/phase within a pipelined loop [see [“Constraining ops to Edges” on page 11-17](#) and [“Constraining ops to Expandable Edges” on page 11-19](#)], the new arrays are constrained in the same manner.

8.3.3.3 Splitting an Array, based on Address Bits

If you specify the **-addr** option to the **split_array** command, all elements in the array whose addresses contain the bit **bit_index** set to **0** are mapped to **array_id_0**, and all elements whose addresses contain the bit **bit_index** set to **1** are mapped to **array_id_1**, as illustrated in [Figure 8-20 on page 8-68](#).

The circuits for the read and write operations, before and after the use of the **split_array** command, are illustrated in [Figure 8-21 on page 8-69](#), [Figure 8-22 on page 8-69](#), [Figure 8-23 on page 8-70](#), and [Figure 8-24 on page 8-70](#).

When an array is split, accesses to the same array will use different read and write ports, which may not create contention.

The data width of the two new arrays will be the same as that of the original array.

The following sections describe more about splitting an array, based on address bits:

- [“Reasons for Splitting an Array, based on Address Bits” on page 8-66](#)
- [“Diagrams Showing How split_array Works, based on Address Bits” on page 8-67](#)
- [“Examples of Using split_array, based on Address Bits” on page 8-71](#)

Reasons for Splitting an Array, based on Address Bits

The reasons you might want to split an array, based on address bits, fall into two categories: based on the LSB and MSB address bits.

Reasons for performing an address split based on LSB:

- You want to read even (odd) words in the same cycle in which you are writing odd (even) words.

However, if you are always reading and writing both the even words (**i**) and the odd words (**i+1**), and **i** is always even, you may want to use the **restructure_array** command ([“restructure_array” on page E-136](#)), instead.

Reasons for performing an address split based on MSB:

- The buffering scheme has the reader accessing the low (high) half of the array, while the writer is accessing the high (low) half of the array.

- The array is not a power of **2** and would benefit from splitting and then flattening the top half of the array.

This is especially true for a 129-word ROM, where you can then propagate special accesses to **array[128]**.

Diagrams Showing How `split_array` Works, based on Address Bits

This section shows a series of diagrams illustrating how arrays are split using the CtoS **`split_array`** command with the **`-addr`** option

Figure 8-20 Splitting an Array, based on Address Bits (Where P Is bit_index)

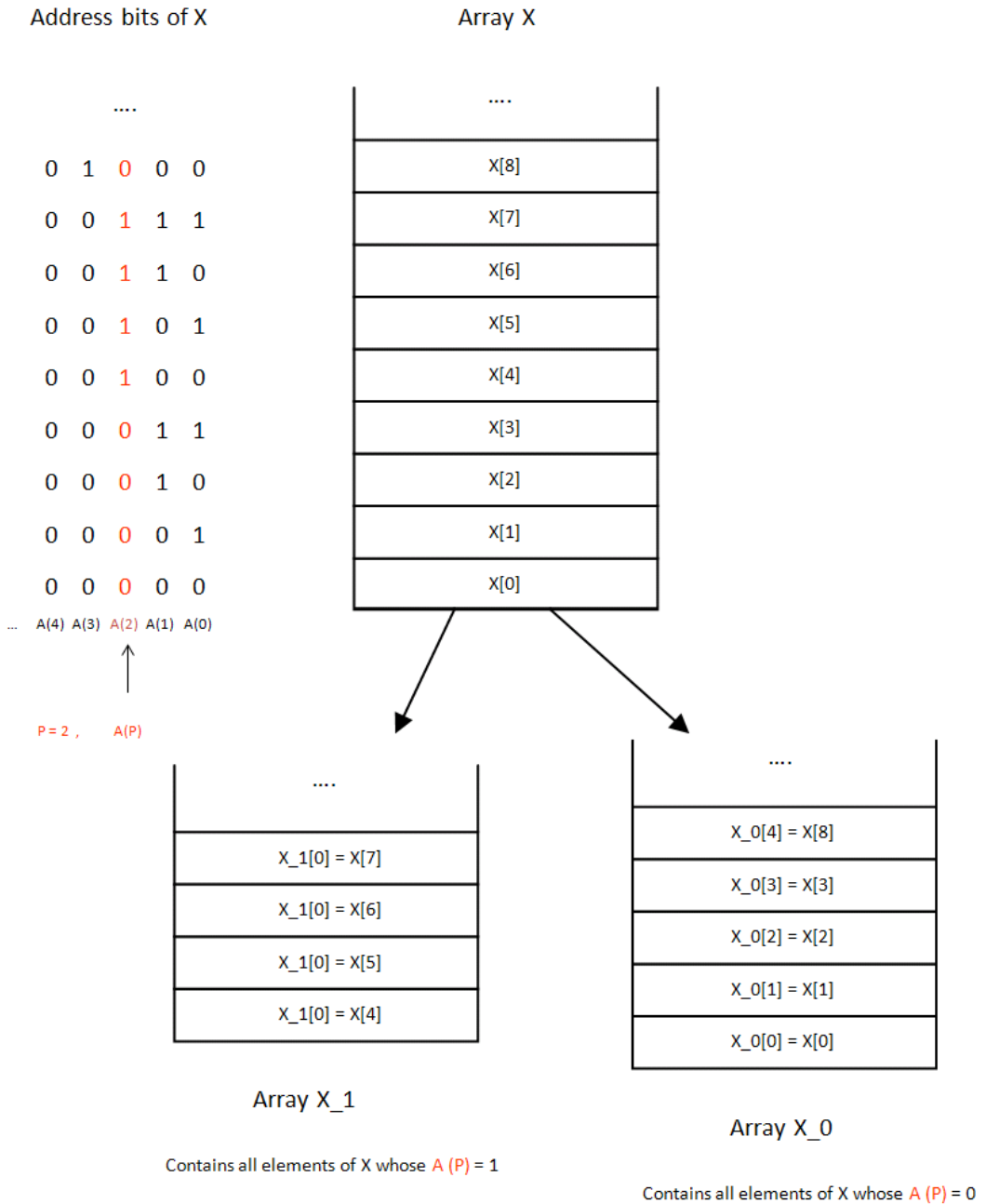


Figure 8-21 Circuit for Read Operation of Original Array (Being Split Using Address Bits)

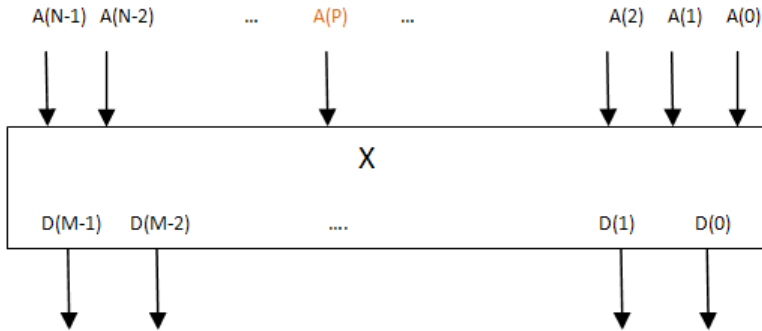


Figure 8-22 Circuit for Read Operation after Using the CtoS split_array -addr Command

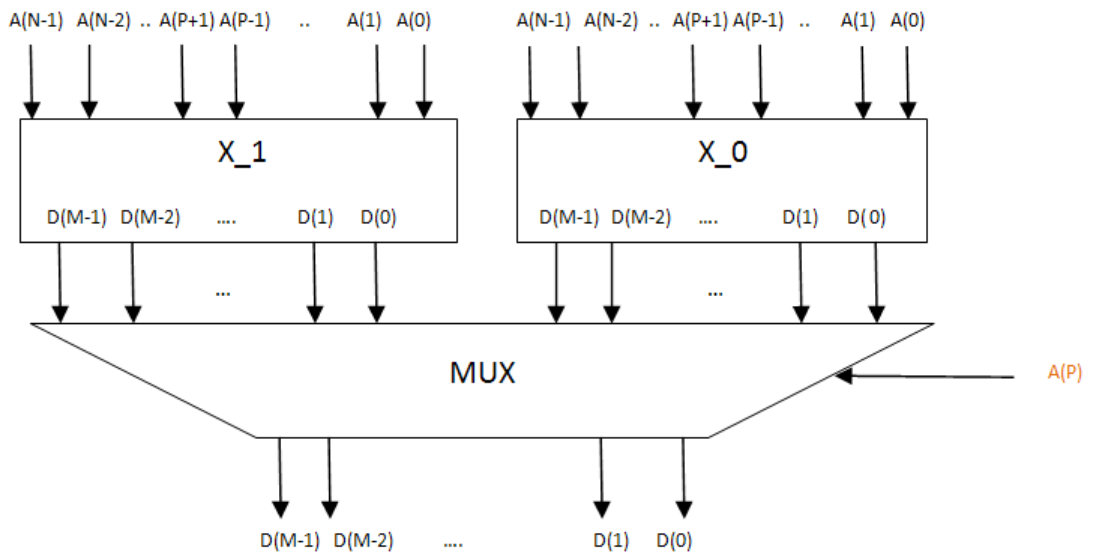


Figure 8-23 Circuit for Write Operation of Original Array (Being Split Using Address Bits)

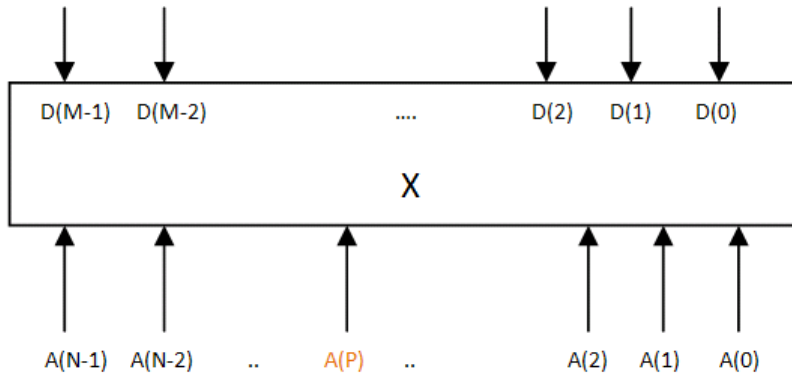
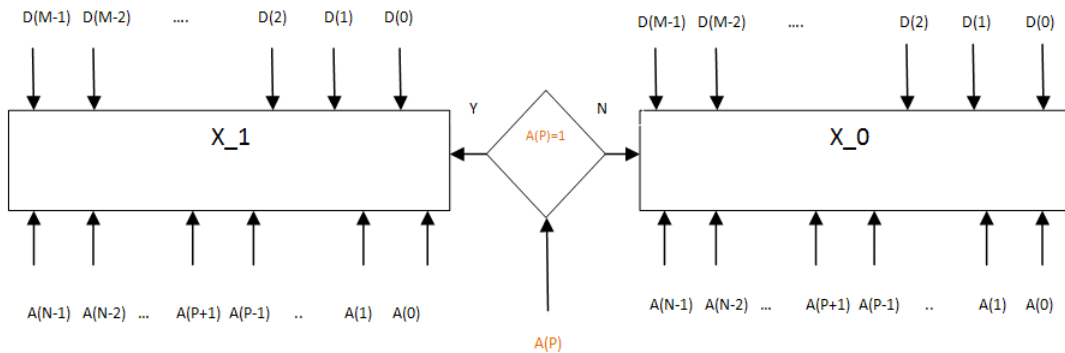


Figure 8-24 Circuit for Write Operation after Using the CtoS split_array -addr Command



Examples of Using `split_array`, based on Address Bits

This section shows three examples of using the CtoS `split_array` command with the `-addr` option.

Figure 8-25 Example: `split_array -addr 0 X`

Address of Array elements in X.

Array X.

1010
1001
1000
0111
0110
0101
0100
0011
0010
0001
0000

X[10]
X[9]
X[8]
X[7]
X[6]
X[5]
X[4]
X[3]
X[2]
X[1]
X[0]

Array X_1

X_1[4]=X[9]
X_1[3]=X[7]
X_1[2]= X[5]
X_1[1]= X[3]
X_1[0]=X[1]

Array X_0

X_0[5]=X[10]
X_0[4]=X[8]
X_0[3]=X[6]
X_0[2]=X[4]
X_0[1]=X[2]
X_0[0]=X[0]

Figure 8-26 Example: `split_array -addr 2 X`

Address of Array elements in X.

Array X.

1010
1001
1000
0111
0110
0101
0100
0011
0010
0001
0000

X[10]
X[9]
X[8]
X[7]
X[6]
X[5]
X[4]
X[3]
X[2]
X[1]
X[0]

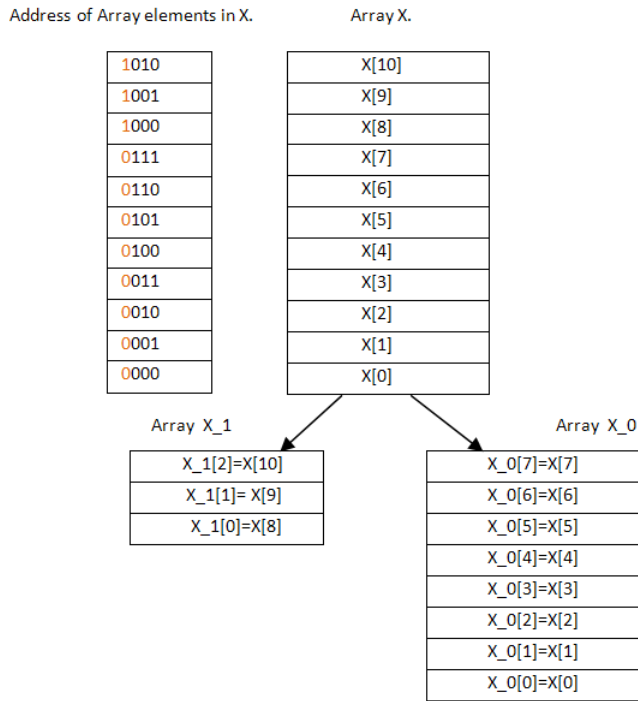
Array X_1

X_1[3]=X[7]
X_1[2]=X[6]
X_1[1]=X[5]
X_1[0]=X[4]

Array X_0

X_0[6]=X[10]
X_0[5]=X[9]
X_0[4]=X[8]
X_0[3]=X[3]
X_0[2]=X[2]
X_0[1]=X[1]
X_0[0]=X[0]

Figure 8-27 Example: `split_array -addr 3 X`



8.3.3.4 Splitting an Array, based on Data Bits

If you specify the **-data** option to the **split_array** command, bits **0** to (**bit_index** - 1) in each array element are placed in **array_id_0**; the remaining bits are placed in **array_id_1**, as shown in [Figure 8-28 on page 8-74](#).

The circuits for the read and write operations, before and after the use of the **split_array** command, are illustrated in [Figure 8-29 on page 8-75](#), [Figure 8-30 on page 8-75](#), [Figure 8-31 on page 8-76](#), and [Figure 8-32 on page 8-76](#).

The data width of **array_id_0** will be **bit_index** bits and of **array_id_1** will be (**M** - **bit_index**) bits, where **M** is the data width of the original array. The total number of words in the two new arrays will be the same as the original array.

The following sections describe more about splitting an array, based on address bits:

- “Diagrams Showing how **split_array** Works, based on Data Bits” on page 8-74
- “Example of Using **split_array**, based on Data Bits” on page 8-77

Diagrams Showing how split_array Works, based on Data Bits

This section shows a series of diagrams illustrating how arrays are split using the CtoS `split_array` command with the `-data` option.

Figure 8-28 Splitting an Array, based on Data Bits (Where Q Is bit_index)

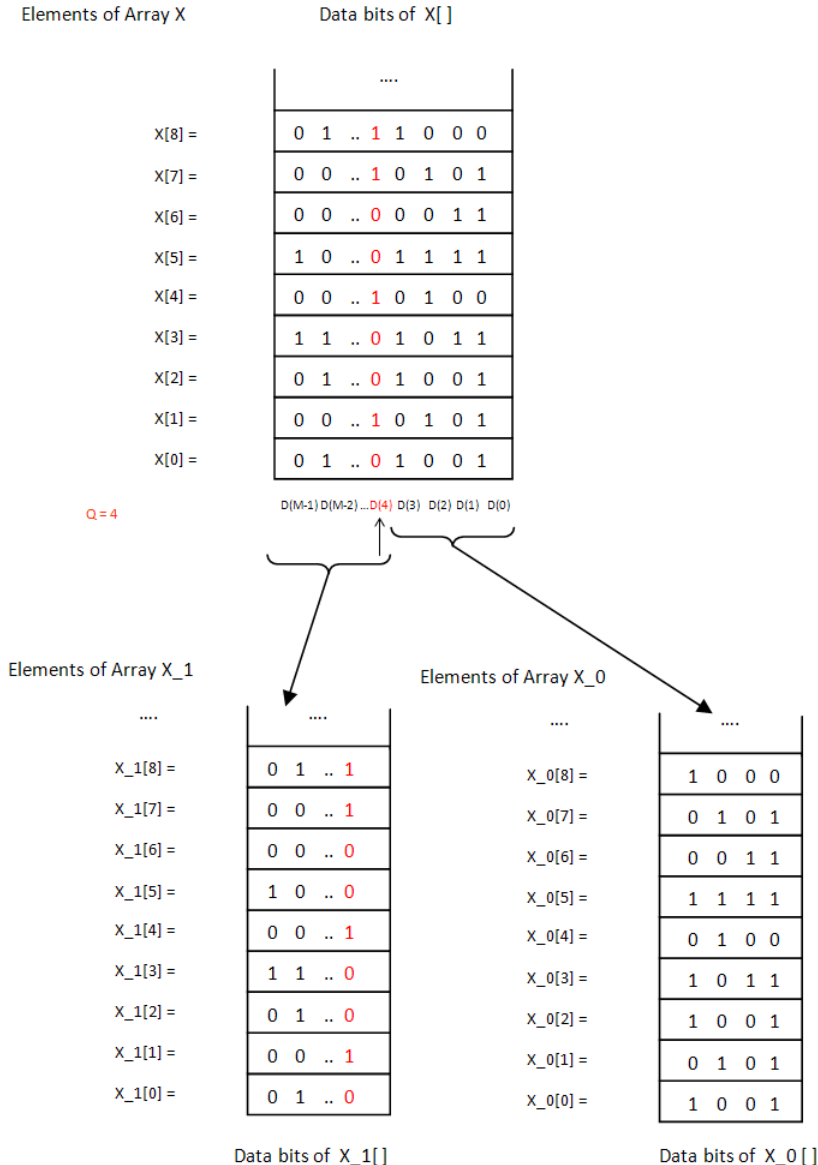


Figure 8-29 Circuit for Read Operation of Original Array (Being Split Using Data Bits)

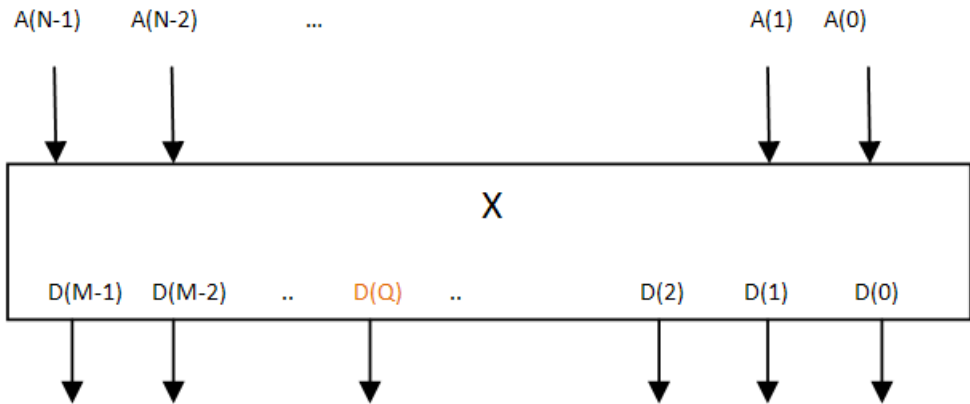


Figure 8-30 Circuit for Read Operation after Using the CtoS split_array -data Command

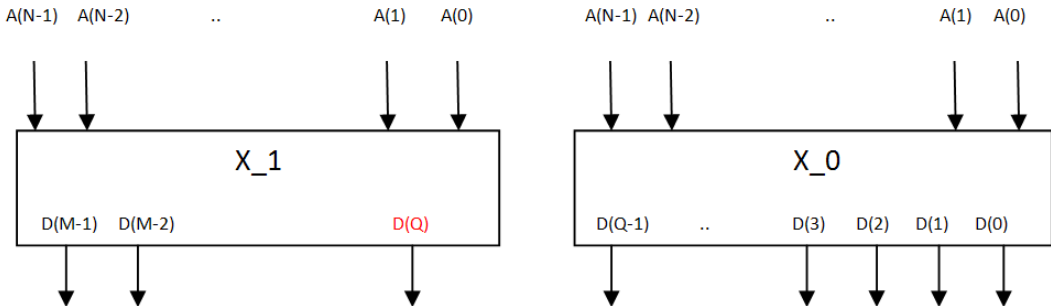


Figure 8-31 Circuit for Write Operation of Original Array (Being Split Using Data Bits)

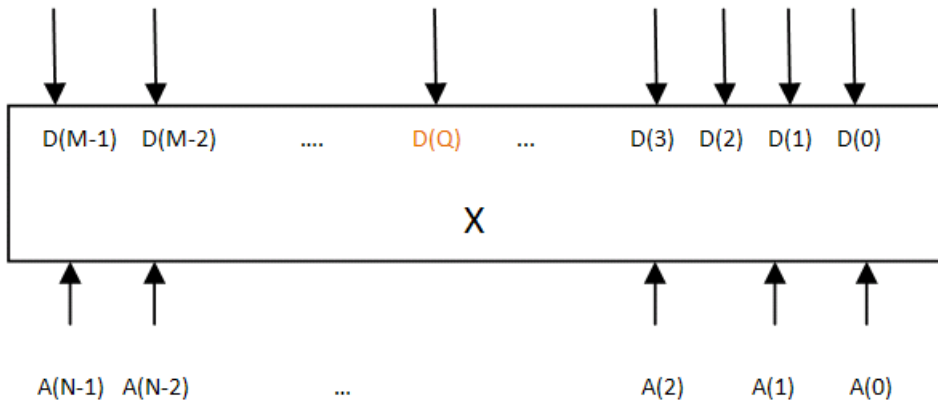
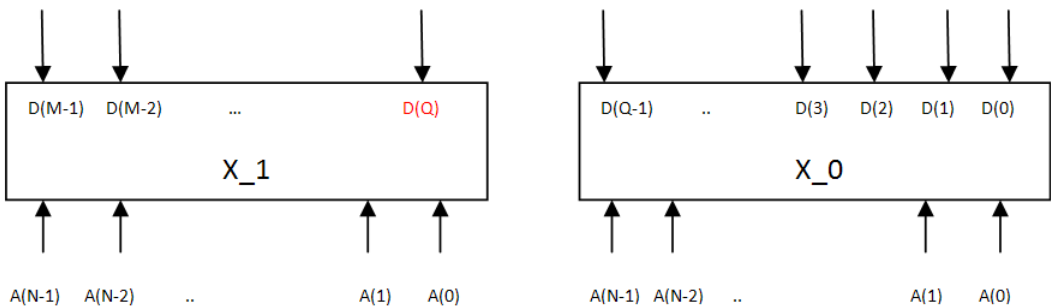


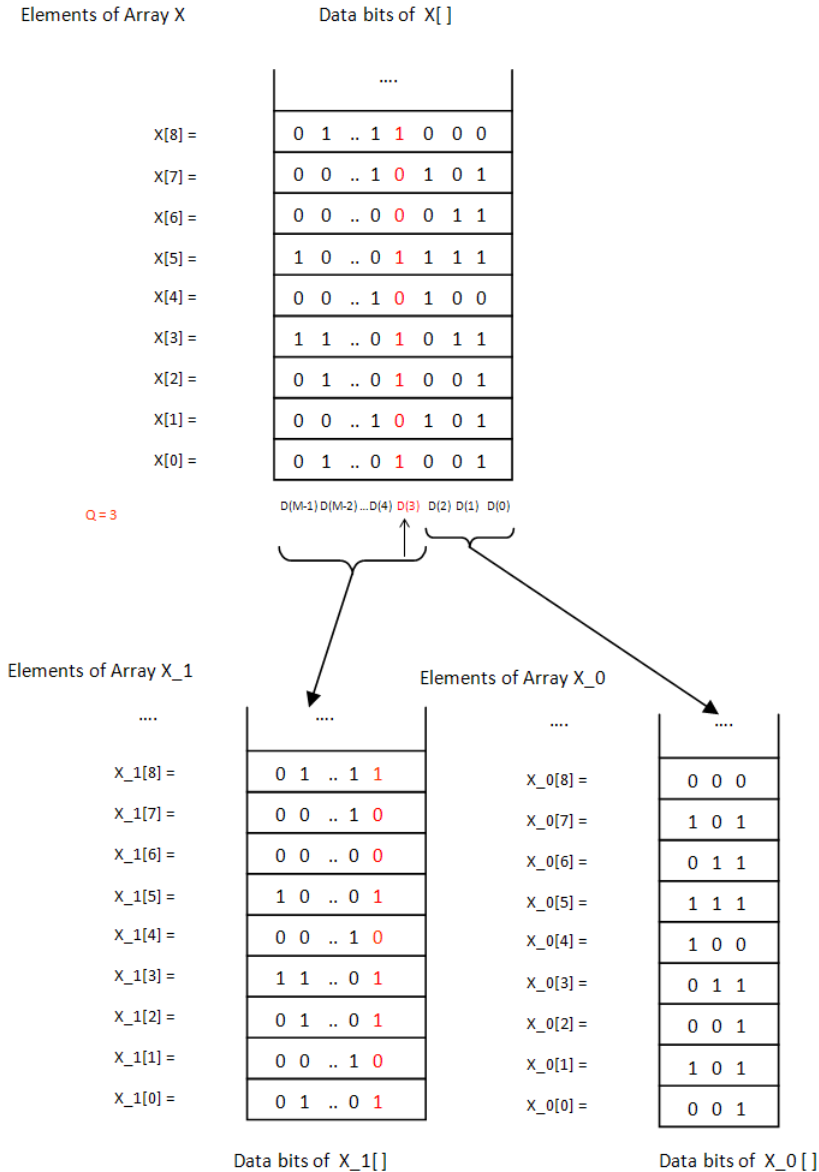
Figure 8-32 Circuit for Write Operation after Using the CtoS split_array -data Command



Example of Using split_array, based on Data Bits

This section shows an example of using the CtoS **split_array** command with the **-data** option.

Figure 8-33 Example: split_array -data 3 X



8.3.4 Restructuring Arrays

CtoS lets you *restructure* an array, that is, change the number of words in the array, using the **restructure_array** command (“[restructure_array](#)” on page E-136).

The array is scaled up (an *increased* number of words) or scaled down (a *decreased* number of words), in multiples of powers of 2.

This feature is further described in:

- “[Requirements for Original and Resulting Array in Restructuring](#)” on page 8-78
- “[Scaling Up an Array \(Increased Number of Words\)](#)” on page 8-78
- “[Scaling Down an Array \(Decreased Number of Words\)](#)” on page 8-86

8.3.4.1 Requirements for Original and Resulting Array in Restructuring

Before restructuring an array, make sure the *original* array *meets*, and the *resulting* array *will meet*, these requirements:

- The *original* array must not be an external array.
- The *original* array must not be marked with **pragma ctos dont_touch**.
- If you are *decrementing* the data width, the width of the *original* array must be a multiple of 2.
- Earlier optimization phases in CtoS may trim the width of the *original* array such that it is not a power of 2. You must be aware of the impact of trimming on the array.
- If you are *decrementing* the data width, the width of the *resulting* array must be a whole number, must be at least 1 bit.
- If you are *incrementing* the data width, the words of the *resulting* array must be more than 2.
- The data width of the *resulting* array must be a multiple of the minimum write width of the original array.
- If you are restructuring an array accessed by multiple processes, it results in multiple read and/or write accesses. Since, the array accesses of an array accessed by multiple processes are fixed on edge, you must use the **float_array_accesses** command to ensure the design can be scheduled.

8.3.4.2 Scaling Up an Array (Increased Number of Words)

To *scale up* an array, you use a *positive* integer for **addr_width_delta** with the **restructure_array** command.

The following sections describe more about scaling up an array:

- “[Results of Scaling Up an Array](#)” on page 8-79
- “[Generic Example of Scaling Up an Array](#)” on page 8-79
- “[Calculating New Memory Address after Scaling Up an Array](#)” on page 8-80

- “Specific Example of Scaling Up an Array” on page 8-82
- “Impact on Memory ops when Scaling Up an Array” on page 8-83

Results of Scaling Up an Array

After scaling up an array, the data width is partitioned into **resize_factor** parts (which is equal to $2^{addr\_width\_delta}$), rearranged, and placed in the new array.

8.3.4.3 Results of Scaling up an Array Are as Follows:

- The **new_array_words** will be increased, as follows:

```
new_array_words = original_array_words * resize_factor
```

- The **new_address_width** will be increased, as follows:

```
new_address_width = addr_width_delta (if original array has one word)
else
new_address_width = original_address_width + addr_width_delta
```

- The **new_data_width** will be decreased, as follows:

```
new_data_width = original_width/resize_factor
```

Note An error will be reported if **new_data_width** is not an integer.

- The new **location** of the array elements will be:

For an array **Arr[X]**, each element will have **resize_factor** equal parts.

Each part will be placed in the new array from location **Arr[X * 2ⁿ + j]**

Each part will have $[(j+1) * (M/2^n) - 1, \dots, (j * M/2^n)]$ bit positions of the original array, where **j** = (0, ..., **resize_factor** - 1) and **X** is the index in the original array.

Generic Example of Scaling Up an Array

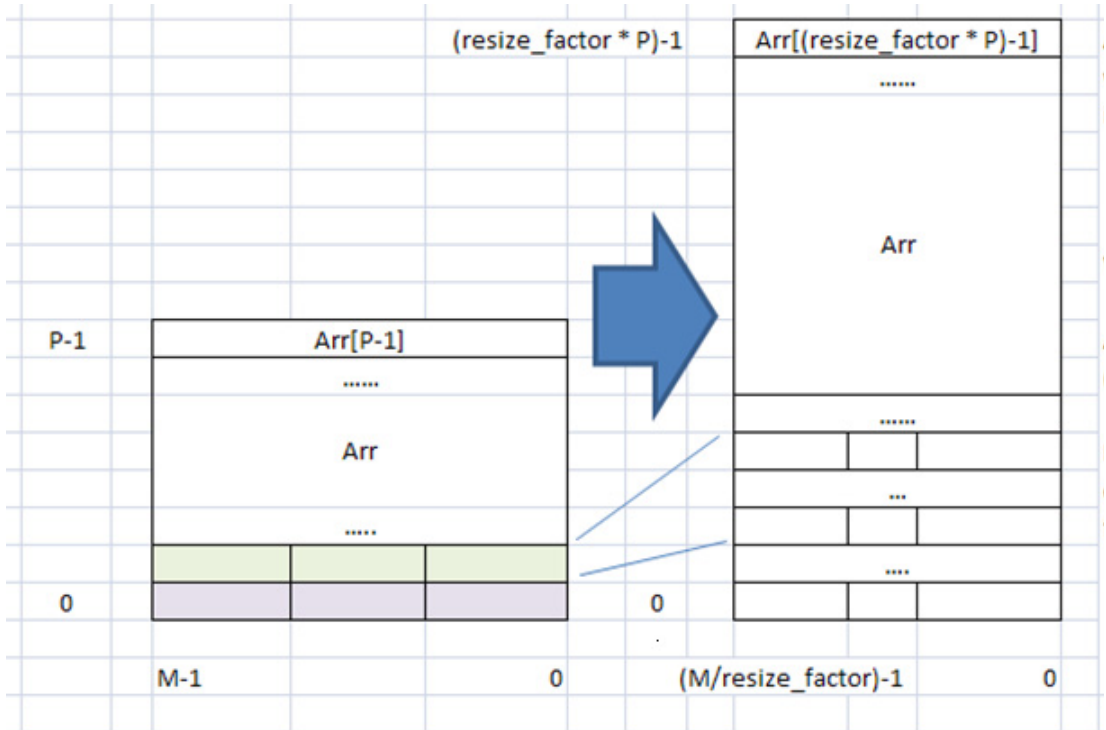
Figure 8-34 on page 8-80 shows a diagram of using the **restructure_array** command to scale up an array **Arr** that has:

- **P** words
- **M** width
- address width of **N** bits
- **resize_factor** = $2^{addr\_width\_delta}$

- $1 < \text{resize\_factor} < M$
- $M \% \text{resize\_factor} == 0$

Note If $\text{resize\_factor} == M$, the new data width would be 1 bit, which would flatten the array and produce an error.

Figure 8-34 Generic Example of Scaling Up an Array



Calculating New Memory Address after Scaling Up an Array

After scaling up an array, the new address starts at $(\text{original\_address} * \text{resize\_factor})$ and ends at $(\text{new\_address} + \text{resize\_factor} - 1)$ where $\text{new\_address} = \text{original\_address} * \text{resize\_factor}$.

Thus, an array element $\text{Arr}[X]$ with data width M will have $[M-1, \dots, 0]$ bits, as shown in [Figure 8-34 on page 8-80](#). There will be resize_factor parts of the original element, and the new array locations of each part are shown in [Table 8-2 on page 8-82](#).

The calculations can be generalized as follows: array location $\text{Arr}[X * 2^n + j]$ will have $\{(j+1) * (M/2^n) - 1, \dots, (j * M/2^n)\}$ bit positions of the original array where $j = (0, \dots, \text{resize\_factor} - 1)$.

Table 8-2 New Array Locations after Scaling Up

part number	new array location	bit positions of original array in new array
1	$\text{Arr}[X * 2^n]$	$[(M/2^n) - 1, \dots, 0]$
2	$\text{Arr}[X * 2^n + 1]$	$[\{2 * (M/2^n)\} - 1, \dots, (M/2^n)]$
3	$\text{Arr}[X * 2^n + 2]$	$[\{3 * (M/2^n)\} - 1, \dots, (2 * M/2^n)]$
...		
$\text{resize\_factor} - 1$	$\text{Arr}[X * 2^n + \text{resize\_factor} - 1]$	$[\{((\text{resize\_factor} - 1) + 1) * (M/2^n)\} - 1, \dots, ((\text{resize\_factor} - 1) * M/2^n)]$

Specific Example of Scaling Up an Array

In [Figure 8-35 on page 8-83](#), the original array **A1** has 12 elements, each with a width of 6 bits. After restructuring, **A1** (the name is not changed) has 24 elements, each with a width of 3 bits.

The elements are arranged so the original **A1[0]<0>** to **A1[0]<2>** is now in **A1[0]** of the new array, the original **A1[0]<3>** to **A1[0]<5>** is now in **A1[1]** of the new array, and so on.

In order to partition the array by 2, you would use the following command:

```
restructure_array 1 A1
```

The original 4 bits are needed to address the array because the array has 12 words. After restructuring, 5 bits are needed to address the array, which now has 24 words.

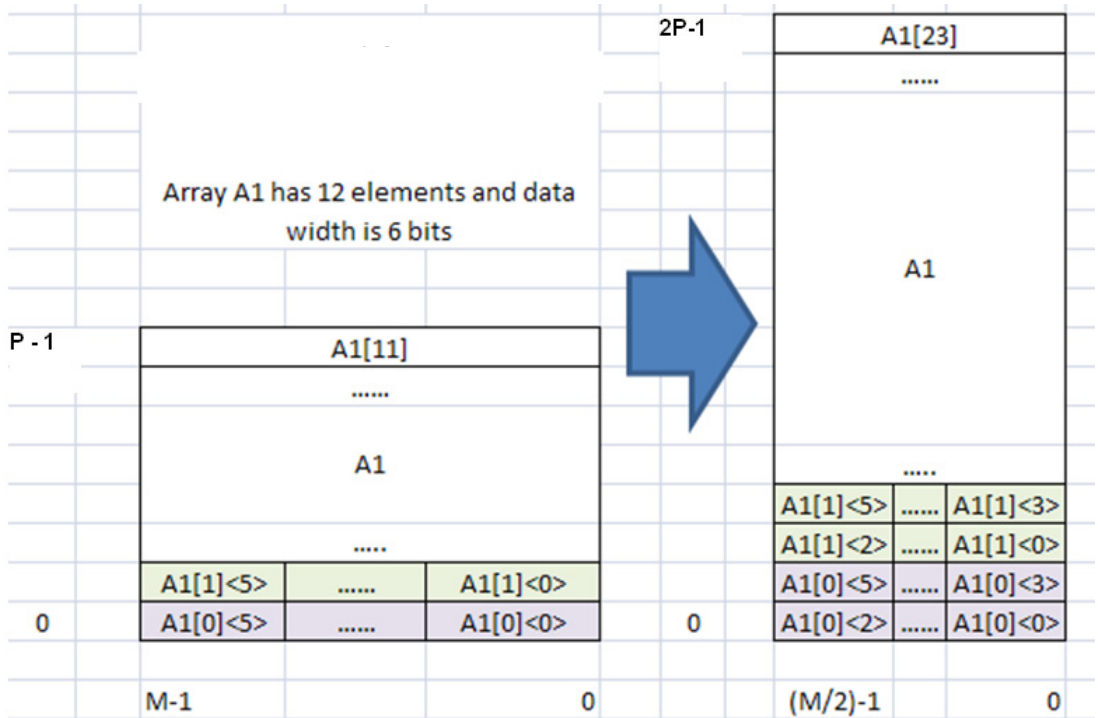
If you used the following command:

```
restructure_array 2 A1
```

The array **A1** would then have 48 elements, and 6 bits would be needed. However, this would be an error, since the new width would not be an integer $[6/(2^2)]$, as the original width was 6 bits.

The new locations of each element will be changed. **A1[2]** in the original array will have two parts and be placed in **A1[4]** and **A1[5]** in the new array. New array locations **A1[4]** contains bits **3...0** and **A1[5]** contains bits **7...4** of the original **A1[2]**.

Figure 8-35 Specific Example of Scaling Up an Array



Impact on Memory ops when Scaling Up an Array

In Figure 8-34 on page 8-80, both the read and write operations need new locations for the new parts of the original elements.

The new address starts at (**original_address** << **n**) and ends at ((**original_address** << **n**) + **resize_factor** - 1).

The parts of the original array **Arr** are placed from location **Arr[new address]**, **Arr[new address + 1]**, ..., **Arr[new address + resize_factor - 1]**.

Figure 8-36 on page 8-84 shows generically how a **read** operation is handled during scaling up of an array, and Figure 8-37 on page 8-84 shows how this is done for the example previously shown in Figure 8-35 on page 8-83. In the specific example, the width of the address port can be reset, and the first **n** pin (**resize_factor** = 2^n) can be set to appropriate values, for example, the first read to value **0**, the second read to value **1**. In general, you are setting the value of new **n** pins from **0** to **resize_factor - 1**.

Similarly, Figure 8-38 on page 8-85 and Figure 8-39 on page 8-85 show the same for a **write** operation.

Figure 8-36 Generic Example of Read Operation during Scaling Up

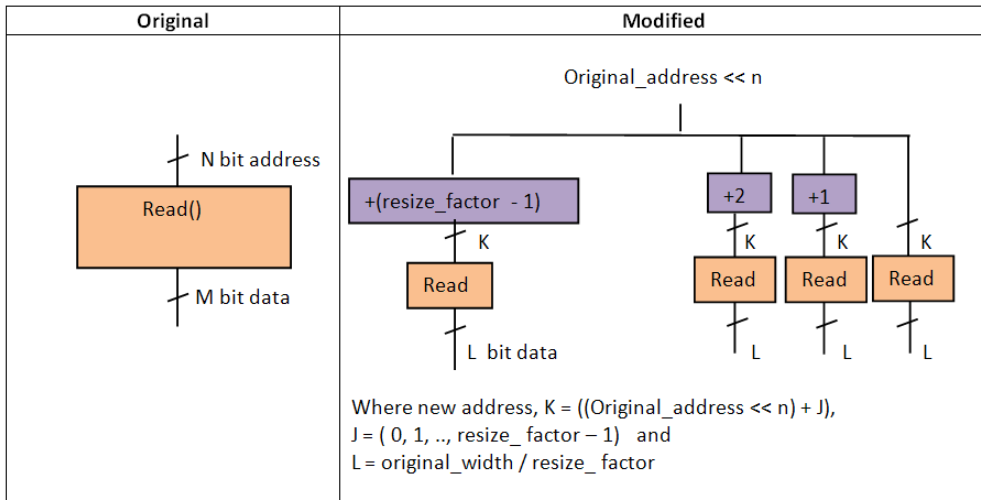


Figure 8-37 Specific Example of Read Operation during Scaling Up

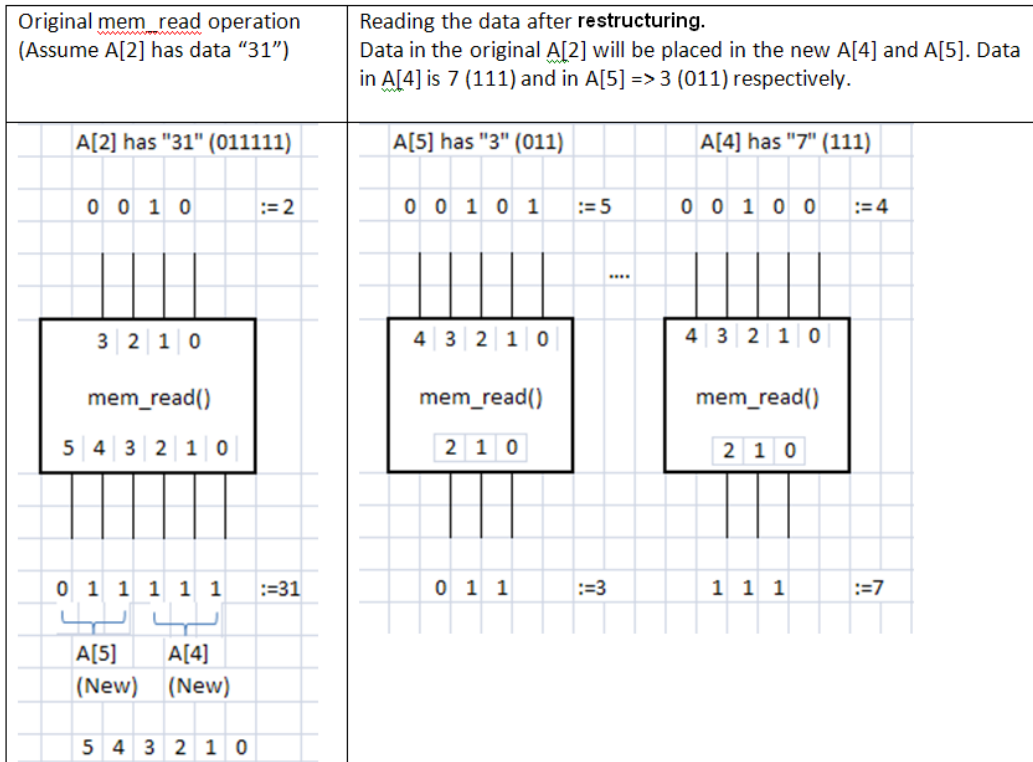


Figure 8-38 Generic Example of Write Operation during Scaling Up

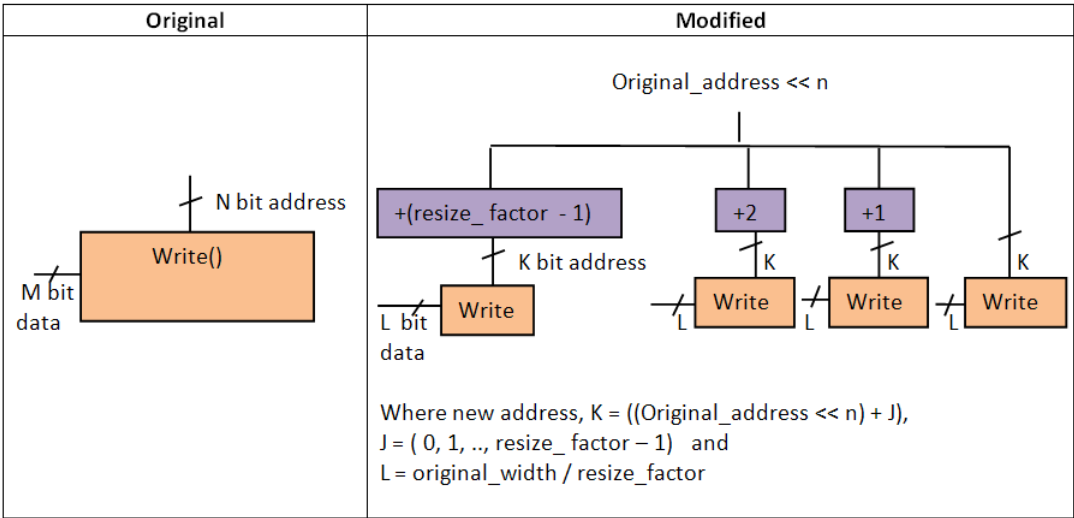
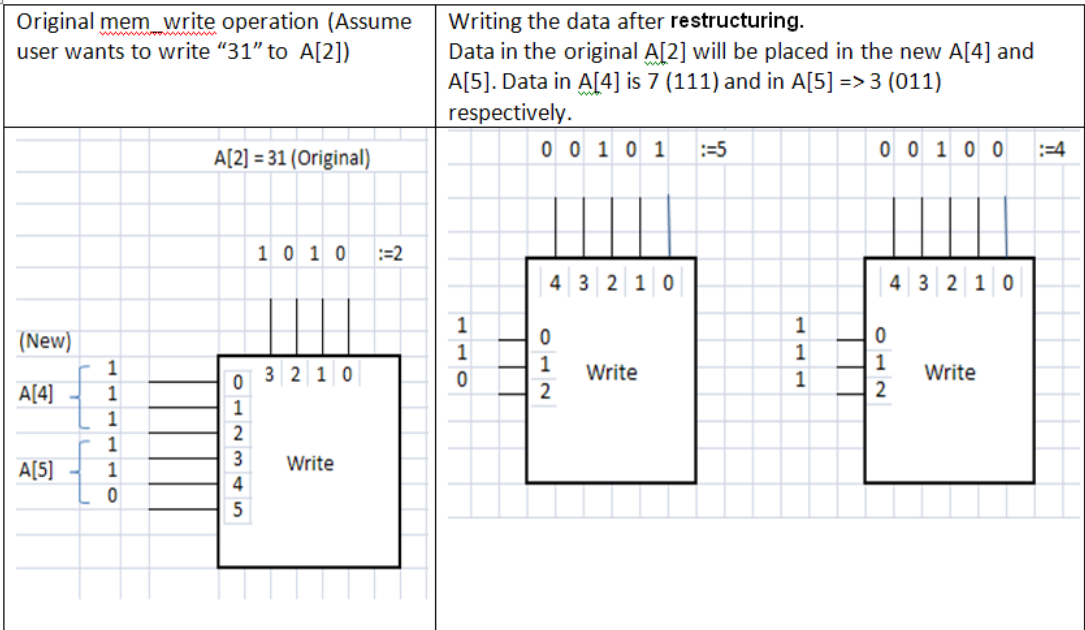


Figure 8-39 Specific Example of Write Operation during Scaling Up



8.3.4.4 Scaling Down an Array (Decreased Number of Words)

To *scale down* an array, you use a *negative* integer for *addr_width_delta* with the **restructure_array** command.

The following sections describe more about scaling down an array:

- [“Results of Scaling Down an Array” on page 8-86](#)
- [“Generic Example of Scaling Down an Array” on page 8-87](#)
- [“Calculating New Memory Address after Scaling Down an Array” on page 8-87](#)
- [“Specific Example of Scaling Down an Array” on page 8-88](#)
- [“Impact on Memory ops when Scaling Down an Array” on page 8-88](#)

Results of Scaling Down an Array

After scaling down an array, the total number of words in the array is partitioned by **resize_factor** parts (which is equal to $2^{\text{addr\_width\_delta}}$), and the data is appended and placed in the new array.

When the $(\text{original\_array\_words} \% \text{resize\_factor})$ is greater than 0, the restructured array will have padded data for the non-existing $[\text{resize\_factor} - (\text{original\_array\_words} \% \text{resize\_factor})]$ array elements of the original array. You will get an error if the **resize_factor** is greater than or equal to the original words.

This results of scaling down an array are as follows:

- The **new_array_words** will be decreased, as follows:

```
if (original_array_words % resize_factor) == 0, then
new_array_words = original_array_words/resize_factor
else
new_array_words = (original_array_words / resize_factor) + 1
```

- The **new_address_width** will be decreased, as follows:

```
new_address_width = original_address_width - addr_width_delta
```

- The **new_data_width** will be increased, as follows:

```
new_data_width = original_width * resize_factor
```

- The new **location** of the array elements will be:

For an array **Arr[X]**, each element will be part of an element in the new array. An element **Arr[X]** of the original array will be placed at location **Arr[X / *resize_factor*]** in the new array and will be at bit position from $((X \% \text{resize\_factor}) * M)$ to $((X \% \text{resize\_factor}) * M) + M - 1$ where X is the index in the original array.

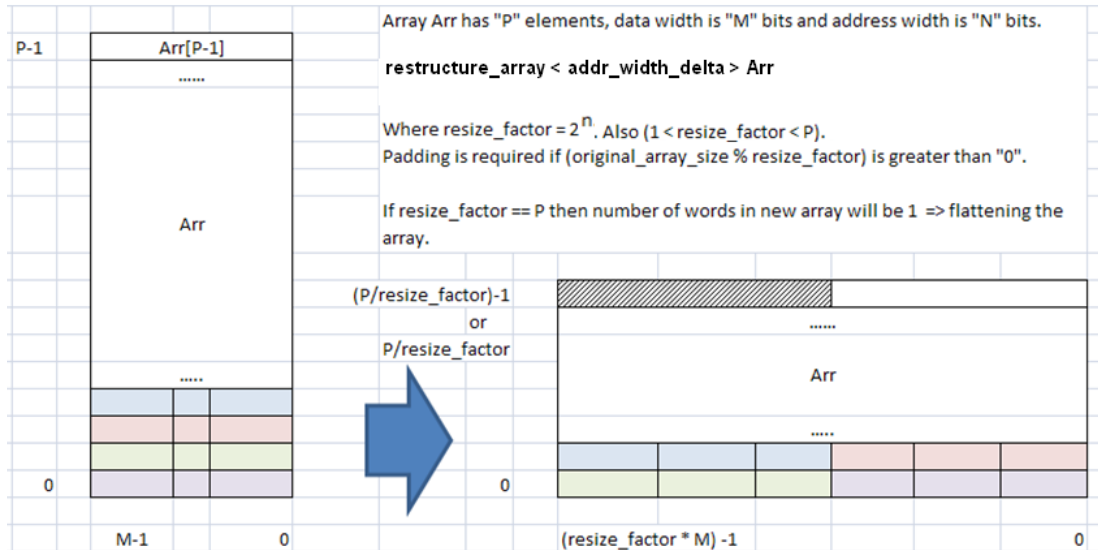
Generic Example of Scaling Down an Array

Figure 8-40 on page 8-87 shows a diagram of using the `restructure_array` command to scale down an array `Arr` that has:

- **P** words
- **M** width
- address width of **N** bits
- **resize_factor** = $2^{\text{addr\_width\_delta}}$
- $1 < \text{resize\_factor} < P$

Note If **resize_factor** == **P**, the number of words in the new array would be 1, which would flatten the array and produce an error.

Figure 8-40 Generic Example of Scaling Down an Array



Calculating New Memory Address after Scaling Down an Array

After scaling down an array, the new address is **old_address/resize_factor**. To place the data at the appropriate bit position within the new address, the start and end bit positions must be calculated, as follows:

```
start_bit = (original_address % resize_factor) * original_data_width
end_bit = start_bit + original_data_width - 1
```

Element **Arr[X]** of the original array will be placed in the new array at location **Arr[X / resize_factor]** and will be at bit position from **((X % resize_factor) * M)** to **((X % resize_factor) * M + M-1)**, where **X** is the index in the original array.

Specific Example of Scaling Down an Array

In [Figure 8-41 on page 8-88](#), the original array **A1** has 11 elements, each with a width of 3 bits.

Using the following command will produce a new **A1** (the name is not changed) with 6 elements, each with a width of 6 bits:

```
restructure_array -1 A1
```

The original **A1**[0]<0> to **A1**[0]<2> and **A1**[2]<0> to **A1**[2]<2> are placed in bit 2 to 0 and bit 3 to 5 of **A**[0] in the new array. The new array **A1**[5]<5,3> is padded with 0 since the original array does not have an element in the 12th position.

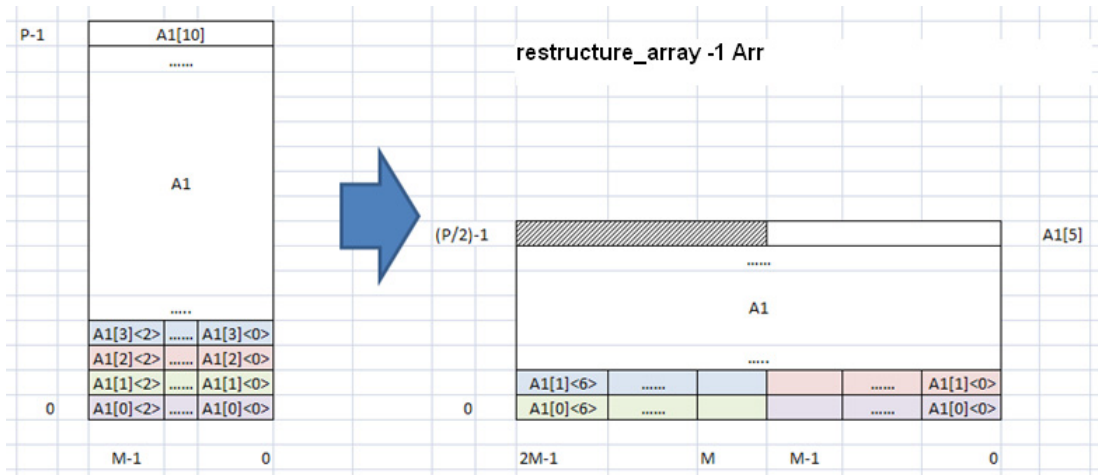
The original 4 bits are needed to address the array because the array has 11 words. After restructuring, 3 bits are needed to address the array, which now has 6 words.

If you used the following command:

```
restructure_array -2 A1
```

The new array **A1** would then have 3 elements, and 2 bits would be needed.

Figure 8-41 Specific Example of Scaling Down an Array



Impact on Memory ops when Scaling Down an Array

In [Figure 8-40 on page 8-87](#), both the read and write operations need new locations for the original elements. An element **Arr**[**X**] of the original array will be placed in the new array at location **Arr**[**X** / **resize_factor**] and will be in bit position from $((X \% \text{resize\_factor}) * M)$ to $((X \% \text{resize\_factor}) * M) + M - 1$ where **X** is the index in the original array.

[Figure 8-42 on page 8-89](#) shows generically how a **read** operation is handled during scaling down of an array, and [Figure 8-43 on page 8-89](#) shows how this is done for the example previously shown in [Figure 8-41 on page 8-88](#). Similarly, [Figure 8-44 on page 8-90](#) and [Figure 8-45 on page 8-90](#) show the same for a **write** operation.

Figure 8-42 Generic Example of Read Operation during Scaling Down

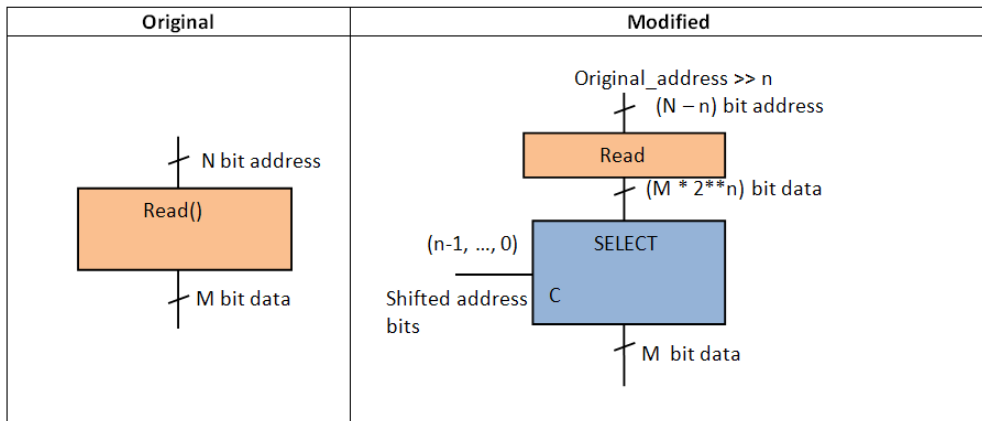


Figure 8-43 Specific Example of Read Operation during Scaling Down

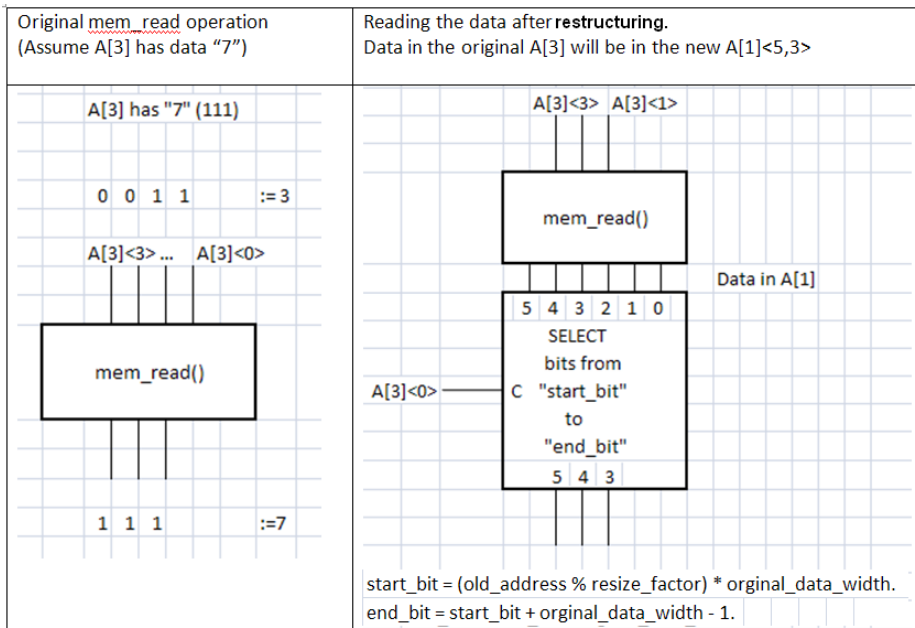


Figure 8-44 Generic Example of Write Operation during Scaling Down

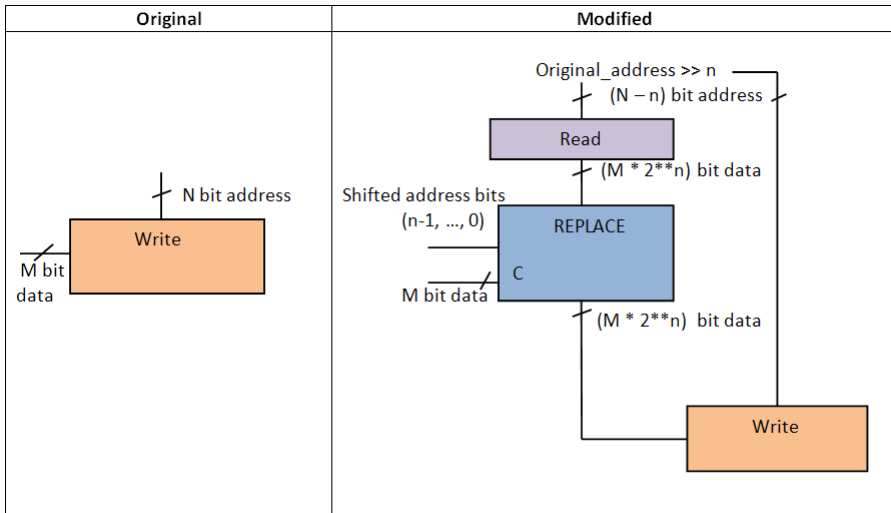
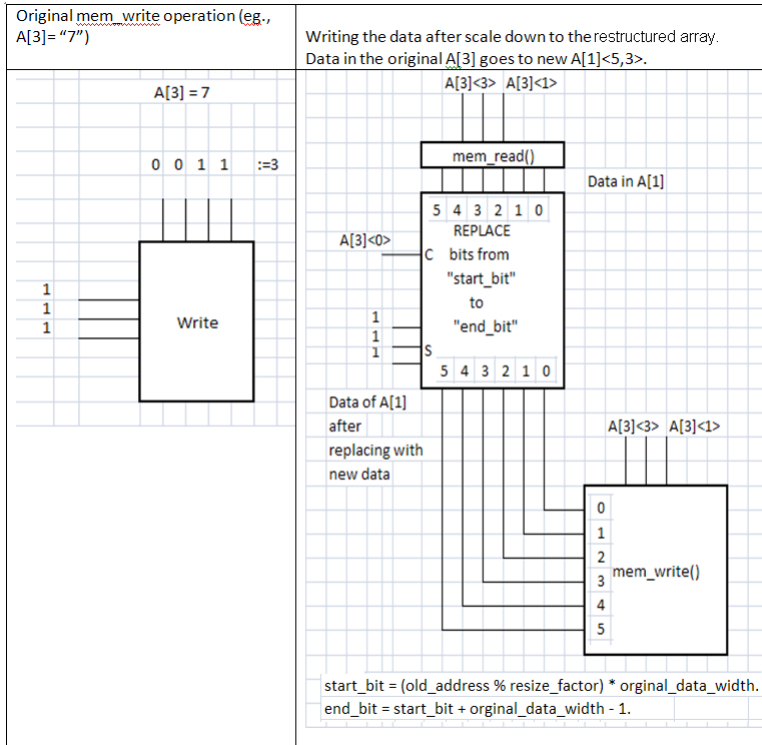


Figure 8-45 Specific Example of Write Operation during Scaling Down



8.3.5 Floating I/O Accesses

Floating I/O accesses is a preliminary feature.

CtoS maintains the cycle timing behavior of a block by scheduling all inputs and outputs on their *birth edges*. However, if you know that some I/O operations can move without breaking the cycle timing behavior of the block, you should apply *floating constraints* to improve QoR via register reduction.

In many designs, certain inputs (for example, registers) are known to be *stable* for many cycles. Reading a stable input will produce the same value in any cycle of the region in which that input is stable.

In addition, certain inputs can change value over many cycles but the values are not used or the design is not sensitive to the accuracy of the value during that period. These inputs are *volatile* within a region.

CtoS gives you the flexibility to *float* I/O accesses, that is, to specify the region of stability or volatility for a set of inputs.

Using Floating Constraints for Stable I/Os

In some designs, inputs are read, and outputs are written, within specific regions, and are then stable for the remainder of the time. For example, input registers are written during setup and are then stable for the remainder of the time. In this case, CtoS can optimize registers that must store read values by taking them directly from read nets at any state of the region, since read values are stable throughout the region.

Similarly, if an output is not being read externally while in the region, then CtoS can optimize scheduling of write operations by moving them to the same cycle in which the value is produced. Writing outputs in the same cycle in which they are produced minimizes the number of registers.

With the floating I/O accesses feature, CtoS provides the option to *float* these types of nets, that is, to specify that these nets *not* be limited to being scheduled on their birth edges.

Checking Stability of I/Os in Simulation

The stability of read values (for the specified floating region) is verified in CtoS-generated Verilog simulation models using an assertion-checker task. The read values are stored when entering a region, and then in every state of the region, the assertion checker task is called to verify that the read values have not changed (by comparing the stored values with the current values).

For output values, the simulation model sets the value of the outputs to **X** when entering the region. This ensures that if there are paths in the region where the outputs are not overwritten, the simulation trace will show **Xs** for these outputs in the waveforms.

Using Floating Constraints for Volatile I/Os

In some designs, certain inputs can change value over many cycles but the values are not used or the design is not sensitive to the accuracy of the value during that period. These inputs are *volatile* within a region. In this case, CtoS can optimize registers that must store read values by taking them directly from read nets at any state of the region, since the read can occur anytime throughout the region.

Similarly, if design isn't sensitive to when an output is read externally, then CtoS can optimize scheduling of write operations by moving them to the same cycle in which the value is produced. Writing outputs in the same cycle in which they are produced minimizes the number of registers.

With the volatile option of the floating I/O accesses feature, CtoS provides the option to *float* these types of nets, that is, to specify that these nets *not* be limited to being scheduled on their birth edges.

When the **volatile** option is specified, the assertion-checker task (as described in the previous section) is *not* added to the simulation model. It is valid for the value of this input to change in this region.

Important Specifying **volatile** option may lead to simulation mismatches if the testbench expects certain values in that region. Furthermore, individual bits may be set at different cycles in that region also leading to simulation mismatches.

Note The volatile option is not supported for region that starts with a fork node.

Requirements for Floating I/O Regions

The specified *floating region* must:

- apply to at least one net; for example, you cannot specify a region that has no I/O ops.
- have only one entry edge.
- not overlap with another floating region that refers to the same nets.
- not overlap with a protocol region that refers to the same nets.
- not overlap with a region preserved, unless you used the **-state_only** option.
- not intersect a pipelined loop containing write ops to which the floating would apply.

Also note that custom ops (calls to functions, combinational or sequential) are allowed if the called functions do *not* contain reads or writes to the floated nets – if they do, you will get an error.

Using the Floating I/O Accesses Feature

To use the floating I/O accesses feature in the CtoS GUI, select **Edit -> Region -> Float I/O Accesses**.

You will see the **Region Window**, with the **Float I/O Accesses** tab selected, as shown in [Figure 8-46 on page 8-93](#), with a listing of all valid end nodes for the floating region.

You can also get to this window using the context menu for loops and nodes.

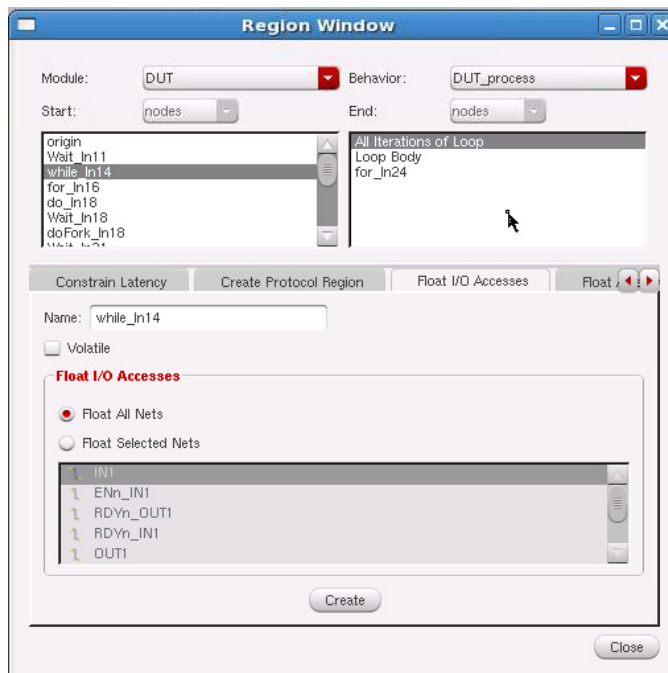
In this window, you can specify a name for the float constraint (if you do not want to use the default), and you can choose either that *all* nets be floated or that only *selected* nets be floated.

To specify that this is a *volatile* access, click on the **Volatile** checkbox.

Notes

- The floating I/O accesses feature will float individual I/O read and write operations, but it will not group them as a floatable I/O protocol (the latter is tentatively planned for a future CtoS release).
- There is a set of behavioral object attributes, **float_constraints**, for this feature; see “[Floating Constraint Object Attributes \(float_constraints\)](#)” on page D-48.
- You can also use the **float_io_accesses** command (“[float_io_accesses](#)” on page E-71).

Figure 8-46 Region Window Dialog Showing the Float I/O Accesses Tab



8.3.6 Floating Array Accesses

If an array mapped to a memory is accessed by multiple processes, CtoS schedules all access operations to the array at the original CDFG edges – where the operations are defined in the source code.

In some cases, this can be too restrictive – for example, if multiple write operations to the array are in the body of a combinational loop, and you break the loop with multiple clock cycles (see [“Breaking Combinational Loops” on page 8-22](#)). Although you intend to use the multiple clock cycles to implement the write operations, those operations are defined on the same CDFG edge in the original source code. Therefore, CtoS tries to schedule the operations in a single clock cycle.

To alleviate such situations, you can choose to *float all array accesses* to a specified array (located in the set of edges E) to be scheduled at any edge of E, even if the array is accessed by multiple processes. The set of edges E is defined as a set of edges *forward reachable* from a specified starting node and *backward reachable* from an optionally specified ending node. The set of edges E does not include the edges of the functions called from the region. Array dependencies will be respected; CtoS will not reorder a write and a read or write unless it can determine the addresses are always unique.

Requirements for Floating Array Accesses

Before floating array accesses, be aware of the following requirements:

- Allowing the specified array accesses to float must not cause functional problems.

Specifically, be sure that other processes are neither reading addresses written by writes in this loop body, nor writing addresses that are read or written by this loop body.

This can be done with another level of protocol (a signal that indicates which process has access or which part of the array each process is working on).

- Several constraints might be issued for the same array. In this case, the set of edges they define *must not overlap*.
- Set constraints *after* all transforms are complete, because they are not preserved during transforms. For example, a constraint set on a function is not preserved if the function is inlined.
- If the array is accessed by *only one process*, you will get a warning, and there is no effect.

When only one process accesses an array, CtoS can float arrays as long as read-write, write-read, or write-write combinations are not reordered to the same address. (If they were reordered, you would get the wrong value when reading from the memory). CtoS allows array accesses to be reordered if it can be determined that they are accessing different addresses.

For example, writes to `A[i]` and `A[i+1]` always access different locations, so they can be reordered. [In cases that cannot be determined by CtoS, but you (the designer) can determine, you can reorder them with the `break_array_dependency` command ([“break_array_dependency” on page E-27](#))].

CtoS cannot currently handle reordering array accesses between iterations of a pipelined loop, so you must move them manually. When you move a *memread*, you must be able to determine that reordering it relative to the *memwrite* of the next iteration will not change the intent. Alternatively, you can move the *memwrite*, thereby preserving the order (delaying it in the loop).

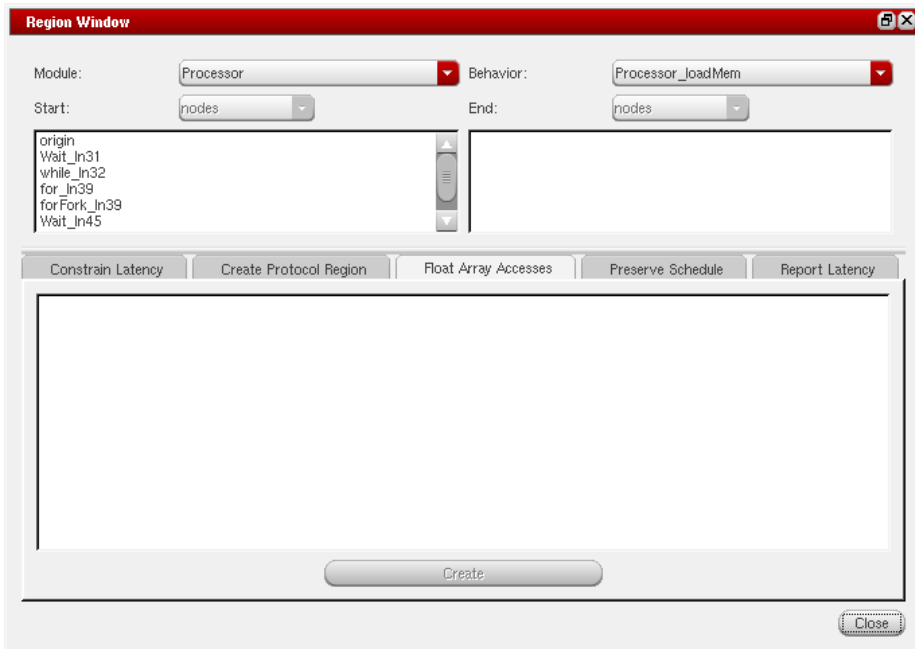
Using the Floating Array Accesses Feature

To use the floating array accesses feature in the CtoS GUI, select **Edit -> Region -> Float Array Accesses**.

The **Region Window** dialog is displayed, with the **Float Array Accesses** tab selected, as shown in [Figure 8-47 on page 8-95](#).

Note You can also use the `float_array_accesses` command ([“float_array_accesses” on page E-70](#)).

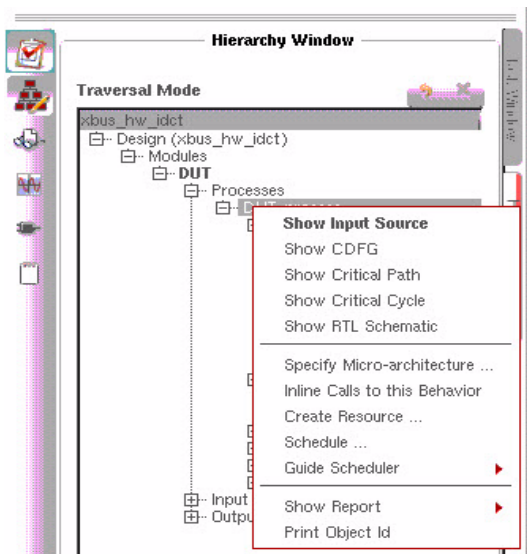
Figure 8-47 Region Window Dialog Showing the Float Array Accesses Tab



8.4 Specifying Micro-Architecture for a Single Behavior

The Micro-Architecture Manager allows you to synthesize one process at a time when specifying actions on loops, arrays and called functions. To synthesize a specific process, right-click the specific process from the **Hierarchy** window and choose the **Specify Micro-architecture...** option.

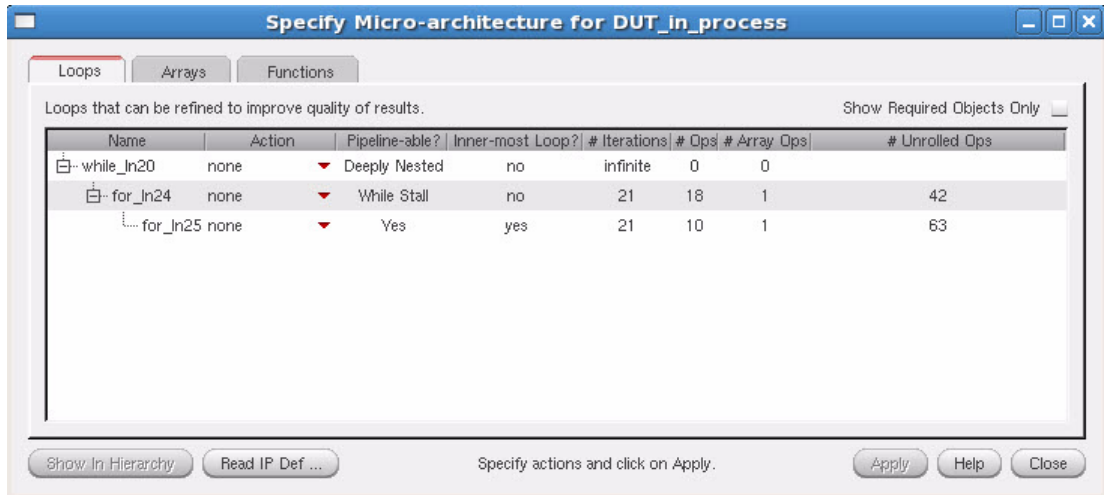
Figure 8-48 The Specify Micro-architecture Option



The **Specify Micro-architecture** dialog appears for your chosen process, as shown in [Figure 8-49](#) and lists:

- Loops that are contained in the specified behavior.
- Arrays that are in the behavior's module and that are accessed by this behavior or any transitively called function.
- Functions that are transitively called in the specified behavior arrays, and called functions of that behavior.

Figure 8-49 Specify Micro-architecture Window



8.5 Semi-Automatic Micro-Architecture

Specify Micro-architecture with Synthesis Modes is a preliminary feature.

CtoS can semi-automatically resolve all micro-architecture choices in one of several prescribed ways called synthesis modes, that correspond to typical design scenarios.

How to choose the synthesis mode and how the micro-architecture is resolved in each of the modes is described in [“Setting Synthesis Modes” on page 8-98](#).

In addition to resolving loops and functions depending on synthesis mode, the arrays are also resolved as described in [“Default Array Actions” on page 8-100](#)

You may override micro-architecture actions selected by the synthesis mode as described in [“Overriding Micro-Architecture Actions” on page 8-100](#).

Once you are satisfied with all the choices, you can apply them as described in [“Applying Micro-Architecture Actions” on page 8-102](#).

The synthesis mode of each behavior is recorded as **synthesis_mode** behavior attribute described in [“synthesis_mode” on page D-44](#).

The micro-architecture action on each loop, function and array is recorded as **uarch_action** attribute on the object.

8.5.1 Setting Synthesis Modes

CtoS provides a set of synthesis modes which can quickly make choices for your micro-architecture. You can specify the synthesis mode on entire design and then override at the module and behavior-level.

See the description of the `set_synthesis_mode` command at: [“set_synthesis_mode” on page E-145](#).

The synthesis mode of each behavior is recorded as `synthesis_mode` behavior attribute described at: [“synthesis_mode” on page D-44](#).

The synthesis modes supported are:

- If your SystemC design is cycle-accurate you may want to use [“Cycle Accurate Mode” on page 8-98](#)
- If your SystemC design consists of a loop that you want to pipeline consider [“Pipelined Mode” on page 8-99](#)
- If your design has no latency or throughput constraints, consider [“Relax Latency Mode” on page 8-98](#)
- If you are only interested in a quick path to (not necessarily optimal) RTL consider [“Simple Mode” on page 8-99](#)

The default value of synthesis mode is manual. In this mode, CtoS makes no automatic micro-architecture choices and you are responsible to resolve micro-architecture, as described elsewhere in this chapter.

8.5.1.1 Cycle Accurate Mode

In this mode, the generated RTL will be cycle-by-cycle equivalent to your SystemC design.

This implies CtoS should not add additional states. When cycle accurate mode is specified on a behavior the following micro-architecture actions will be specified:

- all combinational loops to be unrolled
- functions accessing an array that is not flatten will be inlined
- functions will be inlined if required to make synthesizable
- arrays will be set based on default array actions as described in [“Default Array Actions” on page 8-100](#)

8.5.1.2 Relax Latency Mode

In this mode, CtoS will freely create new states anywhere except in the defined protocol regions. Typically, adding states increases the latency and decreases the throughput.

When relax latency mode is specified on a behavior, the following micro-architecture actions will be specified:

- all loops in reset path (before the first wait) will be unrolled
- all remaining combinational loops will be broken by inserting a state
- functions accessing an array that is not flatten will be inlined
- functions will be inlined if required to make synthesizable
- arrays will be set based on default array actions as described in [“Default Array Actions” on page 8-100](#)

In addition, the **relax_latency** attribute will be specified to direct the scheduler that additional states may be added.

However, relax latency mode does not provide optimality of the implementation and should be used for quick prototyping only. The action of adding states in relax latency scheduling is considered to be more costly than adding resources. The suggested methodology for using relax latency is :

1. Run design with relax latency scheduling.
2. Observe the places where the scheduler added states.
3. Manually copy state additions into the tcl script.
4. Rerun design pursuing QoR with a normal mode of scheduling.

8.5.1.3 Simple Mode

This mode is the simplest path to RTL but does not guarantee optimality. The micro-architecture choices are the same as Relax Latency Mode, except that the **relax_latency** attribute is not set.

8.5.1.4 Pipelined Mode

The SystemC behavior consists of a single top-level loop to be pipelined. The loop cannot be nested in another loop, nor can it be contained in a function called from the thread function. The loop must be infinite and no loop preceding it can be infinite.

Here is an example that meets this criteria:

```
run() {  
    wait();  
    for(...) { ... }  
    ...  
    while(1) {...}  
}
```

When pipelined mode is specified on a behavior the following micro-architecture actions will be specified:

- all loops in reset path (before the first wait) will be unrolled
- all loops nested in the loop to be pipelined will be unrolled
- all combinational loops not contained in the loop to be pipelined will be unrolled
- the top-level loop will be pipelined with `ii = 1`, `min_li` = minimum latency of loop, `max_li` = 100
- sequential functions will be inlined
- functions will be inlined if required to make synthesizable
- arrays will be set based on default array actions as described in [“Default Array Actions” on page 8-100](#)

8.5.1.5 Manual Mode

Use this mode if the design does not fit one of the pre-defined design styles. You must manually specify micro-architecture actions on individual objects before calling **apply_uarch_action**. This is the default synthesis mode for all behaviors.

8.5.1.6 Default Array Actions

To complete specification of micro-architecture, all arrays must be resolved. Setting synthesis mode will also set micro-architecture action on all containing arrays following these rules:

- Small arrays that are not external and not accessed by multiple processes will be flattened

An array is small if total number of bits is less than design attribute **max_auto_flatten_array_size** (default is 256). See description of attribute at: [“max_auto_flatten_array_size” on page D-9](#).

- For designs with ASIC implementation target, arrays will be allocated to prototype memories
- For designs with FPGA implementation target, arrays will be allocated to builtin-RAMs.

8.5.2 Overriding Micro-Architecture Actions

Setting synthesis mode will specify micro-architecture actions on each loop, function and array object based on design style. You can override the micro-architecture action on a particular object.

See the description of the **set_uarch_action** command at: [“set_uarch_action” on page E-146](#).

Note Overriding micro-architecture may cause the behavior to not be synthesizable.

Loop Actions

You can specify the following actions for loops:

- unroll: call unroll_loop
- break: call break_combinational_loop
- pipeline: call pipeline_loop with default values (ii=1, min_li = minimum latency of loop, max_li = 100)
- no_action: no action will be applied to this loop (default)

The micro-architecture action on a loop is recorded as uarch_action attribute as described at [“uarch_action” on page D-55](#).

Function or Custom Op Actions

You can specify the following actions for functions or custom ops:

- inline: call inline all calls to this function or inline the function call specified by custom op
- no action will be applied to this function or custom op (default)

The micro-architecture action on a custom op is recorded as uarch_action attribute as described at [“uarch_action” on page D-38](#).

The micro-architecture action on a function is recorded as uarch_action attribute as described at [“uarch_action” on page D-45](#).

Array Actions

You can specify the following actions for arrays:

- flatten: call flatten_array
- builtin: call allocate_builtin_ram
- prototype: call allocate_prototype_memory
- vendor: issues message that user must explicitly allocate this memory. to Vendor RAM
- rom: issues message that user must explicitly allocate this memory to ROM
- no action will be applied to this array (default)

The micro-architecture action on an array is recorded as uarch_action attribute as described at [“uarch_action” on page D-38](#).

8.5.3 Applying Micro-Architecture Actions

After setting micro-architecture actions on loops, functions, and arrays you can apply all these micro-architecture actions with single call to the **apply_uarch_action** command. You can apply actions at the design, module, or individual behavior level.

See the description of the **apply_uarch_actoin** command at: [“apply_uarch_action” on page E-25](#).

The micro-architecture action on each loop, function, and array is recorded as **uarch_action** attribute.

The following is the order of applying micro-architecture actions:

- Flatten Arrays
- Unroll Loops
- Break Loop
- Inline Functions
- Pipeline Loop
- Allocate Arrays

Note If micro-architecture action on array is vendor or rom, then you must allocate these arrays to make it synthesizable.

8.6 Using Synthesis Directive

This section describes about how to transform a design based on the synthesis directives embedded in the SystemC source code of the design, and how to use the synthesis directive pragmas in your SystemC code. This chapter covers the following:

- “What Is Synthesis Directives” on page 8-103
- “Representation of a Synthesis Directive in the CtoS Database” on page 8-104
- “Elaboration” on page 8-104
- “Applying and Removing a Directive” on page 8-105
- “Use Models” on page 8-105
- “Commands Supported with Synthesis Directives” on page 8-106

8.6.1 What Is Synthesis Directives

A synthesis directive is an executable implementation choice, embedded in the SystemC source as a pragma. It is a CtoS tcl command that follows '#pragma ctos', where an argument is inferred from the position of the pragma.

Syntax:

```
#pragma ctos token1 [ token2 ] [ token3 ] ..  
c++-[statement|declaration]
```

where **token1** is the name of a CtoS Tcl command, and the subsequent tokens (**token2**, **token 3...**) are the options and arguments of the command. The CtoS Tcl command must take at least one argument that is the string-id of a database object. This object argument is implied by the position of the pragma and so it is not specified using the tokens. The pragma applies to the statement or declaration that it precedes.

Example

```
#pragma ctos unroll_loop  
for (int i = 0; i < 8; i++) {  
    m_arr[i] = 0;  
}
```

The directive that will be inferred from this pragma is a follows:

- text: unroll_loop
- inferred object argument: the loop join node inferred from the for loop (since the pragma immediately precedes the for-statement)

For more information about how to specify a pragma for each source constructs, see [“Specifying Synthesis Directive” on page 14-79](#).

8.6.2 Representation of a Synthesis Directive in the CtoS Database

A directive is represented in the CtoS database as a directive object.

- It appears in the directives scope of behavior/array/node/edge/op objects.
- The name of a directive is typically the name of the command embedded in the directive.
- Directive objects can be enumerated using the **find -directive** and **ls** commands

For more information about the directive object attribute, see D.26 Directive Object Attributes.

8.6.3 Elaboration

During build, CtoS creates directive objects from the pragmas. As a Behavior/Array/Node/Edge/Op is inferred from a source construct and the source construct has a pragma associated with it, CtoS decides based on the CtoS Tcl command whether a directive is appropriate.

If the **-verbose** option of the **build** command is specified, CtoS will print an info message containing the source location and ID of each directive created during build. It also provides information about the pragmas that are ignored during elaboration because of any of the following reasons:

- The CtoS Tcl command mentioned is not a correct CtoS Tcl command
- The CtoS Tcl command mentioned is not supported in directives.
- The arguments mentioned is not a settable attribute (if the CtoS Tcl command = `set_attr`)
- The pragma does not make sense at this position in the source (example: an `unroll_loop` pragma on a declaration.)

As a design is transformed, CtoS tries to preserve the directives in the following manner:

- Directives are copied when function calls are inlined, loops are unrolled, and functions are uniquified so that they are called from only 1 process.
- Directives are removed when the object that they are referring to ceases to exist.

8.6.4 Applying and Removing a Directive

The implementation choices embedded in the directives do not become decisions until the user applies the directives using the **apply_directive** command. This does not mean that a directive that is never applied does not affect QoR. Directives do prevent certain optimizations (such as merging of simple nodes). Therefore, the **remove_directive** command allows the user to remove one or more directives from the database.

Notes

- The arguments of the command embedded in the directive are not evaluated until the directive is applied using the **apply_directive** command.
- Syntax checking for pragmas is very limited. Referring to the syntax, only the CtoS Tcl command (token1) is checked during elaboration. Any spelling errors in subsequent tokens will be identified only when the directive is applied.

For more information about these commands, see [“apply_directive” on page E-22](#) and [“remove_directive” on page E-101](#).

8.6.5 Use Models

This section describes the two use models specifying synthesis directives:

Use Model 1

1. Build the design.
2. Examine the directives specified.
3. Remove any unwanted directives that are not required.
4. Execute **apply_directive -step micro_architecture**.
5. Perform manual micro-architecture decisions.
6. Optimize the design.
7. Execute **apply_directive -step allocate_ip**.
8. Perform manual micro-architecture decisions.

Use Model 2

1. Build the design.
2. Do while failing directives.
 1. Execute **remove_directive** on failed directive or performs manual micro-architecture decisions.
 2. Execute **apply_directive -step micro_architecture**.
3. Perform manual micro-architecture decisions.
4. Optimize the design.
5. Do while failing directives.
 1. Execute **apply_directive -step allocate_ip**.
 2. Execute **remove_directive** on any failing directives.
3. Perform manual micro-architecture decisions.

8.6.6 Commands Supported with Synthesis Directives

The following [Table 8-3](#) lists the commands that can be used with synthesis directive pragma.

Table 8-3 Commands Supported with Synthesis Directives.

Command	Description	Inferred Object
inline	inline behavior	behavior
flatten_array		array
allocate_builtin_ram		array
allocate_memory		array
allocate_prototype_memory		array
unroll_loop		loop
pipeline_loop		loop
break_combinational_loop		loop
create_protocol_region		node / node pair
float_io_accesses		node / node pair

Table 8-3 Commands Supported with Synthesis Directives.

Command	Description	Inferred Object
float_array_accesses		node / node pair
constrain_latency		node / node pair
set_attr enable_addsubs		behavior
set_attr enable_const_resources		behavior
set_attr optimize_enable_arithmetic_trees		behavior
set_attr relax_latency		behavior
set_attr scheduling_effort		behavior
set_attr timing_criticality		behavior

9 Allocating Memory and RTL IP

For examples of the Allocating Memory and RTL IP step of the CtoS flow, look in the following directory:

```
install_directory/share/ctos/examples/features
```

Note Before running any CtoS examples, first review [“Setup for Examples” on page F-2](#)

In the Allocating Memory and RTL IP step of the CtoS flow, you can perform the following tasks:

For arrays (memories), see:

- [“Allocating Memory” on page 9-2](#)

For functions, see:

- [“Importing RTL IP into SystemC Designs” on page 9-41](#)

Note *For more on IP definition files required for memories and RTL IP, see the [“IP Definition Files for Memories and RTL IP” on page H-1 appendix](#).*

9.1 Allocating Memory

CtoS provides several options for mapping an array in SystemC to a physical memory implementation; each option is optimal for a different array use case or access pattern, as shown in [Table 9-1 on page 9-2](#).

Table 9-1 Comparison of Commands for Implementing Arrays (Memories)

	flatten array	allocate built-in RAM	allocate prototype memory	allocate vendor RAM
advantages	• best for small arrays, especially if read and write addresses are constant • allows parallel accesses (many read ports)	• best for FPGAs (maps to block RAM) • supports multiple processes • supports many read and write interfaces	• has fastest runtime • intended for design exploration or when vendor model unavailable (must be replaced before final implementation)	• good for final implementation of medium or large arrays
disadvantages	• restricted to a single process • can cause routing congestion due to many muxes and/or lookup resources • does not allow manual overriding of array dependencies • may generate more registers than built-in RAM	• has very long synthesis runtimes for large RAMs	• does not support asynchronous reads • does not support multiple clocks • prevents RTL creation	• requires memory models from ASIC or memory vendors • inefficient for small arrays
synch ports	✓	✓	✓	✓
asynch ports	✓	✓	✗	some vendors
targeted for synthesis		synthesis of large built-in RAMs is very slow	exporting memories is required for synthesis	
FPGA	✓	✓ (maps to block RAM)	special case (see “ CtoS Tutorial for FPGA designs ” on page B-1)	special case (use liberty file, not Verilog file)
# of processes accessing array	1	1 or more	1 or n*	1 or n*
*Vendor RAMs have one or two interfaces, and multiple processes can share an interface with the multiplex option.				
area cost	• high for large arrays • may dramatically increase number of registers	high for large arrays	used only for testing micro-architecture	efficient for large arrays

These options are further described in the following sections:

- [“Allocating Built-In RAM” on page 9-3](#)
- [“Allocating Prototype Memory” on page 9-7](#)
- [“Allocating Vendor RAM” on page 9-10](#)
- [“Allocating Vendor ROM” on page 9-14](#)
- [“Allocating Processes to Memory Interfaces” on page 9-17](#)
- [“Specifying Interfaces for Built-In RAM” on page 9-18](#)
- [“Specifying Interfaces for Prototype Memory” on page 9-19](#)
- [“Specifying Memory Clock for Built-In RAM, Prototype Memory, and Vendor RAM” on page 9-19](#)
- [“Exporting Memories” on page 9-20](#)
- [“Registered Memories” on page 9-21](#)
- [“Stallable Memories” on page 9-22](#)
- [“External Arrays” on page 9-22](#)
- **flatten_array** is described in the previous chapter: see [“Flattening Arrays” on page 8-61](#).

9.1.1 Allocating Built-In RAM

Arrays that are suitably small (nearly always 256 words or fewer) can be implemented as built-in RAMs.

This is a good choice when the number of array words is small, but multi-process access is desired (note, however, that this feature supports both single- and multi-process access).

The RAM is generated for synthesis using behavioral Verilog, accessing an array of necessary dimensions.

Storage of elements in the array *after synthesis* is implemented using flip-flops, and control and datapath logic are primitive gates from the technology library (for example, muxes and bitwise logicals).

Flip-flops carry a large area penalty, compared to SRAM technology cells, so a flip-flop-based RAM will always be less efficient in area and power consumption.

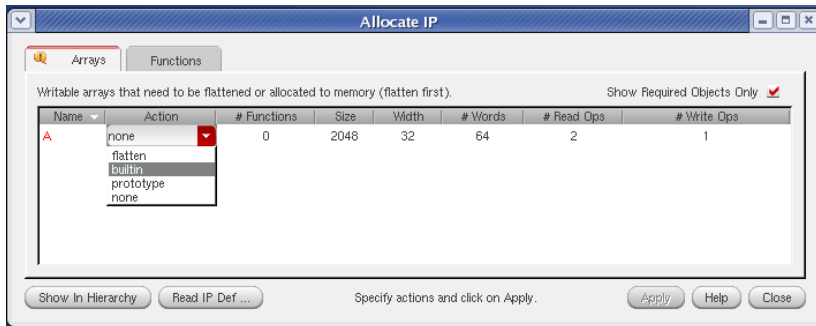
However, built-in RAMs are a viable option when a suitable SRAM technology cell is unavailable, and they have the additional benefit of being quickly re-generated if a SystemC design change implies different array parameters.

The number of write interfaces is determined by the number of processes writing the array. For details about specifying multiple-write and read-write interfaces, see [“Specifying Interfaces for Built-In RAM” on page 9-18](#).

For single-interface memories in FPGA designs, the RAM is implemented through a RAM resource on the FPGA part by the FPGA synthesis tool.

To allocate a Built in RAM with default settings you can use the Action combo box in the Allocate IP Dialog as shown in [Figure 9-1 on page 9-4](#)

Figure 9-1 Allocate IP Dialog - Allocating Built-in RAM with Default Settings



To allocate built-in RAM without all the defaults, use the **Allocate Built In Ram** dialog, as shown in [Figure 9-2 on page 9-5](#).

Figure 9-2 Allocate Built In Ram Dialog

Allocate Built In Ram

Data Width = 32; Number of Words = 64

Memory Interfaces

Specify interfaces for memory that is accessed by 1 process
read-only(0), write-only(0), read+write(1)

Index	Access Type
0	read
1	write

Add Delete

☒ Automatic Interface Allocation

☐ **Synchronous Read**

Memory Read Latency: 1

☐ **Enable Partial Write**

Minimum Write Width: 1

Memory Clock: clk

OK Cancel

You can specify the interfaces in the **Memory Interfaces** section. This is an ordered list of interfaces with interface index on the left. By default a read interface is specified for each behavior reading this array and a write interface for each behavior writing this array.

- You can change the access type of an interface by double clicking the entry and selecting new access type from the combo box. You can also use keyboard shortcut of ‘r’ for read, ‘w’ for write and ‘b’ for both read and write. Read-Write access type is only supported when the **Synchronous Read** box is checked.
- You can add interface by clicking the **Add** button. The new interface is added at the bottom of the list.
- You can delete interfaces by selecting the rows and clicking the **Delete** button.
- If the **Automatic Interface Allocation** check box is selected, interfaces are assigned to processes automatically. Deselect the check box for manual assignment. For more information on manual assignment, see [“Allocating Processes to Memory Interfaces”](#) on page 9-17.

- If the number of interfaces is insufficient, the **OK** button is disabled and an asterisk is displayed next to the appropriate interface type, as shown in [Figure 9-10 on page 9-18](#). [See the first bullet in [Notes on page 9-18](#) for the valid number of interfaces for each type.]
- You can specify a memory clock from the list of compatible clocks (nets) in the **Memory Clock** combo box. For more information, see “[Specifying Memory Clock for Built-In RAM, Prototype Memory, and Vendor RAM](#)” on page 9-19.

CtoS provides an example of the built-in memory in:

```
install_directory/share/ctos/examples/features/arrays/builtin_ram
```

For information on the scheduling rules and restrictions of memory reads, see “[Scheduling Restrictions for Multi-Latency Operations](#)” on page -8.

Notes

- For an example of how to allocate built-in RAM, see “[Binding Arrays \(Option 3\)](#)” on page C-13.
- You can also use the `allocate_builtin_ram` command (“[allocate_builtin_ram](#)” on page E-7).

9.1.1.1 Specifying Read Latency for Built-In RAM

CtoS lets you specify read latency of 1 to 3 (with a default of 1), when you are allocating built-in rams as long as they are synchronous memories. When using built-in RAMs, CtoS will generate the necessary registered memory structure automatically.

Directions for specifying read latency are as follows:

- The **Allocate Built In Ram** dialog, as shown in [Figure 9-2 on page 9-5](#), has a **Read Latency** entry box, which is enabled when you select **Synchronous Read**.
- When the read latency is **2**, the register cycles before the memory is **0** and the register cycles after the memory is **1**; the memory is always one cycle, so the total latency is **2**.
- When the read latency is **3**, the register cycles before the memory is **1** and the register cycles after the memory is **1**; the memory is always one cycle, so the total latency is **3**.
- Setup and launch delays come from RC automatically.
- You can also use the `allocate_builtin_ram` command (“[allocate_builtin_ram](#)” on page E-7) with the option, `-read_latency`.
- The read latency feature is only for *synchronous* memories, so you cannot define `read_latency` for a memory with an interface type of `async_read`.

9.1.2 Allocating Prototype Memory

Allocating Prototype Memory is a preliminary feature.

CtoS supports a notion of *prototype memories*. Prototype memories are useful during the micro-architectural and design exploration stages of a design, when the final implementation of a memory may not yet be complete, or the correct memory technology cell is not available.

Prototype memories allow the CtoS scheduling step to complete by using a prototype memory as a placeholder for the actual memory cell.

CtoS can create a prototype memory for an array, similar to using built-in memory, but a prototype memory does not require logic synthesis for area and timing estimates during resource and timing analysis.

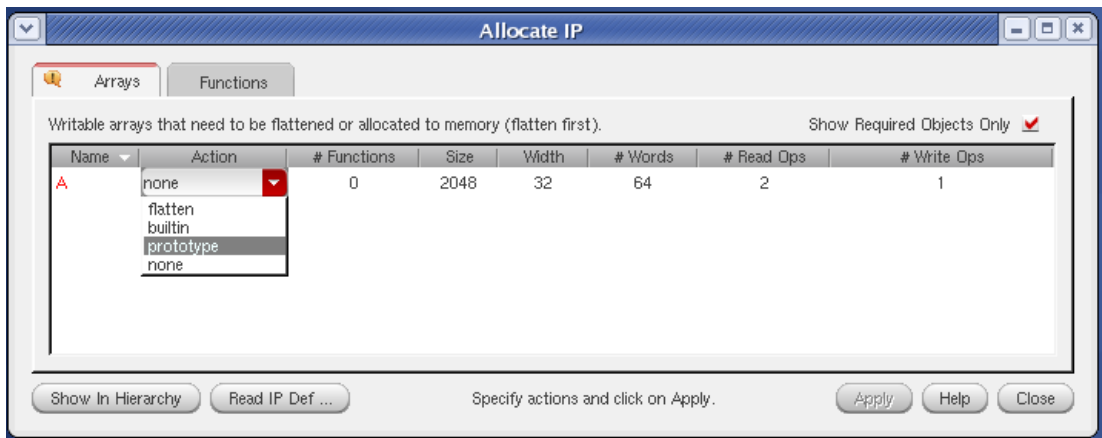
Instead, setup and launch delays are calculated from the clock period and the design attributes **prototype_memory_setup_delay** (“[prototype_memory_setup_delay](#)” on page D-21) and **prototype_memory_launch_delay** (“[prototype_memory_launch_delay](#)” on page D-21).

Area is not estimated and is set to zero.

Prototype memories give you an easy way to allocate memories to arrays without requiring large run times for large arrays; however, they *cannot* be used in the final implementation of a design

Prototype memories can be allocated with default settings from the Action combo box in the **Allocate IP** Dialog, as shown in [Figure 9-3 on page 9-7](#).

Figure 9-3 Allocate IP Dialog - Allocating Prototype Memory with Default Settings



Use the **Allocate Prototype Memory** dialog, as shown in [Figure 9-4 on page 9-8](#) to allocate prototype memories with custom settings.

Figure 9-4 Allocate Prototype Memory Dialog

Allocate Prototype Memory

Data Width = 32; Number of Words = 64

Memory Interfaces

Specify interfaces for memory that is accessed by 1 process
read-only(0), write-only(0), read+write(1)

Index	Access Type
0	read-write

Add Delete

☐ Share Interface
☒ Automatic Interface Allocation

Latency

Memory Read Latency: 1
Register Memory Input: ☐
Register Memory Output: ☐
Total Latency = 1

☐ **Enable Partial Write**

Minimum Write Width: 1

Memory Clock: clk

Active Level

Chip Enable: ☒ high ☐ low
Write Enable: ☒ high ☐ low

Export Memory: ☐

OK Cancel

You can specify the interfaces in the **Memory Interfaces** section. This is an ordered list of interfaces with interface index on the left. By default a read-write interface is specified for each behavior accessing this array.

- You can change the access type of an interface by double clicking the entry and selecting new access type from the combo box. You can also use keyboard shortcut of 'r' for read, 'w' for write and 'b' for both read and write.
- You can add interface by clicking the **Add** button. The new interface is added at the bottom of the list.
- You can delete interfaces by selecting the rows and clicking the **Delete** button.

- One memory interface can be selected to be shared. To share a memory interface, select the **Share Interface** check box. Two additional columns will be added to the Memory Interfaces table. You specify which interface is to be shared by selecting the radio button in the **Share?** column and then specify the share count which must be greater than 1 (default is 2) in the **Share Count** column.
- If the **Automatic Interface Allocation** check box is selected, interfaces are assigned to processes automatically. Deselect the check box for manual assignment. For more information on manual assignment, see [“Allocating Processes to Memory Interfaces” on page 9-17](#).

You specify the latency of the prototype memory in the **Latency** section.

- The read latency of prototype memories can be **1** to **3**. CtoS will generate the necessary registered memory structure automatically.
- In addition, inputs and outputs of prototype memories can be registered through the **Register Memory Input** and **Register Memory Output** check boxes. However, if the read latency is specified as **3**, then the **Register Memory Input** and **Register Memory Output** check boxes are automatically disabled.
- The total latency of the memory is shown as the **Total Latency** on the dialog.
- Setup and launch delays are defined using the **prototype_memory_launch_delay** (“[prototype_memory_launch_delay](#)” on page D-10) and **prototype_memory_setup_delay** (“[prototype_memory_setup_delay](#)” on page D-10) design attributes.

You specify the Prototype memory is to be exported by selecting the **Export Memory** checkbox. For more information on exported memories see [“Exporting Memories” on page 9-20](#).

You can specify a memory clock from the list of compatible clocks (nets) in the **Memory Clock** combo box. For more information, see [“Specifying Memory Clock for Built-In RAM, Prototype Memory, and Vendor RAM” on page 9-19](#).

You can enable partial write by selecting **Enable Write Width** and then specify the **Minimum Write Width** with the spinbox. This matches the functionality of Partial Write of Vendor RAM which is documented at [“Specifying Minimum Write Width in Vendor RAMs” on page H-18](#).

You can specify the active level of the chip enable and write enable signals by selecting the **Chip Enable** and **Write Enable** checkboxes. By default the active level is *high*.

You can also see the **allocate_prototype_memory** command (“[allocate_prototype_memory](#)” on page E-14).

CtoS provides an example of the prototype memory in:

```
install_directory/share/ctos/examples/features/arrays/prototype_ram
```

For information on the scheduling rules and restrictions of memory reads, see [“Scheduling Restrictions for Multi-Latency Operations” on page -8](#).

9.1.2.1 Limitations when Using Prototype Memories

Prototype memories have the following limitations:

- Prototype memories do not support multiple clocks.
- Prototype memories model only synchronous read.
- Prototype memories do not currently support the **reset_all_registers** design attribute ([“reset_registers” on page D-23](#)) which is set in the **Create New Design Wizard** ([“Create New Design Wizard” on page 6-10](#)).
- Depending on whether the memories are exported, the **write_rtl** and **write_rc_script** commands may produce an error or warning when used with prototype memories, as follows:
 - If the memories are *not* exported, the **write_rc_script** command ([“write_rc_run_script” on page E-158](#)) issues an error, and the **write_rtl** command ([“write_rtl” on page E-161](#)) issues a warning.
 - If the memories *are* exported, both commands work without error or warning.

9.1.3 Allocating Vendor RAM

During the exploration of design micro-architecture, it is *not* optimal to map arrays to Vendor RAMs because they may be merged, flattened, or optimized [see [“Resolving Arrays \(Memories\)” on page 8-61](#)], which can alter their dimensions and access properties.

However, when arrays are in their *final* form, you may then allocate them as *Vendor RAMs*.

After allocation, the implementation is understood to be final, and scheduling may begin.

During allocation, the number of interfaces to the RAM are determined, which affects the ability to schedule the design, as well as the *quality of results* (QoR).

To allocate a Vendor RAM to a specified array, you would follow these steps:

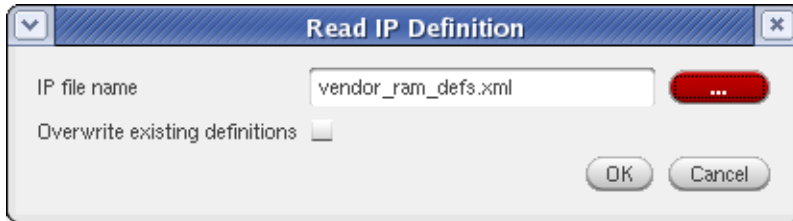
1. Create the RAMDef record for the RAM: see [“Vendor RAMs” on page H-6](#).
2. After you have built the design, read in the IP definition file.

In the CtoS GUI, in the **Allocate IP** dialog, you would click the **Read IP Def** button to display the **Read IP Definition** dialog and use the ... button to browse to the Vendor RAM definition, as shown in [Figure 9-5 on page 9-11](#).

You could also use the **read_ip_defs** command (“[read_ip_defs](#)” on page E-97).

You may do this more than once to read separate **.xml** files.

Figure 9-5 Read IP Definition - Allocating Vendor RAM

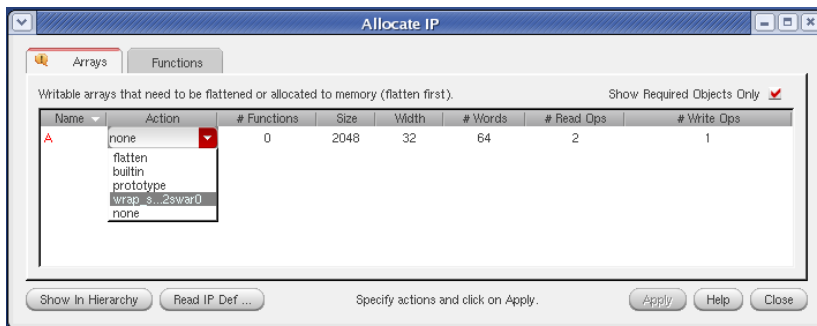


3. Allocate the RAM.

For allocating Vendor RAM with default setting, you would use the Action column in the **Array** tab of the **Allocate IP dialog** as shown in [Figure 9-6](#) on page 9-11.

You could also use the **allocate_** command (“[allocate_memory](#)” on page E-9).

Figure 9-6 Allocate IP Dialog - Allocating Vendor RAM with Default Settings



Use the **Allocate Memory for Array** dialog, as shown in [Figure 9-7](#) on page 9-12 to allocate vendor memories with custom settings.

Figure 9-7 Allocating Vendor RAM

Allocate Memory for Array: A

Share Interfaces: ☐

Memory Definition: wrap_sample_64x32swar0

Index	Access Type
0	sync_write
1	async_read

Automatic Interface Allocation: ☒

Latency

Memory Read Latency: 0

Register Memory Input: ☐

Register Memory Output: ☐

Total Latency = 0

Memory Clock: clk

Export Memory: ☐

Read ... OK Cancel

- The **Memory Definition** combo box lets you to choose the Memory Definition to be allocated. This list contains all Memory Definitions that are compatible to the specified array.
 - The Memory Interfaces table displays the index and access type of interfaces in the memory definition (not editable).
 - The read latency of the memory is specified in the Latency box (not editable) as described in [“Specifying Read Latency for Vendor RAMs”](#) on page 9-13.
- One memory interface can be specified to be shared. To share a memory interface, select the **Share Interfaces** check box.
 - The list of compatible memory definitions is re-evaluated.
 - Two additional columns will be added to the Memory Interfaces table. You specify which interface is to be shared by selecting the radio button in the **Share?** column and then specify the share count which must be greater than 1 (default is 2) in the **Share Count** column.
- Inputs and outputs of memories can be registered through the **Register Memory Input** and **Register Memory Output** check boxes. However, if the read latency is specified as 3, then the **Register Memory Input** and **Register Memory Output** check boxes are automatically disabled.

The total latency of the memory is shown in the **Total Latency** option.

- You can specify a memory clock from the list of compatible clocks (nets) in the **Memory Clock** combo box. For more information, see [“Specifying Memory Clock for Built-In RAM, Prototype Memory, and Vendor RAM” on page 9-19](#).
- You specify the Prototype memory is to be exported by selecting the **Export Memory** checkbox. For more information on exported memories see [“Exporting Memories” on page 9-20](#).
- If the **Automatic Interface Allocation** check box is selected, interfaces are assigned to processes automatically. Deselect the check box for manual assignment. For more information on manual assignment, see [“Allocating Processes to Memory Interfaces” on page 9-17](#). When interfaces are shared, automatic interface allocation is not supported and the checkbox is disabled.
- You can also use the **allocate_memory** command ([“allocate_memory” on page E-9](#)).

The following sections describe some considerations when allocating Vendor RAM:

- [“Specifying Read Latency for Vendor RAMs” on page 9-13](#)
- [“Limitation when Using Vendor Memories” on page 9-14](#)
- [“Using Verilog +nospecify Option for Functional Simulation” on page 9-14](#)

9.1.3.1 Specifying Read Latency for Vendor RAMs

CtoS lets you specify read latency in the XML file, from 1 to 3 (with a default of 1), when you are allocating vendor memories, as long as they are synchronous memories. The latency must match with liberty file and must be the same for all memory interfaces.

Directions for specifying read latency is as follows:

- The body of a vendor memory is not *seen* by CtoS and is considered a black box for the purposes of scheduling and synthesis. You may implement a vendor memory using any technology cell or any synthesizable modeling technique.

There is more than one valid implementation for a given memory with a given read latency, and it is up to you to choose an implementation that is optimal for your specific application.

- Read latency is defined in the XML file, using the XML element, **read_latency**, as shown in the following example (see [“IP Definition \(XML\) File Description” on page H-2s](#)):

```
<interface_types>
  <elem>sync_write</elem>
  <elem>sync_read</elem>
</interface_types>
<read_latency>2</read_latency>
```

- Setup and launch delays are defined conventionally in a **.lib** file.

- The **memory_def** object has a read-only attribute, **read_latency** (“[read_latency](#)” on page D-32), which applies to Vendor RAMs and ROMs; however, it is always **1** for Vendor ROMs.

CtoS provides an example of Vendor RAM in:

```
install_directory/share/ctos/examples/features/arrays/vendor_ram
```

For information on the scheduling rules and restrictions of memory reads, see “[Scheduling Restrictions for Multi-Latency Operations](#)” on page -8.

9.1.3.2 Limitation when Using Vendor Memories

Although the RAMDef/ROMDef specifies the address and data widths of the vendor memory, CtoS currently does not parse the user-supplied Verilog file to verify the widths of the address and data terminals of the RAM/ROM. Due to this limitation, it is your responsibility to ensure that the matching widths are specified in the IP definition file.

If the user-supplied Verilog file contains mismatched bit widths for the address or data terminals, CtoS will be unable to issue a warning or an error. However, during compilation of the models, **ncelab** will issue the following warning:

```
ncelab: *W,CUVMPW (./userMod_rtl.v,184|47): port sizes differ in  
      port connection.
```

If this warning is overlooked, then simulation may fail with mismatches.

9.1.3.3 Using Verilog +nospecify Option for Functional Simulation

When simulating CtoS-generated RTL, it is often desirable to run the simulation without checking for vendor memory timing checks, and to avoid using Verilog delays.

Since almost all vendor Verilog models use Verilog **specify** blocks for timing accuracy, it is recommended that you use the Verilog **+nospecify** option with your Verilog simulator during functional simulation. This will improve simulation performance and eliminate cases where memory delays cause functional simulation problems.

9.1.4 Allocating Vendor ROM

CtoS lets you create a Vendor ROM instance to bind to a specified array, as described below:

Note See the following section, “[Limitations and Notes about Using Vendor ROMs](#)” on page 9-16, for a few restrictions on Vendor ROM arrays.

1. Before you **build** the design, include the following pragma on arrays containing microcode that may be updated through an ECO, to be able to change the ROM programming without re-synthesizing the rest of the RTL:

```
#pragma ctos dont_touch
int my_constants[] = { 1,2,3 };
```

This sets the CtoS **dont_touch** array attribute to true on arrays as they are created by the CtoS elaborator. During optimization, these attributes are preserved when the arrays are uniquified, and such arrays will not be flattened or trimmed.

For example, if you did not use the pragma in this example, CtoS would change the width of this memory to two bits, treat the top 30 bits as zeroes, and optimize the design based on the top 30 bits being zero. (The design would then need to be completely re-synthesized if one of the values were changed to -1.)

Important If calls to **dont_touch** functions (“[dont_touch](#)” on page D-36) evaluate to either constant zero or one, those functions *will* be optimized out.

2. Create the ROM program, using the **create_rom_program** command (“[create_rom_program](#)” on page E-49).
3. Create the ROMDef file; see “[Vendor ROMs](#)” on page H-25.
4. After you have built the design, read in the XML RTL IP .xml file.

In the CtoS GUI, in the **Allocate IP** dialog, you would click the **Read IP Def** button to display the **Read IP Definition** dialog and use the ... button to browse to the Vendor RAM definition, as shown in [Figure 9-8](#) on page 9-15.

You could also use the **read_ip_defs** command (“[read_ip_defs](#)” on page E-97).

You may do this more than once to read separate .xml files.

Figure 9-8 Read IP Definition Dialog - Allocating Vendor ROM

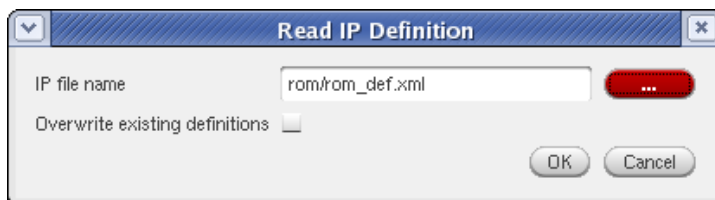
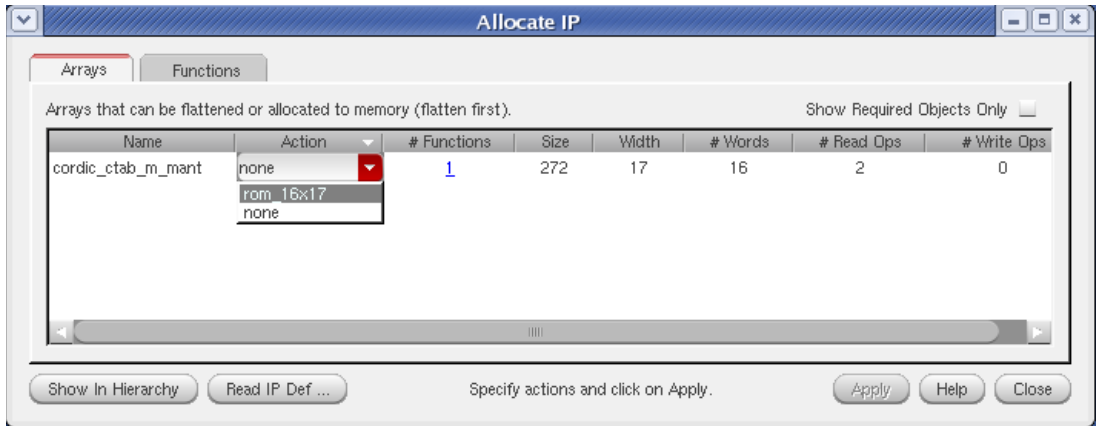


Figure 9-9 Allocate IP Dialog - Allocating Vendor ROM



5. Allocate the ROM.

In the CtoS GUI, you would use the **Allocate IP** dialog, as shown in [Figure 9-9 on page 9-16](#).

You could also use the **allocate_memory** command ("[allocate_memory](#)" on page E-9).

After a specific IP definition file has been allocated for a ROM, it may not be allocated again. ROM definitions are instance-specific.

In digital design flows after netlisting, the name of the ROM is nearly always assumed to be unique to the programming it contains.

This means that two ROM arrays may not allocate to a ROM using the same name, since the programming content is different between them.

Consequently, for each array in the design that will be allocated to a Vendor ROM there must exist a unique IP definition file for each.

9.1.4.1 Limitations and Notes about Using Vendor ROMs

Arrays that are read by combinational processes or functions cannot be allocated to Vendor ROM. Such arrays must be implemented by lookup resources. No specific resource allocation is required to accomplish this.

Important See also the previous sections, "[Limitation when Using Vendor Memories](#)" on page 9-14 and "[Using Verilog +nospecify Option for Functional Simulation](#)" on page 9-14, which apply to Vendor ROMs, as well as Vendor RAMs.

9.1.5 Allocating Processes to Memory Interfaces

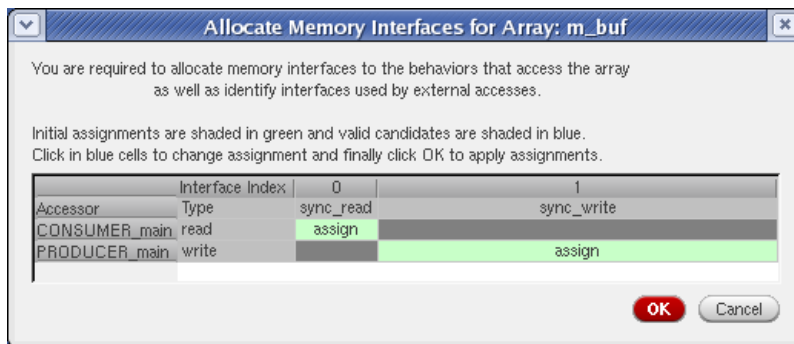
Memory interfaces of Vendor RAMs are defined in the IP Definition file description and the interfaces of prototype memories are specified through the GUI or the **allocate_prototype_memory** command. After a memory is allocated to an array, the memory interfaces must be allocated to the processes that access the array. Memory interfaces are allocated automatically by the GUI or the memory allocation commands by default in most cases. They can also be assigned to processes manually. Most Vendor RAMs have a limited number of interfaces, and designs may have many processes accessing arrays. One of the memory interfaces can be shared by multiple processes, and all other interfaces can be accessed by only one process. Interfaces must be allocated to processes manually if one interface is shared.

Prototype memories can be used as a rough model of Vendor RAMs when the RAM models are not available. The prototype memory can be specified with the same interfaces as the Vendor RAM, and also one interface can be shared by multiple processes.

9.1.5.1 Using the CtoS GUI to Allocate Processes to Memory Interfaces

You can assign memory interfaces after allocating the memory. This can be done by unchecking **Automatic Interface Allocation** in the Allocate Memory Dialog. After allocating the memory, the GUI automatically displays the **Assign Memory Interfaces Dialog**, as shown in [Figure 9-10 on page 9-17](#) to let you choose the memory interfaces for each behavior.

Figure 9-10 Allocate Memory Interfaces Dialog



This dialog has table where rows represent the behaviors accessing the array and columns for each interface index of the memory. Initial assignments are shaded in green and valid candidates are shaded in blue. Click in blue cells to change assignment and finally click **OK** button to apply the assignments. The valid candidates are ensuring that the access type of the behavior matches the access type of the memory interface.

The assignment is complete when all behaviors are assigned to an interface. Behaviors that are not yet assigned are highlighted in red.

If a memory interface is shared then the column header for the interface index will indicate the number of processes that can share that interface.

For arrays with external access see “Using the CtoS GUI to Allocate External Arrays” on page 9-31

9.1.6 Specifying Interfaces for Built-In RAM

You can also use the **allocate_builtin_ram** command (“[allocate_builtin_ram](#)” on page E-7) as follows:

- Use the **-read_interfaces**, **-write_interfaces**, and **-read_write_interfaces** options to specify the number of interfaces.

[Table 9-2 on page 9-18](#) lists the defaults of these interfaces for built-in RAM.

Table 9-2 Defaults for read_, write_, and read_write_interfaces Options for Built-In RAM

interface	type	description	default for built-in	legal range
read_interfaces	unsigned integer	number of read interfaces	number of processes that read from the memory	0+
write_interfaces	unsigned integer	number of write interfaces	number of processes that write to the memory	0+
read_write_interfaces	unsigned integer	number of read-write interfaces	0	0+

Notes

- You must specify a valid number of interfaces, as follows:
 $r + rw \geq \text{read processes}$
 $w + rw \geq \text{write processes}$
 $r + w + rw \geq \# \text{ of processes}$
- Some interfaces may be tied to ground if the number of interfaces is greater than required. Although rare, this might happen during final optimization, when memory accesses are removed.

9.1.7 Specifying Interfaces for Prototype Memory

CtoS lets you specify the number of read, write, and read-write interfaces to be generated for prototype memories.

You can also use the **allocate_prototype_memory** command ([“allocate_prototype_memory” on page E-14](#)), as follows:

- Use the **-write_interfaces** option to specify the number of write interfaces.
- Use the **-read_write_interfaces** option to specify the read-write interfaces.
- Use the **-read_interfaces** option to specify the number of read interfaces.

If the prototype memory has shared interface then you must specify the order of interfaces with explicit access type for each interface.

- Use the **-interfaces_types** option to specify all interfaces in order.

[Table 9-3 on page 9-19](#) lists the defaults of these interfaces for prototype memory.

Table 9-3 Defaults for read_, write_, and read_write_interfaces Options for Prototype Memory

interface	type	description	default for prototype	legal range
read_interfaces	unsigned integer	number of read interfaces	0	0+
write_interfaces	unsigned integer	number of write interfaces	0	0+
read_write_interfaces	unsigned integer	number of read-write interfaces	number of processes that access the memory	0+

9.1.8 Specifying Memory Clock for Built-In RAM, Prototype Memory, and Vendor RAM

CtoS allows users to specify a memory clock for built-in RAM, Prototype Memory, and Vendor RAM that does not have **clocks_per_port** specified. Memories can be driven by a clock that is different from the clocks of the processes that access the memory. All clocks must have the same period and the same edge, but they can be gated differently. Memory clocks are supported for shared memory interfaces and registered memories. The clocks of all processes that access the array must be identical if the clock is not specified.

You can select the memory clock from the list of compatible clocks (nets) in the **Memory Clock** combo box of the following dialogs:

- The **Allocate Built In Ram** dialog, as shown in [Figure 9-2 on page 9-5](#).
- The **Allocate Prototype Memory** Dialog, as shown in [Figure 9-4 on page 9-8](#).
- The **Allocate Memory for Array** dialog, as shown in [Figure 9-7 on page 9-12](#).

The clock net for the memory clock must meet the following conditions:

- The net must be in the module of the array argument.
- The net must be either:
 - hierarchically connected to the net of an existing clock definition.
 - the array does not have external accesses and the net is driven by a CGIC instance.
- The periods of the memory clock and the clocks of all processes that access the array must be identical.
- The edges of the memory clock and the clocks of all processes that access the array must be identical.
- In the case of external arrays, the memory clock must be specified consistently for each array in the family of external arrays. All specified memory clocks must be hierarchically connected to the same net in the top module with the clock definition.
- The memory clock must be enabled whenever any process is accessing the memory, accounting for the latency of the memory and the registers of registered memories. This is not checked by CtoS.

9.1.9 Exporting Memories

Selectively exporting memories is a preliminary feature.

The *exported memories* feature lets you set up memory connectivity at the top level of the design, thereby separating memories – RAMs or ROMs – from the actual design.

Important Built-in RAMs are *not* exported.

Module ports are created to connect memories instead of memory instances in modules.

This feature supports coding styles that require memories to be defined externally to actual functionality. Additionally, external memories can be excluded from RTL synthesis.

By default, memories are not exported. You can change the default by setting it on a design, or you can selectively export memories, as follows:

- Selectively exporting memories can be done with the **Export Memory** checkbox on the **Allocate Prototype Memory** and **Allocate Memory** dialogs [or with the **-export** option to the **allocate_memory** command (“[allocate_memory](#)” on page E-9) or **allocate_prototype_memory** command (“[allocate_prototype_memory](#)” on page E-14)].
- Changing the default can be done by selecting **Default Export** in the **Create New Design Wizard** ([Default Export](#) on page 6-17). [This also sets the **default_export_memories** design attribute (“[default_export_memories](#)” on page D-13).]

If you want to exclude some memories from being exported, you can selectively turn export off by unchecking the **Export Memory** checkbox (**-no_export** option) to the **allocate_memory** or **allocate_prototype_memory** commands.

After allocating memory, the **is_exported** array attribute (“[is_exported](#)” on page D-37) specifies whether a memory will be exported.

Then, to write a wrapper module for RTL simulation, you can select **File -> Generate -> Top Wrapper** [or use the **write_top_wrapper** command (“[write_top_wrapper](#)” on page E-173)]. This wrapper contains an instance of the synthesized RTL model and the set of exported RAM instances

Note See also “[Generating a Top Wrapper for Exported Memories](#)” on page 13-40.

To specify the required suffixes, use the **verilog_top_wrapper_suffix** design attribute (“[verilog_top_wrapper_suffix](#)” on page D-27) and **verilog_rtl_model_suffix** design attribute (“[verilog_rtl_model_suffix](#)” on page D-26).

Important Make sure that these extensions are *not* empty strings, because this can result in a name conflict of the top module and the top wrapper module of the design.

9.1.10 Registered Memories

CtoS supports the specification of *registered memories* – memories whose outputs on reads are buffered using one or more registers – by allowing you to specify *read latency of 1 to 3*, and *stalls on memory interfaces*.

When deciding where to add registers, a primary consideration is often timing. For many memory cells, the clock-to-Q delay (the launch delay) is large, and it may be advantageous to hide this latency by putting the registers on the output side of the memory, which is often the most common case. If further timing improvement is desired, the setup delay can also be hidden by putting registers on the input side.

All CtoS-standard signals have the same naming and meaning, but the timing of these signals will be different with a memory model that implements read latency greater than 1.

For information on the scheduling rules and restrictions of registered memories reads, see “[Scheduling Restrictions for Multi-Latency Operations](#)” on page -8.

9.1.11 Stallable Memories

Stall logic is used when array accesses are scheduled before a stall loop. CtoS supports memory stall inside CtoS for Vendor memories, builtin RAM, and prototype memories. For memories with total latency greater than 1, CtoS automatically generates the stall logic in the RTL instead of in the memory wrapper by the user. Stall logic is not done inside memory, but provided outside the memory by memory stall buffer.

However, stalling can be done only for memory reads of two processes that do not overlap. One process must not issue a memory read request until the read requests from other processes have been fully consumed. The registered memory has produced the data and that data is out of all registers created for the registered memory. CtoS creates an RTL netlist that generates simulation errors if this requirement is violated. Stall logic also leverages low power clock gating if the **low_power_clock_gating** design attribute is enabled.

9.1.12 External Arrays

External arrays is a preliminary feature.

The CtoS *external array* feature allows you to decompose your design into modules that communicate through shared arrays, in addition to through signal interfaces.

The external array feature consists of the following elements:

- a synthesizable pattern for modeling, in the SystemC code, an array declared in one module and accessed in sub-modules.
- options to the **allocate_memory_interfaces** command that let you specify the process assigned to the interface of the memory.
- a consistent simulation flow in the presence of external arrays that covers the whole range from the input SystemC model, the Verilog and SystemC simulation models, and the RTL model.
- synthesis of the modules that make up a hierarchical design in separate CtoS sessions.

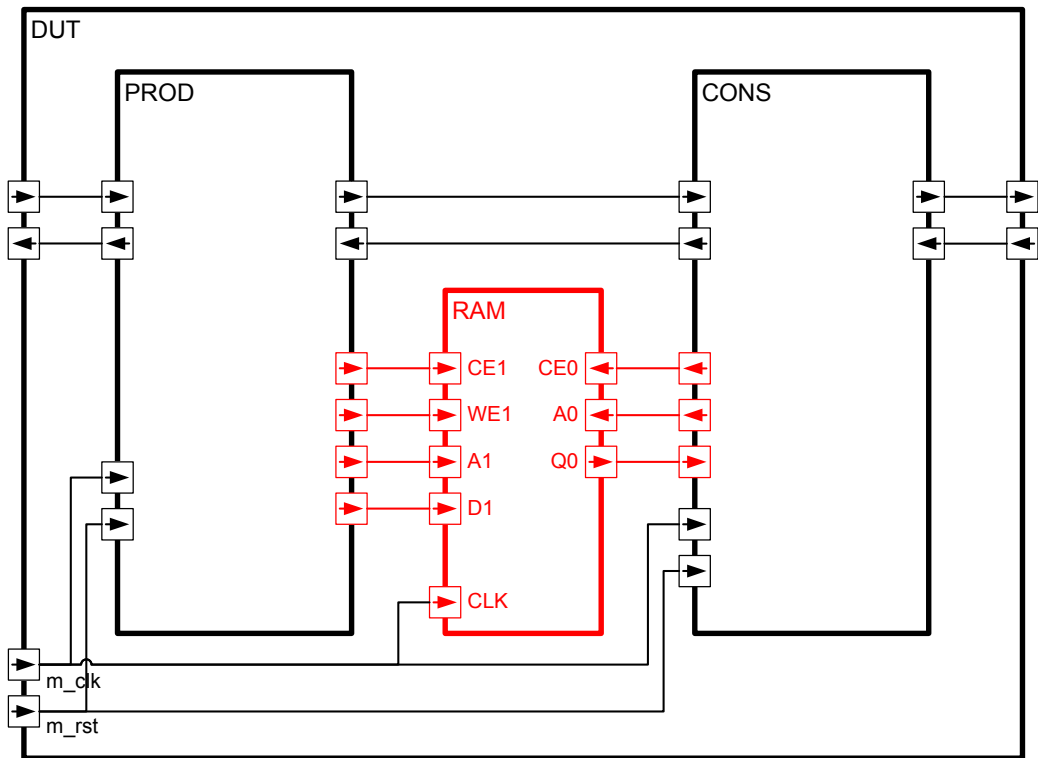
For example, in [Figure 9-11 on page 9-23](#), you can see the RTL resulting from synthesizing three SystemC modules, DUT, PROD and CONS. The modules communicate through signal ports (shown in **black**). In the RTL, the shared array is implemented as a RAM, and the synthesized modules PROD and CONS have ports generated by CtoS (shown in **red**) that are connected directly to the RAM. Unlike regular outputs from thread processes, the memory interface outputs need not be registered (and they are driven directly from the combinational logic of the thread.)

This section has the following subsections:

- [“Modeling with External Arrays” on page 9-23](#)

- [“Optimization/Micro-Architectural Transforms with External Arrays”](#) on page 9-30
- [“Allocating IP with External Arrays”](#) on page 9-30
- [“Scheduling with External Arrays”](#) on page 9-34
- [“Simulation with External Arrays”](#) on page 9-37
- [“Importing RTL IP into SystemC Designs”](#) on page 9-41
- [“Importing RTL IP into SystemC Designs”](#) on page 9-41

Figure 9-11 RTL module structure of design with 3 SystemC modules and external array



9.1.12.1 Modeling with External Arrays

This section contains the following subsections:

- [“External Array Modeling Pattern”](#) on page 9-24
- [“Terminology for the External Array Feature”](#) on page 9-26
- [“Topologies of Designs with External Arrays”](#) on page 9-27

- [“Building a Design with External Arrays” on page 9-27](#)
- [“Naming of External Arrays” on page 9-28](#)
- [“Arbitration of External Arrays” on page 9-28](#)
- [“Bit Representation of External Arrays” on page 9-28](#)
- [“Top-Module Instantiation with External Arrays” on page 9-29](#)
- [“Restrictions when Using the External Array Feature” on page 9-30](#)

External Array Modeling Pattern

The *external array modeling pattern* lets you access an array outside the module in which it is declared. For example, if the array to be shared is of type **T[N]**, the pattern consists of the following elements:

- The module constructor of each module that needs access to the array must take a formal argument of type **T[N]**.
- The module constructor of such a module can use the formal argument in its initializer list to initialize a module member variable of type **T*** or **T*const**. The processes of that module can access the array via that module member variable. It is recommended that the **T*const** variant be used, because it expresses the intent that this module member, which is a pointer to the external array, must not be modified after construction.
- The module constructor of such a module can also use the formal argument in its initializer list as an actual argument of the initializer of another module instantiated in the first module. This allows the second module to access the array.
- The array itself must be declared as a module member of type **T[N]** in a module that is the common ancestor of all modules that need access to the array in the module instantiation hierarchy.

Consider the following example, illustrated in [Figure 9-12 on page 9-25](#) and [Figure 9-13 on page 9-26](#)

- The module DUT instantiates the module PROD.
- The module PROD has an external array of **int** of 128 elements, which is inferred from the formal argument **pbuf** of its constructor. This constructor initializes a field **m_pbuf** of type **int*const**, which allows the processes of PROD to access the external array.
- Module DUT contains an array of **int** of 128 elements (**m_buf**). The constructor of DUT passes **m_buf** as an actual argument to the initializer of submodule instance **m_prod**, which is of type PROD. By doing so, it binds the external array **pbuf** of **m_prod** to the array **m_buf** of DUT.

Figure 9-12 SystemC Code Fragment of Design with Two Modules: Module PROD with External Array Referring to Array of Parent Module DUT

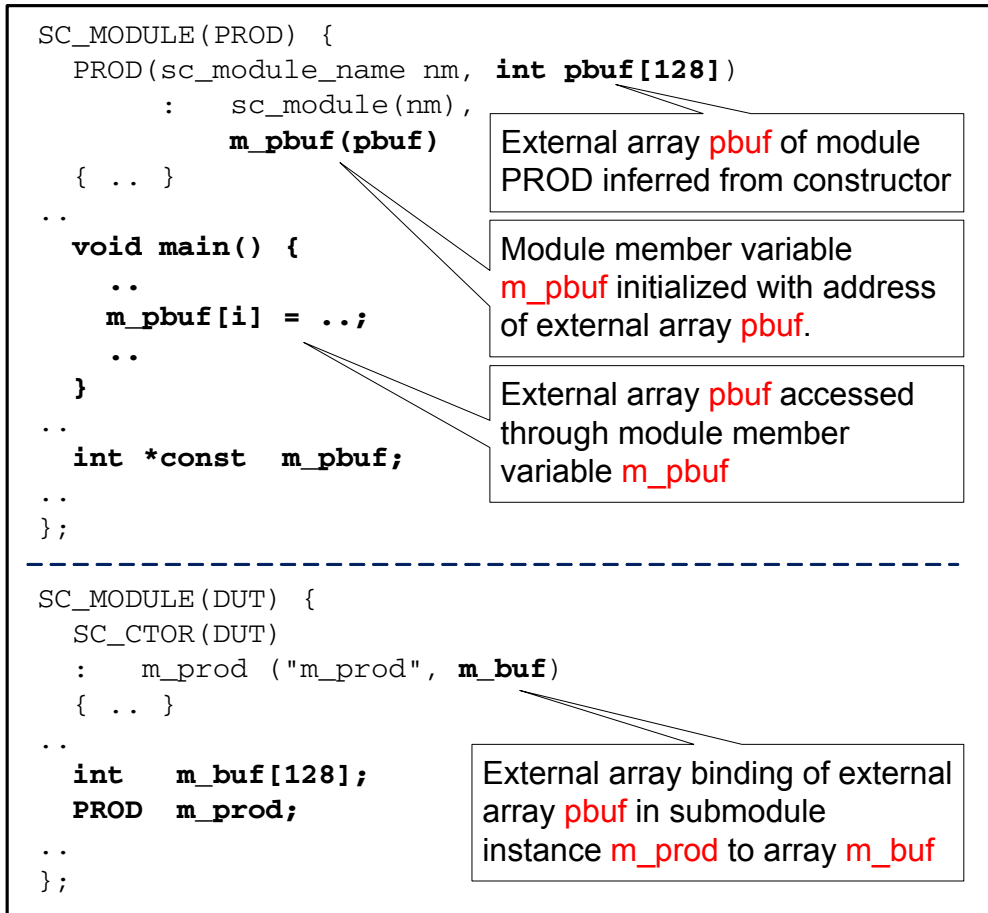
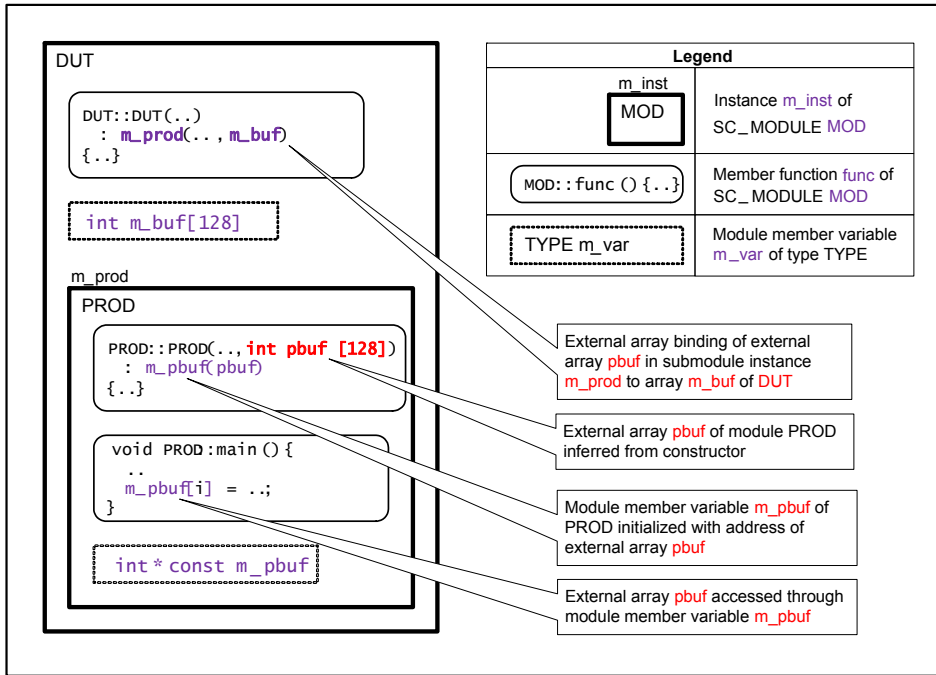


Figure 9-13 SystemC Code Fragment for Top-Module Instantiation of Module with External Array



Terminology for the External Array Feature

The following terminology is used throughout this section.

- **External array:** An array is *external* if it is inferred from a formal argument of a module constructor. In this case, CtoS assumes this array will also be accessed by processes outside the module containing the array. The external array must be implemented by a RAM or prototype memory, which is not part of the synthesized module.
- **Array with external access:** An array has *external access* if it is accessed by processes external to the module containing the array. This is the case for an external array, but it is also the case if a module binds an external array of a submodule instance to a regular array of the first module.
- **External array binding:** An *external array binding* is inferred from the initializer of a sub-module instance where the sub-module contains an external array. It expresses that the external array in the sub-module instance, and the array corresponding to the actual argument, are referring to the same C++ object. In the example in [Figure 9-12 on page 9-25](#), the initializer `m_prod` (“`m_prod`”, `m_buf`) induces an external array binding that states that the external array `pbuf` of `m_prod` refers to the same object as the array `m_buf` of DUT.

- **Family of arrays:** A *family of arrays* consists of arrays, and aliases of that array, in the design that are related through external array bindings. From a SystemC point of view, the members of the family all refer to the same block of C++ memory.
- **Root of a family of arrays:** The *root of a family of arrays* is the array of the family in the module highest in the module hierarchy. Either the family contains a non-external array, which must be the root, or the root of the family must be in the top module of the design.

Topologies of Designs with External Arrays

The members of a family of arrays all refer to the same block of C++ memory; therefore, it follows that the whole family is implemented as a single RAM in the synthesized design. There are two main topologies to consider:

- **Case 1:** The root of the family is an external array.
- **Case 2:** The root of the family is *not* an external array.

In **Case 1**, the root of the family is in the top-module. The members of the family will be implemented by a RAM outside the synthesized design. The top module of the synthesized design has terminals that are not inferred from `sc_in/sc_out` ports in the SystemC, but are inferred from the RAM interfaces allocated to the family of arrays; you must connect those terminals to the RAM during integration.

The array may also be read or written by a process external to the design, thus providing an input or output communication through this array.

Not all interfaces of the RAM are allocated to the synthesized design, because at least one interface must be available for a process outside the design to be synthesized by CtoS. [Figure 9-13 on page 9-26](#) illustrates the topology when synthesizing module PROD as a top-level module.

In **Case 2**, all processes that access the family of arrays are within the design to be synthesized. The family of arrays will be implemented by a RAM instantiated within the synthesized design (by CtoS). The top-level module of the synthesized design does not have extra terminals. [Figure 9-13 on page 9-26](#) illustrates the topology when synthesizing module DUT as a top-level module.

Building a Design with External Arrays

During the **build** process ([“Building a Design” on page 6-28](#)), for each formal argument of an array type `T[N]` of the constructor of a module in the design, CtoS creates an array in the database, as follows:

- The array is named after the formal argument.
- Its `is_external` array object attribute ([“is_external” on page D-37](#)) is set to true.
- The number of elements of the array is equal to `N`.
- The bitwidth of the array is equal to the bitwidth of type `T`, assuming `T` is a SystemC integer data type or a C++ built-in integer data type.

For each external array binding, CtoS creates an **external_array_binding** object (“[External Array Binding Object Attributes \(external_array_binding\)](#)” on page D-39), which is located in the **external_array_bindings** scope of an **array** object (“[Array Object Attributes \(Arrays\)](#)” on page D-36),

In the example in [Figure 9-12 on page 9-25](#), CtoS creates an **external_array_binding** that relates the array **m_buf** of DUT to the external array **pbuf** of PROD. Both of these arrays are logically the same and will be implemented by the same RAM in the synthesized design.

Collapsed modules

If a design is built in non-hierarchical mode [design attribute **build_flat** is true (“[build_flat](#)” on page D-12)], or if a module must be collapsed because it uses TLM constructs, then no external arrays are inferred from formal arguments of type **T[N]** of that module’s constructor. This is because the given module is being collapsed into its parent module, and thus the processes of the given module can directly access the module members.

Naming of External Arrays

External arrays in the CtoS database are named after the formal argument from which they were inferred. However, the name of the external array may not exactly match that of the formal argument because of clashes with other database objects.

Arbitration of External Arrays

Whenever an array is accessed by multiple processes, you must be sure that the design is unambiguous, and its behavior is independent of the order in which the simulator schedules processes. This is usually done through a *protocol*.

For example, the **tlm_fifo** module in the **ctos_tlm** library implements a protocol that ensures the producer and the consumer process will never access the same location in the internal array of the fifo at any given clock cycle. It is your (the designer’s) task to devise a suitable protocol.

This is no different for external arrays. CtoS does not automatically generate arbitration logic, so you must specify it explicitly in your SystemC source code.

Bit Representation of External Arrays

Using the external array feature, you can synthesize a module that interfaces with a multi-ported RAM. The synthesized module contains extra ports that must be connected by you to a multi-ported RAM that implements the array.

Some of the ports of the RAM must be connected to a module other than the first synthesized module. If the second module is handcrafted RTL, rather than CtoS-generated RTL, it is very important that both modules agree on the bit-representation of the array. For this reason, it is recommended that the element type of an external array be a SystemC integer data type (**sc_int<W>**, **sc_bigint<W>**, **sc_bv<W>**, etc.) or a built-in C++ integer data type (**int**, **short**, etc.). This issue is similar to using RTL IP.

Restrictions when Using the External Array Feature

Note the following restrictions when you are using the external array feature:

- In the initializer of a submodule instance, the type of the actual argument and the formal argument must agree on the number of array elements and the type of the array.
- Multi-dimensional external arrays are not supported, but this is not a hard restriction because CtoS supports casting a **T*** pointer that points into an array of **T**, to a pointer **T[M]**, which allows accessing the array as though it were a multi-dimensional array.

Note For more about this, see the **external_arrays** example in the following directory:

`install_directory/share/ctos/examples/features/arrays`

- If a module has an external array, the design must contain no more than one instance of that module.

9.1.12.2 Optimization/Micro-Architectural Transforms with External Arrays

A design consisting of modules that communicate through arrays with external access can be synthesized module by module. In the example in [Figure 9-11 on page 9-23](#), module PROD can be synthesized without the source code for module CONS, and vice versa.

For synthesized modules to operate correctly after integration, the layout of the bits of the arrays onto the bits of the RAM implementing the array must be consistent. Consequently, none of the commands for transforming arrays can be applied to arrays that have external access, and none of the automatic transforms for reducing bit-width and address-width are applied to arrays with external access (even if CtoS is synthesizing both modules accessing an external array).

9.1.12.3 Allocating IP with External Arrays

You must allocate a vendor memory or prototype memory to each array that has external access of that module in the Allocating IP step. IP allocation for an array that has external access consists of two parts, which can be done through the CtoS GUI or with the CtoS Tcl command interface:

- A choice of the vendor memory or prototype memory
- An assignment of the process, accessing the array, to the interface of the memory

You must also make sure the allocation is valid and complete, and some restrictions are met.

For more about these topics, see the following sections:

- [“Using the CtoS GUI to Allocate External Arrays” on page 9-31](#)
- [“Using the CtoS Tcl Command Interface to Allocate External Arrays” on page 9-32](#)
- [“Validity and Completeness of Allocation for Arrays with External Access” on page 9-34](#)
- [“Restrictions when Allocating IP for Designs with External Arrays” on page 9-34](#)

Using the CtoS GUI to Allocate External Arrays

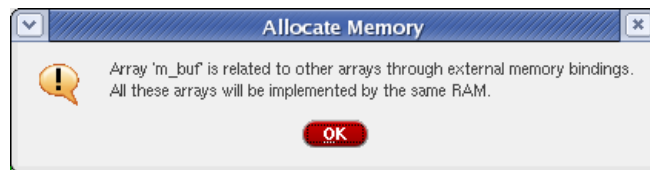
For a complete example of using the CtoS GUI to allocate external arrays, see the **external_arrays** example in the following directory:

`install_directory/share/ctos/examples/features/arrays`

Note Before running any CtoS examples, first review “Setup for Examples” on page F-2

At the Allocating IP step, if your design has an array that has external access, the CtoS GUI allows you to allocate the vendor or prototype memory for the *whole* family, as shown in [Figure 9-15 on page 9-31](#).

Figure 9-15 Allocate Memory - External Array Message



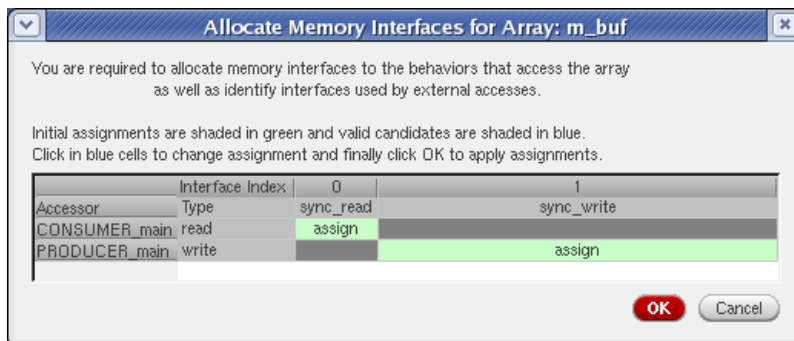
For Vendor RAM, you must select a memory definition with sufficient interfaces for all processes accessing this array. In addition, if the array has external access, the vendor RAM must have at least one interface available for external assignment.

For prototype memory, you must specify in the Memory Interfaces Box sufficient interfaces for all processes accessing this array. In addition, if the array has external access, the prototype memory must be declared with at least one interface available for external assignment.

Note Although, CtoS does support sharing an interface, the external access can not be shared.

For external arrays, automatic interface allocation is not supported. The GUI will automatically display the **Assign Memory Interfaces Dialog** after allocating the memory, as shown in [Figure 9-16 on page 9-31](#) to let you choose the memory interfaces for each behavior.

Figure 9-16 Allocate Memory Interfaces Dialog



This dialog has table where rows represent the behaviors accessing the array and columns for each interface index of the memory. Initial assignments are shaded in green and valid candidates are shaded in blue. Click in blue cells to change the assignment and finally click OK button to apply the assignments. The valid candidates are ensuring that the access type of the behavior matches the access type of the memory interface.

The assignment is complete when all behaviors are assigned to an interface. Behaviors that are not yet assigned are highlighted in red.

For those arrays that have external access, an additional row in the table for “External” will represent interface indices that are dedicated to the external access.

If a memory interface is shared then the column header for the interface index will indicate the number of processes that can share that interface.

Note It is not valid to assign the “External” row to the shared interface index.

Using the CtoS Tcl Command Interface to Allocate External Arrays

In the CtoS Tcl command interface flow for external arrays, you must choose the IP allocation for *each* member of a family of arrays separately; however, CtoS does make sure your choices are consistent.

The external array feature requires that you use the following Tcl commands/options:

- [“Using the -no_auto_interface_allocation to allocate_commands” on page 9-32](#)
- [“Using the allocate_memory_interfaces Command” on page 9-33](#)

An example of using the Tcl command interface flow is found in:

- [“Tcl Command Interface Flow Example” on page 9-33](#)

Using the -no_auto_interface_allocation to allocate_commands

The interfaces of a memory are *not* automatically assigned to processes accessing the array when you specify the **-no_auto_interface_allocation** option with the following commands:

- **allocate_builtin_ram** (“[allocate_builtin_ram](#)” on page E-7)
- **allocate_prototype_memory** (“[allocate_prototype_memory](#)” on page E-14)
- **allocate_memory** (“[allocate_memory](#)” on page E-9)

This option is *required* if the given array has external access.

Using the `allocate_memory_interfaces` Command

The `allocate_memory_interfaces` command ([“allocate_memory_interfaces” on page E-11](#)) requires:

- a memory that has already been allocated to the given array
- none of the specified interfaces has been allocated

If the `-process` option is specified:

- the specified interfaces are allocated to the specified process
- the given process must belong to the same module as the given array

If the `-inst` option is specified:

- the specified interfaces are assigned to the specified submodule instance
- the specified array must have an external array binding to an external array in the specified instance
- the specified instance and array must belong to the same module

Tcl Command Interface Flow Example

Consider the IP allocation for module DUT of [Figure 9-11 on page 9-23](#). A valid allocation can be accomplished with the following Tcl commands:

```
set dut_mod [find -module DUT]
set dut_arr $dut_mod/arrays/m_buf
set prod_inst $dut_mod/insts/m_prod
set cons_inst $dut_mod/insts/m_cons
allocate_prototype_memory $dut_arr -no_auto_interface_allocation \
    -read_interfaces 1 -write_interfaces 1 -read_write_interfaces 0
allocate_memory_interfaces $dut_arr -inst $prod_inst -interfaces {1}
allocate_memory_interfaces $dut_arr -inst $cons_inst -interfaces {0}
```

Note that if more than 1 interface must be assigned to a process, this must be done with a single `allocate_memory_interfaces` command.

Consider the IP allocation for module PROD of [Figure 9-11 on page 9-23](#). A valid allocation can be accomplished with the following Tcl commands:

```
set prod_mod [find -module PROD]
set prod_arr $prod_mod/arrays/pbuf
set prod_proc $prod_mod/behaviors/PROD_main
allocate_prototype_memory $prod_arr -no_auto_interface_allocation \
    -read_interfaces 1 -write_interfaces 1 -read_write_interfaces 0
allocate_memory_interfaces $prod_arr -process $prod_proc -interfaces {1}
```

Note that in this case, interface 0 is assigned implicitly to processes external to module PROD.

Validity and Completeness of Allocation for Arrays with External Access

The allocation commands for the arrays of a *family of arrays with external access* must be consistent with each other. The consistency requirement follows from the following principles:

- All of the arrays are implemented by the same memory instance (outside the design if the root of the family is external, and otherwise inside the design).
- If the current allocation is incomplete, there must be a sequence of allocation commands that will augment the current allocation to an allocation that is complete.
- Multiple processes can be allocated to the same memory interface if the memory interface is shared. A memory interface can be shared through the multiplex option of the **allocate_memory** or the **allocate_prototype_memory** command.

An allocation for a particular array in a family is complete if and only if:

- Every process that accesses that array has been allocated sufficiently capable interfaces, and
- For each external array binding of given array, the specified instance has been allocated sufficiently capable interfaces to satisfy the requirements of processes nested in the module hierarchy rooted at that instance.

If the root of the family is an external array, at least one actual memory interface must be left un-allocated, so it can be used by a process external to the design. This implies that the memory interface used externally cannot be multiplexed.

Restrictions when Allocating IP for Designs with External Arrays

Note the following restrictions when allocating IP for designs with external arrays:

- Vendor memories that support asynchronous read (which are not common) cannot be allocated to arrays that have external access.
- Built-in memories cannot be allocated to external arrays; you must use prototype memories instead.

9.1.12.4 Scheduling with External Arrays

Read and write operations into an external array are scheduled on their birth edge, unless you have specified a **float_array_accesses** command ([“float_array_accesses” on page E-70](#)) for that array.

Make sure array accesses do not float beyond the region in which the arbitration protocol avoids contention with other processes.

9.1.12.5 Support of SystemC Testbenches

To simulate the System C testbench, CtoS inserts the *identity bridges*. The bridge interprets the signals at the RTL side of the bridge (the side connected to the thread logic) and translates them into DPI-C calls for reading and writing in the C++ array.

[Figure 9-17 on page 9-36](#) shows an example of a synchronous read/write identity bridge.

The bridge can be compiled in two ways, depending on whether the Verilog text macro **CTOS_SIM_MULTI_LANGUAGE_EXTERNAL_ARRAY** is defined:

- By default, the bridge simply connects its ports on the memory side to the ports on its RTL side.

This is the view seen by logic synthesis (RC).

- For designs using external arrays, the CtoS-generated Makefile defines the Verilog text macro **CTOS_SIM_MULTI_LANGUAGE_EXTERNAL_ARRAY**, which causes the functionality for performing the array access through DPI-C to be included.

By default, the bridge simply connects the memory side ports to the RTL side ports.

However, at simulation time 0, the verification wrapper may configure a particular bridge instance to execute read and write accesses through DPI-C, in which case, the memory side ports are not used.

More information about the operation of simulation with external arrays is provided in the following section, [“Simulation with External Arrays” on page 9-37](#).

Figure 9-17 Example of Synchronous Read/Write Identity Bridge

```
module identity_sync_read_write_16x16m0(rtl_CE, rtl_A, mem_Q, rtl_D, rtl_WE, CLK, mem_CE,
mem_A, rtl_Q, mem_D, mem_WE);
    input rtl_CE;
    input [3 : 0] rtl_A;
    input [15 : 0] mem_Q;
    input [15 : 0] rtl_D;
    input rtl_WE;
    input CLK;
    output mem_CE;
    output [3 : 0] mem_A;
    output [15 : 0] rtl_Q;
    output [15 : 0] mem_D;
    output mem_WE;

    assign mem_CE = rtl_CE;
    assign mem_A = rtl_A;
    assign mem_D = rtl_D;
    assign mem_WE = rtl_WE;
`ifndef CTOS_SIM_MULTI_LANGUAGE_EXTERNAL_ARRAY
    assign rtl_Q = mem_Q;
`else
    // This is only required when simulating a multi-language design.
    reg [15 : 0] dpi_Q;
    bit use_dpi;
    wire m_ready;

    // Pick which Q drives the RTL Q.
    assign rtl_Q = use_dpi ? dpi_Q : mem_Q;

    initial begin
        use_dpi = 0;
        @(posedge m_ready)
            use_dpi = 1;
    end

    ctos_external_array_accessor #(.ADDR_WIDTH(4), .DATA_WIDTH(16),
    .TRACE(`CTOS_TRACE_EXT_ARRAY)) arr_ref(m_ready);

    always @(posedge CLK)
        begin
            if (use_dpi) begin
                if (rtl_CE && !rtl_WE)
                    arr_ref.read(rtl_A, dpi_Q);
                if (rtl_CE && rtl_WE)
                    arr_ref.write(rtl_A, rtl_D);
            end
        end
`endif
endmodule
```

9.1.12.6 Simulation with External Arrays

At any stage of your synthesis flow, simulation of CtoS-generated models, with the original SystemC testbench, is fully supported for designs with external arrays:

- Simulation of models of designs with external arrays is completely transparent if CtoS-generated Makefiles are used.
- Simulation of these models without CtoS-generated Makefiles requires a few extra flags and arguments to **ncsc_run** or **irun**, as long as the CtoS-generated verification wrapper is used.
- Simulation of these models without using the verification wrapper is not supported.
- Simulation of RTL models requires a design wrapper to *hide* the RAM interfaces of the RTL module from the original testbench. The design wrapper plays a similar role as for designs with exported memories. The design wrapper can be generated using the **write_top_wrapper** command ([“write_top_wrapper” on page E-173](#)).

Simulation of the RTL model, with the RTL testbench, of a design with external arrays is also fully supported and does not require any special simulator flag. The generated RTL contains comments that specify how to connect the ports of the RAM interfaces to the RAMs.

Additional information about simulation with external arrays is provided in the following sections:

- [“Supporting External Arrays in Simulation” on page 9-37](#)
- [“Enabling Debug Tracing of ctos_external_array Library” on page 9-39](#)
- [“Limitations of Simulating Designs with External Arrays” on page 9-40](#)

Supporting External Arrays in Simulation

The requirements for supporting external arrays in simulation are complicated. To explain some of these requirements, consider the hierarchical design shown in [Figure 9-11 on page 9-23](#).

If you are synthesizing module PROD only, the array referred to by the external array of PROD is still in the original SystemC testbench (in module DUT).

If you are simulating the models of PROD in the original SystemC testbench, where the other modules CONS and DUT have not been synthesized, then the Verilog model for PROD needs to read and write into C++ memory and to perform the appropriate data conversion between Verilog bit vectors and the type used in SystemC.

An additional feature is that you can plug the same model of PROD in a testbench where the actual array is also in a synthesized module. This requires the Verilog module of PROD to access a Verilog memory in an ancestor module.

CtoS provides a mechanism to perform accesses into external arrays. It is built on top of the SystemVerilog DPI-C standard. The mechanism is encapsulated into library modules, called the **ctos_external_array** library, instantiated in CtoS-generated modules. The generated models call tasks on the library modules that perform external array access.

However, these library modules must first be configured at the start of simulation. This is done by the verification wrapper. The configuration process is complex, and this is why simulation without the CtoS-generated verification wrapper is not supported.

Simulation of designs with external arrays requires a few extra flags and arguments to the simulator so SystemVerilog DPI-C features are enabled and libraries are compiled. The CtoS-generated Makefile takes care of all of this.

For the specific arguments required for **ncsc_run** and **irun**, see the following sections:

- [“Simulation with ncsc_run” on page 9-38](#)
- [“Simulation with irun” on page 9-39](#)

Simulation with ncsc_run

Simulation of designs that have external arrays with **ncsc_run** requires the following additional flags and arguments:

- **-DPI**

This argument enables the DPI functionality.

- **-ncsim_args,-sv_lib,libncsc_model.so**

This argument tells the simulator where to find the DPI library,

- **"\$CTOSROOT/share/ctos/include/ctos_external_array/*.cpp**

These are the C++ source files of the external array library.

- **"\$CTOSROOT/share/ctos/include/ctos_external_array/*.sv**

These are the SystemVerilog source files of the external array library.

- **"**+define+CTOS_SIM_MULTI_LANGUAGE_EXTERNAL_ARRAY****

This macro must be defined when simulating an RTL model with external arrays.

- **-DCTOS_MODEL=top_wrapper**

This sets the suffix string needed in wrapper instantiation to **_top_wrapper**.

Simulation with irun

Simulation of designs that have external arrays with **irun** requires the following additional flags and arguments:

- **-DPI**

This argument enables the DPI functionality.

- **"-Wcxx,-I\$CTOSROOT/share/ctos/include/ctos_external_array**

This is the include path for the external array library.

- **"\$CTOSROOT/share/ctos/include/ctos_external_array/*.cpp**

These are the C++ source files of the external array library.

- **"-sv \$CTOSROOT/share/ctos/include/ctos_external_array/*.sv**

These are the SystemVerilog source files of the external array library.

- **"+define+CTOS_SIM_MULTI_LANGUAGE_EXTERNAL_ARRAY**

This macro must be defined when simulating an RTL model with external arrays.

- **-DCTOS_MODEL=top_wrapper**

This sets the suffix string needed in wrapper instantiation to **_top_wrapper**.

Enabling Debug Tracing of ctos_external_array Library

To enhance the visibility of the actions being performed inside the **ctos_external_array** library, CtoS lets you turn on debug traces in the library.

In addition to some debug traces signaling library setup at the start of simulation, each access to the external array (read/write) is tracked and printed to the log file. For every access, the following information is reported:

- time of access
- full path of object accessing the array
- type of access - read/write
- array address
- array data

This tracing function can be turned on as follows:

- For **ncsc_run**: Add to the command line:
"-DCTOS_TRACE_EXT_ARRAY -ncvlog_args,' -define CTOS_TRACE_EXT_ARRAY'
- For **irun**: Add to the command line:
"-defineall CTOS_TRACE_EXT_ARRAY"

Limitations of Simulating Designs with External Arrays

There are two limitations with simulating designs with external arrays:

- Side-by-side simulation with a verification wrapper is not supported if a design has an external array.
- No contention checks are performed for arrays with external access
- CtoS-generated SystemC simulation models are not supported for use in a hierarchical synthesis flow.

The SystemC model was designed to be a drop-in replacement of the original SystemC model and therefore, designed to work without the verification wrapper."

9.2 Importing RTL IP into SystemC Designs

The *RTL IP* feature of CtoS lets you implement combinational functions in the SystemC specification of a design using an existing RTL design.

You can therefore provide carefully optimized implementations of performance-critical functions. Additionally, these functions may be pipelined.

CtoS uses the external RTL design for estimating timing and area of the function during implementation of the larger SystemC design. The final RTL output for the SystemC design contains an instance of the external RTL design, which must be included in the simulation and synthesis of the RTL output.

Here are the basic steps to include RTL IP in a design:

1. Before you **build** the design, include the following pragma on any function to be implemented with RTL IP, so you can change the RTL IP without re-synthesizing the rest of the RTL:

```
#pragma ctos dont_touch

int my_special_function(int a, int b) { ... }
```

This sets the CtoS **dont_touch** attribute (“[dont_touch](#)” on page D-41) to true on behaviors as they are created by the CtoS elaborator. During optimization, these attributes are preserved when the behaviors are uniquified, and the behavior interfaces (input/output ports) will not be trimmed or optimized out.

Important If calls to **dont_touch** functions evaluate to either constant zero or one, those behaviors *will* be optimized out.

2. Create the XML RTL IP file (see “[rtl_ip_def XML Elements](#)” on page H-5 and “[RTL IP](#)” on page H-28).
3. After you have built the design, read in the XML RTL IP **.xml** file(s) to create an **rtl_ip_def** for each RTL IP definition (“[RTL IP Definition Object Attributes \(rtl_ip_defs\)](#)” on page D-83) in the directory system for your design.

In the CtoS GUI, you would select **Edit -> Read IP Definitions**, or the **Read IP Def** button from the **Specify Micro-architecture** or **Allocate IP** dialog to display the **Read IP Definition** dialog, as shown in [Figure 9-18 on page 9-42](#). You can also use the **read_ip_defs** command (“[read_ip_defs](#)” on page E-97). For example, for an XML file called **muladd.xml**, the command would be:

```
read_ip_defs -ip_def muladd.xml
```

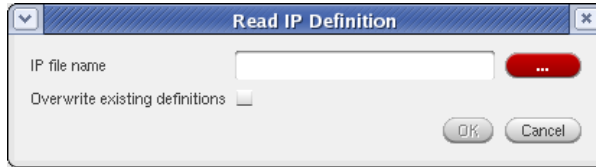
You may do this more than once to read separate **.xml** files.

The following **muladd rtl_ip_def** object would then appear in the directory system:

```
/designs/my_design/rtl_ip_defs/muladd
```

The attributes on the **rtl_ip_def** object contain the values read from the XML description file.

Figure 9-18 Read IP Definition Dialog - Importing RTL IP



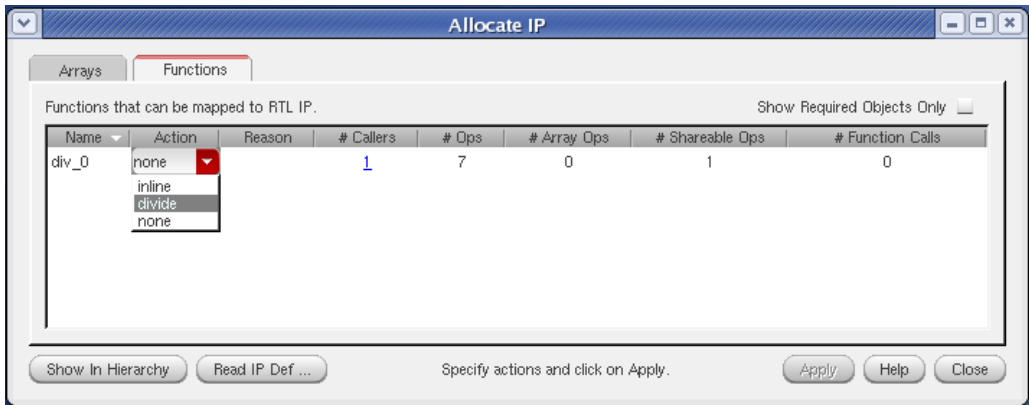
4. Link the behavior that implements the SystemC function to the external RTL, described in the **rtl_ip_def** object. CtoS will compare the function behavior to the RTL IP description to make sure the inputs and outputs in the SystemC code match the name and widths given in the XML description. Also, the parent process clocks and resets are compared to those described in the XML.

In the CtoS GUI, you would use the **Allocate IP** dialog, as shown in [Figure 9-19 on page 9-42](#).

You could also use the **use_ip** command ("[use_ip](#)" on [page E-156](#)). For the **muladd** function, the command would be:

```
use_ip [find -rtl_ip_def muladd] [find -behavior muladd]
```

Figure 9-19 Allocate IP Dialog - Importing RTL IP



The following sections describe some considerations when importing RTL IP:

- “RTL IP Requirements for RTL Designs, SystemC Code” on [page 9-43](#)
- “Effect of Linking Behaviors to RTL IP” on [page 9-43](#)
- “Stall Loop Support in RTL IP” on [page 9-44](#)
- “Using RTL IP on User-Defined Types” on [page 9-46](#)

9.2.1 RTL IP Requirements for RTL Designs, SystemC Code

When used with the RTL IP feature, RTL designs and SystemC code must meet a few requirements, as described below.

RTL designs:

- Must be synthesizable Verilog, supported by the required implementation tool [that is, Encounter RTL Compiler (RC), Xilinx XST, Altera Quartus].¹
- Must conform to SystemVerilog P1800.
- Must be combinational or conforming to a linear pipeline with an initiation interval of 1 (equal number of flops from data input ports to output ports).

SystemC code:

- Must encapsulate code to be replaced in a combinational function (without **wait** statements).
- Must match the behavior between the combinational function and the RTL design.
- Must access only local variables and arguments inside the function (no accesses to module I/Os or memories).
- Must use **SC_THREAD** for the process containing the combinational function.
- Function arguments must be primitive or **sc** types, such as **sc_int** (see “Using RTL IP on User-Defined Types” on page 9-46).

9.2.2 Effect of Linking Behaviors to RTL IP

Once linked to an RTL IP definition, the contents of a function’s behavior remain, but scheduling, timing analysis, and the RTL output are now linked to the RTL IP specification. Behavioral-level commands, such as **write_sim** (“[write_sim](#)” on page E-165), use the original SystemC representation because this model’s abstraction level is higher than RTL.

Executing the **write_rtl** command (“[write_rtl](#)” on page E-161) produces an RTL model that will instantiate the RTL IP. Behavior matching between the RTL IP and SystemC code is necessary because the **write_sim** model uses the SystemC code to generate the behavioral simulation model. Unless the RTL IP matches the SystemC code, the **write_rtl** simulation output will not match the **write_sim** simulation output.

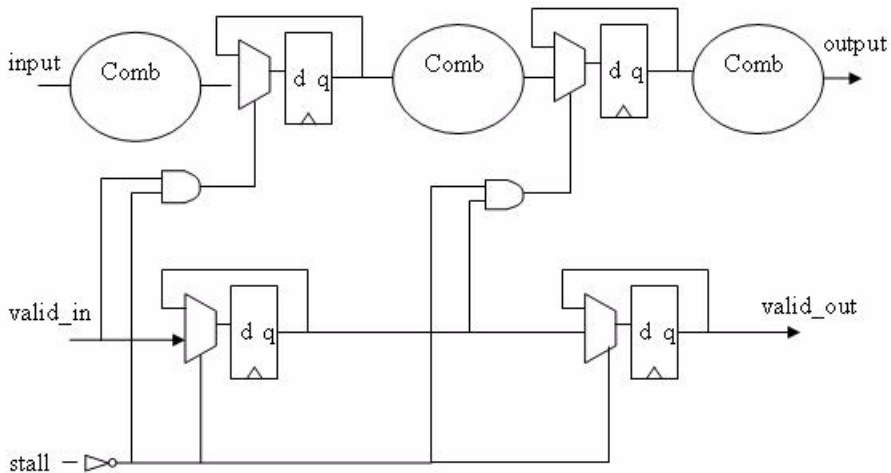
Since the RTL IP is used internally for timing and area estimates, it must be readable and synthesizable by the specified implementation target tool: Encounter RTL Compiler (RC) or an FPGA synthesis tool.¹

1. Support for FPGA designs is a preliminary feature. CtoS does not support Quartus II Web Edition Software.

9.2.3 Stall Loop Support in RTL IP

CtoS supports stall loops for RTL IP. The stall loop can be either *inside* or *outside* of a pipelined loop. [Figure 9-20 on page 9-44](#) shows RTL IP with a pipeline depth of 2 and its **input**, **valid_in**, **stall**, **output**, and **valid_out** ports.

Figure 9-20 RTL IP with Pipeline Depth 2

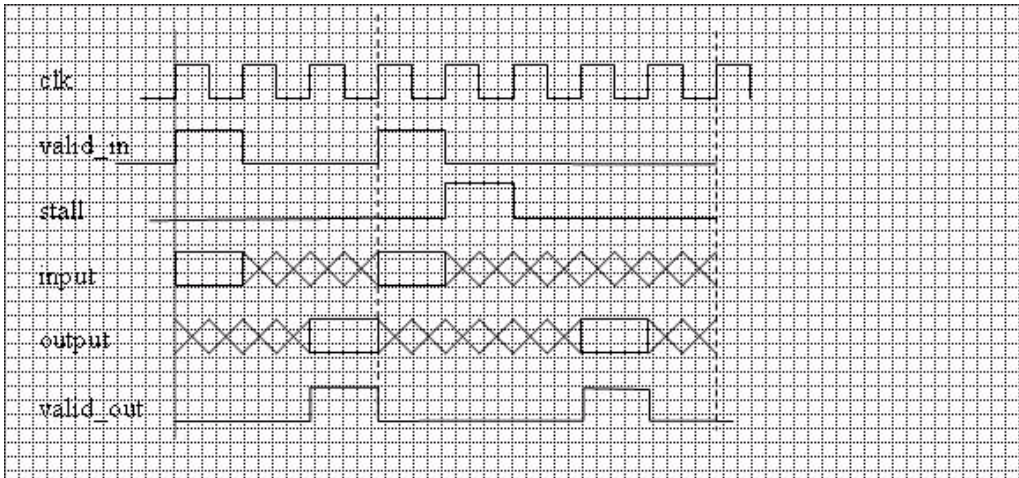


[Figure 9-21 on page 9-45](#) shows the timing when the input is valid and no stall signal is issued, and when the subsequent **valid_in** signal is true and the **stall** signal is true for a single cycle.

The attributes (under the **ports** tag) that support this feature include **clock_port**, **input_port**, **output_port**, **reset_port**, **stall_port**, **valid_in_port** and **valid_out_port**.

Note For descriptions of these and all other RTL IP definition object attributes, see “[RTL IP Definition Object Attributes \(rtl_ip_defs\)](#)” on page D-83.

Figure 9-21 Timing for (1) input valid, no stall; (2) valid_in true, stall true



There are a few current limitations for stall loop support in the RTL IP feature:

- Ops that use RTL IP resources with pipeline depth N must be able to be scheduled on an edge that is followed by N states before the first fork or join.
- The pipelined function must not have side effects. It cannot access member variables, global variables, outputs or arrays.
- The pipeline latency interval (LI) must be greater than the pipeline depth of the RTL IP.
- RTL IP with pipeline depth greater than 0, and without a stall loop, cannot be scheduled in a pipeline loop with a stall loop.
- Restrictions for the pipeline depth of the RTL IP as it relates to the initiation interval (II) of the pipelined loop are shown in [Table 9-4 on page 9-45](#).

Table 9-4 Restrictions of RTL IP Pipeline Depth Related to Pipelined Loop II

RTL IP pipeline depth	pipelined loop II	
	1	>1
0	okay	okay
1	okay	okay
> 1	okay	no

9.2.4 Using RTL IP on User-Defined Types

The Verilog representation of user-defined function arguments in SystemC is not specified and is subject to change.

To avoid using user-defined types for function arguments, use a primitive type, such as **sc_bv**, and define conversion functions and constructors.

For example, consider the following basic floating-point number class and a multiplication function **mul**:

```
template <int M, int E>
class my_float {
public:
    my_float();
    my_float(const sc_bv<M + E> &bv);
    my_float(sc_uint<M> mant,
            sc_uint<E> exp);

    sc_bv<M+E>    to_bv() const;
    ...

};

template <int M, int E>
my_float<M,E>
dut_1::mul(const my_float<M,E> &a,
          const my_float<M,E> &b)
{
    sc_uint<2 * M> m = a.get_mant().to_uint() * b.get_mant().to_uint();
    sc_uint<E>     e = a.get_exp().to_uint() + b.get_exp().to_uint();

    return my_float<M, E>(m.range(2 * M - 1, M), e);
}
```


The function **mul** has user-defined arguments and is not suitable for binding to RTL IP.

The **my_float** class has a constructor and a member function to convert from **and** to **sc_bv**.

The wrapper function **mul_bv** is defined for this purpose, which uses **sc_bv** for its arguments and return value:

```
template <int M, int E>
sc_bv<M + E>
dut_1::mul_bv(sc_bv<M + E> a,
              sc_bv<M + E> b)
{
    return mul(my_float<M,E>(a), my_float<M,E>(b)).to_bv();
}
```

The **mul_bv** function can be used in other functions:

```
my_float<5, 3> a;
my_float<5, 3> b;
my_float<5, 3> p;

p = mul_bv<5, 3>(a.to_bv(), b.to_bv());
```

The **mul_bv** function can be bound to RTL IP, where the relationship between the arguments in SystemC and in Verilog are well-defined.

10 Analyzing Micro-Architecture

After you have specified a desired micro-architecture, CtoS can provide some preliminary reports on the area, timing, and power of your transformed design.

Although analysis at this point in the CtoS design flow is less accurate than analysis after scheduling, the results can provide comparisons between various micro-architectural choices.

This step is optional, and final optimizations will be automatically applied if you move directly to the Scheduling Step.

Here are the reports that you can see, at this point in the design flow:

- [“Check Design Report” on page 10-2](#)
- [“Prescheduled Timing Report” on page 10-3](#)
- [“Prescheduled Area” on page 10-4](#)
- [“Prescheduled Power Report” on page 10-5](#)
- [“Summary Report” on page 10-6](#)

Note Analysis *after* scheduling is described in [“Analyzing and Implementing Designs” on page 13-1](#).

10.1 Check Design Report

To review a design for inconsistencies, select **Report -> Check Design**. CtoS reviews the design for the following problems and issues the appropriate error or warning:

- *A process does not have a proper reset state.* A reset state of a process is a state node reachable from the origin node via a path that does not contain any state nodes. For synthesis, the reset state must be unique, and there must be only one path in the CFG from the origin node to the reset state. If a node in a reset path has the **preserve** node attribute set to *true*, you may get multiple reset paths. To avoid this problem, set the **preserve** node attribute to *false* and run the **optimize** command [ERROR (CTOS-7026)].
- *An operation is illegal on the reset path in a process.* Ops on the reset path of a process are restricted to assignments of constants. If the process has multiple resets, these assignments can be conditional, and the condition tests whether one of the reset conditions is active [WARNING (CTOS-7027)].
- *A loop was found between combinational SC_METHODs.* Loops between combinational SC_METHODs lead to RTL that is not synthesizable. If this is a false cycle, consider splitting the SC_METHODs [ERROR (CTOS-7102)].
- *A combinational process cannot be synthesized as it contains loops.* Unroll the loop using the **unroll_loop** command, or rewrite your source code so it becomes a thread process that will allow loops to be broken [WARNING (CTOS-19026)].
- A combinational process cannot be synthesized as it accesses an array. Either change the array to an array of `sc_signals`, flatten it if it is a local array, or change the process to a thread process [WARNING (CTOS-19027)].
- *A variable is never set.* The value of an unassigned variable is being read. Rewrite your code to assign a value to this variable before it is used [WARNING (CTOS-19029) or WARNING (CTOS-19030)].
- *A function call must be inlined so a process can be properly reset.* All function calls in a thread process before the first **wait** statement must be inlined [WARNING (CTOS-19046)].
- *A write op is writing a non-constant value during reset.* Assignments to module outputs or signals that appear before the first **wait** statement of a process must assign constant values [ERROR (CTOS-19047)].
- *An operation is not consistent with the reset priorities of a process.* A process with multiple resets must respect the priorities of these resets while testing reset conditions on the reset path. Reset priorities are inferred from the first **if** statement in the body of the process. For example, if a process has two synchronous reset conditions, **RST1** and **RST2**, and both are active high, then the process should test that **RST2** is active only after testing that **RST1** is inactive [ERROR (CTOS-19097)].
- *An edge will be split during scheduling to allow for possible state insertion.* Edges on which both input and output ops are born will be split during scheduling by adding a simple non-state node. CtoS may thus add one or more states between read and write if more latency is needed [WARNING (CTOS-19110)].

Notes

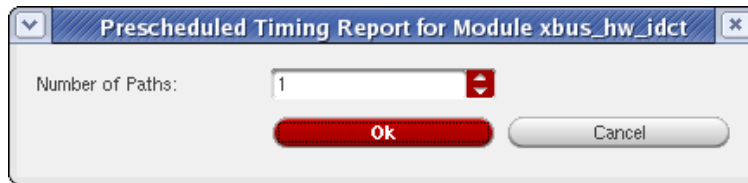
- This works well after the Set up Design Step, but may be more accurate after the Allocate IP Step.
- You can also use the **check_design** command ("[check_design](#)" on page E-32).

10.2 Prescheduled Timing Report

To get a *Prescheduled Timing Report*, select **Report -> Timing -> Report**.

As shown in [Figure 10-1 on page 10-3](#), the **Prescheduled Timing Report** dialog is displayed, in which you can select the number of paths to report.

Figure 10-1 Prescheduled Timing Report



The **Prescheduled Timing Report**, as shown in [Figure 10-2 on page 10-3](#), is displayed for all of the paths you have chosen; you can access them using the red arrows at the top right of the window.

Figure 10-2 Prescheduled Timing Report

Prescheduled Timing Report for Module xbus_hw_idct					
xv_idct_run with slack 22920 and total available time 28000		Timing report for critical path #5 in process xbus_hw_idct_run with slack -7726 and total available time 112000			
Op Name	Op Type	Master	Delay (ps)	Arrival (ps)	Slack (ps)
1 read_xbus_hw_idct_ms_in53	read	-	0	0	24633
2 if_in53	if	if_then_else	0	0	24633
3 ctrlAnd_1_in53	unary_and	unary_and_2	213	25266	24213
4 ctrlAnd_1_in52	unary_and	unary_and_2	213	74538	-7726
5 mux_inptr_in209	mux	mux_2_6	259	74751	-7726
6 add_in238	add	add_3x2	859	75010	-7726
7 memread_xbus_hw_idct_coef_block_in238	memRead	ram_64x16m0_1ar_1w_r	1813	75869	-7726
8 add_in243	add	add_17	4357	77682	-7726
9 add_in245	add	add_18	4633	82039	-7726
10 mul_in245	mul	smul_32x18x15	8378	86672	-7726
11 add_in256	add	add_32	8018	95050	-7726
12 add_in259	add	add_32	8018	103068	-7726
13 add_in259_0	add	add_31	7812	111086	-7726
14 add_in272	add	add_32	8018	118898	-7726
15 add_in272_0	add	add_22x1	3188	126916	-7726
16 memwrite_workspace_in272	memWrite	ram_64x21m0_1ar_1w_r	1852	130104	-7726

Note You can also use the **report_timing** command ("[report_timing](#)" on page E-130).

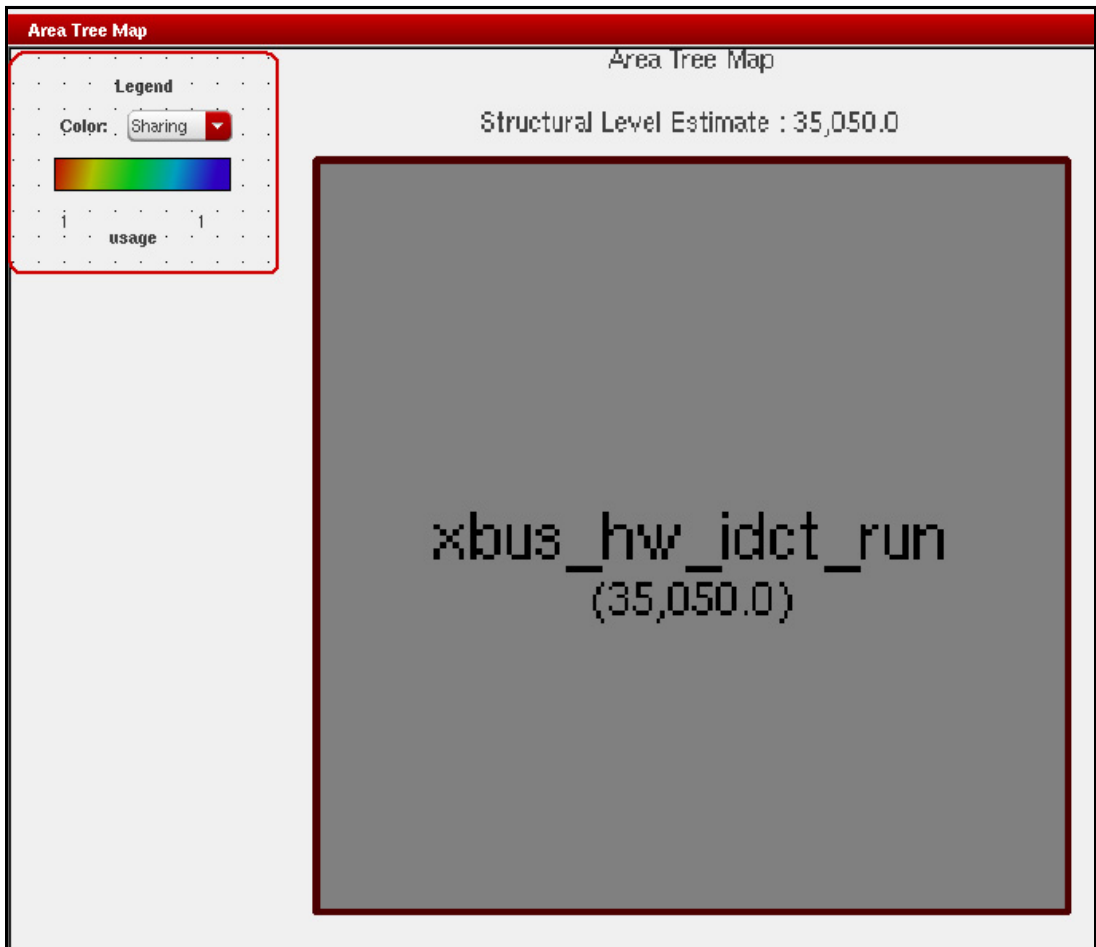
10.3 Prescheduled Area

The Tree Map is a preliminary feature.

To see an area report, in the form of a *Tree Map*, as shown in [Figure 10-3 on page 10-4](#), select **Report->Area**.

Warning The report will *not* be as helpful at this point in the design flow, as report generated after scheduling.

Figure 10-3 Area Tree Map - Preschedule



Note For more information about the Tree Map, see [“Reporting Power and Area Using the Tree Map” on page 13-29](#).

10.4 Prescheduled Power Report

You can get power estimates on behavioral-level designs after making micro-architecture decisions and allocating arrays, but before scheduling. This provides an analysis of power that can be used to compare with other micro-architecture alternatives. Doing power estimates earlier in the flow (that is, before scheduling) has the advantage of not waiting to complete all the scheduling phases and only running timing phase, as well as faster simulation time in generating toggling activity counts.

The following is the step-by-step Behavioral-Level Power Estimation flow:

1. Set up a design. For more information see, [“Creating a New Design” on page 6-9](#).
2. Specify the micro-architecture. For more information, see [Chapter 8 “Specifying Micro-Architecture”](#).
3. Allocate IP. For more information see [Chapter 9 “Allocating Memory and RTL IP”](#).
4. Perform all steps upto the Analyze Micro-architecture step.

Note Step 1 through Step 4 should be invoked on the same design on which the simulation model was generated; therefore, do not make any changes to your design while performing these steps.

5. Perform one of the following:
 - **Set Default Toggling Probabilities:** See [“Setting Default Toggle Probability” on page 18-5](#).
 - **Simulate to get more accurate toggling counts:**
 1. Run the **write_sim** command with the **-tcf** option (see [“Using the -tcf Option of write_rtl and write_sim Commands” on page 18-6](#)), which promotes registers to module level and adds behavioral-level tracing information (for example, execution frequency for basic blocks in the SystemC source code) to the simulation file.
 2. Run INCISIV (specifically, run the **dumpptcf** command from ncsim) to save a record of the toggling activity and of the probability of being at logic level 1 for each net. For more information on using dumpptcf, see [“Considerations when Using INCISIV dumpptcf” on page 18-9](#) or see the Cadence INCISIV documentation.
 3. Run the **read_tcf** command to read toggling count from simulation dump(see [“Using the read_tcf Command” on page 18-7](#)).
6. Run the **report_behavioral_power** command to request that RC estimates the power for a default implementation of every operation in the CDFG, given the switching activity at its inputs. For more information, see [“report_behavioral_power” on page E-106](#).

10.5 Summary Report

To see a **Summary Report** for a design, select the *Summary Report* icon (along the left side of the main CtoS GUI window – the last icon, which looks like a note pad), or select **Window -> Summary Report**.

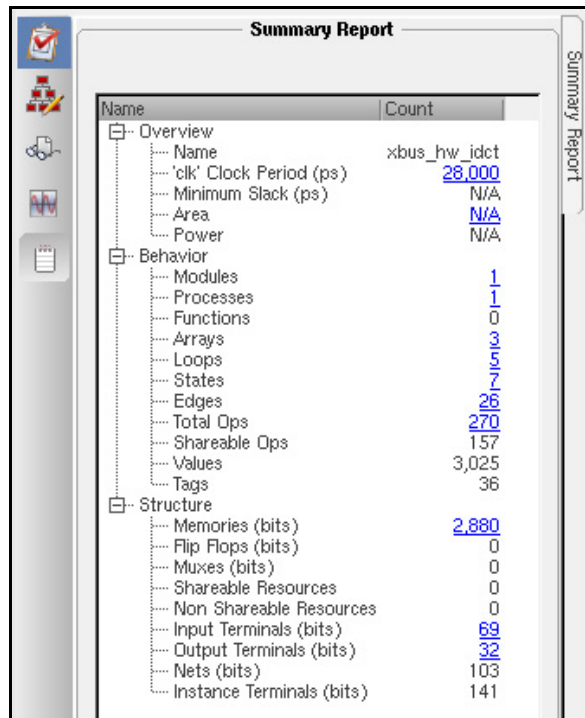
The **Summary Report**, as shown in [Figure 10-4 on page 10-6](#), gives you a sense of the design complexity – both in terms of the number and kind of behavioral objects appearing in the source code and in terms of the number and kind of structural elements that CtoS uses to implement that behavior.

Before scheduling, you will see *only* behavioral information; after scheduling, you will also get structural information.

Notes

- See “[Summary Report](#)” on [page 13-35](#) for an example of the **Summary Report** after scheduling.
- You could also use the `report_summary` command (“[report_summary](#)” on [page E-128](#)).

Figure 10-4 Summary Report



Name	Count
Overview	
Name	xbus_hw_idct
'clk' Clock Period (ps)	28,000
Minimum Slack (ps)	N/A
Area	N/A
Power	N/A
Behavior	
Modules	1
Processes	1
Functions	0
Arrays	3
Loops	5
States	7
Edges	26
Total Ops	270
Shareable Ops	157
Values	3,025
Tags	36
Structure	
Memories (bits)	2,880
Flip Flops (bits)	0
Muxes (bits)	0
Shareable Resources	0
Non Shareable Resources	0
Input Terminals (bits)	69
Output Terminals (bits)	32
Nets (bits)	103
Instance Terminals (bits)	141

11 Advanced Features for Guiding the CtoS Scheduler

Before you schedule a design, you may want to use some of the following advanced features to give the CtoS scheduler more precise instructions:

- [“Creating and Managing Array Dependencies” on page 11-2](#)
- [“Creating and Managing States” on page 11-6](#)
- [“Creating and Managing Resources” on page 11-20](#)

The last section of this chapter is designed to help you resolve potential problems before you try to schedule your design:

- [“Potential Sources of op span Scheduling Failures” on page 11-27](#)

11.1 Creating and Managing Array Dependencies

To create and manage array dependencies, you have the following options:

- “Creating Array Dependencies” on page 11-2
- “Producing an Array Dependency Report” on page 11-2
- “Breaking Array Dependencies” on page 11-3

11.1.1 Creating Array Dependencies

Prior to scheduling a design, you can create *array dependencies* for the design. These dependencies are *internal* data structures that insure array operations will be done in a safe order during scheduling.

If you do not create array dependencies, CtoS *automatically* creates them during scheduling.

The advantage of creating dependencies *prior* to scheduling is that you can analyze them to determine if you want to *break* some of them before continuing. The section, “[Breaking Array Dependencies](#)” on page 11-3, explains why you might want to do this and provides instructions for how to do it.

Important Redundant memory ops are optimized out, during the optimization process, but memory ops are not optimized across join nodes in a design.

To create array dependencies, in the CtoS GUI, select **Edit -> Guide Scheduler -> Create Array Dependencies**.

Note You can also use the `create_array_dependencies` command (“[create_array_dependencies](#)” on page E-41).

11.1.2 Producing an Array Dependency Report

After you have created array dependencies, you can see a report of these dependencies as pairs of memory ops or memory ops and sequential function calls that contain array accesses, in the context of the specified design, module, behavior, or array.

In the CtoS GUI, select **View -> Arrays**, and you will see the **Array Dependency Report**, as shown in [Figure 11-1 on page 11-2](#).

Figure 11-1 Array Dependency Report

Array Dependency Report for Design DUT	
Array dependencies for behavior DUT_proc	
ArrayOp1	ArrayOp2
1 memwrite_DUT_A_In22	memread_DUT_A_In27
2 memwrite_DUT_A_In22	memread_DUT_A_In27_0

Note You can also use the `report_array_dependencies` command ([“report_array_dependencies” on page E-105](#)).

11.1.3 Breaking Array Dependencies

As described in the preceding section, you can create array dependencies before scheduling. However, this CtoS process may be too conservative and may infer a dependency that you *know* does not exist. The following three examples show cases in which you might want to *break* a created dependency.

Example 1

A classic case in which you might want to break a created dependency is shown in the following example. Because CtoS does limited analysis to determine if there is overlap between array indexes, an assumption is made that reads from **A** and writes to **A** must be done in order:

```
for (i=1; i<100k; i++) {
    A[i] = A[i+200] + 37;
}
```

Example 2

Another typical case in which you might want to break an array dependency is when you know the two addresses are different. In the following example, the read can take place before or after the write, and the design will be correct (assuming there is more than one word in the array).

```
for(i=0; i<10; i++){
    for(j=i+1; j<10; j++){
        x = mem[i];
        mem[j] = y;
    }
}
```

Example 3

CtoS does limited array dependency analysis between the caller and the sequential function containing array accesses. If both the caller and callee access the same array, false dependencies such as read-read array accesses are detected and would not be recorded. However, write array accesses to disjoint array addresses would not be detected.

The following example illustrates a false dependency between the caller and the sequential function, which needs to be broken by the user.

```
void main() {
    ...
    mem[i] = x;
    wait();
    seq_func(i+2);
}
```

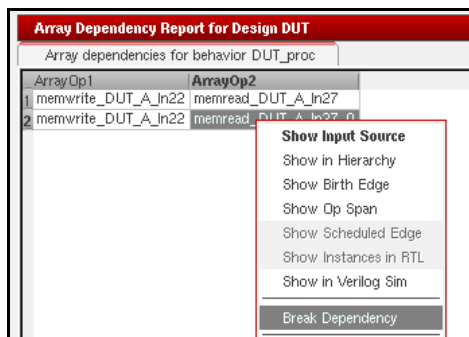
```
void seq_func(int i) {  
    mem[i] = y;  
    wait();  
}
```

Note that array dependencies between the caller and a sequential function can only be broken, if all array accesses inside the sequential function are independent of the array access in the caller. For example, the dependency cannot be broken for the following sequential function:

```
void seq_func(int i) {  
    wait();  
    int value = mem[i-2];  
    wait();  
    mem[i] = value;  
    wait();  
}
```

To reinstate an array dependency, you can simply create array dependencies again, as described in the preceding section, “[Creating Array Dependencies](#)” on page 11-2. To remove an overly conservative dependency, in the CtoS GUI, select **View -> Arrays**, right-click on the array, and select **Break Dependency**, as shown in [Figure 11-2](#) on page 11-4.

Figure 11-2 Using the Array Dependency Report to Break a Dependency



Note You can also use the **break_array_dependency** command (“[break_array_dependency](#)” on page E-27).

Example 4

A classic case in which you might want to break inter-iteration loop array dependencies is shown in the following example. Because CtoS does limited analysis to determine if there is an overlap between array indexes from one iteration to the other, in case a loop is pipelined, the user might want to set the independent distance for the following example:

```
LOOP: for (i=1; i<100k; i++) {
```

```
A[i] = A[i+4+input.read()] + 37;  
}  
break_array_inter_iteration_dependencies -num_iterations 3 LOOP_for_begin A
```

Note that in the following example, CtoS is able to understand that the number of independent iterations is equal to 3.

```
LOOP: for (i=1; i<100k; i++) {  
  A[i] = A[i+4] + 37;  
}
```

11.1.4 Scheduling Sequential Functions with Array Accesses

When the function with array accesses is not inlined it is guaranteed that it must be scheduled before its callers. This means that issuing a schedule command for a caller will first trigger issuing the scheduling command for all the functions that it calls, which contain memory accesses and only after that the scheduling of the caller will proceed.

However, the user must note that the results of scheduling sequential functions with array accesses can impact the scheduling of the caller (that is, the behavior that calls the sequential function) and the callee (the called sequential function) in the following ways:

- If the sequential function has accesses to array A scheduled on the call cycle edge (that is, the edge combinationally adjacent to the origin node of the sequential function) then all edges combinationally adjacent to the edge of the sequential function call will be unavailable for scheduling any operations accessing array A in the caller.
- The return cycle edge (that is, the edge combinationally adjacent to the end node of the sequential function) is given to the caller unless the caller has no accesses to array A. This means that during scheduling a sequential function with no memory accesses in the function can be scheduled combinationally adjacent to the return edge of the function. Additionally, even if there is no memory contention between the caller and the callee, the scheduler will ensure that no memory ops can be scheduled on the return cycle edge inside the sequential function.

11.2 Creating and Managing States

There are several ways to add and manage states in CtoS.

For example, you can add a state by breaking a loop, explicitly creating a state, or adding a latency constraint.

Creating states in *combinational* functions has some specific limitations, which are described in the following section:

- [“Limitations When Creating States in Combinational Functions” on page 11-7](#)

After that section are the following sections on creating and managing states, in general:

- [“Creating Required States” on page 11-8](#)
- [“Creating Individual States” on page 11-9](#)
- [“Specifying External Delay for Process I/O Nets” on page 11-10](#)
- [“Specifying External Delay for Pipelined or Sequential Functions” on page 11-11](#)
- [“Constraining Latency” on page 11-12](#)
- [“Producing a Latency Report” on page 11-13](#)
- [“Creating Protocol Regions” on page 11-14](#)
- [“Constraining ops to Edges” on page 11-17](#)
- [“Constraining ops to Edges” on page 11-17](#)
- [“Constraining ops to Expandable Edges” on page 11-19](#)

11.2.1 Limitations When Creating States in Combinational Functions

Creating states in combinational functions has some specific limitations.

Before describing these limitations, note the following related details about how functions and processes are handled in CtoS:

functions

- when you add a state to a combinational function, the function then becomes sequential.

processes

- *Combinational* **SC_METHOD** processes (sensitive to *inputs*) are passed directly to RC to generate a combinational block.
- *Clocked* **SC_METHOD** processes (sensitive to *clock*) are treated exactly as **always@(posedge clk)**, that is, combinational logic plus one register is created.
- If you need to add a state to a function, you must write it as an **SC_CTHREAD** process, which is the only construct actually scheduled by CtoS.

All other processes are optimized, while retaining the same cycle behavior, into something that can be processed by RC.

- For more information about the types of SystemC processes used as input to CtoS, see [“Processes” on page 14-9](#).

Here, then, are the limitations when creating a state in a combinational function:

- The CFG must have more than two nodes (origin and end node) or transitively call a function whose CFG has more than two nodes.

To add a state to a simple function with only an origin and an end node, you should add either a label or a state to the function source.

- The behavior must not be transitively called from within a pipelined loop.
- The behavior must not have RTL IP associated with it, or be transitively called from a behavior that has RTL IP associated with it.

This is necessary in order to keep consistent cycle behavior between the simulation model and the generated RTL.

- No op constraints should be associated with calls to the behavior.

States cannot be added to a combinational behavior that has calls constrained to an edge or to a resource.

- Combinational behaviors can be converted to sequential only before final optimization; therefore, you cannot create a state after final optimization.

11.2.2 Creating Required States

If you have defined a latency constraint on an affected path(s) (see “[Constraining Latency](#)” on page 11-12) or are running in relaxed latency scheduling mode (see “[Relaxed Latency Scheduling Mode](#)” on page -4), you can insert states to increase the latency between ops, as long as constraints are not violated. This is required to avoid *sequential conflicts* or conflicts due to *resource contention* between array ops.

Examples of these conflicts include:

- a write to an array location and a read from it within the same cycle (*sequential conflict*)
- two read ops in the same cycle on an array with only one read port (*resource contention conflict*)

For *sequential conflicts*, the decision on which edges to break is made based on a cost function that considers the following metrics, in this order:

1. the maximum number of conflicts resolved by the same insertion
2. the minimum increase of maximum latency on the resulting path
3. the minimum number of edges affected
4. the maximum number of loops affected
5. the earliest of two edges

For *resource contention conflicts*, a simplified scheduling of the behavior that looks only at array ops and ignores resource sharing and timing is attempted. If that fails, one or more states will be added on suitable edges. It will succeed if, at the end, every op has at least one edge on which it can be scheduled.

Important CtoS cannot proceed if *sequential conflicts* are not resolved. For *resource contention conflicts*, the algorithm is conservative, and additional conflicts can be detected and resolved later on, during scheduling.

To insert states, in the CtoS GUI, select **Edit -> Guide Scheduler -> Create Required States**, and you will see the **Create Required States** dialog, as shown in [Figure 11-3 on page 11-8](#). Note that you can also **Enable Relax Latency** in this dialog (see “[Relaxed Latency Scheduling Mode](#)” on page -4).

Figure 11-3 Create Required States Dialog



To stop this process, select the **Interrupt** button (see “[Interrupt Button](#)” on page 6-32), and CtoS will leave the states it has already created. If you then start the process again, it will continue where it left off.

Note You can also use the `create_required_states` command ([“create_required_states” on page E-46](#)).

11.2.3 Creating Individual States

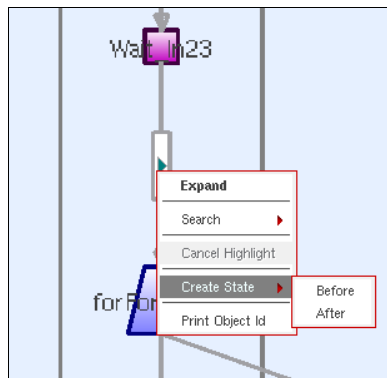
You can insert a state before or after a edge, if latency constraints allow it, for certain behaviors.

The edge must meet the following qualifications:

- The edge must be expandable due to a latency constraint.
- The edge cannot be a backward edge, or a busy loop edge of a behavior call, or belong directly or indirectly (via a function call) to a clocked `SC_METHOD`.
- The edge cannot be associated with an op constraint. You just first insert the desired edge and then apply the op constraint.

Important Review the restrictions for the specified behavior in [“Limitations When Creating States in Combinational Functions” on page 11-7](#).

Figure 11-4 Creating a State in the CDFG



To create a state, bring up the **CDFG (View -> CDFG)** for the appropriate process. Right-click on the desired edge, and select **Create State ->Before** or **After**, as shown in [Figure 11-4 on page 11-9](#).

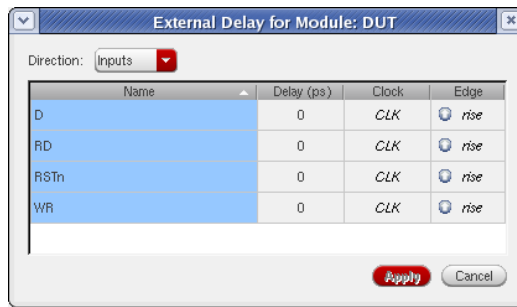
Note You can also use the `create_state` command ([“create_state” on page E-50](#)).

11.2.4 Specifying External Delay for Process I/O Nets

You can set the external delay of either a user-created net or port on a module.

Arrival times apply to any net or port on a module that is an input to a process, because either the process reads the input port (**sc_port**) associated with the net or port on a module, or the net or port on a module represents an **sc_signal** that is read and written by different processes. The *required* time applies to all nets or ports on a module that are outputs of a process, because either the process writes the output port (**sc_port**) associated with the net or port on a module, or the net or port on a module represents an **sc_signal** that is read and written by different processes.

Figure 11-5 External Delay Dialog - Process I/O Nets



To set external delay in the CtoS GUI, right-click on any *user-created* net or port in the **Hierarchy Window** and select **Edit External Delay**. In the **External Delay** dialog, as shown in [Figure 11-5 on page 11-10](#), note the following:

- The **Name** column is read-only because names are specified by the design.
- The **Delay** column can be any integer from **0** to the **clock_period**. It can also be a range (for example, **100 - 200**) when individual bits are set differently, and it is editable.
- The **Clock** is the name of the clock in the design. It can also be **Mix Values** when individual bits are set differently, and it is also editable.
- The **Edge** is either **rise** or **fall**. It can also be **Mix Values** when individual bits are set differently. Double-clicking on the item causes it to toggle to the other value.
- When you modify a value, the font is bolded, and the **Apply** button is enabled. However, when you are editing bits of a terminal, all bits must be set before the **Apply** button will be enabled.
- If a delay has not been set on a terminal, a clock and edge are inferred from the behaviors to which the terminal is connected, and the inferred value is displayed in *italics*. If a clock cannot be inferred, the **Delay** is **Not Set** (in red), and **Clock** and **Edge** are empty. A tooltip (when hovering over **Not Set**) displays the reason a clock could not be inferred, as follows:
 - No clocks in the design.
 - There is more than one clock in design, and the terminal is not connected to a clocked behavior.

- There is more than one clock in design, and the terminal is connected to multiple clocked behaviors that are sensitive to different clocks.
- You can perform interactive editing (similar to the **Clock** tab of the **Design Property** dialog).
- You can set external delay on terminals of a module when selected from **Input Terminals** or **Output Terminals** in the **Hierarchy Window**. You can also switch between **Inputs** and **Outputs**.
- You can set individual bits of a terminal when selecting the terminal name in the **Hierarchy Window**.

Note You can also use the `external_delay` command ("[external_delay](#)" on page E-58).

11.2.5 Specifying External Delay for Pipelined or Sequential Functions

You can set external delay on the behavior terminals of a pipeline function or sequential function.

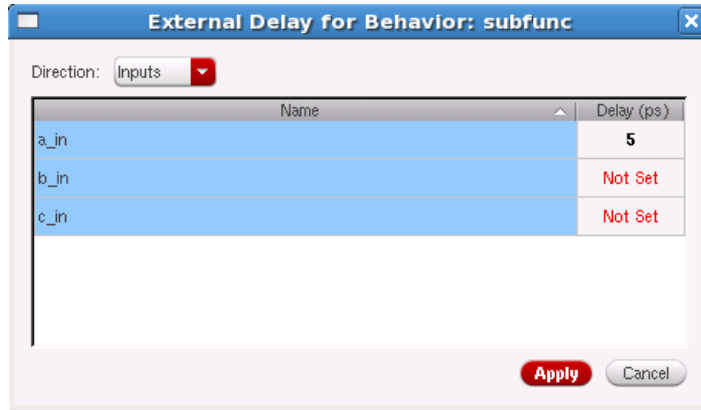
The external delays are copied to the module terminals of the pipeline implementation module when it is created during final optimize. The external delays for both the behavior terminals of the specification behavior and the module terminals of the implementation module are obtained from and set on the module terminals of the implementation module from then on.

During schedule of the pipeline function module, the timing analyzer initializes the external delays of the module terminals of the shadow module for the behavior terminals of the specification behavior with the external delays of the implementation module terminals. At the end of schedule, the calculated delays of the shadow module terminals are copied to the input and output delay of pipeline module terminals. And, the max delay of the pipeline function module is also updated.

During the first report timing after register allocation, the input and output delay of pipeline module terminals are updated with more accurate values. At this time, the max delay of the pipeline function module is not updated.

To set external delay in the CtoS GUI, click the **Input Terminals** or **Output Terminals** node under **Pipelined Functions** in the **Hierarchy Window**. The **External Delay** dialog is displayed, as shown in [Figure 11-6 on page 11-12](#).

Figure 11-6 External Delay Dialog - Behavior Terminal



In the **External Delay** dialog, note the following:

- The **Name** column is read-only because the names are specified by the design.
- The **Delay** column can be any integer from 0 to the clock_period. It can also be a range (for example, 100 - 200) when individual bits are set differently, and it is editable.
- When you modify a value, the font is bolded, and the **Apply** button is enabled. However, when you are editing bits of a terminal, all bits must be set before the **Apply** button will be enabled.

11.2.6 Constraining Latency

You can constrain the latency – set a bound for the maximum latency allowed on a set of paths – between two control-flow nodes. The CtoS scheduler may then insert state nodes on any path for which a constraint is defined, as long as an inserted state node does not cause the path to exceed the maximum latency. These states provide opportunities to share resources and to enable reads and writes.

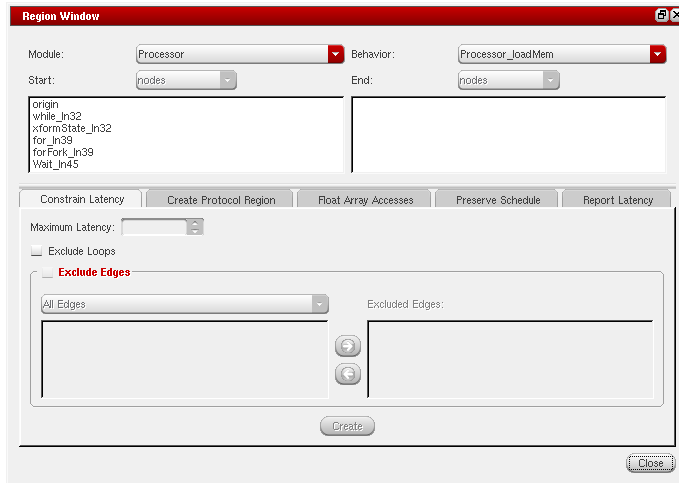
The set of paths, to which the constraint is applied, contains all paths from the start node to the *optional* end node. If you do *not* specify an end node, the start node *must* be the start of a loop (you will get an error if not), and CtoS will automatically set the end node to the end of the loop.

If you choose to exclude certain edges, the set of paths will *not* include any paths that cross those edges. If you exclude loops, the set of paths will contain only *simple* paths, that is, paths that do not cross a backward edge; therefore, no loops will be included. If no path meets all of the conditions you have set, the constraint will have no effect because the maximum latency can be neither determined nor enforced.

During some transforms, not all constraints are preserved; namely, a constraint made on a set of paths that includes an edge on which a function is called will be preserved when the function is inlined, but a constraint made on a set of paths that includes a loop will *not* be preserved after the loop is unrolled. In the latter case, constraints should be set *after* all unrolling transforms have been completed.

To set a latency constraint, select **Edit -> Region -> Constrain Latency**. The **Region Window** dialog, showing the **Constrain Latency** tab, is displayed, as shown in [Figure 11-7 on page 11-13](#).

Figure 11-7 Region Window Dialog, the Constrain Latency Tab



Important

- Review the restrictions for the specified behavior in [“Limitations When Creating States in Combinational Functions” on page 11-7](#).
- For an example, see [“Setting a Latency Constraint \(Option 4\)” on page C-19](#).
- To insert states in a specific location, see [“Creating Individual States” on page 11-9](#).
- You can also use the **constrain_latency** command ([“constrain_latency” on page E-36](#)).

11.2.7 Producing a Latency Report

CtoS provides a report of the latency of paths between operations in the data flow, or between nodes in the control flow.

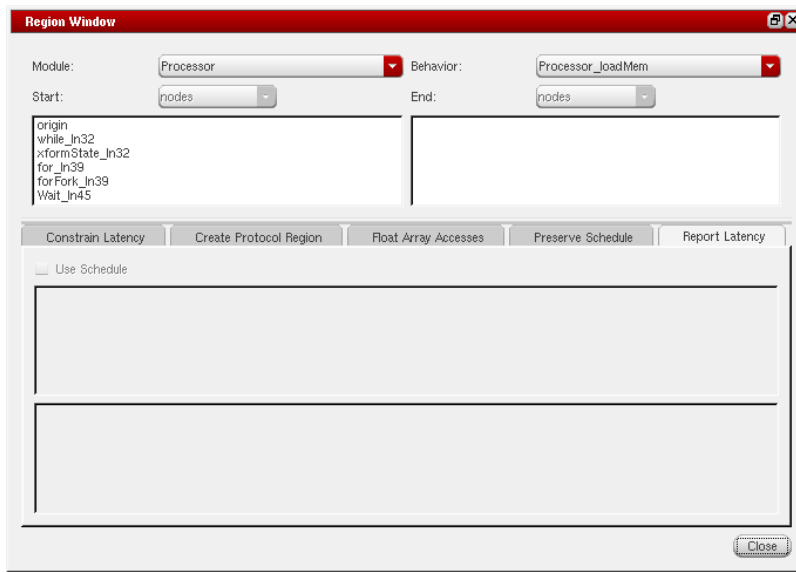
This report shows the minimum and maximum number of states on any path from start to end, not including the start node if it is a state, and displays a sample path for both the minimum and maximum latency.

If any inner loops are intersected in any of the paths, an error is issued.

To see this report, in the CtoS GUI, select **Report -> Region Latency**.

The **Region Window** dialog is displayed, with the **Report Latency** tab selected, as shown in [Figure 11-8 on page 11-14](#).

Figure 11-8 Region Window Dialog, the Report Latency Tab



Note You can also use the **report_latency** command ([“report_latency” on page E-108](#)).

11.2.8 Creating Protocol Regions

Protocol regions provide a mechanism to ensure that CtoS transformations do not break I/O protocols. If a part of the CDFG is designated as a protocol region, CtoS will enforce the following two rules in the generated RTL:

- Any two I/O operations in the region that are executed in the same cycle in the input SystemC model will be executed in the same cycle in the generated RTL as well.
- If two I/O operations in the region are not executed in the same cycle in the input SystemC model, then the number of cycles between them will stay the same in the generated RTL.

If violating the second rule does not break the protocol, the protocol is said to be stallable. The **-stallable** option to the **create_protocol_region** command indicates to CtoS that only the first rule needs to be enforced.

For example, assume that in the following piece of code ‘valid’ and ‘data’ are outputs, ‘ready’ is an input:

```
BEGIN:
    valid = 1;
    data = a; ①
```

```
wait();
while (!ready) wait();②
valid.write = 0;
END:
```

Here is the declaration of a protocol region (without the **-stallable** option) between BEGIN and END labels:

```
create_protocol_region [find -node BEGIN] [find -node END]
```

The first rule requires that outputs ‘valid’ and ‘data’ will always be assigned in the same cycle. To ensure this, CtoS will never create a state between them.

The second rule requires that ‘ready’ input is checked in every cycle after ‘data’ and ‘valid’ are set, until it is asserted and then ‘valid’ is set to 0. This is necessary to enforce this particular protocol (called valid-ready) which requires that ‘valid’ and ‘ready’ be asserted together for exactly one cycle for each data token transferred between the sender and the receiver. To ensure this, CtoS will not create states inside the protocol region.

11.2.8.1 Using Protocol Regions in a Pipelined Loop

The valid-ready protocol discussed above could be broken if protocol is in a pipelined loop with multiple stall loops. There is no guarantee that ‘ready’ will be checked in every cycle, and there could be multiple cycles between the receiver setting ‘ready’ to 1 and the sender setting ‘valid’ to 0. Therefore, CtoS will report an error if you try to pipeline a loop, which contains both multiple stall loops and a protocol region declared without the **-stallable** option.

For example, this is a protocol inside a pipelined loop. The **pipeline_loop** command will add backpressure logic that will stall later iterations when ‘ready2’ is false. This will break the protocol because ‘valid’ is not set to zero immediately after ‘ready’ is true.

```
PIPELINE: while (1) {
...
BEGIN:
    valid.write(1);
    data.write(a);
    wait();
    // This while loop is added implicitly by the pipeline_loop command
    while (backpressure_from_ready2) {wait();}
    while (!ready.read()) wait();
    valid.write(0);
END:
...
    while (!ready2.read()) wait();
}
```

To resolve this problem, CtoS supports a design technique to make the protocols stallable. The **set_stall_value** command can be used to set a signal to a particular value when the loop is stalled.

The following commands will create stallable protocol region in a pipelined loop.

```
create_protocol_region -stallable [find -node BEGIN] \
                        [find -node END]
set_stall_value -value 0 [find -net valid]
pipeline_loop [find -node PIPELINE_while_begin]
```

In this example, the protocol would be maintained if ‘valid’ signal is set to **0** whenever the loop is stalled by an unrelated stall loop in the same pipelined loop. Once the **set_stall_value** command is applied to the ‘valid’ signal with *value* set to **0**, the protocol region can be safely declared with the **-stallable** option. The generated RTL will maintain the protocol even if the code appears inside a pipelined loop with stalls outside the protocol region.

The protocol region is shown below with the effect of the **pipeline_loop** and **set_stall_value** commands. The **pipeline_loop** will insert the backpressure stall loop and the **set_stall_value** command will add the write of valid to this loop, making the protocol stallable. Note that the **set_stall_value** introduces a combinational path from ‘ready2’ to ‘valid’ and that the write in the backpressure loop is temporary.

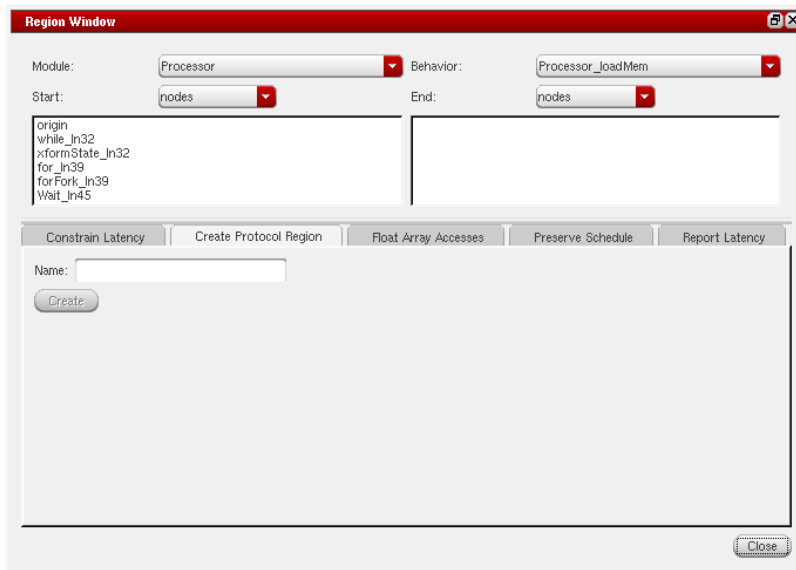
```
PIPELINE: while (1) {
...
  BEGIN:
    valid.write(1);
    data.write(a);
    wait();
    // The while loop is added implicitly by the pipeline_loop command.
    // The valid.write(0); is added implicitly by the set_stall_value command.
    while (backpressure_from_ready2) {valid.write(0); wait();}
    while (!ready.read()) wait();
    valid.write(0);
  END:
...
  while (!ready2.read()) wait();
}
```

Warning Do not add the **-stallable** option, unless you have verified that your protocol is indeed stallable. Otherwise, the generated RTL will not behave correctly.

To define the cycle behavior of I/O accesses as fixed, in the CtoS GUI, select **Edit -> Region -> Protocol Region**.

The **Region Window** dialog is displayed, with the **Create Protocol Region** tab selected, as shown in [Figure 11-9 on page 11-17](#).

Figure 11-9 Region Window Dialog, the Create Protocol Region Tab



Notes

- There must be no pipelined loops in the region to be defined. A protocol region *can be fully contained within* a pipelined loop; in this case, it cannot include any states [**wait()** statements] unless the protocol is stallable or the pipeline loop does not have any stall loops, and it cannot include the state insertion point of the pipeline (see “How to Pipeline a Loop” on page 8-32).
- You can also use the **create_protocol_region** command (“create_protocol_region” on page E-43).

11.2.9 Constraining ops to Edges

You may guide the scheduler to implement a particular op on an edge in the CDFG, using the **constrain_op** command with the **-edge** option (“constrain_op” on page E-38) with the following choices for the constraint:

- The constraint is a requirement, that is, a *hard* constraint.
- The constraint is just a “hint,” that is, a *soft* constraint.
- The constraint should not be shared with any other ops, that is, *restrict sharing*.

When a constraint is designated as *hard*, the attribute **is_respected** will always be *1 (true)*. If a constraint designated as *hard* cannot be respected due to a conflict (resource contention or negative slack), the scheduler will issue an error and fail.

The scheduler will attempt to respect a *soft* constraint, but if it cannot, it will silently bind it to a different edge; however, soft constraints will never be moved to an *earlier* edge. If the scheduler chooses to ignore a soft constraint, the attribute **is_respected** will be *false (0)* for any such constraint.

If an op is *not* within a pipelined loop, you can simply specify an edge on which to constrain the op.

The edge must meet the following criteria:

- The edge must belong to the span of the op.
- The edge cannot belong to a combinational behavior or a checking process.
- The edge cannot be a backward edge or a busy loop edge of a behavior call, or belong directly or indirectly (via a function call) to a clocked **SC_METHOD**.
- The op specified for the edge must not be an I/O op, part of the FSM control logic, or a join mux (the scheduling for such ops is fixed).

If an op *is* within a pipelined loop, you can specify stage and phase numbers on which to constrain the op. You must have already performed pipelining ([“**Pipelining Loops**” on page 8-26](#)), and the pipeline must have valid II and LI values, that is, the stage and phase numbers must satisfy:

$$1 \leq [\text{stage\_number}] \leq \text{ceil}(\text{LI}/\text{II}) \quad 1 \leq [\text{phase\_number}] \leq (\text{LI} \bmod \text{II}) + 1$$

If you pipeline a loop multiple times, the existing constraints will be checked for consistency, as follows:

- If the new pipelined loop has an initiation interval *different from* the previous pipelined loop, the stage and phase constraints of all ops from this pipelined loop are reset.
- If the new pipelined loop has the *same* initiation interval as the previous pipelined loop, but *different* latency intervals, the constraints are reset if they constrain a stage inconsistent with a new latency interval (that is, it exceeds the maximum stage number).

Note For preconditions for this command, see [“**constrain_op**” on page E-38](#).

11.2.10 Constraining ops to Expandable Edges

Constraining ops to regions in the CDFG that are *expandable* is a special case within CtoS.

Expandable edges (which includes pipeline stages/phases), by definition, have no fixed relationship to adjacent edges in the CDFG.

Therefore, trying to constrain an op (using the **constrain_op -edge** command) to an expandable edge will have no meaningful interpretation.

There are two types of expandable edges, depending on whether the pipeline feature is being used in the design, as follows:

- *a non-pipelined edge* that is expandable as a result of a specified latency constraint that includes this edge
- *a pipeline stage/phase* after the expansion point in a pipeline with a variable latency interval

CtoS will issue either a warning or an error due to this inconsistency between constraints and expandable edges, depending on the design flow:

- First scenario: An edge has been identified as one of the two types of expandable edges, and you have then constrained this edge to an op.

In this case, the edge on which the op is constrained will be marked as non-expandable, and you will get a warning.

- Second scenario: You have defined an op constraint and want to add a state, by either [“Creating Individual States” on page 11-9](#) or [“Constraining Latency” on page 11-12](#) through the specified edge.

In this case, CtoS will treat the edge as non-expandable after you have constrained it to the op; therefore, the edge may be excluded from the expandable region when you attempt to constrain its latency.

If you then try to create a state explicitly on this edge, CtoS will fail with an error.

11.3 Creating and Managing Resources

To create and manage resources, you have the following options:

- “Creating Initial Resources” on page 11-20
- “Creating Individual Resources” on page 11-21
- “Controlling addsub Resources” on page 11-23
- “Controlling Embedded RC Timing Engine” on page 11-23
- “Constraining ops to Resources” on page 11-23

11.3.1 Creating Initial Resources

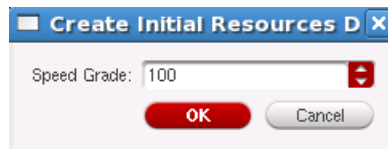
When array dependencies are known for particular behaviors, you can run the resource estimator on the specified behaviors to create the minimum number of shareable resources for use by the CtoS scheduler.

Depending on the setting of **Scheduling Effort**, this will either:

- allow complex ops to share the same resource, if they are mutually exclusive (box checked), or
- create a unique resource for every shareable op (box unchecked)

To create initial resources, in the CtoS GUI, select **Edit -> Guide Scheduler -> Create Initial Resources**. You will see the **Create Initial Resources** dialog, as shown in [Figure 11-10 on page 11-20](#).

Figure 11-10 Create Initial Resources Dialog



These settings affect the design attribute **default_speed_grade** (“[default_speed_grade](#)” on page D-14). To understand more about setting the **Speed Grade**, see “[Relaxed Latency Scheduling Mode](#)” on page -4.

Notes

- This will automatically create required states, if you have not already created them in the CtoS GUI or with the **create_required_states** command (“[create_required_states](#)” on page E-46).
- If you want to stop this process (and it will stop *after processing the current behavior*), select the **Interrupt** button (“[Interrupt Button](#)” on page 6-32). This will leave the design in the state of having some behaviors with initial resources and others without.
- You can also use the **create_initial_resources** command (“[create_initial_resources](#)” on page E-42).

11.3.2 Creating Individual Resources

You can create individual resources, for several purposes:

- to craft a custom resource to provide specific capabilities for how you want to implement your design
- to make another resource, part of the basic resource set, to bind to a particular operation in the source

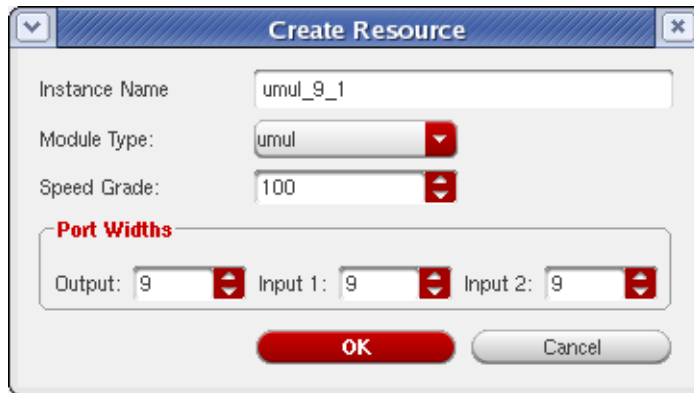
You can also create constant resources for values that are constant. This ensures faster and smaller resource implementation especially for FPGA design targets. And, also improves area and timing during scheduling. Additionally, you can share the constant resource among ops, if each op has an identical constant input value. However, constants that are larger than 64-bits cannot be implemented as const resources.

To create individual resources, in the CtoS GUI, first open the **Op Constraint Viewer (View -> Resources)**.

Right-click on a resource (use the **Expand** button to see the resources), and select **Create Resource**.

You will see the **Create Resource** dialog, as shown in [Figure 11-11 on page 11-21](#).

Figure 11-11 Create Resource Dialog



For supported resource types and required widths, see [Table 11-1 on page 11-22](#).

Notes

- Before you create an individual resource, the array dependencies must be known for the object specified.
- You can create an individual resource before or after creating initial resources.
- You can also use the **create_resource** command ([“create_resource” on page E-47](#)).

Table 11-1 Resource Types and Required Widths for Creating Resources

resource_type	widths	significant terminals	compatible op type
add	2	{Z, A}, B	add
add_const		A, K	add
sub	2	{Z, A}, B	sub
sub_const		A, K	sub
addsub	2	{Z, A}, {BADD, BSUB}	addsub
gtle	2	A, B	gt lt ge le
gt_const		A, K	gt lt
le_const		A, K	ge le
umod	2	A, B (Z = min(A,B))	mod
smod	2	A, B (Z = min(A,B))	mod
udiv	2	{Z, A}, B	div
sdiv	2	{Z, A}, B	div
umul	3	Z, A, B	mul
umul_const		A, B	mul
smul	3	Z, A, B	mul
smul_const		A, B	mul
lsh	3	Z, A, B	lsh
ursh	3	Z, A, B	rsh
srsh	3	Z, A, B	rsh
lrot	2	{Z, A}, B	lrot
rrot	2	{Z, A}, B	rrot

In this table:

- The {Z, A} format means the first width is for both Z and A.
- The second width is for B, and it must be smaller than A.

- **K** is the decimal value of the constant input.

11.3.3 Controlling addsub Resources

CtoS lets you have control over the use of *addsub* resources.

Some designs are highly constrained or have aggressive timing targets. For these designs, it is useful to specify that CtoS not use *addsub* resources, since they are marginally slower implementations for *add* and *sub*. There is a better chance of meeting timing and getting better *quality of results* (QoR) by not considering these resources. For FPGA based implementations, use of *addsub* resources is already off by default.

The use of the **enable_addsubs** behavior attribute controls whether the **create_initial_resources** and **schedule** commands use *addsub* resources.

You can manually create *addsubs* with the **create_resource** command; however, no ops in the design will be mapped onto any existing *addsub* resource.

If you have already created an op constraint (either soft or hard) to an *addsub* resource, before resource allocation, CtoS will disallow the disabling of this attribute.

Note See “Behavior Object Attributes (Behaviors)” on page D-41.

11.3.4 Controlling Embedded RC Timing Engine

Before generating RTL, CtoS uses RC to time all components in sequential paths. And, CtoS can specify RC commands that need to be applied after a design is read by RC, such as forcing a wireload model.

During the creating resources portion of the CtoS scheduler, a Tcl procedure hook can be called after RC elaboration, for example:

```
proc rc_post_elab_proc {module} {  
  puts "Setting wireload on $module."  
  set_attr force_wireload my_wire_load $module  
}
```

11.3.5 Constraining ops to Resources

You may guide the CtoS scheduler to implement a particular op with a particular resource by creating a resource *constraint*, using the **Op Constraint Viewer**, as shown in [Figure 11-12 on page 11-24](#).

If the constraint is successful, the scheduler creates a resource *binding*.

Some of the reasons to constrain a resource include:

- if you want to constrain an op to a resource, even though it is oversized (widths are larger than necessary), to facilitate the sharing of a resource
- if you want to constrain an op to a resource of a particular speed grade (slower speed grade means smaller area), based on the timing criticality of the resource

You have the option of designating that the constraint:

- be a requirement, that is, a *hard* constraint. In this case, the attribute **is_respected** will always be *true (1)*. If a constraint designated as *hard* cannot be respected due to a conflict (resource contention or negative slack), the scheduler will issue an error and fail.
- be just a “hint,” that is, a *soft* constraint. In this case, the scheduler will attempt to respect the constraint, but if it cannot, it will silently bind it to a different edge – however, soft constraints will never be moved to an *earlier* edge. If the scheduler chooses to ignore a soft constraint, the attribute **is_respected** will be *false (0)*.

Figure 11-12 Op Constraint Viewer

Op	Resource Type	add	add_c 1	sub_c 1	gtle
add ops	Expand/Collapse	Expand	Expand	Expand	
add_c 1 ops					
sub_c 1 ops					
le ops					
lt ops					
subfunc ops					

You have the option of designating that the resource:

- be used *exclusively* with the specified op, that is, restrict sharing with other ops, or
- be *shared* with other ops

The op must meet the following qualification:

- Custom ops transitively calling the specified behavior must not have a resource binding.

The resource instance must meet the following qualifications:

Note Unlike constraining an op to an edge (“[Constraining ops to Edges](#)” on page 11-17), whether the loop has been pipelined makes no difference when constraining an op to a resource – these same qualifications still apply.

- The resource instance must be associated with the same behavior as the specified operation.
- The resource instance must be able to implement the operation. See “[Resource Types and Required Widths for Creating Resources](#)” on page 11-22 for characteristic resource widths affecting legal constraints.
- If you specify both a resource instance and an edge, the resource instance must not have any constraints containing other edges that are combinationally reachable from the specified edge.
- The resource instance must not be a memory. Memory instances cannot be constrained, since the binding of a memory op to a memory instance is already fixed by CtoS.

Figure 11-13 Op Constraint Viewer Expanded

Op Constraints						
DUT_proc						
Op	Resource Type	add	add_c 1	sub_c 1	gtle	
Constraints	Expand/Collapse	Collapse	Collapse	Collapse	Collapse	
	Widths	32	32	5	32	32x6
Shareable Ops	\ Instance Name	(*)	(*)	(*)	(*)	(*)
add ops						
add_in44						
add_in64						
add_c 1 ops						
add_in41						
add_in62						
sub_c 1 ops						
sub_in66						
le ops						
lt_in40						
lt ops						
lt_in62						
subfunc ops						
subfunc_in45						

To use the **Op Constraint Viewer**, after you have built a design and specified the micro-architecture, select **View -> Resources** (if necessary, click *Yes* on the **Resource Viewer** dialog box) to see the **Op Constraint Viewer**, as shown in [Figure 11-12 on page 11-24](#).

Right-click anywhere in the viewer, and select **Expand All** from the context menu to see the **Op Constraint Viewer Expanded**, as shown in [Figure 11-13 on page 11-25](#).

To create a constraint between the op and the resource, double-click on any blue cell, and choose the type of constraint to create (**restrict_sharing**, **hard**, or **soft**).

Notes and Caveats

- If you are planning to inline a function with *custom ops*, you must do this *before* constraining the op. CtoS does not preserve *custom op* constraints during inlining (see [“Inlining Functions” on page 8-3](#)).
- If you are planning to unroll a loop, you must do this *before* constraining the op. CtoS does not preserve op constraints during inlining (see [“Unrolling Loops” on page 8-17](#)).
- For more preconditions, or to use the **constrain_op** command with the **-inst** option, see [“constrain_op” on page E-38](#).

11.4 Potential Sources of op span Scheduling Failures

This section lists potential problems that can occur during scheduling – with diagnostics pointing to op span failures – and provides some recommendations for how to prevent or resolve these problems.

The first three problems can occur during two of the steps in this chapter (“[Creating and Managing States](#)” and “[Creating and Managing Resources](#)”) or during scheduling (“[Scheduling](#)”). They can “slip” to the scheduling step because the problems are not always noticed in the previous steps.

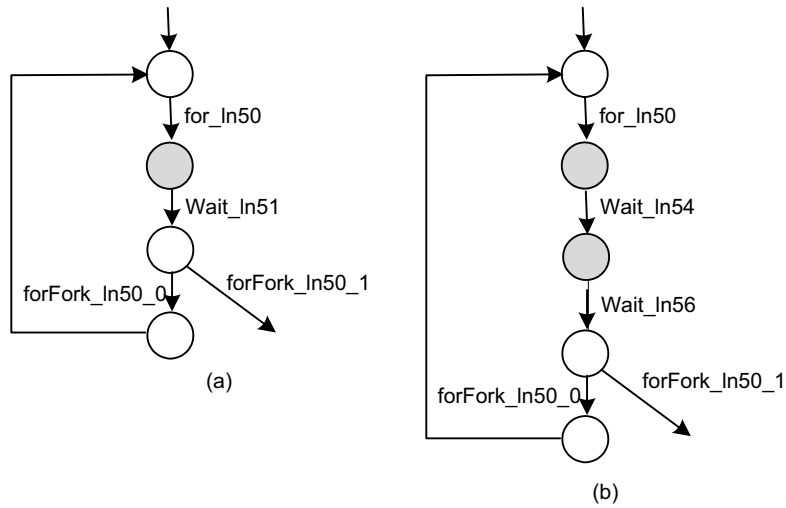
The last two problems, related to pipelining, can be seen only during scheduling.

Here are the problems discussed:

- “[Inconsistent Edge Constraints](#)” on page 11-29
- “[Sequential Distance Violations for Resources with Non-Zero Latency](#)” on page 11-29
- “[Memory Contention](#)” on page 11-31
- “[Pipelining Restrictions](#)” on page 11-31

The two CFGs in [Figure 11-14 on page 11-28](#) will be used to illustrate some of the examples.

Figure 11-14 Two CFGs for Examples of Source Code with op span Failures



11.4.1 Inconsistent Edge Constraints

The following example, “[Example 1](#)” on page 11-29, corresponds to CFG (a) in [Figure 11-14](#) on page 11-28, where shaded circles denote state nodes.

Example 1

```
for (int i = 0; i < numWords; i++) {  
    wait(); // line 51  
    sum += data1.read();  
    sum += data2.read();  
    out.write(sum);  
}
```

For this example, suppose you try to balance additions inside the loop by moving the first add op to the first edge of the loop with an op constraint:

```
constrain_op -edge edges/for_ln50 ops/add_ln52
```

During scheduling, an error would be issued regarding this constraint because it breaks data causality.

The first add op depends on the result of reading from input **data1**. However, all read operations are assumed to be fixed, and **data1.read()** happens only at edge **Wait_ln51**. Moving the first add op to an edge before the memory read violates the causality of the computation.

You could fix this failure by:

- constraining the read op to the first edge of the loop (see “[Constraining ops to Edges](#)” on page 11-17)
- dropping the op constraint on the add op
- changing your source code

11.4.2 Sequential Distance Violations for Resources with Non-Zero Latency

When a resource has non-zero latency (memory or RTL IP), it takes one or more cycles to get its results ready. If there are not enough states in the CFG before the results of the operation are to be used, the CtoS scheduler will fail.

This is illustrated by the following example, “[Example 2](#)” on page 11-30, in which array **M1** is allocated to regular synchronous memory (has latency equal to 1), while array **M2** is allocated to a registered memory with **read_latency** equal to 2.

Example 2

```
for (int i = 0; i < numWords; i++) { //ln_50
    sc_uint<8> mem1 = M1[i];
    delta += mem1;
    out1.write(delta);
    wait(); // ln_54
    sc_uint<8> mem2 = M2[addr.read()];
    wait(); // ln_56
    prod *= mem2;
    out2.write(prod);
}
```

The read from **M1** (**mem1**) cannot happen earlier than the first edge of the loop (value **i** must be known), and it is written to the output **out1** in the very same cycle (remember that write to the output is a fixed operation, and it cannot be moved from the edge where it is born, unless you specify it, for example with [“Floating I/O Accesses” on page 8-91](#)).

However, the result of reading from synchronous memory is available only in the next cycle. Therefore the op span computation for the memory read from **M1** will return an empty op span, and a corresponding diagnostic message.

A similar analysis for the read from **M2** shows that the earliest edge for the read is edge **Wait_In54** (because **addr.read()** is fixed on this edge) and the result of the memory read must be available at **Wait_In56** (because **out2.write()** is fixed on it).

However, it needs two states to make the result from the memory read available (the **read_latency** of this memory is 2), and the op span computation for the read from **M2** would also fail, returning an empty op span.

You could fix this failure by:

- inserting more states in the loop in the source code
- adding more states to the loop, as described in [“Creating Individual States” on page 11-9](#)
- specifying *Relaxed Latency Scheduling Mode* ([“Relaxed Latency Scheduling Mode” on page -4](#)). In this case, CtoS will properly insert states in edge **for_In50** and **wait_In54** to satisfy the sequential distance for memory operations.

Note For more on the rules of scheduling of multi-latency operations, see [“External Arrays” on page 9-22](#).

11.4.3 Memory Contention

Resource contention is resolved during scheduling by adding resources as needed.

However, this does not apply to *single-interface* memories.

Therefore, if too many operations are specified to happen at a particular place in the CFG, the op span computation will fail, as illustrated by the following example, “[Example 3](#)” on page 11-31, for synchronous memory **M**.

Example 3

```
for (int i = 0; i < numWords; i++) { //ln_50
    ...
    sum = M[i] + M[i+1];
    wait(); // ln_54
    out.write(sum);
    wait(); // ln_56
}
```

This example corresponds to CFG (b) in [Figure 11-14 on page 11-28](#). Memory reads get ready at the first edge of the loop and are used in the next cycle (at the edge **Wait_ln54**). As it is possible to schedule only one memory read in a single cycle, this specification is infeasible, which is confirmed by the op span failure, reporting the memory contention.

You could fix this failure by:

- allocating to memory with a *multiple-read* interface (“[Allocating Memory](#)” on page 9-2)
- adding more states to the loop, as described in “[Creating Individual States](#)” on page 11-9
- specifying *Relaxed Latency Scheduling Mode* (“[Relaxed Latency Scheduling Mode](#)” on page -4)

11.4.4 Pipelining Restrictions

Loop pipelining imposes additional restrictions on op spans. The major limitations come from:

- “[Strongly Connected Components \(SCC\) Scheduling Limitations](#)” on page 11-32
- “[Memory Operations Scheduling Limitations](#)” on page 11-33

Note See also “[Resolving Pipelining Scheduling Failures](#)” on page 8-42

11.4.4.1 Strongly Connected Components (SCC) Scheduling Limitations

Consider the task of pipelining the loop in the following example, “[Example 4](#)” on page 11-32, with initiation interval $II=1$.

Example 4

```
for (int i = 0; i < numWords; i++) { //loop_to_pipeline
    sum += data1.read();
    wait();
    sum += data2.read();
    out.write(sum);
    wait();
}
```

In this example, the computation of **sum** creates a computational cycle [an SCC (*strongly connected component*; see “[SCC](#)” on page N-14)], because the old value of **sum** is needed to compute the new one.

All SCCs in a pipelined loop must be scheduled within II states (see “[Ops of SCCs Must Be Scheduled within a Stage](#)” on page 8-42), which for this case of $II=1$ means all operations that compute **sum** must be scheduled within a single state.

This is clearly impossible, because **sum** depends on the values read from inputs **data1** and **data2**, which are fixed in consecutive states.

The op span computation will return an empty op span for add operations that are part of **sum** evaluation.

You could fix this failure by:

- changing your source code
- moving all **sum** computations into a single stage of the pipeline using op constraints (see “[Constraining ops to Expandable Edges](#)” on page 11-19)
- increasing the initiation interval (II) when pipelining the loop

11.4.4.2 Memory Operations Scheduling Limitations

Reads and write from the same memory must occur in the same stage of a pipeline because CtoS conservatively estimates that there is a dependency between any memory read and write performed in different iterations (see “Ops with same Memory (Implicit SCC) Must Be within Single Stage”).

This is illustrated by the following example, “Example 5” on page 11-33, which uses synchronous memory and a pipelined loop with $II=1$.

Example 5

```
for (int i = 0; i < numWords; i++) { //loop_to_pipeline
    update = M[i] + delta.read();
    M[i+3] = update
    wait();
}
```

There is no direct dependency between the memory read and memory write in a single iteration because reads and writes are performed at different memory addresses.

However, in pipelining, several loop iterations are executed simultaneously.

CtoS conservatively estimates that there is a potential dependency between the memory read and write in different iterations and demands these two be scheduled in the same stage.

This is impossible because the results of the memory read are available only in the next cycle, and therefore the memory read and write cannot be scheduled in a single pipeline stage for $II=1$.

Op span computation would thus fail.

You could fix this failure by:

- changing your source code
- increasing the initiation interval (II) when pipelining the loop
- manually scheduling the memory read and memory write inside the pipelined loop (using op constraints). In this case, CtoS will ignore the inter-iteration dependencies.

a. Scheduling and Managing Registers

This chapter describes the Scheduling and Managing Registers steps of the CtoS flow:

- [“Scheduling” on page -2](#)
- [“Results of Scheduling” on page -17](#)
- [“Managing Registers” on page -18](#)

Before proceeding with this chapter, you might consider the following:

- If you want to create or manage array dependencies, states or resources during your scheduling process, see [“Advanced Features for Guiding the CtoS Scheduler” on page 11-1](#)
- For an example of how to work through scheduling messages resulting from tight constraints, see:

`install_directory/share/ctos/examples/analysis/scheduling`

Note Before running any CtoS examples, first review [“Setup for Examples” on page F-2](#).

12.1 Scheduling

When all nets are properly driven, all array dependencies are known, and all resources have been created for a design, module, or behavior, then that object is *schedulable*, and you can use the CtoS scheduler to *schedule* that object.

The CtoS scheduler maps ops to resources and binds those resources to states. By default, the CtoS scheduler does *not add states*, nor does it *move I/Os or shared memories*; however, there are options that let you instruct the scheduler to add states and provide more flexibility in moving certain objects.

In the CtoS GUI, you schedule your design by selecting **Edit -> Schedule**, which then displays the **Schedule** dialog, as shown in [Figure 12-1 on page -2](#).

During scheduling, CtoS uses the following information, which you may have already specified earlier in your design flow:

- timing information specified during the Set Up Design Step, or with the **define_clock** command (“[define_clock](#)” on page E-53) and **external_delay** command (“[external_delay](#)” on page E-58).
- RC information specified during the Set Up Design Step: the Technology Library, RC Startup Script, and Working Directory [if you are using Encounter RTL Compiler (RC), that is, you have selected **Implementation Aware**.

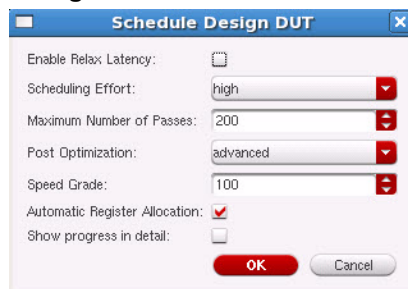
Note See also “[Creating a New Design](#)” on page 6-9.

If CtoS succeeds in creating a feasible schedule, it will calculate the op spans of each operation, which you can examine using the report commands in “[Analyzing and Implementing Designs](#)” on page 13-1.

Important Notes

- The CtoS scheduler works only on threads and sequential functions called from threads, and it does nothing to clocked methods and combinational methods.
- If, at any time, you want to stop the scheduling process, select the **Interrupt** button (“[Interrupt Button](#)” on page 6-32). The CtoS scheduler stops at the end of the current pass. The design state is *Failed Schedule*, and you can view the failed ops in the CDFG viewer.
- You can also use the **schedule** command (“[schedule](#)” on page E-138).

Figure 12-1 Schedule Dialog



In the **Schedule** dialog, you guide the CtoS scheduler using the following options:

- **Enable Relax Latency:** Lets you enable *relaxed latency scheduling mode*, which lets the scheduler add states as long as they do not violate any latency constraints, protocol definitions, etc., that you have already explicitly defined. See [“Relaxed Latency Scheduling Mode” on page -4](#).
 - **Scheduling Effort:** Specifies the number of phases the scheduler should perform. By default, this value will be the value that you have set for the **default_scheduling_effort** design attribute during setup. You can specify any of the following options:
 - **high** performs all phases of the multiple-phase scheduler: Memory, Timing, and Sharing Feasibility and Final Phases.
 - **medium** performs Memory, Timing and Sharing Feasibility.
 - **low** performs Memory Feasibility phase. It also performs the Timing Feasibility phase if technology libraries are specified. See [“default_scheduling_effort” on page D-14](#) and [“Multi-Phase Scheduler” on page -11](#).
 - **Maximum Number of Passes:** Specifies the maximum number of scheduling passes you want to allow. The default is 200.
 - **Post Optimization:** Specifies the post-schedule optimization level:
 - **advanced** (the default) runs all optimizations.
 - **simple** skips the *Unsharing* optimization.
 - **none** skips all optimizations.
 - **Speed Grade:** Lets you set the speed grade. A speed grade of 100 (the default) forces CtoS to utilize the fastest possible, but area-expensive, resources to implement the given operation. A speed grade of 0 instructs CtoS to choose the slowest implementation, with probably the smallest area requirement.
- Note** See also the following section, [“Relaxed Latency Scheduling Mode” on page -4](#).
- **Automatic Register Allocation:** Allocates registers automatically, that is, registers are allocated for all processes and non-inlined sequential functions, and a netlist is automatically generated. To have more options during this process, use the **allocate_registers** command, instead.
- Note** See also the following section, [“Allocating Registers” on page -20](#).
- **Show progress in detail:** Enables verbose mode, in which the CtoS scheduler will display each op scheduled with negative slack, and each op that cannot be scheduled, with an explanation.

The following sections describe these Scheduling options, as well as other choices, in more detail:

- [“Timing Analysis” on page -4](#)
- [“Relaxed Latency Scheduling Mode” on page -4](#)
- [“Scheduling Restrictions for Multi-Latency Operations” on page -8](#)
- [“Multi-Phase Scheduler” on page -11](#)
- [“Controlling Speed Grades” on page -14](#)

- “Scheduling with op Delays Larger than Clock Cycle” on page -16
- “Scheduling for Low Power Design” on page -16

12.1.1 Timing Analysis

CtoS performs timing analysis during scheduling and generating timing reports. For timing results to correlate with logic synthesis, the same technology libraries should be specified to CtoS. You can specify technology libraries in the **Create New Design Wizard** or **Design Properties** dialog, or directly sets the design attribute “`tech_lib_names`” on page D-24.

CtoS automatically calls the Cadence Encounter RTL Compiler (RC) when you are scheduling and analyzing a design, using RC for logic synthesis of the individual resources.

Here is details of timing analysis:

- For *evaluation of area and max delay*, it uses RC.
- For *wire load models*, it employs a very basic wire load model, which uses the $\log_{10}$ number of loads times the delay of an inverter.
- For *static timing analysis*, it uses a single max delay value to model the delay from each input pin to the output of each resource, with the exception of mux, select, replace, and sequential resources. For mux, select, and replace resources, it uses control delays for C to Z pins and data delays for A to Z pins. For sequential devices, it uses setup delays for that specify constraints on the latest arrival time of sequential resource data input to clock edge and launch delays for the time when outputs of a sequential resource get asserted after the clock edge.

Note See the RC documentation for more detail about the different wire load models RC uses for each type of timing analysis.

12.1.2 Relaxed Latency Scheduling Mode

Relaxed latency scheduling mode is a preliminary feature.

In *relaxed latency scheduling mode*, which is essentially a *last resort* for the CtoS scheduler, states will be added for the following reasons:

- to avoid scheduling with negative slack
- to avoid contention among memory accesses

It is important to note, however, that the scheduler will not add states when limited by certain user directives or design constraints.

Details for these limitations and additional information about *relaxed latency scheduling mode* are described in the following sections:

- [“User Directives/Design Constraints that Override Relaxed Latency” on page -5](#)
- [“Maintaining Design Interface Protocols Using Protocol Regions” on page -6](#)
- [“Understanding Why Schedule Is Still Infeasible” on page -6](#)
- [“Understanding Why Scheduled Design Still Has Empty States” on page -7](#)
- [“Setting Schedule Slack Margin” on page -7](#)
- [“Additional Notes on Relaxed Latency Scheduling Mode” on page -8](#)

12.1.2.1 User Directives/Design Constraints that Override Relaxed Latency

In relaxed latency scheduling mode, the CtoS scheduler will automatically add states wherever needed to achieve positive slack (timing), except where limited by the following user directives and design constraints:

- Read/Write operations for shared memories default to being unmovable from their original state.
- Pipelined loop restrictions take precedence – more specifically, the rules for state insertion in a pipelined loop *do not change* in relaxed latency scheduling mode.

The rules for state insertion are still determined by the minimum and maximum latency interval and are done only at the points that you specify in the **Pipeline Loop** dialog ([“How to Pipeline a Loop” on page 8-32](#)) or with the **pipeline_loop** command ([“pipeline_loop” on page E-92](#)).

Note The **Pipeline Loop** dialog has areas clearly marked for setting the **Latency Interval** and **Additional State Insertion Location**. For the **pipeline_loop** command, minimum and maximum latency are set using the **-min_lat_interval** and **-max_lat_interval** options, and insertion points are specified using the **-expand_after**, **-expand_before** and **-expand_at** options or labels in the code.

- Latency constraints take precedence.

Note Latency constraints are set using the **Constrain Latency** tab in the **Region Window** or with the **constrain_latency** command (see [“Constraining Latency” on page 11-12](#)).

- Op constraints may lock an op to a specific state.

Note Op constraints are set using the **Op Constraint Viewer** or with the `constrain_op` command (see [“Constraining ops to Resources” on page 11-23](#)).

- States will not be added within protocol regions.

Note Protocol regions are set using the **Create Protocol Region** tab in the **Region Window** or with the `create_protocol_region` command (see [“Creating Protocol Regions” on page 11-14](#)).

12.1.2.2 Maintaining Design Interface Protocols Using Protocol Regions

Before using relaxed latency scheduling mode, if you want to maintain protocols at the input and output design interfaces, you should define protocol regions for all of your I/O accesses.

If you do not define protocol regions, the CtoS scheduler may insert states between I/O accesses, which may inadvertently lead to simulation mismatches if the testbench requires a particular timing protocol.

Note Protocol regions are set using the **Create Protocol Region** tab in the **Region Window** or with the `create_protocol_region` command (see [“Creating Protocol Regions” on page 11-14](#)).

12.1.2.3 Understanding Why Schedule Is Still Infeasible

When using relaxed latency scheduling mode, scheduling may *still* be infeasible, due to the fact that a design has been over-constrained by one or more of these actions:

- setting a protocol region (see also [“Creating Protocol Regions” on page 11-14](#))

Important Although it *can* create an over-constrained design, it *is* recommended that you set a protocol region if you need to create such a region (see the previous section, [“Maintaining Design Interface Protocols Using Protocol Regions” on page -6](#)).

- pipelining loops (see also [“Pipelining Loops” on page 8-26](#)):
 - setting a maximum **LI** (latency interval) for a pipelined loop
 - specifying an incorrect state insertion point for a pipelined loop, where states do not help scheduling the loop
 - pipelining a loop with an SCC (*strongly connected component*; see [“SCC” on page N-14](#)) that cannot complete in one **II** (initiation interval)

- constraining an op (see also [“Creating and Managing States” on page 11-6](#))
- constraining latency (see also [“Constraining Latency” on page 11-12](#))
- preserving a region (see also [“Constraining ops to Edges” on page 11-17](#))

12.1.2.4 Understanding Why Scheduled Design Still Has Empty States

After scheduling in relaxed latency scheduling mode, you may still see empty states, for the following reasons:

- if the states are user-defined
- when a multi-cycle op exists in a behavior
- in pipelined designs:
 - in the last state of the pipelined loop
 - when the Π (initiation interval) is greater than 1

This is due to the fact that in pipelines with $\Pi > 1$, ops in different pipeline phases can share the same resource. Removing an empty state could change the phases of forthcoming states, thus violating the sharing decisions.

- if you specify a minimum latency greater than the number of states needed to schedule the loop

12.1.2.5 Setting Schedule Slack Margin

For better *QoR*, some negative slack may be acceptable – and actually preferable – because the RTL Compiler (RC) can close the timing.

The `schedule_slack_margin` design attribute ([“schedule_slack_margin” on page D-23](#)) lets you set your acceptable *slack margin* as a percentage of the clock cycle, from **0** to **100**. This attribute applies only to paths with negative slack.

Here is the suggested methodology for using the **schedule_slack_margin** design attribute:

1. For your first scheduling run, use the default of **0** for **schedule_slack_margin**.

Note To set an attribute, use the **set_attr** command (“[set_attr](#)” on page E-140).

2. On successive runs, experiment with larger slack margins to reduce states (in relaxed latency scheduling mode) and area (in all scheduling modes). Better QoR may be realized because the scheduler will *not* try all corrective actions (including adding resources). However, note that:
 - Scheduling may still complete with negative slack, even with a large slack margin, because the scheduler may run out of corrective actions.
 - It is recommended that you try increasing slack margin to reduce area before changing speed grades or constraining ops.
3. Keep increasing the slack margin, and rerun, until RC completes with negative slack.

12.1.2.6 Additional Notes on Relaxed Latency Scheduling Mode

Here are a few more notes on relaxed latency scheduling mode:

- When you select relaxed latency scheduling mode, the behavior object attribute, **relax_latency**, is set to *true* (the default is *false*). See “[relax_latency](#)” on page D-43 for more on this behavior attribute.
- You can also select relaxed latency scheduling mode when you are creating required states. See “[Creating Required States](#)” on page 11-8 for more information.

12.1.3 Scheduling Restrictions for Multi-Latency Operations

Scheduling of multi-latency operations (RTL IPs or memory reads/writes including registered memories) must satisfy the following restriction: If an operation **O** has latency **n**, it must be scheduled not closer than **n** states before reaching a forward fork or join node in the CFG.

The only exception to this rule concerns the use of stall loops in the CFG: If an operation **O** is specified with a stall pin, the stall loop is ignored when counting the latency to the closest fork or join node.

The following two examples illustrate this scheduling rule for the case of a registered memory **M** with read latency 3 and a stall pin.

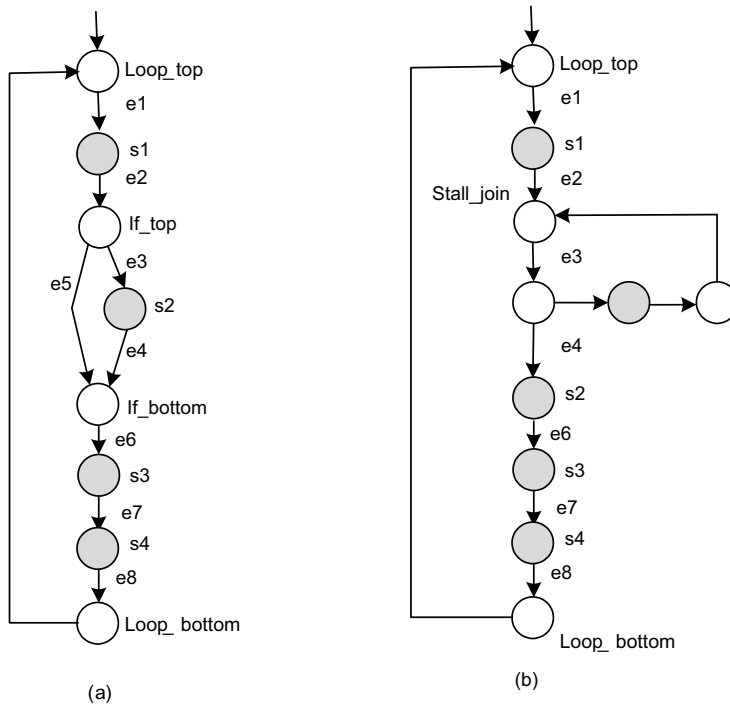
Example 1

```
void interpolation1::thread() {
    wait();
    sc_uint<16> aver_sum = 0;
    for (int i = 0; i < numWords; i++) {
        sc_uint<8> mem = M[i];
        wait(); // s1
    }
}
```

```
        if (mem * scale.read() > threshold) {
            mem = threshold;
            wait(); // s2
        }
        mem *= coef.read();
        wait(); //s3
        wait(); //s4
        out.write(mem);
        aver_sum += mem;
        aver_sum /= i;
        res.write(aver_sum);
    }
}
```

Example 2

```
void interpolation2::thread() {
    wait();
    sc_uint<16> aver_sum = 0;
    for (int i = 0; i < numWords; i++) {
        sc_uint<8> mem = M[i];
        wait(); // s1
        while (valid_out.read() == false) { // stall loop
            wait();
        }
        mem *= coef.read();
        wait(); //s2
        wait(); //s3
        wait(); //s4
        out.write(mem);
        aver_sum += mem;
        aver_sum /= i;
        res.write(aver_sum);
    }
}
```

Figure 12-2 Multi-Latency Scheduling Restrictions

In “[Example 1](#)” on page -8, the **memread** can happen as early as the first edge of loop **e1**, as this is the first operation done in the loop. However, scheduling will fail because, in the CFG diagram (a), in [Figure 12-2 on page -10](#), there is no linear fragment to satisfy the restriction that **M** have latency=3. Indeed, edge **e1** has only a single state before forward reaching the fork node **If_top**, while edge **e6** has only two states (**s3**, **s4**) before reaching the join node **Loop_top**.

In “[Example 2](#)” on page -9, the **memread** can also happen as early as the first edge of loop **e1**. As memory **M** is specified with a stall pin, in analyzing available linear fragments in the CFG diagram (b), in [Figure 12-2 on page -10](#), the stall loop should be ignored. In this case, three edges {**e1**, **e2**, **e4**} satisfy the restriction that latency=3 for memory **M**. The read from memory **M** could be scheduled in any of these three edges.

12.1.4 Multi-Phase Scheduler

CtoS supports multi-phase scheduler where scheduling problem progresses in complexity when going from one phase to the next one. The scheduler phases are:

- Memory Feasibility
- Timing Feasibility
- Sharing Feasibility
- Final Schedule

The multi-phase scheduler provides the following:

- Improves the clarity of failure reporting to the user because phase separation lets you identify failure diagnostics more precisely.
- Reduces the scheduling time significantly in case of a failure. For example, if the design is over constrained around memories, running the first phase (Memory Feasibility) would be sufficient to identify it.

By default, all phases are run and the scheduling stops at the first phase that fails. The number of phases to be performed is controlled by setting the **default_scheduling_effort** design attribute. (“[default_scheduling_effort](#)” on page D-14). The scheduling effort is set on the **Schedule** dialog.

Since the Sharing Feasibility and Final phases do not improve the timing, you may want the scheduler to stop after Timing Feasibility when the scheduler reports large negative slack. The **max_negative_slack** option of the **schedule** command lets you specify the tolerance for starting the sharing phase as percentage of clock cycle. The default value is 100 (that is, a clock cycle).

- If negative slack < max_negative_slack, a warning is displayed and the scheduling stops with a valid RTL without sharing.
- If the negative slack is between 0 and max_negative_slack, a warning is displayed but the scheduling continues to the sharing phase.

For more information on the **max_negative_slack** option, see “[schedule](#)” on page E-138.

Note In incremental synthesis design flow, you must set the same scheduling effort for both the baseline and the new design (see “[Incremental Synthesis](#)” on page 17-1).

12.1.4.1 Memory Feasibility

In the Memory Feasibility phase, the scheduler tries to answer the question whether the design can be scheduled considering only memory contention without optimizing for timing and area. The scheduler in this phase assumes all resources have zero delay but does consider the contention of accessing a single memory interface. For more information see [“Memory Contention” on page 11-31](#).

In this phase, the work of the CtoS scheduler is greatly simplified and typically it completes in a single pass.

If the CtoS scheduler fails to find a feasible schedule in this phase there is a memory contention problem which must be resolved to get a feasible schedule. You can fix the design by adding states or changing the source code or memory architecture.

The design with memory contention problems will not succeed when considering timing or area. Thus, the multi-phase scheduler will fail faster because if the design is not memory feasible, it will not continue to subsequent phases.

You can also explicitly specify to only run the memory feasibility phase by setting **scheduling effort** to *low* and not providing technology libraries. This is useful on large designs to more quickly determine if a design is possibly unschedulable due to memory contention.

12.1.4.2 Timing Feasibility

In the Timing Feasibility phase, the scheduler tries to answer the question whether the design can be scheduled considering only timing constraints without optimizing for area. The scheduler in this phase assumes that every operation is implemented by its own dedicated resource. The scheduler does not consider sharing of resources between several operations of the same type. For example, in the following code, each addition would be bound to a separate adder instance.

```
wait();  
if (n) {x=a+b;} else {y=c+d;}  
wait();
```

In this phase, the work of the CtoS scheduler is greatly simplified and is typically faster (will have fewer passes).

If the CtoS scheduler fails to find a feasible schedule in this phase there is a timing problem which must be resolved to get a feasible schedule. This design will definitely not close timing when considering sharing because resource sharing introduces additional timing penalty. Thus, the multi-phase scheduler will fail faster if the design is not timing feasible because it will not continue to subsequent phases.

You can also explicitly specify to only run the timing feasibility phase by setting **scheduling effort** design attribute to *low* and providing technology libraries. This is useful on large designs to more quickly determine if a design is possibly unschedulable due to timing problems.

12.1.4.3 Sharing Feasibility

In Sharing Feasibility phase, the scheduler considers mapping multiple operations on the same resource that implements them and models the timing impact of the sharing multiplexer. The timing of this design closely matches the timing considering sharing. While using sharing saves area, the additional delay of the multiplexer may cause timing problems.

This phase is faster than final scheduling phase because it is not mapping to set of shared resources. It only needs to model the timing of the sharing multiplier. The only difference to the Final Phase is that timing penalties from false paths are ignored.

If the CtoS scheduler fails to find a feasible schedule in this phase then the design will definitely not succeed when considering sharing. Thus, the multi-phase scheduler will fail faster if the design is not sharing feasible and will not continue to Final Schedule phase.

You can also explicitly specify to run the sharing feasibility phase by setting **scheduling_effort** design attribute to *medium* and providing technology libraries. This is useful for timing critical designs to more quickly determine if a design is possibly unschedulable due to timing problems when sharing is considered.

If there is a timing problem due to sharing, you can direct the scheduler to use a dedicated resource for a given operation. Set the attribute **use_dedicated_resource** to *true* on the op.

12.1.4.4 Final Schedule

In Final phase, the scheduler considers mapping multiple operations on the same resource that implements them. Sharing potentially providing better quality of results (QoR), because it uses less area. For example, in the following code, the two additions can *share* the same adder instance, because they are *mutually exclusive*.

```
wait();  
if (n) {x=a+b;} else {y=c+d;}  
wait();
```

By default, the scheduler runs all phases including the Final schedule. This is specified with **scheduling_effort** design attribute set to *high*.

If there is a timing problem due to false paths, you can direct the scheduler to use a dedicated resource for a given operation. Set the attribute **use_dedicated_resource** to *true* on the op.

12.1.4.5 Recommended Scheduling Effort

For some behaviors, the scheduler will advise you to rerun with a different **scheduling_effort** (for a particular behavior) if QOR is estimated to be better.

For more detail, the scheduler outputs QOR estimates for each level of scheduling effort so that you can decide if the QOR improvement is worth the time of rerunning the scheduler. The estimates include area and timing numbers as well as the latencies of pipelined loops in the behavior.

The area is normalized to user-specified effort level. Note that a recommendation is not provided when the user-specified effort level has the best estimated QOR.

Here is an example of message advising user (who selected effort as **high**) to rerun with effort **medium** because the area is slightly smaller.

Post-schedule QoR Estimates:

Metric	Scheduling Effort		
	low	medium	high
Area (%)	100.02	99.78	100.00
Slack (ps)	> 0	> 0	> 0
Latency (foreverTop_ln23)	4	4	4

For better QoR, consider rerunning with `scheduling_effort` set to 'medium' for behavior `always_block`.

There will be other scenarios, where advice is given because timing is met with lower effort.

12.1.5 Controlling Speed Grades

For an example of this feature, look in the following directory:

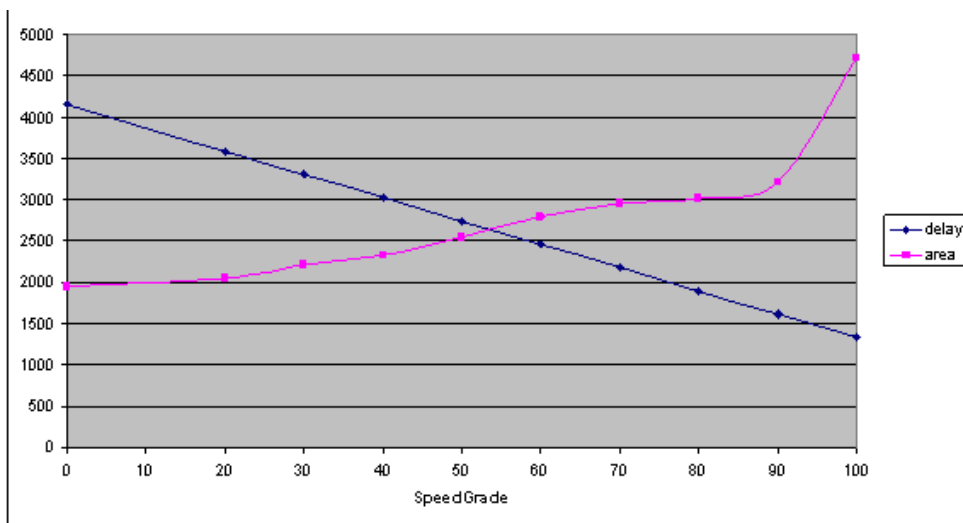
```
install_directory/share/ctos/examples/features/schedule/speedgrade
```

Note Before running any CtoS examples, first review “[Setup for Examples](#)” on page F-2.

When scheduling, CtoS will, by default, use the fastest (and largest) resources to implement a given operation. This helps in closing timing; but, sometimes for designs that are not timing critical, it is better to direct CtoS to use slower (and smaller) resources, instead. This is especially true for area-critical designs with large positive slack.

With *variable speed grades*, CtoS provides a way to trade-off area with respect to timing of individual resources. A speed grade of 100 (the default) forces CtoS to utilize the fastest possible resources to implement the given operation, but of course these resources will be area-expensive. Conversely, a speed grade of 0 instructs CtoS to choose the slowest implementation, which would probably result in the smallest area requirement.

The graph shown in [Figure 12-3 on page -15](#) illustrates this trade-off curve for a custom operator consisting of several primitive operations. The inverse relationship between delay and area is clearly visible. It is important to note that area is not a linear function, which makes the best speed grade value highly design- and technology-dependent.

Figure 12-3 Speed Grade Trade-Off

The selection of the speed grade, as well as the potential creation of resources with specific individual speed grades, can have a vast impact on the quality of results (QoR) of a design. The CtoS scheduler takes actions based on available slack within cycles and considers the area cost of a resource when resolving scheduling conflicts. As result, CtoS might create a very different schedule, given that a resource is large but fast, versus a smaller resource with larger delay.

Altering the speed grade lets you fine-tune the QoR after the overall micro-architecture has been chosen. CtoS will potentially choose an alternative schedule based on the speed grade you select and the available freedom within the design.

Any change of speed grade *after scheduling has completed* [for example, by the **analyze** command (“analyze” on page E-20)] has *no* effect on the RTL. For a given resource and schedule, with any speed grade, the RTL is always exactly the same. What changes is the target clock cycle used when RC synthesizes the resource, which affects both the timing and area estimates returned by RC and used by the area and timing analyzers. This target clock cycle tries to reflect the budget of this resource in the final module, that is, the percentage of the total clock cycle for the module that it will use.

For example, for speed grade 100, RC will use a 0 clock cycle, providing the fastest available implementation and the largest area. For speed grade 0, RC will use an infinite clock cycle, providing the smallest and slowest implementation. Speed grades in between are obtained by interpolation. But there is no change in the resource and hence in the RTL.

On the other hand, selecting a different resource grade *before scheduling* [for example, using the **create_resource** command (“create_resource” on page E-47) or changing the **default_speed_grade** design attribute (“default_speed_grade” on page D-14)] *does* have an effect on the schedule, and hence on the RTL.

Summary of Speed Grade Use

- To control the speed grade of a *specific resource*, use the **create_resource** command with the **-speed_grade** option ([“create_resource” on page E-47](#)).
- To see the *speed grade per resource*, use the **report_resources** command with the **-detail** option ([“report_resources” on page E-119](#)).
- To set the best speed grades on *instances that do not make the slack (more) negative*, use the **analyze** command ([“analyze” on page E-20](#)).
- To set the speed grade for an *entire design*, use the **set_attr** command ([“set_attr” on page E-140](#)) to set the **default_speed_grade** design attribute ([“default_speed_grade” on page D-14](#)).

12.1.6 Scheduling with op Delays Larger than Clock Cycle

When an operation requires a resource whose delay is larger than a clock cycle, you can set the **allow_op_delays_exceeding_clock_period** design attribute ([“allow_op_delays_exceeding_clock_period” on page D-11](#)) to *true*, to prevent scheduling from exiting with an error.

Scheduling will continue; but you will still get one of two different warnings, depending on whether the operation is constrained to an edge.

Warning Setting the **allow_op_delays_exceeding_clock_period** design attribute to *true* has no effect on scheduling in *low power mode* (see [“Scheduling for Low Power Design” on page -16](#)).

12.1.7 Scheduling for Low Power Design

Power Estimation is a preliminary feature.

The CtoS scheduler has enhanced techniques to enable *Scheduling for Low Power Design*, using the **-low_power** option to the **schedule** command ([“schedule” on page E-138](#)).

With this feature, the scheduler performs *pre-scheduling multi-cycle timing analysis* and evaluates the *criticality of all operations*. Then, based on the results, the scheduler chooses the best set of resources in terms of delay, area, and power trade-offs to implement the given set of operations. This analysis is redone on a regular basis during the scheduling process.

The power trade-offs enable the criticality analysis to be aware of scheduler decisions, such as sharing resources, accurate slack of already scheduled operations, etc. This helps to keep a consistent view between the scheduler and the multi-cycle timing analysis.

Expectations of these techniques are twofold:

- Non-critical operations are to be implemented on more efficient (but slower) resources, which in general helps to achieve smaller area.
- The negative slack (if any) becomes more balanced between scheduled states.

12.2 Results of Scheduling

After the CtoS scheduler has finished, there are three possible outcomes:

- “Scheduling Completes with Positive Slack” on page -17
- “Scheduling completes with negative slack” on page -17
- “Scheduling Fails with Failed Resource Bindings” on page -18

Note See also “Resolving Pipelining Scheduling Failures” on page 8-42 and “Potential Sources of op span Scheduling Failures” on page 11-27

12.2.1 Scheduling Completes with Positive Slack

When the CtoS scheduler completes with *positive slack*, you are then ready to analyze and implement your design. You may want to look at the results of both timing and area (see “Timing Report” on page 13-16 and “Reporting Power and Area Using the Tree Map” on page 13-29).

You are now also ready to implement your design (see “Generating Models, Wrappers, RTL, SLEC after Scheduling” on page 13-36).

12.2.2 Scheduling completes with negative slack

The default behavior of the CtoS scheduler is to schedule with *negative slack* as a last resort. In other words, when the scheduler runs out of actions, it will overrun the negative slack for the operations that cannot be pushed to later edges. In this case, note the following:

- If the scheduler finishes with negative slack, CtoS will run a few more passes to balance the slack, that is, it will distribute the negative slack more evenly across the edges of its op span. If this situation arises, you will get an information message (INFO) stating the following:

```
Running additional $(num_passes) passes to improve negative slack.
```

Note This does not apply to Low Power flows.

- The CtoS scheduler does not fail when an implementation does not satisfy clock constraints, that is, at the end of scheduling, it returns a successful result. To distinguish between scheduling results *with* or *without* clock violations, the CtoS scheduler sets the slack parameter for the scheduled behavior. You can access this parameter at the Tcl level:

```
get_attr slack $behavior
```

- To improve the diagnostic at the end of scheduling, the set of primary failures is printed in the log and trace files (*primary* failures do not depend on other failed ops). When timing violations are allowed, the only expected primary failures should be due to array dependencies and memory contention.

- Scheduling with negative slack may produce non-optimal (in terms of timing) results for pipelined designs.

Note To analyze this situation, you may want to review the timing report (see [“Timing Report” on page 13-16](#)) and the **Cycle Analysis** viewer (see [“Cycle Analysis Viewer” on page 13-20](#)).

12.2.3 Scheduling Fails with Failed Resource Bindings

The CtoS scheduler may sometimes fail due to failed resource bindings.

Note See [“Using the Resource Bindings Viewer after a Failed Schedule” on page 13-11](#) for more information.

12.3 Managing Registers

In the Manage Registers Step of CtoS, you have the following options:

- [“Creating Specific Registers” on page -18](#)
- [“Binding a Value” on page -18](#)
- [“Resetting Registers” on page -19](#)
- [“Allocating Registers” on page -20](#)
- [“Connecting Terminals to Add Power or Test” on page -20](#)
- [“Reporting Registers \(Primary and Secondary\)” on page -21](#)

12.3.1 Creating Specific Registers

To create a register with a specific name and width, you can use the **create_register** command ([“create_register” on page E-45](#)).

Then, you can use the **bind_value** command to bind values to this register, which is described more fully in the next section, [“Binding a Value” on page -18](#).

12.3.2 Binding a Value

After a behavior containing a value is fully scheduled, you can use the **bind_value** command ([“bind_value” on page E-26](#)) to create an appropriate value-to-register binding.

You can bind the value to a register you have created using the **create_register** command (which is described in the previous section, “[Creating Specific Registers](#)” on page -18), or CtoS will automatically create a new register resource for it.

Here are some of your options, and related conditions, when using the **bind_value** command:

- If you do specify the register, it must be available for each use.
- If you specify a state, a register binding must exist for the value in that state.
- If you do not specify a state, the default is every state in which a register is needed for that value.
- You may specify an index, but if you do not, the default is 0.
- All of the objects you specify must be from same behavior.

12.3.3 Resetting Registers

Some test methodologies require *all registers to be reset*. However, since adding reset logic increases the size of a design, CtoS does not, by default, reset registers for a design.

If you would like to reset registers, you can use the **set_attr** command (“[set_attr](#)” on page E-140) to set the **reset_registers** design attribute to *true* (“[reset_registers](#)” on page D-23), or you can select **Reset Registers** (“[Reset Registers](#)” on page 6-15) in the **Create New Design Wizard**, when you are setting up a design in the CtoS GUI.

When this attribute is set to **internal** resets are added to internal registers to make the design more testable, but it does *not* change the functionality of the design or of the CtoS-generated RTL. The reset state of these internal registers is observable only in test mode.

When this attribute is set to **internal** and there are **sc_signals** that are not reset, you will receive a warning.

Sometimes there are **sc_signals** that are not reset by the SystemC source. When the **reset_registers** attribute is set to **internal_and_outputs** the internal registers and the registers for **sc_signals** are reset. However, this may cause differences in behavior between the SystemC and synthesis simulation after reset (Such **sc_signal** will have the pre-reset value in SystemC simulation and the reset value in the synthesis simulation). The user needs to review these signals and ensure that resetting these signals is acceptable.

The **check_design** command (“[check_design](#)” on page E-32) will also warn about these registers and recommend that you modify your source code to reset these variables.

Notes on resetting registers

- If a process has multiple resets (no matter whether they are *asynchronous* or *synchronous*), only the *primary* reset (the highest priority reset) is used for the additional registers reset when this attribute is **internal** or **internal_and_outputs**.

- Registers/RAMs belonging to *single state processes* that do *not* have reset are *not* affected by this attribute.
- Built-in RAMS *are* affected by this attribute, but Vendor RAMs are *not*.

12.3.4 Allocating Registers

When a design, module, or behavior is fully scheduled, you can allocate registers for all processes and sequential functions that are not inlined. If this step is successful, a netlist is generated for the object.

In the CtoS GUI, you can select **Edit -> Allocate Registers**, although you can specify that this be done automatically in the **Schedule** dialog (see [Figure 12-1 on page -2](#)).

For more control over this process, you can use the **allocate_registers** command ("**allocate_registers**" on [page E-18](#)), which lets you choose the *minimization*, using the **-min** option with one of these arguments:

- **regs** allocates the *absolute minimum* number of registers and is recommended when a register is *expensive* with respect to a mux. This is the default for ASIC targets.

[The choices continue on the next page.]

- **muxes** allocates one register for each output bit of a *resource* whose result is produced in one clock cycle and used in another clock cycle. This is the default for FPGA targets
- **regs_and_muxes** strikes a balance between the two previous options, sometimes achieving better results than just **regs**. *Note that the -min regs_and_muxes option is a preliminary feature.*
- **share** allocates one register for each output bit of a *source code operation* whose result is produced in one clock cycle and used in another clock cycle. As compared with minimizing muxes, minimizing sharing is much less efficient, because if a resource is shared among three ops, with **-min muxes** you get *one* register, but with **-min share** you get *three* registers. However, this may be useful to minimize interconnect cost on FPGAs by reducing input muxes, albeit at the expense of registers.

12.3.5 Connecting Terminals to Add Power or Test

After netlist generation (the last step of register allocation), you can use the **connect** command ("**connect**" on [page E-34](#)) to edit the netlist to add power or test management signals that were not described in the original SystemC source.

The **connect** command connects two terminals (SystemC ports) or instance terminals (SystemC ports on an instance of a module). One terminal should be a driver (either an input terminal or an output instance terminal), and the other terminal should be a load (either an output terminal or an input instance terminal). If the load is connected to another driver, then that driver is disconnected before the load is connected to the new driver. Both the driver and the load should be in the same module.

Limitation The **connect** command handles only scalar (1-bit) terminals and instance terminals.

12.3.6 Reporting Registers (Primary and Secondary)

After a design has been scheduled, to display the register bits used per state for a design, module, or behavior, you can use the **report_registers** command ([“report_registers” on page E-117](#)).

The action of this command varies slightly, depending on whether you are using the CtoS GUI or command-line CtoS, as follows:

- If **report_registers** is called on a module when you are using the CtoS GUI, and you are using the **Register Viewer**, registers for each behavior in that module are reported in a separate tab. This is also true if no *object_id* is provided, since the default is the top-level module. In this case, the CtoS GUI will provide a report of the registers for each behavior in the top-level module in a separate tab.
- If **report_registers** is called on a module when you are using CtoS at the command line, multiple listings are also reported.

To see a nested list of *secondary register bindings*, you can use the **-detail** option to the **report_registers** command. The report includes a **Kind** heading indicating **primary**, **secondary inverted**, or **secondary non-inverted**. Four register binding attributes support this feature: **is_primary**, **is_inverted**, **primary_binding**, and **secondary_bindings** ([“Register Binding Object Attributes \(reg_bindings\)” on page D-63](#)).

13 Analyzing and Implementing Designs

In the Analyzing and Implementing step of the CtoS flow, you can use the many helpful CtoS GUI reports and viewers to analyze your design.

You can also generate models, wrappers, RTL, and SLEC.

And CtoS allows you to generate gates, and to review the QoR, from CtoS-generated RTL.

This chapter has the following sections:

- [“Using Reports to Analyze Designs” on page 13-2](#)
- [“Generating Models, Wrappers, RTL, SLEC after Scheduling” on page 13-36](#)
- [“Generating Gates in CtoS” on page 13-42](#)

13.1 Using Reports to Analyze Designs

The CtoS Graphical User Interface (CtoS GUI) provides many reports and viewers to aid in your analysis.

This section walks you through a simple example to illustrate these features and is organized as follows:

- “Starting the CtoS GUI and Setting Up the Design” on page 13-2
- “Resource Bindings Viewer” on page 13-4
- “Register Bindings Viewer” on page 13-11
- “Path Summary Report” on page 13-15
- “Timing Report” on page 13-16
- “Cycle Analysis Viewer” on page 13-20
- “RTL Schematic Viewer” on page 13-23
- “Reporting Power and Area Using the Tree Map” on page 13-29
- “Summary Report” on page 13-35

13.1.1 Starting the CtoS GUI and Setting Up the Design

The example used in this section is included with the CtoS release.

From your CtoS installation area, change to the following directory:

```
install_directory/share/ctos/examples/training/analysis
```

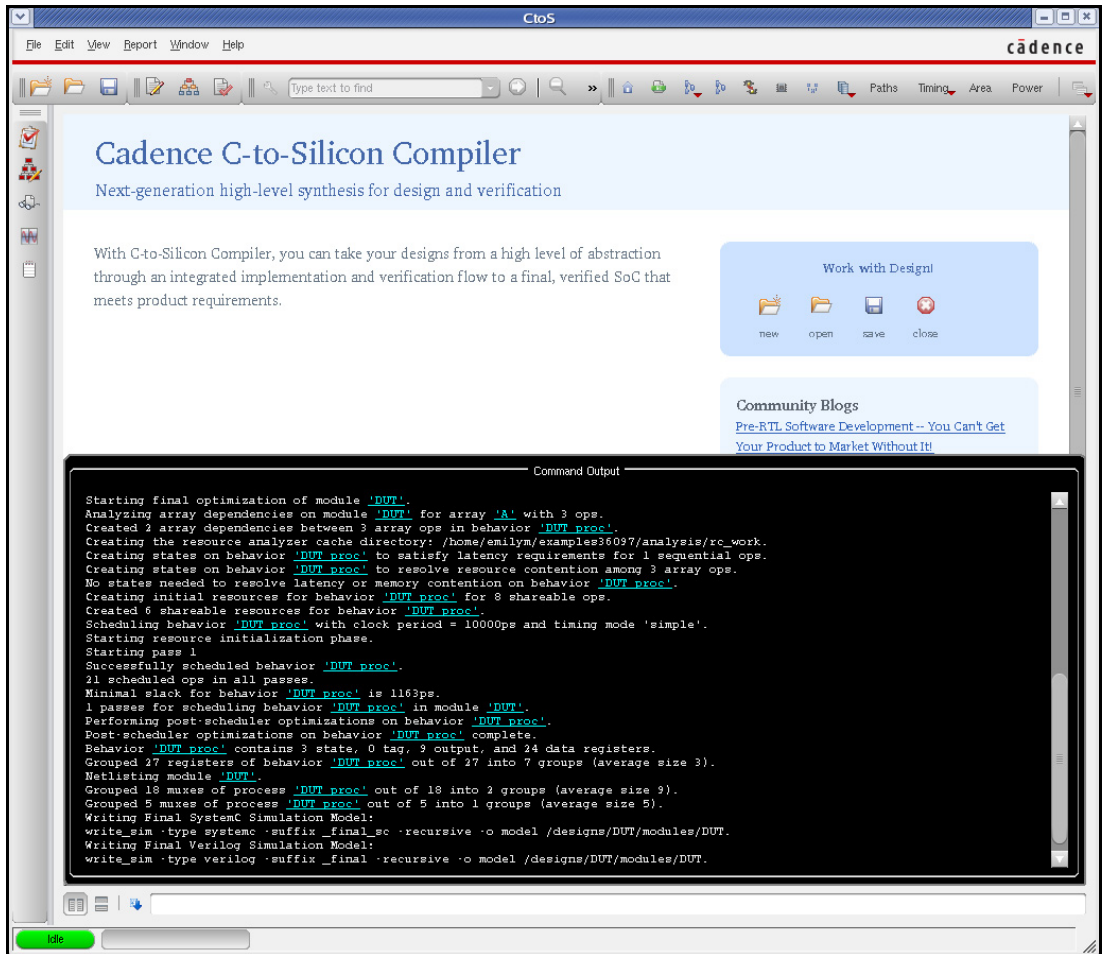
Note Before running any CtoS examples, first review “Setup for Examples” on page F-2.

This is the directory in which you should start the CtoS GUI.

Type at the command line:

```
ctosgui ctos.tcl
```

Figure 13-1 Main View of the CtoS GUI



This script loads the sample design and completes all design steps, through scheduling.

You should see a window similar to the one shown in [Figure 13-1 on page 13-3](#).

Note For more detail, see “Starting the CtoS GUI” on page 6-2.

13.1.2 Resource Bindings Viewer

Select **View -> Resources**, and you will see the **Resource Bindings Viewer**, as shown in [Figure 13-2 on page 13-4](#).

When there are multiple processes in a design, you will see a table for each process. This design has only one process, **DUT_proc**, defined by the **proc** function in **DUT.cpp**, so only one table is displayed.

Figure 13-2 Resource Bindings Viewer

Resource Bindings					
DUT_proc					
Resource Bindings	Resource Type	umul	add	sub	gtle
	Expand/Collapse	Expand	Expand	Expand	Expand
	Widths	9			
Shareable Ops	Instance Name	umul(s)	add(s)	sub(s)	gtle(s)
mul ops					
add ops					
sub ops					
lt ops					

The following sections describe the **Resource Bindings Viewer** in detail:

- “Understanding the Resource Bindings Viewer” on page 13-4
- “Configuring the Resource Bindings Viewer” on page 13-8
- “Cross-Linking the Resource Bindings Viewer with Other Viewers” on page 13-8
- “Using the Resource Bindings Viewer after a Failed Schedule” on page 13-11

13.1.2.1 Understanding the Resource Bindings Viewer

The *rows* of the **Resource Bindings Viewer** correspond to the *shareable ops* of the process, categorized by op type, while the *columns* correspond to the *shareable resource instances* used in the process.

The constant resources/ops are listed separately from non-const resources/ops. The constant resources/ops have a suffix following the pattern ‘**c <delta>**’; where ‘**c**’ denotes constant and **delta** is the constant value. For example “add c 1” is an adder that increments by constant 1 and “add c -1” is an adder that decrements by constant 1.

The multi-level heading of this viewer groups the resources by *Resource Type*, (*Port*) *Widths*, and *Instance Name*.

The items in the *Resource Type* group can be expanded to list all of the resource instances of that resource type. Clicking the **Expand** button under the **umul** resource type shows that two multiplier resources are instantiated in the design, as shown in [Figure 13-3 on page 13-5](#). Each multiplier implements a single multiply operation as indicated by the green-colored entry marked **bound**.

This design requires two multipliers, because both multipliers are scheduled on the same edge.

Right-click on the **mul_In22** row header, and select **Show Scheduled Edge** (and it is best to select the *On the side* icon) from the context menu to see the edge in the CDFG, as shown in [Figure 13-4 on page 13-6](#). If you also do this for **mul_In22_0** (right-click, select **Show Scheduled Edge**), you can see that the second multiplier op is on the same edge.

Figure 13-3 Resource Bindings Viewer Showing umul Expanded

Resource Bindings						
DUT_proc						
Resource Bindings	Resource Type	umul		add	sub	gtle
	Expand/Collapse					
	Widths	9				9
Shareable Ops	\ Instance Name	(*)	(*)_0	add(s)	sub(s)	gtle(s)
mul ops						
mul_In22		bound				
mul_In22_0			bound			
add ops						
sub ops						
lt ops						

Figure 13-4 CDFG Showing Scheduled Edge for mul_In22

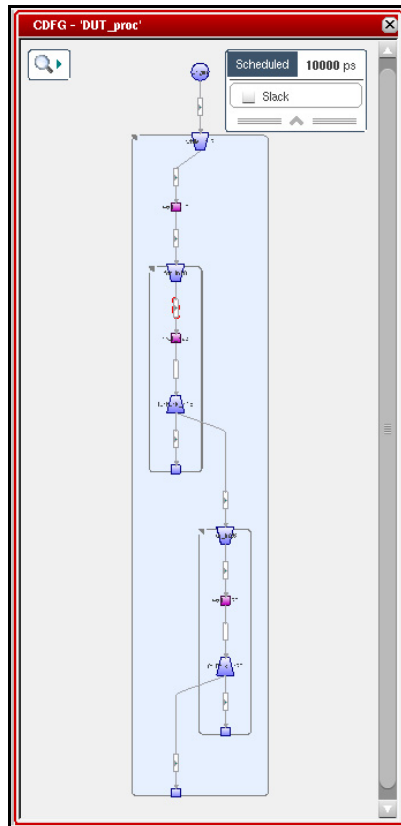


Figure 13-5 Resource Bindings Viewer Showing Tooltip for add_In18

Resource Bindings						
DUT_proc						
Resource Bindings	Resource Type	umul		add		addsub
	Expand/Collapse	Collapse		Collapse		Expand
	Widths	9		9	6x1	9
Shareable Ops	Instance Name	(*)	(*)_0	(*)	(*)	addsub(s)
mul ops						
mul_In22		bound				
mul_In22_0			bound			
add ops						
add_In22				bound		
add_In27_0						
add_In18					bound	
add_In26					bound	
sub ops						
it ops						
		Operation: add_In18 Width: IN - A (5) B (7) DUT - Z (6) Location: DUT.cpp:18:25 Instance: add_6x1 Max Delay: 545 Slack: undefined				

Also, in the **Resource Bindings Viewer**, *multiple green entries* in a column means the resource implements multiple operations. Clicking the **Expand** button under the **add** resource type shows two **add** resources. The first adder has output port size of **9** and implements one adder, as indicated by the one green cell.

The second added has output port size of **6x1** and implements two adders, as indicated by the two green cells. Selecting **Show Scheduled Edge** shows each op is scheduled on different edges and thus can be shared by the same resource.

Positioning the cursor over an op or resource instance displays a tooltip about the object, as shown in [Figure 13-5 on page 13-7](#). For example, the tooltip for an op shows the number of bits for the inputs or outputs of the op, and the location of the op in the SystemC input code, when applicable.

For each operation in the table, additional resource instances available in the current design that could also implement the operation are indicated by blue-colored entries in the table.

A resource instance can implement an operation if:

- The functionality of the resource instance is sufficient to implement the functional type of the operation and the bit widths of the inputs, *and*
- The outputs of the resource instance are wide enough for the inputs and the output of the operation.

For example, locate the operation **add_In18** listed under the **add ops**. In this example, **add_In18** could be implemented by either **add_9** (in blue) or **add_6x1** (in green); the CtoS scheduler chose **add_6x1**.

Alternate resource information is useful when you want to explore the possibility of sharing, or not sharing, resource instances for a particular type of operation to reduce the number of resource instances in a design, or to reduce the timing criticality of particular operations.

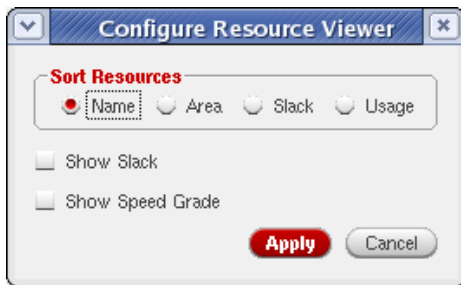
13.1.2.2 Configuring the Resource Bindings Viewer

The rows and columns of the **Resource Bindings Viewer** can be sorted, based on various metrics.

Right-click in the table, and select **Configure Viewer** from the context menu. You will see the **Configure Resource Viewer**, as shown in [Figure 13-6 on page 13-8](#).

This sorting capability is useful to determine timing-critical states or resource instances or to find resource instances whose areas are more dominant in the design than others.

Figure 13-6 Resource Bindings Viewer - Configure Resource Viewer



13.1.2.3 Cross-Linking the Resource Bindings Viewer with Other Viewers

For objects shown in the **Resource Bindings Viewer**, you can select particular links from the context menus to relate them to the following:

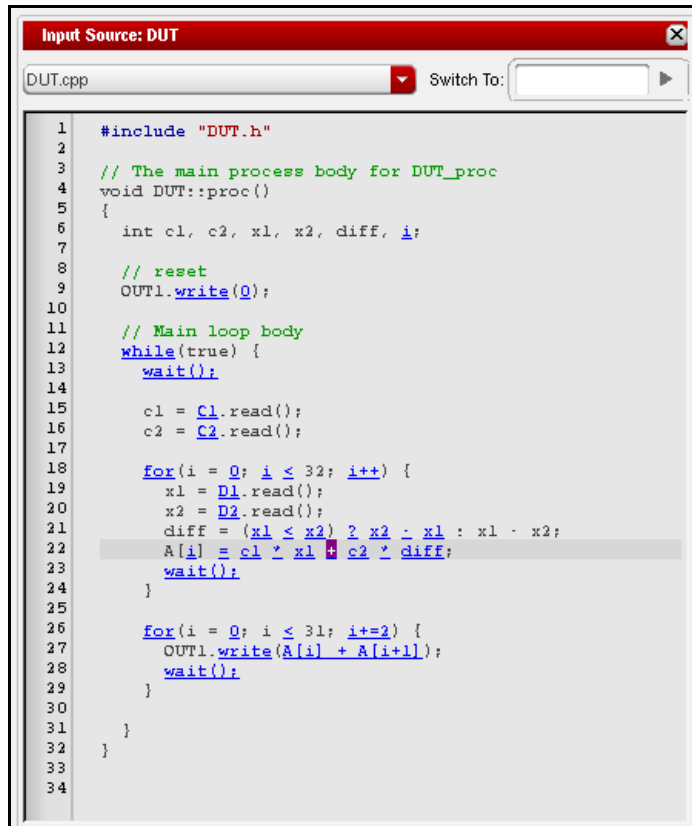
- the original SystemC input code
- the output models
- the internal data structure of CtoS

For operations, you can see the SystemC input code.

For example, right-click operation **add_In22** and select **Show Input Source**.

This opens a new tab for the SystemC input code, with the addition operation at line 22 highlighted, as shown in [Figure 13-7 on page 13-9](#).

Figure 13-7 Input Source for add_In22



```

1  #include "DUT.h"
2
3  // The main process body for DUT_proc
4  void DUT::proc()
5  {
6      int c1, c2, x1, x2, diff, i;
7
8      // reset
9      OUT1.write(0);
10
11     // Main loop body
12     while(true) {
13         wait();
14
15         c1 = C1.read();
16         c2 = C2.read();
17
18         for(i = 0; i < 32; i++) {
19             x1 = D1.read();
20             x2 = D2.read();
21             diff = (x1 < x2) ? x2 - x1 : x1 - x2;
22             A[i] = c1 * x1 + c2 * diff;
23             wait();
24         }
25
26         for(i = 0; i < 31; i+=2) {
27             OUT1.write(A[i] + A[i+1]);
28             wait();
29         }
30     }
31 }
32
33
34

```

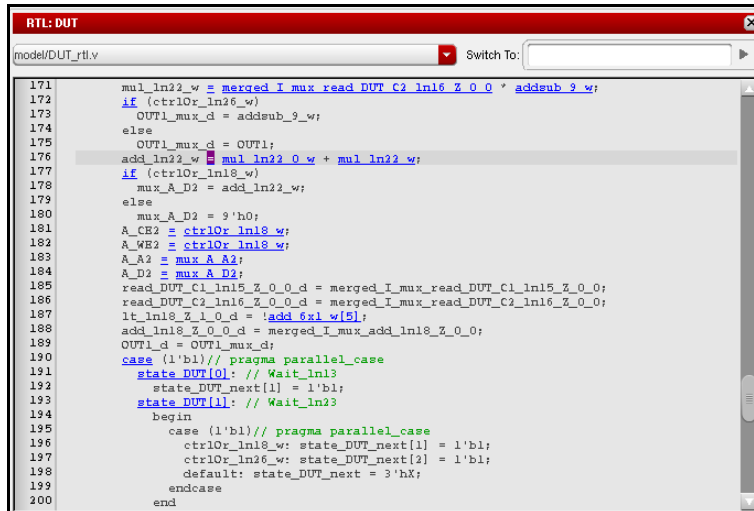
For resource instances, you can see both SystemC input code and RTL output.

For example, to see RTL output, back in the **Resource Bindings Viewer**, right-click on the instance name field for **add_9**, which is represented by an asterisk (*). [This name is due to the fact that the instance is an **add** and the width is **9**, and no additional information is required.]

Select **Show RTL Source** from the context menu, and a new tab for the RTL output opens, with the assignment highlighted inside a statement starting with **add_22_w** =, as shown in [Figure 13-8 on page 13-10](#).

This means the output of this resource instance is represented by **add_22_w** in the RTL model.

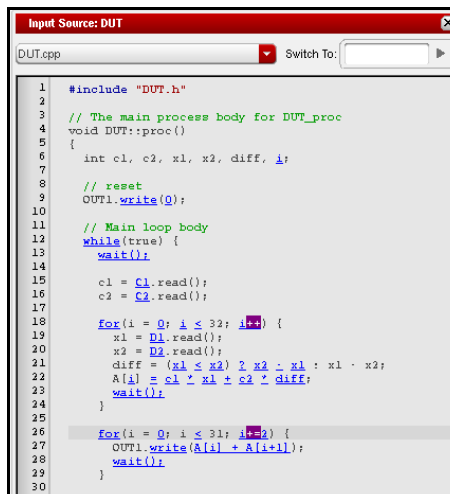
Figure 13-8 RTL for add_9



```
171 mul_ln22_w = merged_I_mux_read DUT_C2_ln16_Z_0_0 * addsub_9_w;
172 if (ctrl0r_ln26_w)
173     OUT1_mux_d = addsub_9_w;
174 else
175     OUT1_mux_d = OUT1;
176 add_ln22_w = mul_ln22_0_w + mul_ln22_w;
177 if (ctrl0r_ln18_w)
178     mux_A_D2 = add_ln22_w;
179 else
180     mux_A_D2 = 9'h0;
181 A_CE2 = ctrl0r_ln18_w;
182 A_WE2 = ctrl0r_ln18_w;
183 A_A2 = mux_A_A2;
184 A_D2 = mux_A_D2;
185 read_DUT_C1_ln15_Z_0_0_d = merged_I_mux_read DUT_C1_ln15_Z_0_0;
186 read_DUT_C2_ln16_Z_0_0_d = merged_I_mux_read DUT_C2_ln16_Z_0_0;
187 lt_ln18_Z_1_0_d = !add_6x1_w[5];
188 add_ln18_Z_0_0_d = merged_I_mux_add_ln18_Z_0_0;
189 OUT1_d = OUT1_mux_d;
190 case (1'b1) // pragma parallel_case
191     state_DUT[0]: // Wait_ln13
192         state_DUT_next[1] = 1'b1;
193     state_DUT[1]: // Wait_ln23
194         begin
195             case (1'b1) // pragma parallel_case
196                 ctrl0r_ln18_w: state_DUT_next[1] = 1'b1;
197                 ctrl0r_ln26_w: state_DUT_next[2] = 1'b1;
198                 default: state_DUT_next = 3'hK;
199             endcase
200         end
```

Returning to the **Resource Bindings Viewer**, this time, right-click **add_6x1**, and select **Show Ops > in Input**. This highlights, in magenta, all operations implemented by this resource instance, as shown in [Figure 13-9 on page 13-10](#) – in this case, the two addition operations at line 18 and line 26.

Figure 13-9 Input Source for add_6x1



```
1 #include "DUT.h"
2
3 // The main process body for DUT_proc
4 void DUT::proc()
5 {
6     int c1, c2, x1, x2, diff, i;
7
8     // reset
9     OUT1.write(0);
10
11     // Main loop body
12     while(true) {
13         wait();
14
15         c1 = C1.read();
16         c2 = C2.read();
17
18         for(i = 0; i < 32; i++) {
19             x1 = D1.read();
20             x2 = D2.read();
21             diff = (x1 < x2) ? x2 - x1 : x1 - x2;
22             A[i] = c1 * x1 + c2 * diff;
23             wait();
24         }
25
26         for(i = 0; i < 31; i++) {
27             OUT1.write(A[i] + A[i+1]);
28             wait();
29         }
30     }
```

13.1.2.4 Using the Resource Bindings Viewer after a Failed Schedule

The **Resource Bindings Viewer** shows, in green, ops that were successfully scheduled on particular instances. It also highlights, in red, ops that *failed* to schedule, as shown in [Figure 13-10 on page 13-11](#).

Figure 13-10 Resource Bindings Viewer after a Failed Schedule

Resource Bindings								
DUT_proc								
Resource Bindings	Resource Type	umul						
	Expand/Collapse	Collapse						
	Widths	9						
Shareable Ops	\ Instance Name	(*)	(*)_0	(*)_1	(*)_2	(*)_3	(*)_4	(*)
mul ops								
mul_in22		bound						
mul_in22_0			bound					
add ops								
add_in22								
add_in27_0								
add_in18								
add_in26								
sub ops								
sub_in21								
R ops								
lt_in21								

13.1.3 Register Bindings Viewer

Select **View -> Registers**, and you will see the **Register Bindings** viewer, as shown in [Figure 13-11 on page 13-11](#).

Figure 13-11 Register Bindings Viewer

Register Bindings									
DUT_proc									
	ctrlOr_in12_Z_0	ctrlOr_in18_Z_0	ctrlOr_in26_Z_0	add_in18_Z_0	add_in18_Z_0	add_in18_Z_0	add_in18_Z_0	add_in18_Z_0	read_DUT_C1_in15_Z_0
	0	0	0	0	1	2	3	4	0
origin									
Wait_in13									
Wait_in23									
Wait_in28									

If there are multiple processes in a design, you will see a table for each process. For this design, there is only one process, **DUT_proc**, defined by the **proc** function in **DUT.cpp**, so only one table is displayed.

The following sections describe the **Register Bindings** viewer in detail:

- “Understanding the Register Bindings Viewer” on page 13-12
- “Cross-linking the Register Bindings Viewer with other Viewers” on page 13-13

13.1.3.1 Understanding the Register Bindings Viewer

The *rows* of the **Register Bindings** viewer represent the control states of the process, while the *columns* represent the registers used in the resulting RTL.

The registers are shown in terms of individual bits. In the RTL, however, some of the individual registers are grouped together and represented as a single **reg** variable in Verilog. This typically happens when the output of a single operation in the RTL consists of multiple bits, and the output is fed into registers. Therefore, the **Register Bindings** viewer also groups such registers.

For example, five columns are grouped together under the name **add_in18_Z_0**, where the second row under this name shows the five individual registers using the integers from **0** to **4**.

This group corresponds to the 5-bit reg variable **w_add_in18_Z_0** in the RTL output.

If you double-click on any of the **add_in18_Z_0** entries in the table, the individual registers are collapsed, as shown in [Figure 13-12 on page 13-12](#).

Figure 13-12 Register Bindings - Individual Registers Collapsed

Register Bindings												
DUT_proc												
	ctrlOr_in12_Z_0	ctrlOr_in18_Z_0	ctrlOr_in26_Z_0	add_in18_Z_0					read_DUT_C1_in15_Z_0	read_DUT_C1_in15_Z_0	read_DUT_C1_in15_Z_0	read_DUT_C1_in15_Z_0
	0	0	0	0	1	2	3	4	0	1	2	3
origin												
Wait_in13												
Wait_in23												
Wait_in28												

This is useful to consolidate columns of the table to reduce the width of the table.

Each entry of the table represents the value of the corresponding register stored at the control state given in the row of that entry. If an entry does not show a value, it means the corresponding register does not receive any new value in that state.

Note Registers may sometimes have values in multiple entries at different states. For example, the registers under the group **add_in18_Z_0** have values at the state **Wait_in23**, as well as at **Wait_in28**. When this happens, it means such registers are *shared*, that is, the registers are used to hold values produced by different operations executed at different control states of the design.

Register sharing is an optimization technique that can reduce the number of registers in the resulting implementation, although sharing will introduce multiplexers in front of the registers, creating additional delay on the paths through the registers.

13.1.3.2 Cross-linking the Register Bindings Viewer with other Viewers

The **Register Bindings** viewer has a cross-link capability, similar to the **Resource Bindings Viewer**.

First, right-click a state in the first column, for example **Wait_In13**, to open the context menu. Select **Show Input Source** in the menu. In the **Input Source** viewer, you will see the **wait()** statement highlighted at line **13**. This capability is same as that for the **Resource Bindings Viewer** (see “Cross-Linking the Resource Bindings Viewer with Other Viewers” on page 13-8).

The cross-link capability is also available for the entries of the table. For example, in the column headed by **add_In18_Z_0**, right-click the entry for the value **add_In18_Z_1** at row **Wait_In23** (you can double-click **add_In18_Z_0** to expand this area if it is still collapsed).

In the context menu for the value, select **Show driver**. You will see the **Resource Bindings Viewer** with the operation **add_In18** highlighted, as shown in [Figure 13-13 on page 13-13](#).

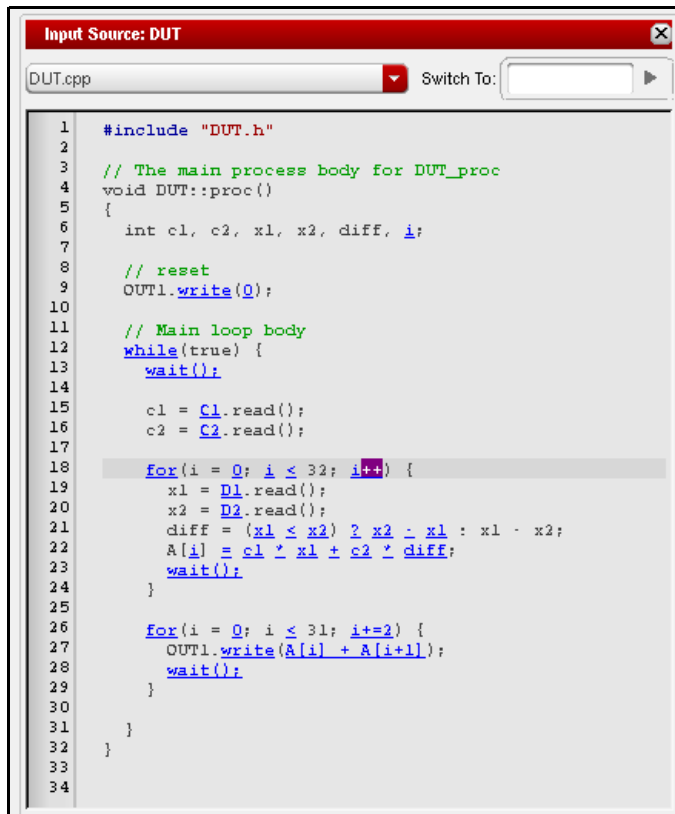
Figure 13-13 Resource Bindings - add_In18 driver from Register Bindings Highlighted

Resource Bindings							
DUT_proc							
Resource Bindings	Resource Type	umul		add		addsub	gtle
	Expand/Collapse	Collapse		Collapse		Collapse	Collapse
	Widths	9		9	6x1	9	9
Shareable Ops	\ Instance Name	(*)	(*)_0	(*)	(*)	(*)	(*)
mul ops							
mul_in22		bound					
mul_in22_0			bound				
add ops							
add_in22				bound			
add_in27_0						bound	
add_in18					bound		
add_in26					bound		
sub ops							
sub_in21						bound	
lt ops							
lt_in21							bound

This means that value **add_In18_Z_1**, selected from the **Register Bindings** viewer, is driven by the operation **add_In18**. You can use an additional cross-link capability of the **Resource Bindings Viewer** by selecting **Show Input Source** in the context menu of the operation **add_In18**.

You will see the SystemC input code, and the addition operation at line 18 for incrementing the loop index **i** will be highlighted, as shown in [Figure 13-14 on page 13-14](#).

Figure 13-14 Input Source - add_In18



```
1  #include "DUT.h"
2
3  // The main process body for DUT_proc
4  void DUT::proc()
5  {
6      int c1, c2, x1, x2, diff, i;
7
8      // reset
9      OUT1.write(0);
10
11     // Main loop body
12     while(true) {
13         wait();
14
15         c1 = C1.read();
16         c2 = C2.read();
17
18         for(i = 0; i < 32; i++) {
19             x1 = D1.read();
20             x2 = D2.read();
21             diff = (x1 < x2) ? x2 - x1 : x1 - x2;
22             A[i] = c1 * x1 + c2 * diff;
23             wait();
24         }
25
26         for(i = 0; i < 31; i+=2) {
27             OUT1.write(A[i] + A[i+1]);
28             wait();
29         }
30     }
31 }
32
33
34
```

You can thus see that the value in question in the **Register Bindings** viewer is produced as the result of the addition operation highlighted at line 18 of **DUT.cpp**, and this value is stored at the control state corresponding to the **wait()** statement at line 23.

The context menu also provides another option: **Show receiver**. This is similar to **Show driver**, except it highlights operations for which the entry values are used as inputs of the operations.

By using these cross-link capabilities, you can analyze, for a given register, which operations *produce* the values to be stored in the register and which operations *use* those values.

Furthermore, by combining the cross-link capabilities of the **Register Bindings** and **Resource Bindings** viewers, you can observe those operations in terms of the input SystemC code, as well as of the RTL output.

13.1.4 Path Summary Report

Select **Report->Paths** to see the **Path Summary Report - Summary tab** (Figure 13-15 on page 13-15).

Figure 13-15 Path Summary Report - the Summary Tab

Path Summary Report for /designs/DUT/modules/DUT				
Summary of paths reported for module DUT			Details of paths reported for module DUT	
Type	Source	Dest.	Logic?	#Paths
Port to Reg	C1	r CLK	yes	54
	C2	r CLK	yes	54
	D1	r CLK	yes	81
	D1	r CLK/sync lo RSTn	yes	81
	D2	r CLK	yes	81
	D2	r CLK/sync lo RSTn	yes	81
Port to Port				
Reg to Reg				
Reg to Port				
	r CLK/sync lo RSTn	OUT1	no	9

The **Path Summary Report** reports types of *paths* in the RTL model, where a path is defined only in terms of the start and end points.

The start point is either a single-bit register or a single bit of an input port of the top module of the RTL; the end point is either a single-bit register or a single bit of an output port of the top module of the RTL.

There are two types of path reports: **Summary** and **Details**. In either type of report, paths are classified in terms of the following aspects:

- the types of the start and end instances
- the name of the input or output port, if it is the start or end instance
- the clock and reset used in the register, if it is the start or end instance
- the presence or absence of combinational logic between the start and end instances

The **Summary** tab reports a list of types of paths classified this way, and the number of paths in each type. For those paths whose start and end instances are both registers, the report includes such paths only if different clock or reset is used between the start and end registers.

Note In this case, a *path* is defined in terms of the start and end instances, and the number of paths is reported in terms of the number of distinct pairs of start and end instances of the paths.

In the column **Dest.** (destination), if the end instance of the path is a register, the clock and reset domains of the end instance (if the reset is available for the register) are shown, where clock information is given in terms of the name of the clock terminal and the polarity of the edge (**r** for rise and **f** for fall), and reset information is given in terms of the name of the reset terminal, whether it is synchronous or asynchronous, and the active level (**lo** for active low).

Figure 13-16 Path Summary Report - the Details Tab

Path Summary Report for /designs/DUT/modules/DUT			
Summary of paths reported for module DUT		Details of paths reported for module DUT	
Type	Source	Dest.	Logic?
☒ Paths originating from a port			
.....	C1[0]	A[0]	yes
.....	C2[0]	A[0]	yes
.....	D1[0]	A[0]	yes
.....	D1[0]	OUT1[0]	yes
.....	D1[0]	OUT1[1]	yes
.....	D1[0]	OUT1[2]	yes
.....	D1[0]	OUT1[3]	yes
.....	D1[0]	OUT1[4]	yes
.....	D1[0]	OUT1[5]	yes
.....	D1[0]	OUT1[6]	yes
.....	D1[0]	OUT1[7]	yes
.....	D1[0]	OUT1[8]	yes
.....	D2[0]	A[0]	yes
.....	D2[0]	OUT1[0]	yes
.....	D2[0]	OUT1[1]	yes
.....	D2[0]	OUT1[2]	yes
.....	D2[0]	OUT1[3]	yes
.....	D2[0]	OUT1[4]	yes
.....	D2[0]	OUT1[5]	yes
.....	D2[0]	OUT1[6]	yes
.....	D2[0]	OUT1[7]	yes
.....	D2[0]	OUT1[8]	yes
☒ Paths terminating at a Port			
.....	OUT1[0]	OUT1[0]	no
.....	OUT1[1]	OUT1[1]	no
.....	OUT1[2]	OUT1[2]	no
.....	OUT1[3]	OUT1[3]	no
.....	OUT1[4]	OUT1[4]	no
.....	OUT1[5]	OUT1[5]	no
.....	OUT1[6]	OUT1[6]	no
.....	OUT1[7]	OUT1[7]	no
.....	OUT1[8]	OUT1[8]	no
☐ Paths between registers with different clock/reset			

The **Details** tab, as shown in [Figure 13-16 on page 13-16](#), lists individual paths of the following types, in terms of the names of the start and end instances of the paths:

- The start is an input port, with information about whether combinational logic is present on the path.
- The end is an output port, with information about whether combinational logic is present on the path.
- The start and end instances are registers, but associated with different clock or reset.

It is important that you view this report after writing an RTL model, in order to see the report with the same names used for registers shown in the RTL model. If you view this report before creating an RTL model, the names of the internal database of the tool are used.

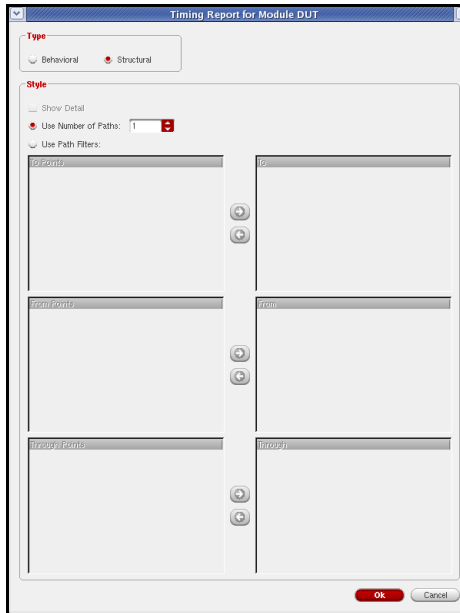
The paths terminating at a port section of the detailed report will list the same name as the source and destination for registered outputs, where the first name refers to the register, and the second name refers to the output.

13.1.5 Timing Report

To get a timing report, select **Report -> Timing -> Report** to see the **Timing Report** dialog, as shown in [Figure 13-17 on page 13-17](#).

In the first field, you select the type of report, which is either **Behavioral** or **Structural**.

Figure 13-17 The Timing Report Dialog



- The *structural* type reports a path in terms of resource instances, and is described in “[Structural Timing Report](#)” on page 13-17.
- The *behavioral* type reports a path in terms of operations executed on the path, and is described in “[Behavioral Timing Report](#)” on page 13-19

13.1.5.1 Structural Timing Report

To see a *Structural Timing Report*, select **Structural** in the **Type** field. (The **Show Detail** field is grayed out, because structural details are available only for a **Behavioral** type.)

You can then choose the **Number of Paths** to be displayed in the report. If you select an integer greater than **1**, CtoS will show the most critical paths up to that number, one for each tab. For this example, you will leave the number of paths at **1**, so only one of the most critical paths is reported.

Use Path Filters is a search capability that lets you choose the start and end points of paths, as well as intermediate points through which the paths should go. This capability is convenient when you are interested in particular paths. For example, using the RTL model CtoS produces, if you further synthesize the design using downstream tools, such as logic synthesis or routing, and if you apply a timing analysis to the resulting netlist, you might find certain paths violating the timing requirements.

Figure 13-18 Structural Timing Report

Structural Timing Report for Module DUT				
Timing report for most critical path in module DUT				
Instance	Terminal	Master	Fanout	Delay (ps) / Arrival (ps)
1	add_in18_2_0/Q[4]	flipflop_5	3	143 143
2	add_in18_2_0_inv_4/A	unary_not_1		42 190
3	add_in18_2_0_inv_4/Z	unary_not_1	1	0 190
4	ctrlAnd_0_in26/A[1]	unary_and_2		0 190
5	ctrlAnd_0_in26/Z	unary_and_2	2	213 403
6	ctrlOr_in26/A[1]	unary_or_2		26 429
7	ctrlOr_in26/Z	unary_or_2	12	210 639
8	l_mux_A_A1/C[0]	mux_2_5		96 735
9	l_mux_A_A1/Z[0]	mux_2_5	1	256 991
10	A/A1[0]	ram_32x9_2ar_1w		0 991
11	A/Q1[0]	ram_32x9_2ar_1w	1	1239 2230
12	l_mux_addsub_9_A/A_01[0]	mux_2_9		0 2230
13	l_mux_addsub_9_A/Z[0]	mux_2_9	1	245 2475
14	addsub_9/A[0]	addsub_9		0 2475
15	addsub_9/Z[0]	addsub_9	2	1469 3944
16	umul_9/B[0]	umul_9		26 3970
17	umul_9/Z[0]	umul_9	1	1899 5869
18	add_9/B[0]	add_9		0 5869
19	add_9/Z[0]	add_9	1	1024 6893
20	l_mux_A_D2/A_00[0]	mux_2_9		0 6893
21	l_mux_A_D2/Z[0]	mux_2_9	1	245 7138
22	A/D2[0]	ram_32x9_2ar_1w		0 7138
23	A/CLK	ram_32x9_2ar_1w		959 8097

Typically, the names of the ports or registers of such paths have correspondence to names used in the CtoS-produced RTL model. In that case, you can specify the start and end points of such a path in this search field, so you can obtain a timing report for those paths within CtoS.

For this example, click **OK** without specifying instances to search for, to get the critical path report. This will open a new tab that reports a single path in a table format, as shown in [Figure 13-18 on page 13-18](#).

Here is a description of the columns in this report:

- The first column shows the sequence of terminal pins (inputs and outputs) of the resource instances on the path.
- The second column shows the names of the resource instances.
- The third column shows the number of fanouts for each output terminal pin of the resources on the path.

The last two columns show delay information:

- The fourth column shows delay in picoseconds from the preceding terminal pin listed in the table, according to whether it is an *output* or *input* pin:
 - For an *output* terminal pin, it shows the delay of the resource instance of the output pin from the input pin of the resource listed one row above.
 - For an *input* terminal pin, it shows **0** if the output pin that drives the input pin has no other fanout, while it shows the delay with respect to the fanout capacitance if the output pin has other fanouts.
- The last column shows the total arrival time from the first terminal pin of the path.

13.1.5.2 Behavioral Timing Report

You can also see the operations in the RTL actually executed on the path in the timing report.

To do this, close the previous timing report, and again select **Report -> Timing -> Report**. This time, select **Behavioral** in the **Type** field, leave the other defaults, and click **OK** to get a *Behavioral Timing Report*, as shown in Figure 13-19 on page 13-19.

Figure 13-19 Behavioral Timing Report

Behavioral Timing Report for Module DUT				
Timing report for most critical path in module DUT				
Op Pin	Op Type	Delay (ps)	Arrival (ps)	
1 if_in26/Z[1]	if	0	148	
2 ctrlAnd_0_in26/A[1]	unary_and	42	190	
3 ctrlAnd_0_in26/Z[0]	unary_and	213	403	
4 ctrlOr_in26/A[1]	unary_or	26	429	
5 ctrlOr_in26/Z[0]	unary_or	210	639	
6 memread_DUT_A_in27_0/A[0]	memRead	352	991	
7 memread_DUT_A_in27_0/Q[0]	memRead	1239	2230	
8 add_in27_0/A[0]	add	245	2475	
9 sub_in21/Z[0]	sub	1469	3944	
10 mul_in22/B[0]	mul	26	3970	
11 mul_in22/Z[0]	mul	1899	5869	
12 add_in22/B[0]	add	0	5869	
13 add_in22/Z[0]	add	1024	6893	
14 memwrite_DUT_A_in22/D[0]	memWrite	245	7138	

Here is a description of the columns in this report:

- The first column lists individual operations and their input and output terminal pins on the path.
- The second column displays the types of the operations.
- The last two columns show the delay information in the same way as the structural report.

You can right-click any of the individual operations listed in the table, and select one of the following:

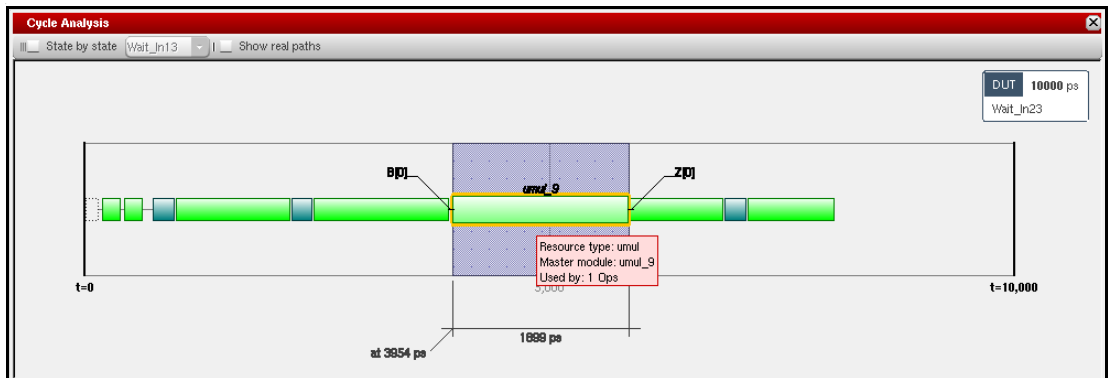
- If you select **Show Operation in Input**, you will see the selected operation highlighted in the **Input Source** viewer. You can thus see which operations in the SystemC input are on the critical path.
- If you select **Show Path > in Input**, all operations in the SystemC input code shown in the table are highlighted. For any of these highlighted operations in the SystemC input code, you can use the cross-link capability in the **Input Source** viewer to see the corresponding operations in the RTL, in the **Resource Bindings Viewer**, and so on.
- If you select **Show Op Instances in RTL**, the operation instance(s) is highlighted in the **RTL**.

13.1.6 Cycle Analysis Viewer

The Cycle Analysis viewer is a preliminary feature.

Select **Report -> Timing -> Cycle Analysis**, and you will see the **Cycle Analysis** viewer, which loads real structural critical paths. When you move the mouse over this diagram, you will see each section highlighted, and a tooltip provided, as shown in [Figure 13-20 on page 13-20](#), for **umul_9**.

Figure 13-20 Cycle Analysis Viewer, Highlighting umul_9



The following sections describe the **Cycle Analysis** viewer in detail:

- [“Color Coding in the Cycle Analysis Viewer” on page 13-20](#)
- [“Annotation of Time Slice in the Cycle Analysis Viewer” on page 13-21](#)
- [“Total Time and Slack in the Cycle Analysis Viewer” on page 13-22](#)
- [“Additional Features of the Cycle Analysis Viewer” on page 13-22](#)

13.1.6.1 Color Coding in the Cycle Analysis Viewer

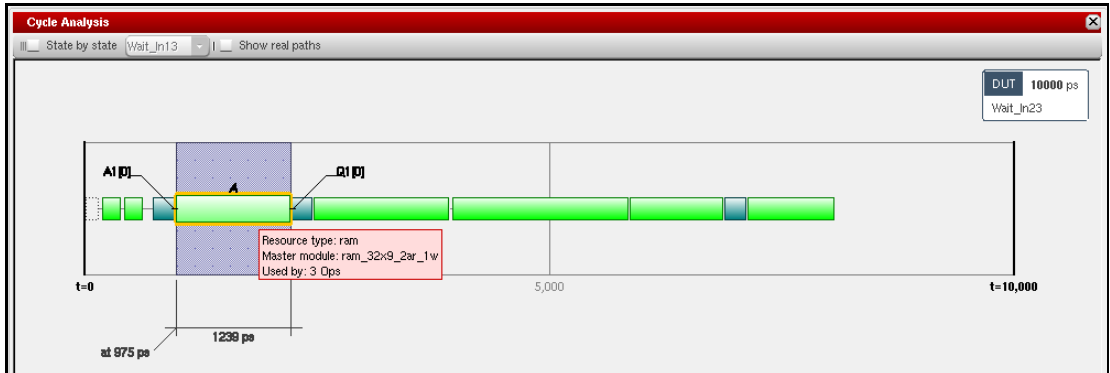
The elements on the paths in the **Cycle Analysis** viewer are color-coded, as follows:

- **bright green** - for normal instances
- **dark teal** - for sharing muxes
- **grey** (with a dotted box around) - for things just outside the bounds of the path

13.1.6.2 Annotation of Time Slice in the Cycle Analysis Viewer

In the **Cycle Analysis** viewer, the **time slice** is annotated with both the *start time* and the *duration*, as shown in [Figure 13-21 on page 13-21](#).

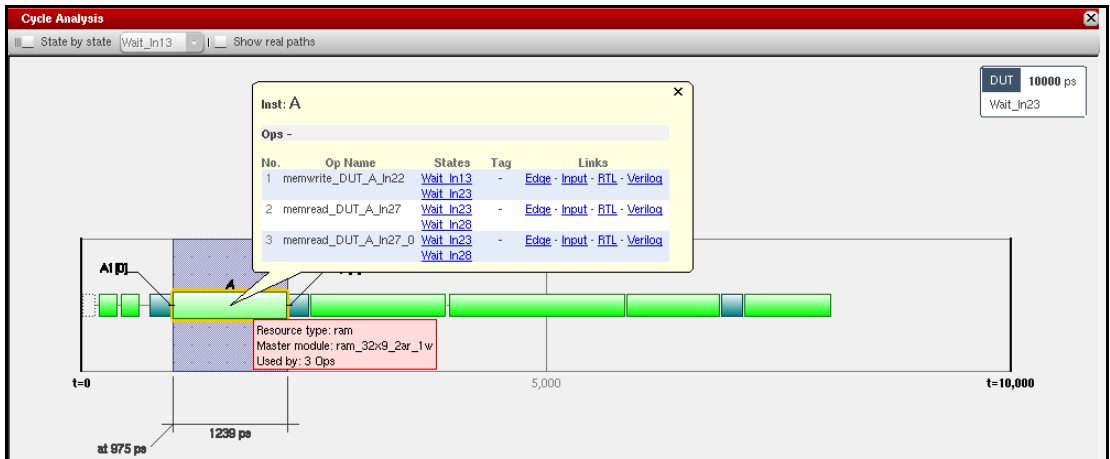
Figure 13-21 Example of Time Slice Annotation



You can also get information about the corresponding ops and states of each resource in the *detail bubble*, which is displayed if you click on the resource, as shown in [Figure 13-22 on page 13-21](#).

Note The **Tag** column, which in this case is not applicable, helps to disambiguate a case in which a resource is used by multiple ops in a given state.

Figure 13-22 Detail Bubble Showing Ops and States

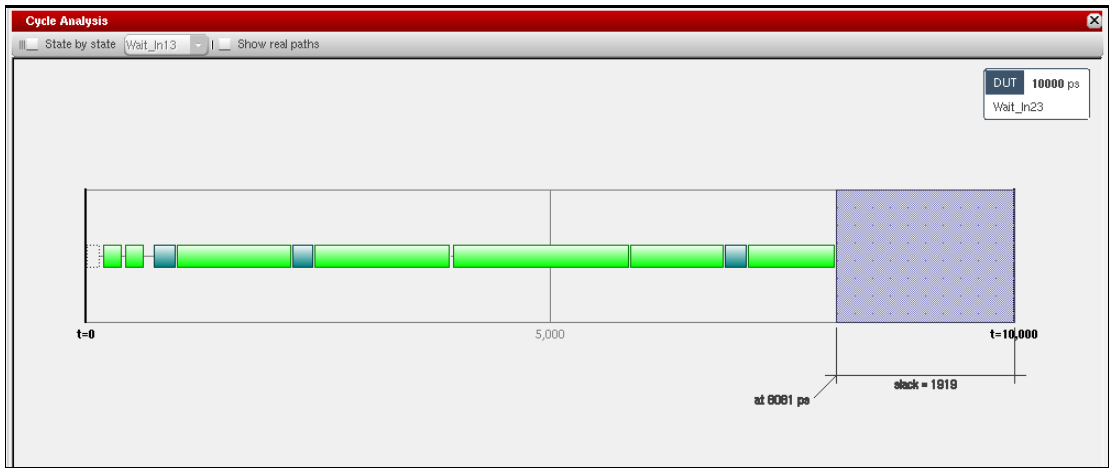


13.1.6.3 Total Time and Slack in the Cycle Analysis Viewer

In the Cycle Analysis viewer, the total time and slack are displayed when you hover at the end of the cycle, as shown in [Figure 13-23 on page 13-22](#).

Note If the slack were negative, it would be indicated with a reddish color.

Figure 13-23 Total Time and Slack for Critical Path



13.1.6.4 Additional Features of the Cycle Analysis Viewer

The **Cycle Analysis** viewer also has tooltips and links to navigate to the rest of the CtoS GUI.

From an op, you can jump to its states and its scheduled edge and see it in the input source, Verilog, and RTL source.

In addition to the zoom feature on the *Tool Bar*, you can use the **Ctrl** button with the mouse wheel to zoom in and out.

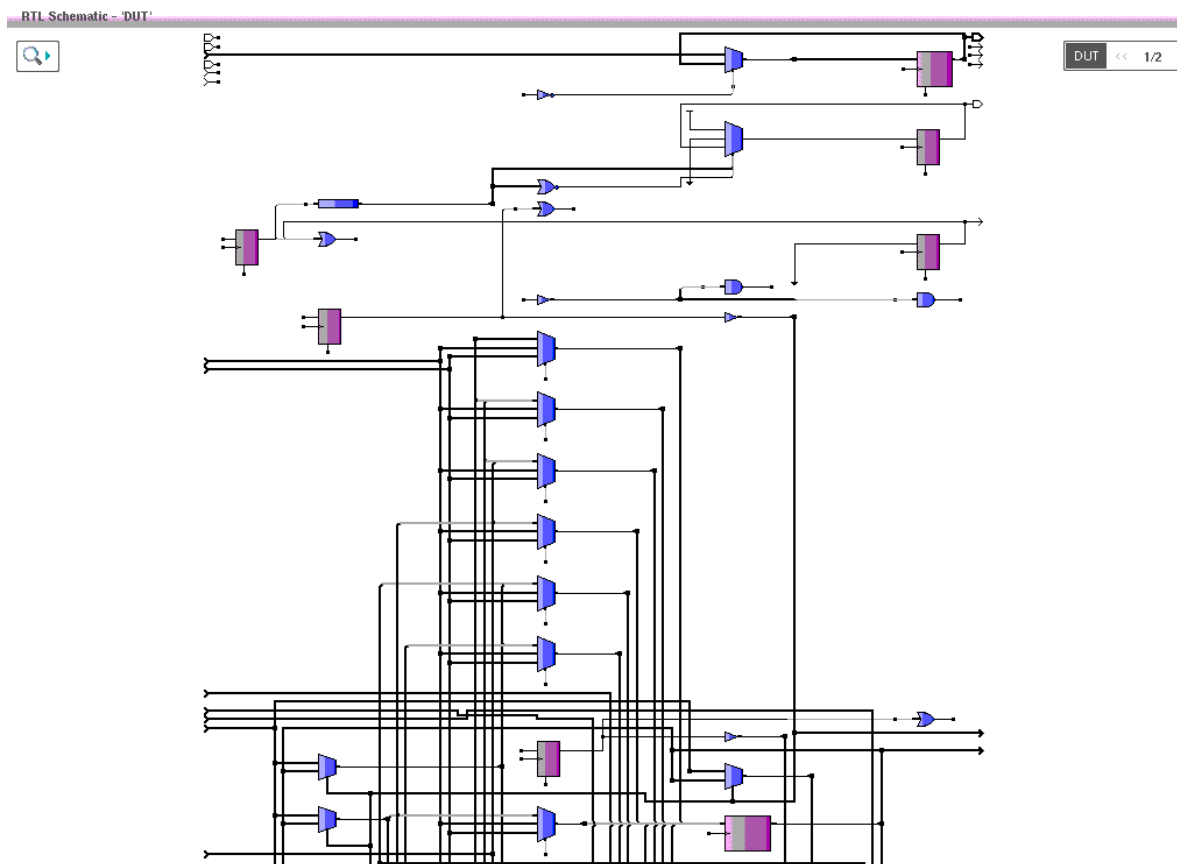
13.1.7 RTL Schematic Viewer

The RTL Schematic Viewer is a preliminary feature.

Select **View -> RTL Schematic**, and you will see the RTL Schematic Viewer of the complete design, [Figure 13-24 on page 13-23](#).

You can also view the RTL Schematic of specific modules. In the **Hierarchy Window**, right-click the module and select **Show RTL Schematic**. The RTL Schematic of modules allows users to quickly grasp the structure of the design and understand the complexity of each process.

Figure 13-24 RTL Schematic Viewer



The following sections describe the **RTL Schematic Viewer** in detail:

- [“Understanding the RTL Schematic Viewer” on page 13-24](#)
- [“Viewing Critical Paths in RTL Schematic” on page 13-25](#)

- [“Additional Features of the RTL Schematic Viewer” on page 13-25](#)

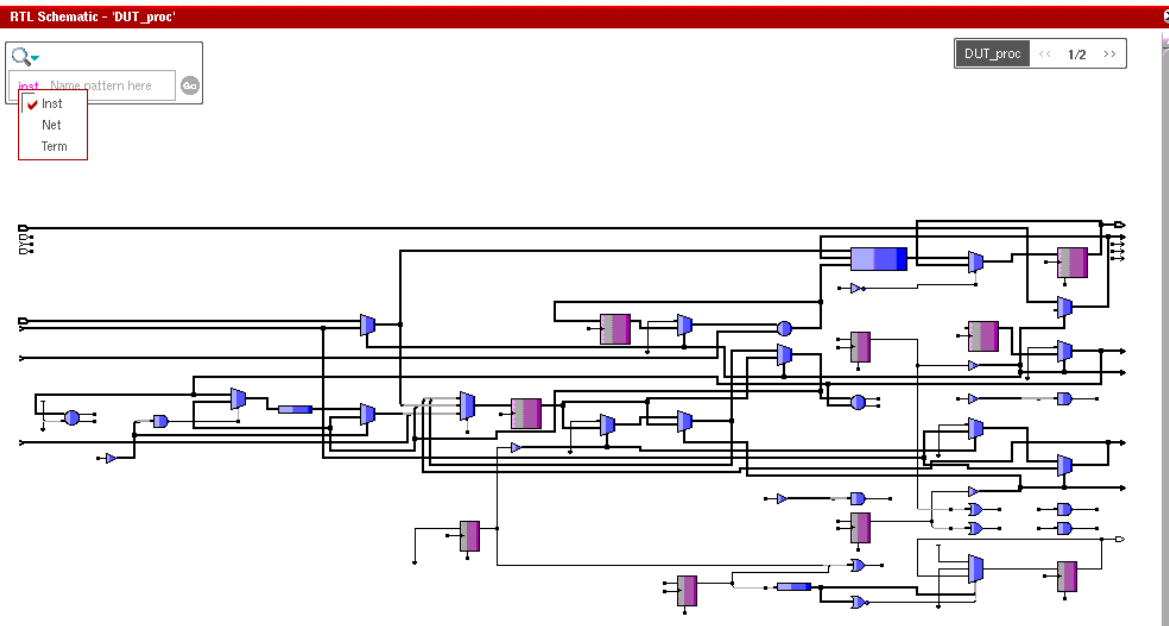
13.1.7.1 Understanding the RTL Schematic Viewer

The module-level RTL Schematic Viewer is a hierarchical representation of the Verilog modules and the primitive symbols represent registers, RTL IP blocks, arithmetic resources (adder, subtractor, and so on), logic resources (ands, ors), and multiplexers.

The sequential instances are displayed in red and combinational instances are displayed in blue. The scalar nets are displayed as a thin line; whereas, the bus or vector nets are displayed as a thick line. And, the module terminals are indicated by arrows, which indicate their direction (input, output).

For more information on the additional features, see [“Additional Features of the RTL Schematic Viewer” on page 13-25](#).

Figure 13-25 RTL Schematic Viewer of a Module



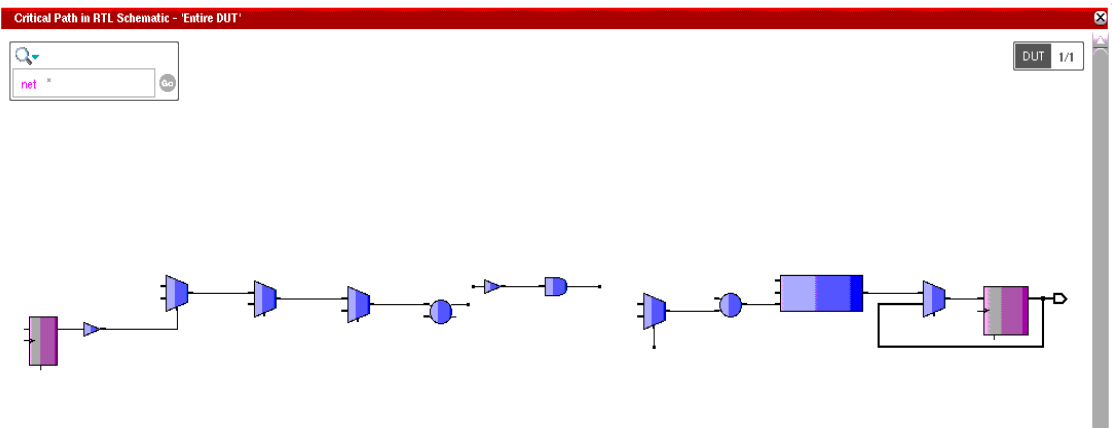
13.1.7.2 Viewing Critical Paths in RTL Schematic

To view the RTL Schematic of the critical paths, right-click anywhere in the **Cycle Analysis Viewer** (see [“Cycle Analysis Viewer” on page 13-20](#)), and select **Show Critical Path > in RTL Schematic**. The critical paths are displayed in RTL Schematic, as shown in [Figure 13-26 on page 13-25](#).

Additionally, to highlight a specific instance of the **Cycle Analysis Viewer** in an RTL Schematic, right-click the instance and select **Show Inst in RTL Schematic**. The RTL Schematic view of the entire design is displayed highlighting the selected instance.

For more information on the additional features, see [“Additional Features of the RTL Schematic Viewer” on page 13-25](#).

Figure 13-26 Critical Path in RTL Schematic



13.1.7.3 Additional Features of the RTL Schematic Viewer

In addition to the zoom feature on the *Tool Bar*, you can use the **Ctrl** button with the mouse wheel to zoom in and out.

The RTL Schematic Viewer includes the following options:

- [“Search for an Inst, Net, or Term” on page 13-26](#)
- [“Provides Tooltips” on page 13-26](#)
- [“Show Clock Signals, Reset Signals, and Instances/Nets in FSM” on page 13-27](#)
- [“Auto-Split of Large Schematics into Multiple Pages” on page 13-27](#)
- [“Print and Save an RTL Schematic View” on page 13-27](#)
- [“View Resources and Ops” on page 13-28](#)

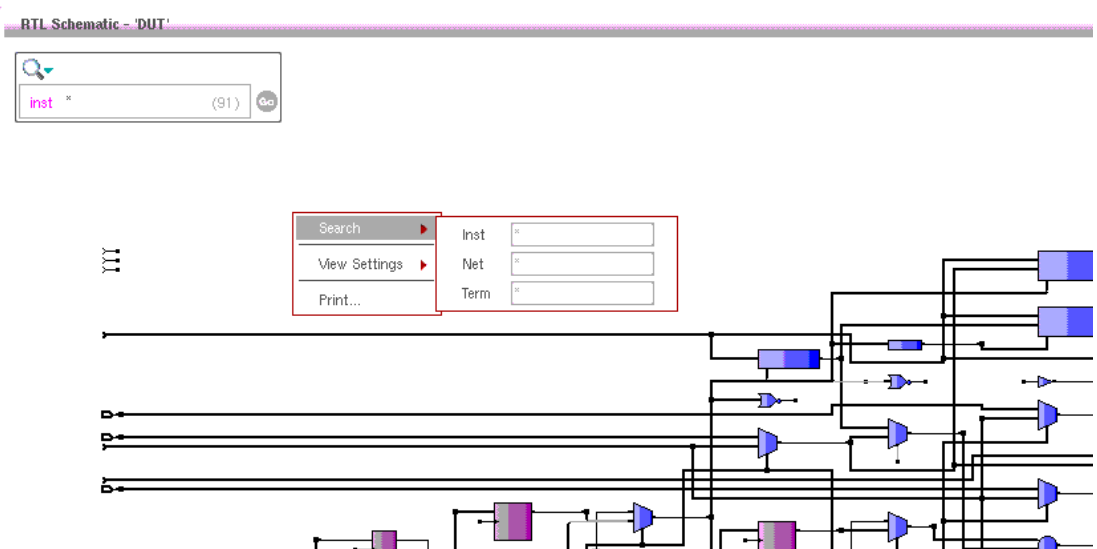
- “View Register Receivers in Input” on page 13-28

Search for an Inst, Net, or Term

The RTL Schematic Viewer includes a **Search** box. To search for a specific **Inst**, **Net**, or **Term**, click the **Search** icon that is included in the top left corner of the **RTL Schematic Viewer**. Then, select the type that you want to search, enter the name, and click **Go**.

Additionally, you can also search for an **Inst**, **Net**, or **Term** by right-clicking within the **RTL Schematic Viewer** (not on any symbol), select **Search**, and enter the name of the inst, net or term, as shown in [Figure 13-27 on page 13-26](#).

Figure 13-27 RTL Schematic Viewer - Search



If there is a matching item, the search box expands to show list of matching items. You can click the item to highlight the item in Red in the schematic. Once the list of matching items is listed, click the **Clear** button to delete the listing.

You can also search by the object type, **Inst**, **Net**, **Term** by clicking the object type and selecting from drop-down box. The default object type is **Inst**. Search for object supports regular expression matching.

To minimize or maximize the search box, click the magnifying glass icon.

Provides Tooltips

The **RTL Schematic Viewer** provides tooltips, which displays more information when you hover the cursor over an instance or a line.

Show Clock Signals, Reset Signals, and Instances/Nets in FSM

To show clock signals, reset signals, or instances and nets in FSM, right-click within the **RTL Schematic Viewer** (not on any symbol), select **View Settings**, and then select **Clock**, **Reset**, **FSM**, respectively, as shown in [Figure 13-28 on page 13-27](#).

Figure 13-28 RTL Schematic Viewer - View Settings



Auto-Split of Large Schematics into Multiple Pages

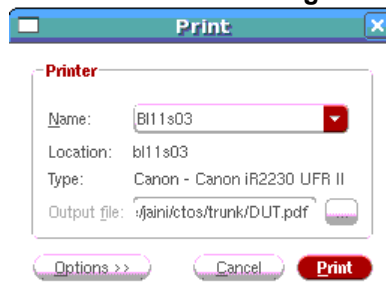
The top right corner of the **RTL Schematic Viewer** provides an indication of the current page and total number of pages for the RTL view. For example, the RTL schematic view shown in [Figure 13-24 on page 13-23](#) indicates << 1 / 2 >>, which means that there are 2 pages of view and the current display is that of page 1.

Users can move to next and previous page by clicking the left (<<) and right (>>) double arrows. Nets that flow across pages are indicated by page connector.

Print and Save an RTL Schematic View

To print an RTL Schematic view, right-click within the **RTL Schematic Viewer** (not on any symbol) and select **Print...**. The **Print** dialog is displayed, as shown in [Figure 13-29 on page 13-27](#). Select the printer in the **Name** combo box and click **Print**.

Figure 13-29 RTL Schematic Viewer - Print Dialog



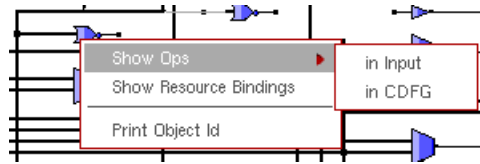
To save a PDF copy of the RTL Schematic view, select **Print to File (PDF)** in the **Name** combo box and click **Print**.

To specify additional options such as the number of pages, number of copies, or color mode for print, click **Options>>**.

View Resources and Ops

To view ops in the input source or CDFG, right-click any resource item and select **Show Ops > in Input** or **in CDFG**, respectively, as shown in [Figure 13-30 on page 13-28](#).

Figure 13-30 RTL Schematic Viewer - Show Ops



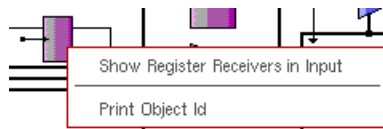
Show Resource Bindings

Print Object Id

View Register Receivers in Input

To view register receivers in the input source, right-click any register and select **Show Register Receivers in Input**, as shown in [Figure 13-31 on page 13-28](#).

Figure 13-31 RTL Schematic Viewer - Show Register Receivers in Input



13.1.8 Reporting Power and Area Using the Tree Map

The Tree Map is a preliminary feature.

As shown in Figure 13-32 on page 13-29 and Figure 13-33 on page 13-30, the CtoS GUI displays a *Tree Map* for power or area, when you select **Report->Power** or **Report->Area**, respectively.

Figure 13-32 Power Tree Map (Structural Level, Instance Level)

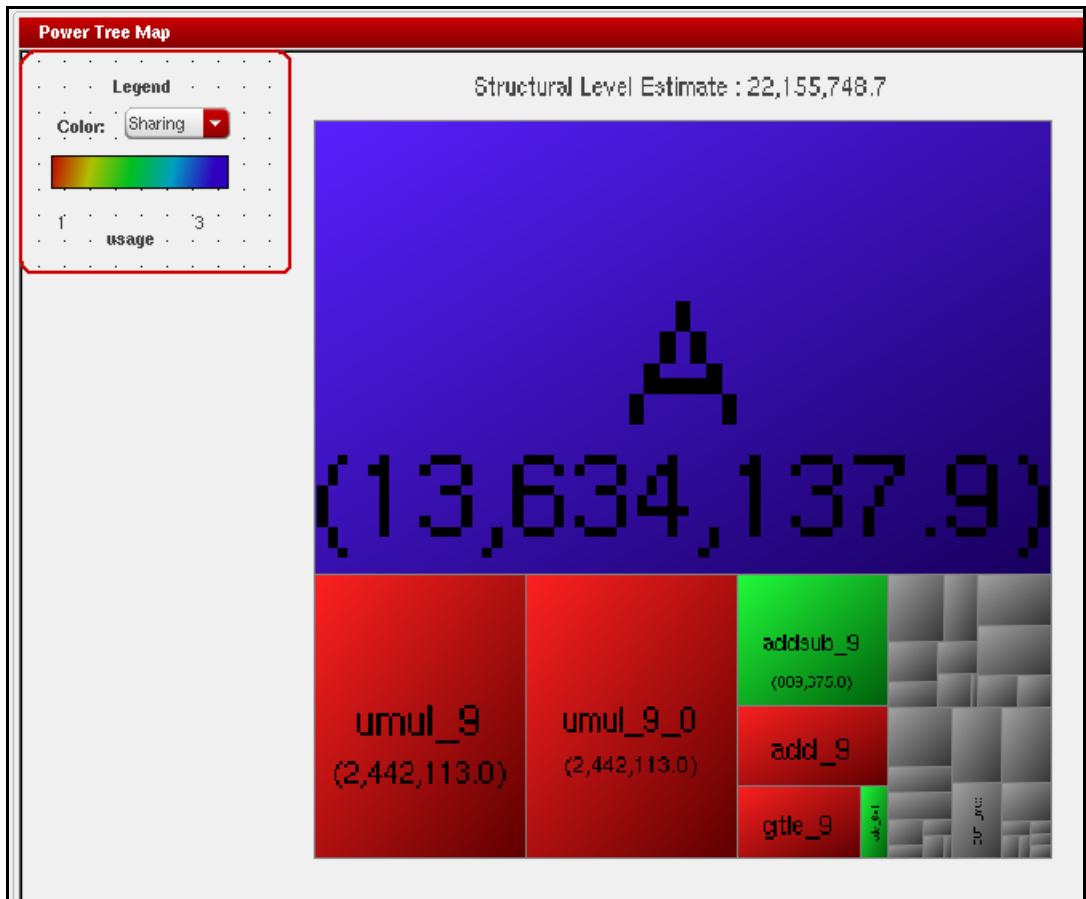
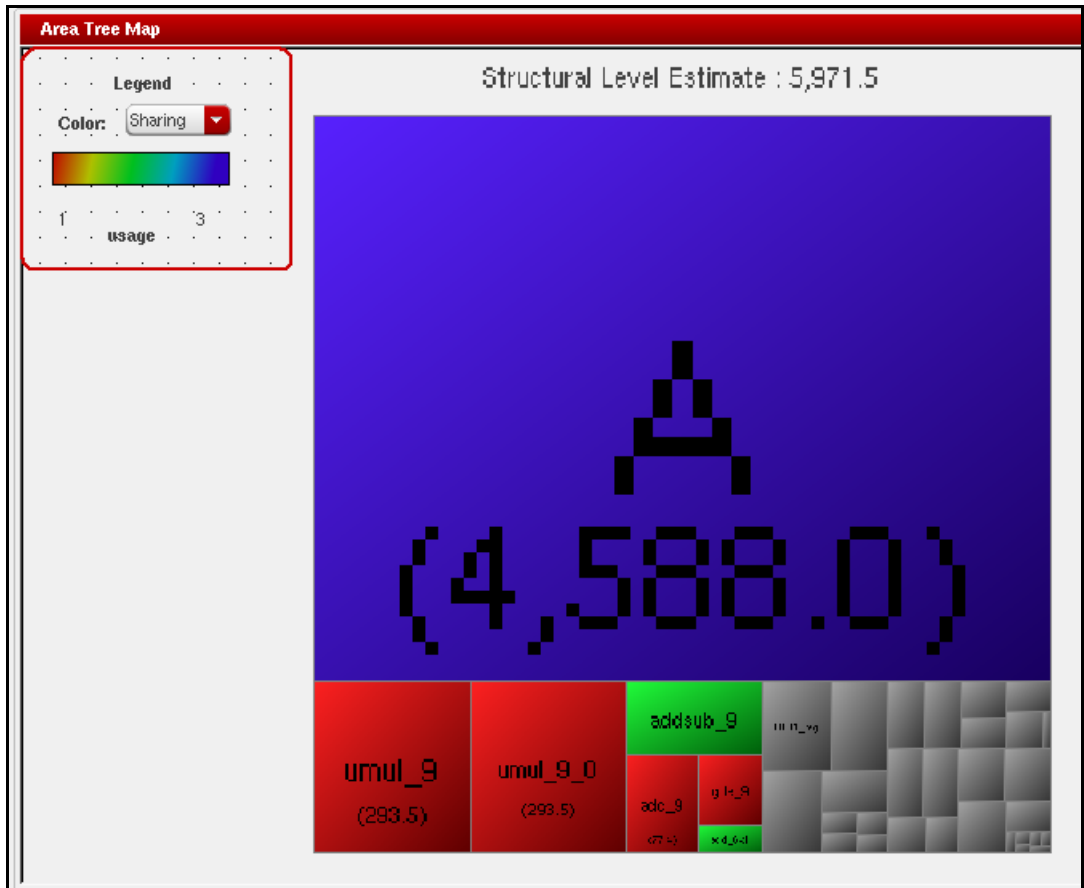


Figure 13-33 Area Tree Map (Structural Level, Instance Level)



The following sections describe the **Tree Map** in more detail:

- “Color Coding in the Tree Map” on page 13-31
- “Split-Layout Algorithm in the Tree Map” on page 13-33
- “Three-Level Display in the Tree Map” on page 13-33
- “Additional Features of the Tree Map” on page 13-34

13.1.8.1 Color Coding in the Tree Map

In the **Tree Map**, instances are color-coded by different cost functions: **Sharing**, **Slack**, and **Grade**.

This can be changed by double-clicking in the combo box in the **Legend**.

If any items are displayed at the instance level, you should see the color changing to reflect the new cost function.

Examples of color-coding are shown in [Figure 13-34 on page 13-32](#), [Figure 13-35 on page 13-32](#), and [Figure 13-36 on page 13-33](#).

- **Sharing:** Displays **red** for items not shared, gradually moving across the color spectrum (**red**, **yellow**, **green**, **cyan**, to **blue**) up to the instance most shared in this design.

Looking at the legend, you can see the value represented by **blue**.

All instances *not worth sharing* are marked in gray.

Not only are those the ones in the *Not Shareable* category, but also others (for example, muxes).

The minimum for sharing is 1 (not zero). Thus $\text{shared} == 1$ is displayed as **red**.

Note that, in the tool tip for instances, sharing will be specified only if the instance is worth sharing.

- **Slack:** Displays **green** for 0 slack.

Negative slack gradually moves across the color spectrum (**red**, **yellow**, to **green**) while positive slack is in the color spectrum (**green**, **cyan**, to **blue**).

Looking at the legend, you can see the values represented by **red** and **blue**.

If no items have negative slack, then the legend starts with **green**.

If no items have positive slack, then the legend ends with **green**.

- **Grade:** Displays **red** for items minimally graded, gradually moving across the color spectrum (**red**, **yellow**, **green**, **cyan**, to **blue**) up to the instance that is most graded.

Looking at the legend, you can see the values represented by **red** and **blue**.

Figure 13-34 Area Tree Map, Showing Color Coding

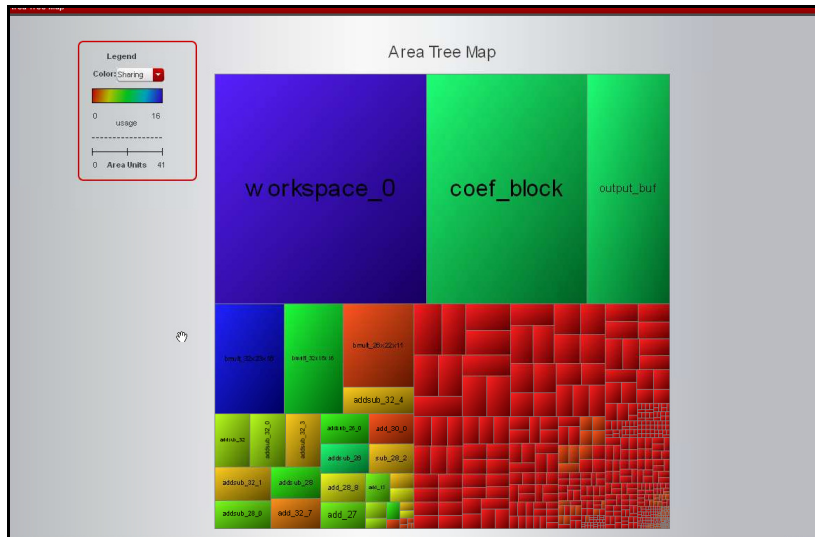


Figure 13-35 Power Tree Map, Showing “slack” as the Coloring Objective

Note the instance in red, which indicates negative slack.

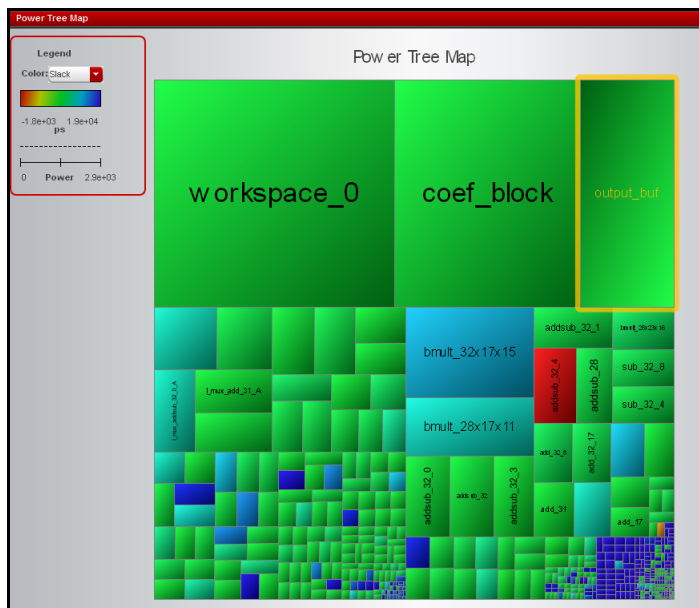
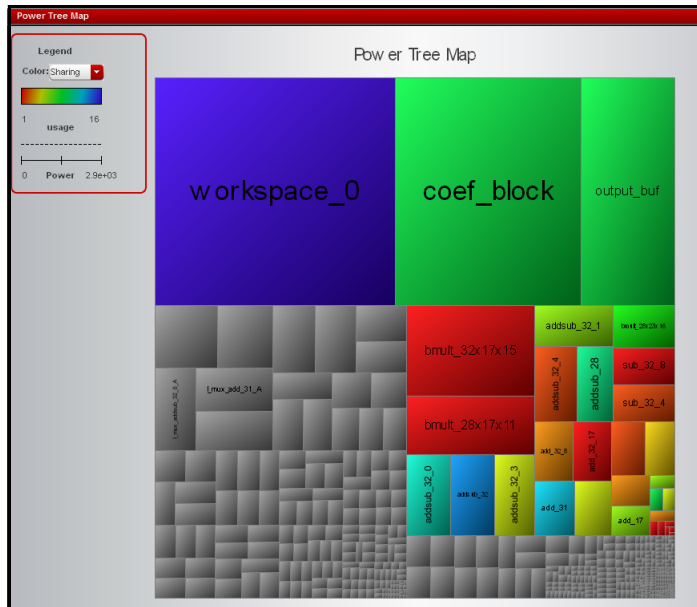


Figure 13-36 Power Tree Map, Showing “sharing” as the Coloring Objective



13.1.8.2 Split-Layout Algorithm in the Tree Map

The **Tree Map** has a *split-layout algorithm*, in which the size of each rectangle is proportional to the area of the instance resource.

The *progress bar* is updated while performing the layout of rectangles in the Tree Map, as well as coloring the map.

There is an output message to the *Command Window* for each behavior processed.

13.1.8.3 Three-Level Display in the Tree Map

The **Tree Map** is displayed at three levels:

- **Behavior:** Displays all behaviors in the module, as well as a *preview* of categories, shown by grey dotted lines.
- **Category:** Displays each of the resource types, as well as a *preview* of the instances, shown by grey dotted lines.
- **Instance:** Displays instances with coloring as specified by *color objective* (see “Color Coding in the Tree Map” on page 13-31).

13.1.8.4 Additional Features of the Tree Map

Here are some additional features of the **Tree Map**:

Categorization of Instance Hierarchy in the Tree Map

The **Tree Map** categorizes instance hierarchy in behaviors, then **resource_type**, as follows:

Memory, Flip Flop, Muxes, Shareable, NonShareable

Expanding and Collapsing in the Tree Map

On individual items in the **Tree Map**, you can **Expand/Collapse** these items by double-clicking. You can use the right-mouse context menu to **Expand**.

To change all to the same level, you can use the right-mouse context menu to **Show at Level X** actions.

Name Visibility, Saving, and Printing in the Tree Map

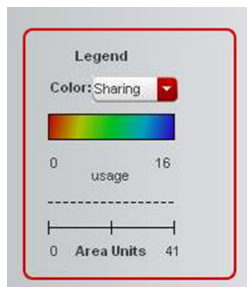
The context menu in the background of the scene lets you

- Toggle the name visibility
- Save to PNG
- Print

Zooming

In addition to the zoom feature on the *Tool Bar*, you can use the **Ctrl** button with the mouse wheel to zoom in and out. The **Legend** shows the scale of the current zoom level, as shown in [Figure 13-37 on page 13-34](#). It is also movable – you can simply click it and drag it to a new location.

Figure 13-37 Area Tree Map - Legend



13.1.9 Summary Report

From the icons along the left side of the main CtoS GUI window, select *Summary Report* (the bottom-most icon, which looks like a note pad), or select **Window -> Summary Report**, and you will see the **Summary Report**, as shown in [Figure 13-38 on page 13-35](#).

The **Summary Report** gives you a sense of the design complexity – both in terms of the number and kind of behavioral objects appearing in the source code and in terms of the number and kind of structural elements CtoS uses to implement that behavior.

Figure 13-38 Summary Report

Name	Count
Overview	
Name	DUT
'CLK' Clock Period (ps)	<u>10,000</u>
Minimum Slack (ps)	<u>1,903</u>
Area	<u>5,971.5</u>
Power	<u>22,155,748.3</u>
Behavior	
Modules	<u>1</u>
Processes	<u>1</u>
Functions	<u>0</u>
Arrays	<u>1</u>
Loops	<u>3</u>
States	<u>3</u>
Edges	<u>14</u>
Total Ops	<u>31</u>
Shareable Ops	<u>11</u>
Values	154
Tags	1
Structure	
Memories (bits)	<u>288</u>
Flip Flops (bits)	<u>36</u>
Muxes (bits)	<u>99</u>
Shareable Resources	<u>6</u>
Non Shareable Resources	<u>14</u>
Input Terminals (bits)	<u>38</u>
Output Terminals (bits)	<u>9</u>
Nets (bits)	254
Instance Terminals (bits)	646

Note You could also use the **report_summary** command (“[report_summary](#)” on page E-128).

13.2 Generating Models, Wrappers, RTL, SLEC after Scheduling

After scheduling, CtoS also automatically produces models, or you can manually produce them. You can also generate verification wrappers and RTL descriptions, as described in the following subsections:

- “Generating a Simulation Model after Scheduling” on page 13-36
- “Generating a Verification Wrapper after Scheduling” on page 13-36
- “Generating an RTL Description after Scheduling” on page 13-36
- “Generating a SLEC Script and XML File” on page 13-38
- “Generating a Top Wrapper for Exported Memories” on page 13-40

13.2.1 Generating a Simulation Model after Scheduling

In addition to generating a simulation model after a successful build, CtoS also automatically generates a simulation model after a successful scheduling and allocation of registers.

You may also *manually* generate a simulation model after scheduling. The only difference is that the **birthday** option is not required *after* scheduling, while it is required *before* scheduling.

Note See “Generating Models” on page 7-7.

13.2.2 Generating a Verification Wrapper after Scheduling

In addition to generating a verification wrapper before scheduling, you may also generate one after scheduling.

The generated wrapper uses SystemC code and has a parameterized constructor to allow different forms of instantiation.

Note See “Generating a SystemC Verification Wrapper” on page 7-11.

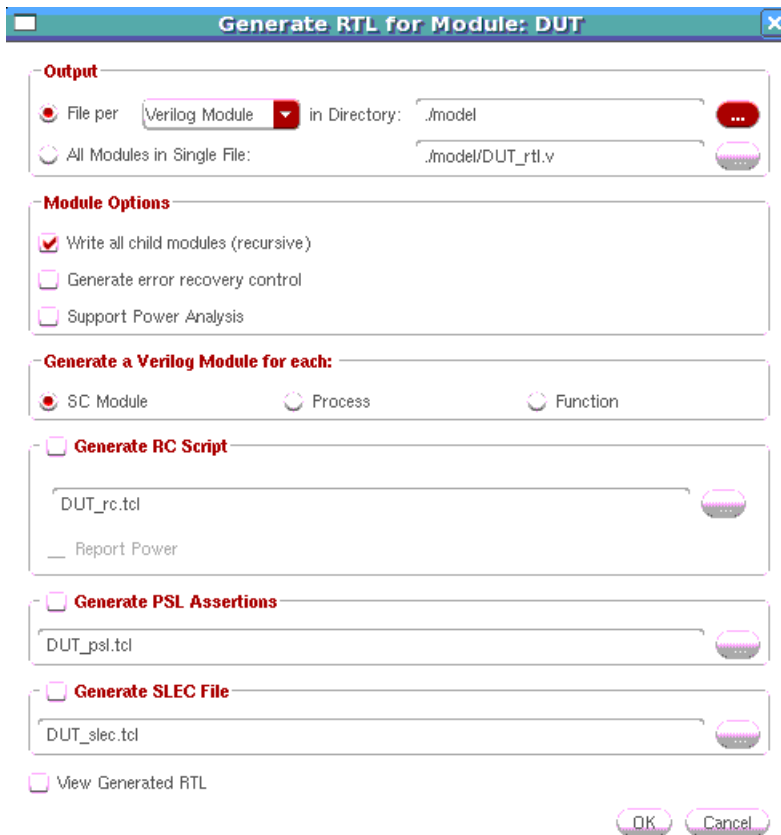
13.2.3 Generating an RTL Description after Scheduling

After the successful generation of a schedule, including the allocation of registers and netlist generation, for a module or all modules in a design, you can create a synthesizable RTL description for the specified module or all modules in a design.

Select **File -> Generate -> RTL**, and complete the **Generate RTL** dialog, as shown in [Figure 13-39 on page 13-37](#).

Note You can also use the **write_rtl** command (“[write_rtl](#)” on page E-161).

Figure 13-39 The Generate RTL Dialog



In the **Generate RTL** dialog, you would complete the fields, as follows:

- **Output:**
 - **File Per:** Select the **Verilog Module** option if you want CtoS to generate separate files for each Verilog module. Select the **SC Module** option if you want CtoS to generate a separate file for each SystemC module. The file for each SystemC module contains the Verilog module for the SystemC module and other modules referenced by the module such as for combinational functions and memory bridges. And, specify the directory name in which to generate models.
 - **All Modules in a Single File:** Select this option if you want CtoS to generate a single output file for all modules. Specify the filename with path to generate models.
- **Write all child modules (-recursive):** Specifies that a model be generated for the given database module, and every module transitively instantiated from that module.

By default, it is set to recursive. Unchecking this option will make the behavior non-recursive. CtoS will not descend into module hierarchy and a model is generated for the specified module only. For more information, see “[write_rtl](#)” on page E-161.

- **Generate error recovery control:** If the state machine ever enters an illegal state, the state machine will be reset, and an error will be signaled on the output terminal defined by the behavior attribute **error_recovery_terminal** (“[error_recovery_terminal](#)” on page D-42) [which can be set using the **define_control_error_terminal** command (“[define_control_error_terminal](#)” on page E-54) and cleared using the **undefine_control_error_terminal** command (“[undefine_control_error_terminal](#)” on page E-149)].
- **Support Power Analysis:** Generates RTL compatible with TCF (toggle count format) power analysis. TCF is the Cadence standard format to describe switching activity information in a design. The switching activity information contained in the file is required for accurate power analysis or power optimization of a design. This instructs RTL generation to promote all process-local registers up to the module scope. In addition, it causes the creation of extra registers, which are assigned a value of 1 in every clock cycle for which an edge or tag of the CDFG is active.
- **Generate a Verilog Module for each:** Maps SystemC module, process, or function constructs to verilog modules. The default value is **SC Module**, and CtoS generates one Verilog module for every user defined SystemC module.
- **Generate PSL Assertions:** Generates the properties of the synthesized design as PSL assertions. An assertion that the control FSM has a 1-hot encoding is produced.
- **Generate SLEC File:** Generates a script for verifying the generated RTL using SLEC and a SLEC XML file (see “[Generating a SLEC Script and XML File](#)” on page 13-38).
- **Generate RC Script:** Generates an RC script by calling the **write_rc_script** command (see “[Understanding RC Run Scripts](#)” on page 13-51).
- **Report Power:** In the generated RC script, if a TCF file has been read, generates power reporting commands (see **-report_power** in “[write_rc_run_script](#)” on page E-158).
- **View Generated RTL:** Specifies whether you want CtoS to display the new RTL in the **RTL** window.

13.2.4 Generating a SLEC Script and XML File

CtoS generates a SLEC script and a SLEC XML file, in addition to the RTL, if you select **Generate SLEC File** in the **Generate RTL** dialog, as shown in [Figure 13-39 on page 13-37](#). [You can also use the **-slec** option to the **write_top_wrapper** command (“[write_top_wrapper](#)” on page E-173).]

Important The SLEC flow does *not* support designs with multiple user-defined modules. The design must have a single user-defined module, which may result from collapsing many SystemC modules when the **build_flat** design attribute is *true*.

If the design has no exported memories, this is all you need to do.

However, if some or all of the vendor RAMs in the design are exported, see the following section, “Generating SLEC Files with Exported Memories” on page 13-39.

The characteristics of a CtoS-generated SLEC script are described in “Characteristics of a CtoS-Generated SLEC Script” on page 13-39.

13.2.4.1 Generating SLEC Files with Exported Memories

If a design has exported memories (either some or all are exported) use the following flows:

- “SLEC Flow when all Vendor RAMs Are Exported” on page 13-39
- “SLEC Flow when some but not all Vendor RAMs Are Exported” on page 13-39

SLEC Flow when *all* Vendor RAMs Are Exported

1. When you generate the RTL for the top module, do *not* select the **Generate SLEC File** group box. [If you did select it, CtoS would issue a warning that you should not use this XML because the module has exported memories.]
2. When you generate the top wrapper, *do* select the **Generate SLEC File** group box.

SLEC Flow when *some but not all* Vendor RAMs Are Exported

1. When you generate the RTL for the top module, *do* select the **Generate SLEC File** group box. In this scenario, an alternate model for the vendor RAMs that are *not* exported is included in the RTL (selection between the two models is done via `ifdef CALYPTO_SLEC`). However, you should not actually use the XML, and you will get an INFO message to remind you to re-generate the XML output (using either the **Generate Top Wrapper** dialog or the `write_top_wrapper` command).
2. When you generate the top wrapper, *do* select the **Generate SLEC File** group box. Use the generated XML from this step, rather than the XML generated as part of step 1.

13.2.4.2 Characteristics of a CtoS-Generated SLEC Script

A CtoS-generated SLEC script has the following characteristics:

- Comments documenting changes in loop latencies introduced by scheduling are included.
- If a loop is pipelined, the details of the loop’s initiation and latency intervals are also included.
- **-sample_start** is always used with **create_waveform**. If no specific value is required, the default is 0.

- I/O maps are generated according to Cadence's SLEC DFTL (detectable fixed throughput and latency) specification, that is, for each input read from and output written to in the steady-state loop, the following map is generated:

```
#####  
# I/O maps  
#####  
create_map -input \  
[create_waveform -sample_start L_in1_spec spec.in1] \  
[create_waveform -sample_start L_in1_impl impl.in1]  
create_map -output \  
[create_waveform -sample_start L_out1_spec spec.out1] \  
[create_waveform -sample_start L_out1_impl impl.out1]
```

L_in1_spec and **L_in1_impl** are latencies of **in1** in **spec** and **impl** respectively (similar for **out1**);

For example if the spec code is:

```
while(1) {  
x = in1.read();  
wait();  
out1.write(x);  
wait();  
}
```

In the implementation, if the loop is pipelined with **II**=1 and **LI**=6, and the output operation is constrained to the 5th pipeline stage, then the parameters should be:

```
L_in1_spec = L_in1_impl = 0  
L_out1_spec = 1  
L_out1_impl = 5
```

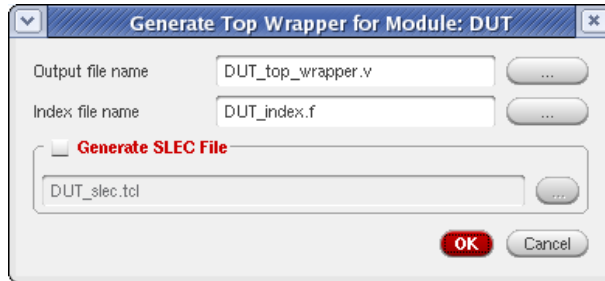
13.2.5 Generating a Top Wrapper for Exported Memories

If any of the memories in your design is exported outside the top model (that is, the **is_exported** array attribute is set to *true* for that memory), you need an additional *top wrapper* to connect the memories and the RTL design to your SystemC testbench.

Note See “Exporting Memories” on page 9-20 for more about exporting memories. Also note that built-in RAMs are not exported.

The top wrapper module instantiates all RAMs of a synthesized design, as well as the rest of the RTL of the design. The index file lists each of the RTL files of the design and is typically used as an **-F** argument for the simulator.

Figure 13-40 The Generate Top Wrapper Dialog



You can generate a top wrapper after you have generated an RTL description, by selecting **File -> Generate -> Top Wrapper** and completing the **Generate Top Wrapper** dialog, as shown in [Figure 13-40 on page 13-41](#).

Note You can also use the **write_top_wrapper** command ([“write_top_wrapper” on page E-173](#)).

13.3 Generating Gates in CtoS

CtoS allows you to generate gates, and to review the QoR, from CtoS-generated RTL. This feature is accessed through the **Synthesis Monitor** in the CtoS GUI, which, along with other aspects of this process, is described in the following sections:

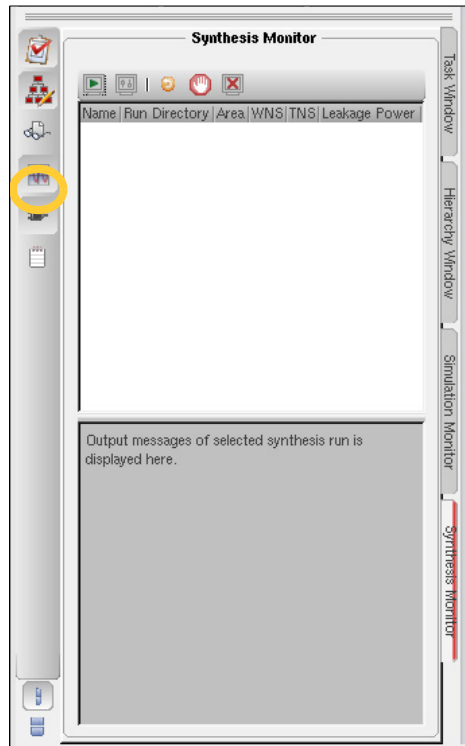
- “Using the Synthesis Monitor” on page 13-42
- “Generating Gates” on page 13-44
- “Configuring Logic Synthesis Runs” on page 13-46
- “Using Foundation Flows” on page 13-47
- “Understanding RTL Design Configurations” on page 13-47
- “Configuring Foundation Flows” on page 13-48
- “Using Plugins to Customize Foundation Flows” on page 13-50
- “Understanding RC Run Scripts” on page 13-51
- “Viewing Reports Generated by Default Synthesis Flow” on page 13-52
- “Generating RC Scripts” on page 13-53
- “Generating Gates from Logic Synthesis Makefiles” on page 13-53

13.3.1 Using the Synthesis Monitor

You set up synthesis runs, configure synthesis, and monitor synthesis progress using the **Synthesis Monitor**, as shown in [Figure 13-41 on page 13-43](#).

The **Synthesis Monitor** is displayed when you select, from the icons going down the left side of the CtoS GUI, the fifth icon from the top (the “gate symbol”).

Figure 13-41 Synthesis Monitor



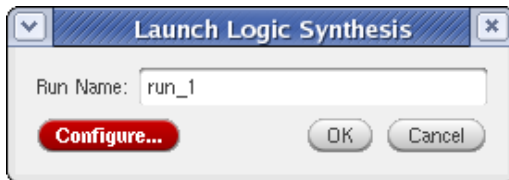
The five icons going across the top of the monitor – from left to right – let you:

- **Launch (green arrow)** a logic synthesis run to generate gates, as described in [“Generating Gates” on page 13-44](#).
- **Configure (flow chart)** a logic synthesis run, as described in [“Configuring Logic Synthesis Runs” on page 13-46](#)
- **Re-run Selected Item (circular arrow)** – if you select a row and click this button, CtoS will restart synthesis with the same configuration.
- **Stop Selected Item (stop sign)** – if you select a row and click this button, CtoS will stop the run.
- **Remove Selected Item (red X)** – if you select a row and click this button, CtoS will remove the run. If it is still running, then you will see a message box that will let you choose to also stop the run.

13.3.2 Generating Gates

To generate gates in CtoS, in the **Synthesis Monitor** (Figure 13-41 on page 13-43), select the first icon on the left (**green arrow**). The **Launch Logic Synthesis** dialog, as shown in Figure 13-42 on page 13-44, will be displayed.

Figure 13-42 The Launch Logic Synthesis Dialog



If you want to use the default run name and configuration, you can simply click **OK**. To customize either the run name or configuration, you can:

- enter a different name for the **Run Name**. This is a unique identifier for the synthesis run, so it must be a unique name within your CtoS session.
- customize the *Synthesis Configuration* by selecting the **Configure** button, which will display the **Configure Logic Synthesis** dialog (described in the next section, “Configuring Logic Synthesis Runs” on page 13-46).

When you are satisfied with your choices, click **OK**. CtoS will then:

- invoke RC using a *Foundation Flow* (see “Using Foundation Flows” on page 13-47) or an RC script (see “Generating RC Scripts” on page 13-53).
- after RC starts, display the entry in the top section, and the log file in the lower section, of the **Synthesis Monitor**, as shown in Figure 13-43 on page 13-45.
- while running, display the entry with a **blue** background color and display the **Name** of the run and the **Run Directory** (to identify where temporary and generated files will be written) in the **Synthesis Monitor**, again as shown in Figure 13-43 on page 13-45.

Figure 13-43 Synthesis Monitor during a Synthesis Run

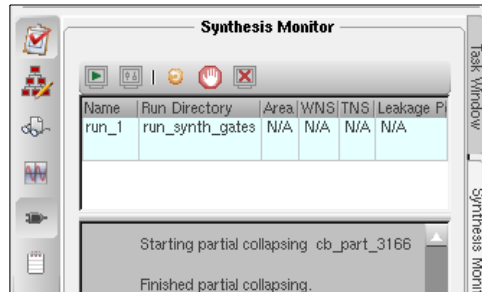
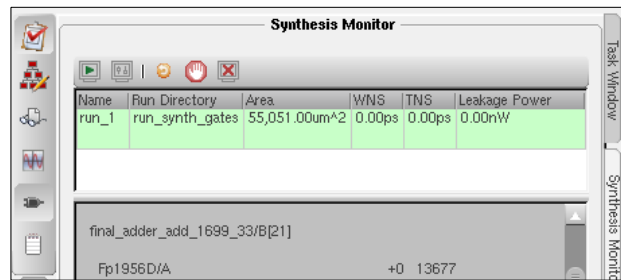


Figure 13-44 Synthesis Monitor after a Successful Synthesis Run



- when synthesis has completed, update the background color to **green** for success, as shown in [Figure 13-44 on page 13-45](#), and **red** for failure.
- if synthesis is successful, display the QoR for **Area**, **WNS** (Worst Negative Slack), **TNS** (Total Negative Slack), and **Leakage Power**, again as shown in [Figure 13-44 on page 13-45](#).
- if you are using a *Foundation Flow* and synthesis is successful, generate a **Gates File**, named `<topMod><VerilogRTLSuffix>.vg`, in the run directory specified in the *Synthesis Configuration*. The output for the *RC Script* method is dependent on the content of the *RC Script*.

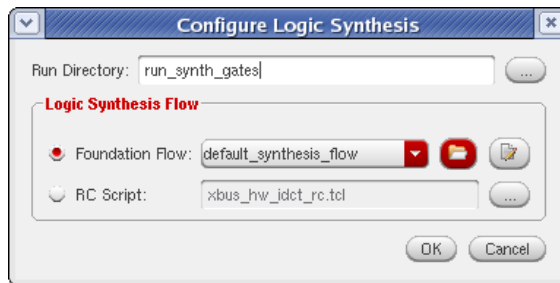
Note You can also use the **write_gates** command ([“write_gates” on page E-157](#)). Be aware that this Tcl command will block other processes until RC exits, whereas the **Synthesis Monitor** method will not.

13.3.3 Configuring Logic Synthesis Runs

To customize the configuration of your logic synthesis runs, use the **Configure Logic Synthesis** dialog, as shown in [Figure 13-45 on page 13-46](#).

This dialog is displayed when you select the second icon from the left in the **Synthesis Monitor** ([Figure 13-41 on page 13-43](#)) or the **Configure** button in the **Launch Logic Synthesis** dialog ([Figure 13-42 on page 13-44](#)).

Figure 13-45 The Configure Logic Synthesis Dialog



In the **Configure Logic Synthesis** dialog, you can specify a **Run Directory** and either a **Foundation Flow** or an **RC Script**, as described below:

- For the **Run Directory**, specify the directory in which you will run RC. The default is **run_synth_gates**. All temporary and generated files will be written to this run directory.
- For the **Foundation Flow**, specify your standard *Foundation Flow* for running synthesis. The default is **default_synthesis_flow**. The two icons to the right of the flow's name let you configure the flow. Details for *Foundation Flow* usage are described in [“Using Foundation Flows” on page 13-47](#).
- For the **RC Script**, browse to the desired RC script, using the browse (...) button. [See also the **write_rc_script** command ([“write_rc_script” on page E-159](#)).

When you click **OK**, the *Synthesis Configuration* is updated, if applicable, and if you have selected the *Foundation Flow*, several other files are created, as described in the section, [“Configuring Foundation Flows” on page 13-48](#).

Notes

- You can also use the **define_synth_config** command ([“define_synth_config” on page E-56](#)).
- The *Synthesis Configuration* objects are stored in the CtoS database in **synthesis_configs** (see [“Default Synthesis Configuration Object Attributes \(synthesis_configs\)” on page D-86](#)).
- You could also use the **write_rc_run_script** command ([“write_rc_run_script” on page E-158](#)).

13.3.4 Using Foundation Flows

A *Foundation Flow* is a standard baseline flow for generating gates from RTL that allows sophisticated customization. The main advantage of using a *Foundation Flow* is that inputs are separated from flow execution. This lets you use a single common baseline flow to execute on a number of different designs or revisions of the same design.

A *Foundation Flow* consists of the following:

- the *Foundation Flow Control File*, which is a list of steps to direct RC to generate gates (see “[Logic Synthesis Steps](#)” on page K-1 for the complete list of steps).
- the *RTL Design Configuration*, which describes the RTL to synthesize, including specification of RTL files and technology libraries and references to the SDC file (see “[Understanding RTL Design Configurations](#)” on page 13-47).
- optional *customization of the Foundation Flow* by
 - setting variables, in the *Synthesis Configuration*, such as effort levels for each synthesis step. CtoS shows you the default values and then lets you copy the defaults and modify them (see “[Configuring Foundation Flows](#)” on page 13-48).
 - inserting additional steps anywhere in the flow. CtoS shows you the empty plugins and lets you fill in the Tcl (see “[Using Plugins to Customize Foundation Flows](#)” on page 13-50).
- the *RC run script* to run the *Foundation Flow* in RC (see “[Understanding RC Run Scripts](#)” on page 13-51).

13.3.5 Understanding RTL Design Configurations

An *RTL Design Configuration* describes the RTL to synthesize.

This file is generated by the **write_rc_run_script** command (“[write_rc_run_script](#)” on page E-158), in the run directory specified in the *Synthesis Configuration*, and is named *topModule_rtl_config.tcl*.

An *RTL Design Configuration* specifies the following:

- RTL Verilog files for the design and the RTL IP
- technology libraries, which include those referenced by memories in the design
- constraints representing clocks and external delays, using an SDC file, named *topModule_rc.sdc*
- RC startup script, so it matches what is applied during CtoS resource characterization.
- (optionally) CtoS clock gating insertion, consistent with the use of the **low_power_clock_gating** design attribute (“[low_power_clock_gating](#)” on page D-17).
- (optionally) TCF file for the power flow declared if the TCF file has been read using the **read_tcf** command (“[read_tcf](#)” on page E-98), and the **-report_power** option has been specified with the **write_rc_run_script** command (“[write_rc_run_script](#)” on page E-158).

13.3.6 Configuring Foundation Flows

As mentioned in the previous section, “[Configuring Logic Synthesis Runs](#)” on page 13-46, you can select a specific *Foundation Flow* using the **Configure Logic Synthesis** dialog ([Figure 13-45 on page 13-46](#)). This dialog also provides two icons to the right of the *Foundation Flow*’s name that let you:

- **View** the selected *Foundation Flow* with an external editor (see “[Using External Editors on SystemC Files](#)” on page 6-36).

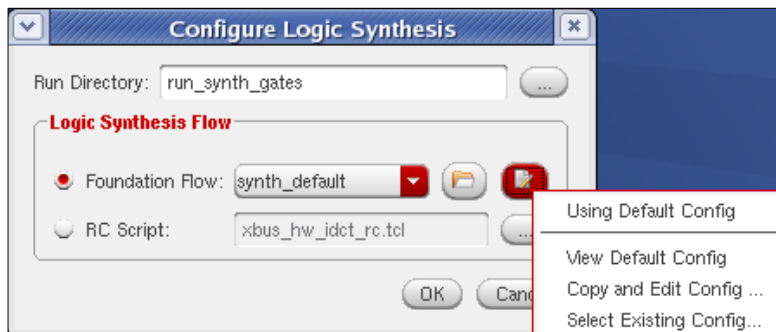
Note A set of standard *Foundation Flows* is provided in the following directory:

```
install_dir/share/synth/ff_install/flows
```

- **Configure** certain variables for your *Foundation Flow*. The **default_setup.tcl** file defines default values for these variables. You can simply use the default values, or you can use a modified version of this file. Examples of configuration file variables include effort levels for various synthesis steps, such as **syn2gen_effort** [see “[Logic Synthesis Steps](#)” on page K-1 for a list of all logic synthesis steps].

You can also insert additional steps into the flow by defining pre- or post-actions (plugins). This is useful for additional report generation or QoR analysis. For more detail, see the following section, “[Using Plugins to Customize Foundation Flows](#)” on page 13-50.

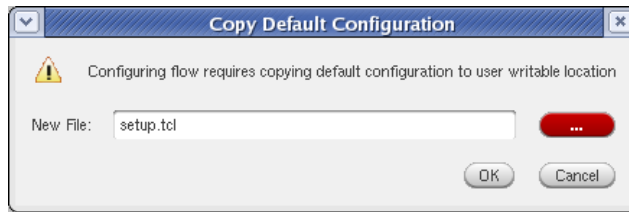
Figure 13-46 The Configure Logic Synthesis Dialog, showing Configuration Options



When you right-click on the **Configure** icon, you will see the following choices, as shown in [Figure 13-46 on page 13-48](#), for configuring this file:

- **View Default Config** lets you view the default values for the *Foundation Flow* variables, using an external editor in read-only mode.
- **Copy and Edit Config** lets you customize the default configuration file (**default_setup.tcl**). You are first prompted for a new filename with the **Copy Default Configuration** dialog, as shown in [Figure 13-47 on page 13-49](#). Then, the **default_setup.tcl** file is copied to this new filename, and an external editor is started in edit mode so you can customize the new file.

Figure 13-47 The Copy Default Configuration Dialog

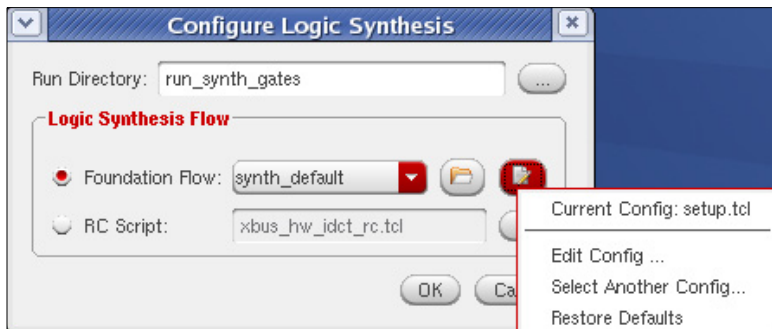


- **Select Existing Config** lets you select an existing configuration file.

If you have selected a user-specified configuration file, when you right-click the **Configure** icon, the choices, as shown in [Figure 13-48 on page 13-49](#), will change to:

- **Edit Config** lets you edit the current configuration file.
- **Select Another Config** lets you select a different configuration file.
- **Restore Defaults** lets you revert to the default configuration file.

Figure 13-48 The Configure Logic Synthesis Dialog, after Specifying User Configuration



Notes

- You could also use the **define_synth_config** command ("[define_synth_config](#)" on page E-56).
- The *Synthesis Configuration* objects are stored in the CtoS database in **synthesis_configs** (see "[Default Synthesis Configuration Object Attributes \(synthesis_configs\)](#)" on page D-86).

13.3.7 Using Plugins to Customize Foundation Flows

Foundation Flows can be customized without having to edit the *Foundation Flow Control File*.

One way to customize a flow is by using *plugins*.

In the default synthesis flow, every step supports a pre-plugin (**pre_step_name**) and a post-plugin (**post_step_name**), for example, for loading libraries, you would have the following:

```
step_name = ff_load_library
pre-plugin = pre_ff_load_library
post-plugin = post_ff_load_library
```

Note See [Appendix K “Logic Synthesis Steps”](#) for a list of all of the steps in the logic synthesis flow.

The advantage of using plugins is that they can be written for specific flow needs, yet they still take advantage of the basic *Foundation Flow* infrastructure.

Plugins are implemented as Tcl procs that call RC-specific commands to produce the desired function.

The full RC command set is available, including database access commands, reporting commands, manipulation commands, as well as the full Tcl interface.

From a scripting perspective, there is no difference between a plugin and a user-defined operation in a standard RC Tcl script.

Any script that executes in an RC shell should be able to execute as a plugin.

To enable a plugin to be run before or after a particular step in the *Foundation Flow*, the plugin must be set by a foundation variable.

For example, to generate custom reports at the end of a synthesis run, you can define a **post_ff_syn_incr** plugin to be your Tcl proc **my_report_gen**, as follows:

```
set var(FFF::post_ff_syn_incr) my_report_gen
```

where **my_report_gen** is a Tcl proc sourced in the startup script, and **set var** is called in the *Synthesis Configuration* (as described in [“Configuring Foundation Flows” on page 13-48](#)).

13.3.8 Understanding RC Run Scripts

An *RC Run Script* instructs RC to generate gates using a *Foundation Flow* or a *Direct RC Script*.

An RC Run Script is generated by the **write_rc_run_script** command (“[write_rc_run_script](#)” on page E-158). The script is named **rc_run.tcl** and is written to the run directory specified in the *Synthesis Configuration*.

This script works differently, depending on whether you are using a *Foundation Flow* or a *Direct RC Script*:

- “[RC Run Script for a Foundation Flow](#)” on page 13-51
- “[RC Run Script for a Direct RC Script](#)” on page 13-52

13.3.8.1 RC Run Script for a Foundation Flow

An *RC Run Script* directs RC to run a *Foundation Flow*, as follows:

- loads the *Foundation Flow Control File* (see also “[Using Foundation Flows](#)” on page 13-47).
- loads the *RTL Design Configuration*, named **topModule_rtl_config.tcl**, which describes the RTL to synthesize (see also “[Understanding RTL Design Configurations](#)” on page 13-47).
- loads the *User Configuration File* that customizes the flow – pre-action; post-action; override variables (see also “[Configuring Foundation Flows](#)” on page 13-48).
- runs the *Foundation Flow Control File* to generate gates.

Notes

- The *Foundation Flow Control File* is specified in the *Synthesis Configuration*.
- The *RTL Design Configuration* is automatically generated by the **write_rc_run_script** command (“[write_rc_run_script](#)” on page E-158).

Note The *SDC File*, named **topModule_rc.sdc**, which represents the clocks and external delays, referenced by the *RTL Design Configuration*, is also automatically generated by the **write_rc_run_script** command.

- The *User Configuration File* is specified in the *Synthesis Configuration*.

13.3.8.2 RC Run Script for a Direct RC Script

An *RC Run Script* directs RC to run a *user-specified RC Script*, as follows:

- changes the directory to the location of the *user-specified RC Script*, which ensures that relative file paths within the *user-specified RC Script* are interpreted relative to the location of this script.
- runs the *user-specified RC Script*.

The **write_hdl** RC command, called from within the *user-specified RC Script*, is extended to not only write the gates file, but also to write the *QoR Metrics File*, which can be displayed in the **Synthesis Monitor**.

Notes

- The *user-specified RC Script* is specified in the *Synthesis Configuration*.
- You can use the **write_rc_script** command ("**write_rc_script**" on page E-159) to generate an *RC Script* for a design (see "[Generating RC Scripts](#)" on page 13-53).

13.3.9 Viewing Reports Generated by Default Synthesis Flow

The default logic synthesis flow generates the following reports in the run directory specified in the *Synthesis Configuration*:

- **html/metrics.xml** – the XML file report describing the final QoR (a subset is displayed in the **Synthesis Monitor**)
- **reports/opMod_rtl.summary_qor.html** – the HTML file report, which can be displayed in an internet browser, describing the QoR for all stages
- **reports/syn2gen.qor** – the QoR report after the **syn2gen** stage
- **reports/syn2map.qor** – the QoR report after the **syn2map** stage
- **reports/syn2map_incr.qor** – QoR report after the **syn2map incr** stage
- **reports/full_time_info.rpt** – the runtime performance of the RC run, including total time and memory usage for each stage
- **lec/do** – the files for running LEC to prove logic equivalency between RTL and the gate-level netlist.

13.3.10 Generating RC Scripts

You can generate *RC Scripts* in the following ways:

- by selecting **Generate RC Script** in the **Generate RTL** dialog, as shown in [Figure 13-39 on page 13-37](#), or by using the **write_rc_script** command ([“write_rc_script” on page E-159](#)).
- by running the *Foundation Flow* (**write_foundation_template**), which is enabled by the *Foundation Scripts* shipped with RC. The *Foundation Flow* generates an RC script named **synthesis.tcl** in the run directory.
- by manually writing them.

After generating an RC script, you can use it to generate gates, by selecting it in the **Configure Logic Synthesis** dialog (see [“Configuring Logic Synthesis Runs” on page 13-46](#)) and then following the instructions in [“Generating Gates” on page 13-44](#).

Note If the **low_power_clock_gating** design attribute ([“low_power_clock_gating” on page D-17](#)) is set to *true*, the following line will be added to an RC script:

```
set_attr lp_insert_clock_gating true
```

13.3.11 Generating Gates from Logic Synthesis Makefiles

If you would like to run **make** from a Unix prompt or regression system, you can create a logic synthesis makefile using the **write_synth_makefile** command ([“write_synth_makefile” on page E-169](#)).

The **write_synth_makefile** command bases the makefile on the *Synthesis Configuration* of the specified design [see [“Default Synthesis Configuration Object Attributes \(synthesis_configs\)” on page D-86](#)].

You can edit these settings using the **define_synth_config** command ([“define_synth_config” on page E-56](#)).

The following sections describe the logic synthesis makefile:

- [“Targets of CtoS-Generated Synthesis Makefiles” on page 13-54](#)
- [“Overriding Certain Variables in CtoS-Generated Synthesis Makefiles” on page 13-54](#)
- [“Example of Synthesis Makefile for Foundation Flow” on page 13-55](#)

13.3.11.1 Targets of CtoS-Generated Synthesis Makefiles

There is only one target for a CtoS-generated logic synthesis makefile: **synth_gates**.

The **synth_gates** target:

- calls RC in 64-bit mode. By default, it runs RC from the same install directory in which CtoS generated the makefile.
- runs the RC Run Script named *run_dir/rc_run.tcl*
- generates the following output in the run directory specified in *Synthesis Configuration*:
 - a gates file with the name *<topMod><VerilogRTLSuffix>.vg*, where *<VerilogRTLSuffix>* is the design object attribute, **verilog_rtl_model_suffix** ([“verilog_rtl_model_suffix” on page D-26](#)).
 - (only for a *Foundation Flow*) additional reports (see [“Viewing Reports Generated by Default Synthesis Flow” on page 13-52](#)).
- writes the following log files to **LOG_DIR**:
 - **synth_gates.log** (RC log file)
 - **synth_gates.cmd** (RC commands)

13.3.11.2 Overriding Certain Variables in CtoS-Generated Synthesis Makefiles

You can override the following variables in CtoS-generated synthesis makefiles at the command line:

Table 13-1 Variables in Synthesis Makefiles that can Be Overridden

variable	description	default
RC_INSTALL_DIR	provides the directory from which to run RC	CTOS_ROOT
LOG_DIR	provides the directory for log and command files	<i>run_dir</i> /LOGS
SYNTH_EXIT	provides the command to run after generating gates	exit with -post exit

For example, the following line would prevent RC from exiting:

```
make synth_gates SYNTH_EXIT=
```

13.3.11.3 Example of Synthesis Makefile for Foundation Flow

Here is an example of a CtoS-generated logic synthesis makefile when the *Synthesis Configuration* specifies a *Foundation Flow*:

```
#####
# This is a stand-alone makefile for running logic synthesis in batch mode.
#
# #####
# ***** RC Section *****
# #####
RC_INSTALL_DIR ?= /home/user/Install
RC_EXE = $(RC_INSTALL_DIR)/tools/bin/rc
RUN_DIR = .
LOG_DIR ?= ${RUN_DIR}/LOGS
SYNTH_EXIT ?= -post exit
# #####
# ***** Rules and Targets Section *****
# #####
all: synth_gates

setup: info clean
      mkdir -p $(LOG_DIR)

info:
      @echo Test run at `date` on `hostname`
      @echo Test run in `pwd`

clean:
      rm -rf core* $(LOG_DIR)/run_synth_gates.log $(LOG_DIR)/run_synth_gates.cmd

${RUN_DIR}/my_design_rtl.vg:    ${RUN_DIR}/rc_run.tcl
                                ${RUN_DIR}/my_design_rtl_config.tcl
      @echo "=====
      @echo "Synthesize Gates"
      @echo "=====
      mkdir -p ${RUN_DIR}
      mkdir -p ${LOG_DIR}
      rm -f ${LOG_DIR}/synth_gates.log
      rm -f ${LOG_DIR}/synth_gates.cmd
      ${RC_EXE} -nogui -f ${RUN_DIR}/rc_run.tcl \
        -64 -log ${LOG_DIR}/synth_gates.log \
        -cmd ${LOG_DIR}/synth_gates.cmd \
        $(SYNTH_EXIT)

synth_gates:    ${RUN_DIR}/my_design_rtl.vg
```


14 Authoring SystemC for CtoS

This chapter describes the synthesizable subset of SystemC. It also describes the coding style recommended for getting the greatest benefits from CtoS, as well as for minimizing design errors.

CtoS accepts a wide range of C++ and SystemC coding styles as input. From the rich expressiveness of C++, this may include templates, classes, user-defined types, and some pointer usage. SystemC extends C++ to add many convenient features for describing intended hardware behavior. This includes a standardized way to define design interfaces and bit-slicing of integer data types, plus the ability to control concurrency and process communications.

CtoS transforms this input code into models of varying levels of abstraction, from fast I/O cycle-accurate simulation models, down to synthesizable Verilog RTL. These models can then be synthesized using a traditional logic synthesis and physical design flow. CtoS also generates assertions, utility files, and scripts to aid in the downstream verification flow.

To perform these transforms, CtoS must necessarily impose some restrictions on coding styles in order to create predictable, consistent results. Most of these restrictions can be thought of as allowing CtoS to statically determine the intended behavior. For example, objects dynamically created at the runtime of the program do not have a corresponding structure in a persistent hardware implementation, and thus must be excluded from the input of CtoS.

Verifying Timed vs. Untimed Models

Verifying the output of CtoS is an important part of the design process. In addition to generating Verilog RTL, CtoS can generate a SystemC or Verilog simulation model at any point after the original source has been read. This can help you verify and explore design alternatives after each step of the process.

A cycle-accurate model (that is, a design with a timed top-level interface, but whose bulk is pure untimed C++ function calls) is relatively easier to verify than an untimed model, since outputs can be compared at each clock using the verification wrapper file generated by CtoS. Alternately, an untimed or partially timed model is easier to write and gives CtoS more flexibility to synthesize a design, but requires more care in verification, since outputs could arrive at different clock cycles. (See [“Verifying Designs” on page 7-1](#) for more detail on this subject.)

Requirements for Designs Being Input into CtoS

All designs for input into CtoS must have exactly one top-level **sc_module**. This input model must have a signal-level interface by declaring its ports using **sc_in** and **sc_out**. If you are using *Transaction-level Modeling (TLM)*, the transaction-based *model under test* must be instantiated with a transactor that converts transactions to signals at the top-level.

CtoS will generate a corresponding model with the same I/O to communicate with the outside environment or with higher-level modules if the design is hierarchical. The top-level **sc_module** input into CtoS will then contain one or more processes or other **sc_module** instances, plus the communication between them, which could be **sc_signal** data types or TLM interfaces. C++ is used to describe the internal behavior of the processes, adding SystemC where convenient.

The constructor for the **sc_module** can construct (1) **sc_module** instances declared as members of this **sc_module**, (2) define connectivity of the **sc_module** instances, and (3) register processes defined in this module. While a constructor of an **sc_module** can specify more than this, CtoS currently supports only these three definitions made in a constructor of an **sc_module**. SystemC **sc_signal** data types provide a race-free way to communicate between SystemC processes.

This top-level **sc_module** must then be instantiated so CtoS can know the top level of the design or design hierarchy to synthesize.

For example, you can instantiate the top module using a macro, called **SC_MODULE_EXPORT**, provided in INCISIV, as follows:

```
SC_MODULE_EXPORT(top_level_module)
```

You must also set two attributes:

- **source_files** (“[source_files](#)” on page D-24), which defines the SystemC source file that includes this instantiation, as well as other source files for your design to be input into CtoS.
- **top_module_path** (“[top_module_path](#)” on page D-25), which specifies the **instance** name (see “[Instantiation of the Top Module](#)” on page 14-4).

Notes

- Review the example in the “[Using Reports to Analyze Designs](#)” on page 13-2 to see how **DUT.h** uses this type of instantiation and **ctos_setup.tcl** sets the attributes:
- Review the example in the “[CtoS Tutorial for ASIC designs](#)” on page A-1 to see how the instantiation is made in **src/xbus_hw_idct.cc**, and the attributes are set in **ctos.tcl**.

Alternatively, if the top-level **sc_module** is already instantiated in your SystemC code, you do not need to explicitly instantiate it again for CtoS. Instead, you can include the SystemC file that instantiates the top-level module in the **source_files** design attribute and set the instance of the top-level module in the **top_module_path** design attribute.

For example, suppose in an `sc_main()` function, a module called **whole_design** is instantiated:

```
int sc_main(int argc, char *argv[]) {
    ...
    whole_design whole_design_instance("whole_design_name");
    ...
}
```

Also, in a module **whole_design**, an `sc_module` called **hardware_top** is instantiated:

```
SC_MODULE(whole_design) {
    ...
    hardware_top hardware_top_instance;
    ...
    SC_CTOR(whole_design) :
    ...
    , hardware_top_instance("hardware_top_instance_name")
    ...
}
```

To specify the **hardware_top_instance** as the top-level module for CtoS, you would set the **top_module_path** attribute to **whole_design_instance.hardware_top_instance** and the **source_files** attribute to a list of SystemC source files that include the functions and constructors that instantiate all of the modules given in the path specified in **top_module_path**.

Details of How CtoS Handles SystemC Components

The following sections provide detailed descriptions of how CtoS handles specific SystemC components:

- “Modules” on page 14-4
- “Ports” on page 14-6
- “Processes” on page 14-9
- “Mealy Communication” on page 14-37
- “Inputs, Outputs, and Multiple Drivers” on page 14-37
- “Variables” on page 14-37
- “External Arrays” on page 9-22 [in the “Allocating Memory and RTL IP” on page 9-1 chapter]
- “Object Models” on page 14-45
- “Data Types” on page 14-53
- “Using the CtoS Pragmas” on page 14-71
- “Constructs” on page 14-85
- “Using C++ Labels” on page 14-94
- “Coding Tips for High Quality Synthesis” on page 14-98
- “Specifying Synthesis-Specific and Simulation-Specific Code (`__CTOS__` macro)” on page 14-102
- “SystemC Standard Language Restrictions” on page 14-106

Note The term *LRM* refers to the *IEEE Standard 1666-2005 for SystemC Language Reference Manual*, which describes the SystemC language.

14.1 Modules

A module is the primary unit of synthesis. A SystemC module is simply a C++ class that derives from SystemC class **sc_module**, which can be conveniently specified using the **SC_MODULE** macro:

```
::::::::::::
my_top_module.h
::::::::::::
#include "systemc.h"

SC_MODULE(my_top_module) {
    sc_in<bool>  clk;
    sc_out<bool> rst;
    ..

    SC_CTOR(my_top_module)
        clk("clk"),
        rst("rst")
    {
        ..
    }
    ..
private:
    void      main();
};
::::::::::::
```

This section has the following subsections:

- “[Instantiation of the Top Module](#)” on page 14-4
- “[Module Constructor](#)” on page 14-5

14.1.1 Instantiation of the Top Module

CtoS requires that the source code of a design contain an instance of the top SystemC module to be synthesized and that the path to this instance be specified in the **top_module_path** design attribute (“[top_module_path](#)” on page D-25). A convenient way to instantiate the top module is by using the **SC_MODULE_EXPORT** macro in the **.cpp** file associated with this module. In the following example, the value for design attribute **top_module_path** is *my_top_module*:

```
::::::::::::
my_top_module.cpp
::::::::::::
#include "my_module.h"
..
void
my_top_module::main()
{
    ..
}
```

```

}
..
#ifdef __CTOS__
SC_MODULE_EXPORT(my_top_module);
#endif
:::

```

14.1.2 Module Constructor

A *module constructor* is simply a constructor of the C++ class that defines the module. A module constructor can be specified most conveniently using the **SC_CTOR** macro within the module class definition (as shown in “Modules” on page 14-4). This is appropriate if the module constructor has a single formal argument of type **sc_module_name**. If the constructor requires additional arguments you need to define the module constructor without using the **SC_CTOR** macro.

The information that CtoS infers from the module constructor is as follows:

- The processes of the module.
- The static sensitivity of processes.
- The reset specification of thread processes.
- The asynchronous reset events of **SC_METHOD** processes.
- The port bindings of modules instantiated within given module.
- Dynamically allocated ports, signals, and module.
- The values of module fields that are of type **const bool**, **const int**, and **const unsigned int**.

CtoS infers the above information by symbolically simulating the constructor call associated with the instance of the top module specified by the **top_module_path** design attribute. This analysis supports a more limited subset of C++/SystemC than the supported subset for constructs appearing in processes or functions called from processes.

Apart from the SystemC constructs for specifying processes, static sensitivity and reset specifications (which are discussed in section 14.2), CtoS supports the following constructs in module constructors and functions called from module constructors:

- Port bindings
- The control statements: **if**, **else**, **for**, **continue**, **break**, and **return**,
- Calls to user-defined functions
- Declarations of variables of type **bool**, **int**, and **unsigned int**.
- Initializations of module fields of type **const bool**, **const int**, and **const unsigned int**.

- Dynamic allocation of modules, ports, and signals.

The analysis of module constructors ignores the following constructs:

- Initializations of non-const fields of a module.
- Initializations of const **T** fields, where **T** is a SystemC datatype, an enumerated type, or a user-defined class/struct, or array.
- Declarations of local variables of type **T**, where **T** is a SystemC datatype, an enumerated type, or a user-defined class/struct or array.

The analysis of module constructors does not support the following constructs and the presence of any of these constructs in a module constructor, or a function called from a module constructor will result in the design to be rejected:

- Control statements **while**, **do**, **switch**, **case**, **try**, **catch**.
- Control statements that depend on expressions of types different from **bool**, **int**, and **unsigned int**.

The set of constructs supported in module constructors is sufficiently expressive to describe modules where the port bindings, processes, and the number of ports, signals, and sub modules is configurable as a function of template parameters (of type **int** or **bool**), or formal arguments of the constructor, where the formal arguments are of type **bool**, **int**, and **unsigned int**.

It is recommended that the constructor of the top-level module has the conventional signature of a single formal argument of type **sc_module_name**, possibly augmented with additional arguments for specifying external arrays, but no other formal arguments. This is because the **write_wrapper** command assumes that module constructors follow these restrictions.

14.2 Ports

This section describes the following elements of ports:

- “Instantiation” on page 14-7
- “Binding Array Ports Using a for Loop” on page 14-7
- “User-Defined Port Binding Function” on page 14-8
- “Port Promotion” on page 14-8

14.2.1 Instantiation

Ports are usually instantiated as fields of **sc_modules**. Ports can also be instantiated as fields of classes or structs that are not **sc_modules**. In this case, the port is considered to be a port of the module that instantiates that class or struct, and the class or struct must not be used as a local variable in a process or as a formal argument of a function called from a process. Ports can be allocated dynamically in the constructor of a **sc_module**; such ports are considered to be ports of the **sc_module** whose constructor allocates them.

14.2.2 Binding Array Ports Using a for Loop

For an **sc_module** that has a field of type **array-of- sc_in/sc_out** it is often convenient to bind these ports using a loop. CtoS supports this if the loop is a for loop and the iteration variable is of type int or unsigned int. In the example below, **sc_module** SUB has an **array-of- sc_in SUB::INS**. The **sc_module** DUT instantiates SUB and binds the ports **SUB::INS** using a for loop.

```
template<int N>
SC_MODULE(SUB) {
    sc_in<in>    INS[N];
    SC_CTOR(SUB) { .. }
    ..
};

template<int N>
SC_MODULE(DUT) {
    ..
    SC_CTOR(DUT) {
        for (int i = 0; i < N; i++) {
            sub1.INS[i](sub1_ins[i])
        }
        ..
    }
    ..
    SUB sub1;
    sc_in<in>    sub1_ins[N];
};
```

14.2.3 User-Defined Port Binding Function

Ports can be bound using a user-defined port-binding function. In the example below, **sc_module** defines has a member function **DUT::bind_clk_rst()** for binding its **CLK** and **RST** ports to the **CLK** and **RST** ports of the formal argument. This user-defined port binding function is called from the constructor of **sc_module** DUT which instantiates SUB.

```
SC_MODULE(SUB) {
    sc_in<bool> CLK, RST;
    ..
    template<typename PAR>
    void bind_clk_rst(PAR &par) {
        CLK(par.CLK);
        RST(par.RST);
    }
    SC_CTOR(SUB) { .. }
    ..
};

SC_MODULE(DUT) {
    sc_in<bool> CLK, RST;
    ..
    SC_CTOR(DUT): CLK("CLK"), RST("RST"), sub1("sub1") {
        sub1.bind_clk_rst(*this); // binds sub1.CLK and sub1.RST
        ..
    }
    ..
    SUB sub1;
};
```

14.2.4 Port Promotion

Conventionally, the ports of an **sc_module** are bound by the constructor of the **sc_module** that instantiates the first module. Thus, given a top-level **sc_module**, the only **sc_in** or **sc_out** ports that are not bound after running the constructor of that module are the ports of that module. In this case, there is a one-to-one correspondence between the **sc_in/sc_out** ports of the top-level module and the terms of module in the CtoS database.

There are situations where it is appropriate that some of the **sc_in/sc_out** ports of an **sc_module** (M1) are still unbound after running the constructor of an **sc_module** (M2) that instantiates M1. The expectation is that those ports are bound by the constructor, a module that instantiates M2. CtoS allows the situation described and in this case the database module inferred from M2 will have terms inferred from the unbound ports of M1 so that those can be connected when the module inferred from M2 is instantiated. So, the unbound ports of M1 are promoted to be ports of M2 also.

14.3 Processes

The types of SystemC processes used as input to CtoS are **SC_METHOD** and **SC_CTHREAD**.

The primary difference between these process types is that **SC_METHOD** processes *cannot* call **wait** and therefore must return in zero time. **SC_CTHREAD** processes, however, must not return; instead, they must call **wait** to suspend themselves and to allow other processes to execute.

The choice of process type depends on the type of logic that you are modeling:

- *Pure combinational logic* should be modeled as an **SC_METHOD** process with static sensitivity consisting of all input signals of that logic.

This is called a *combinational SC_METHOD process*.

- *A sequential process with explicit control state* should be modeled as an **SC_METHOD** process with static sensitivity consisting of the clock event and, optionally, the asynchronous reset event.

This is called a *clocked SC_METHOD process*.

- *A sequential machine with implicit control state* should be modeled as an **SC_CTHREAD** process.

This modeling style is most suitable for behavioral synthesis.

This section has the following subsections:

- [“SC_CTHREAD Processes” on page 14-10](#)
- [“SC_THREAD Processes” on page 14-19](#)
- [“Combinational SC_METHOD Processes” on page 14-19](#)
- [“Clocked SC_METHOD Processes” on page 14-25](#)
- [“Resetting Fields of Modules” on page 14-31](#)
- [“Multiple Resets” on page 14-33](#)
- [“Internally Generated Reset Signals” on page 14-36](#)
- [“Clocks with SC_METHOD and SC_CTHREAD” on page 14-36](#)
- [“dont_initialize\(\) Function Calls” on page 14-36](#)

14.3.1 SC_CTHREAD Processes

The most common type of SystemC process used in CtoS is an **SC_CTHREAD**. A thread process is specified by calling the **SC_CTHREAD** macro and the **reset_signal_is** or **async_reset_signal_is** functions in the constructor of the module containing the process. The **SC_CTHREAD** macro specifies:

1. the *name* of the function that implements the behavior of the process, and
2. the *clock event* of the process

The **reset_signal_is** and **async_reset_signal_is** functions specify the reset conditions of the process.

An **SC_CTHREAD** process is started at the beginning of simulation and is also restarted whenever a reset is asserted. Since this process describes intended hardware, the top function of the thread should never return, as this would terminate the thread (and a terminated process does not restart on reset in SystemC). The function, or a function that it calls transitively, should call **wait** to mark the end of a clock cycle and suspend the process until the next clock event.

Thread processes are used for behavioral modeling of hardware. CtoS is able to perform more advanced transforms and optimizations on thread processes than on **SC_METHOD** processes. For example, CtoS can move operations to different states to share hardware, insert additional cycles to resolve timing problems, and pipeline loops to improve performance. Also, it is easier to convert legacy C or C++ code to a SystemC thread process than to an **SC_METHOD** process.

This section has the following subsections:

- [“Example of SC_CTHREAD Process” on page 14-10](#)
- [“Specifying SC_CTHREAD Processes” on page 14-12](#)
- [“Clock Specification of SC_CTHREAD Processes” on page 14-12](#)
- [“Reset Specification of SC_CTHREAD Processes” on page 14-12](#)
- [“Function Associated with SC_CTHREAD Processes” on page 14-14](#)
- [“Calling wait Function in SC_CTHREAD Processes” on page 14-16](#)
- [“Specifying Multiple Resets for SC_CTHREAD Processes” on page 14-16](#)

14.3.1.1 Example of SC_CTHREAD Process

An **SC_CTHREAD** process is specified with the **SC_CTHREAD** macro, which takes a function and a clock edge.

In the example on the following page, the function **main** describes the behavior of the process. Each time the process calls **wait**, it is suspended until the **clk** signal transitions from 0 to 1. The process then resumes execution at the statement following that **wait** statement. The call to **reset_signal_is()** specifies that the process has a synchronous reset, which is active when the **rst** signal samples true. This means that any time **clk** transitions from 0 to 1, and the **rst** signal samples true, the process restarts.

Example: SC_CTHREAD Process

```
SC_MODULE(pulser) {
public:
    sc_in<bool>          clk;
    sc_in<bool>          rst;
    sc_in<bool>          set_rate;
    sc_in<  sc_int<16> > rate;
    sc_out<  sc_int<16> > out;

    SC_CTOR(pulser)
    :   clk("clk"),
        rst("rst"),
        set_rate("set_rate"),
        rate("rate"),
        out("out")
    {
        SC_CTHREAD(main, clk.pos());
        reset_signal_is(rst, true);
    }

    void          main();
    int           cur_rate;
};

void
pulser::main()
{
    out.write(0);
    wait();
    while (!set_rate.read()) {
        wait();
    }
    cur_rate = rate.read();
    while (1) {
        for (int i = 0; i < cur_rate - 1; i++) {
            wait();
        }
        out.write(1);
        wait();
        out.write(0);
    }
}
```

14.3.1.2 Specifying SC_CTHREAD Processes

An **SC_CTHREAD** process is specified in the constructor of the module containing this process, as follows:

```
sc_cthread_instantiation ::=  
SC_CTHREAD(module_member_function_name, clock_sensitivity);  
[ async_reset_signal_is( reset_signal, reset_level_bool ); ]  
[ reset_signal_is(reset_signal, reset_level_bool); ... ]
```

14.3.1.3 Clock Specification of SC_CTHREAD Processes

The second argument of the **SC_CTHREAD** macro specifies the clock event.

The clock event must be the rising (**pos**) or falling (**neg**) edge event of a **sc_in<bool>** or an **sc_in_clk** member of the module containing the process.

```
clock_sensitivity ::= sc_in_port_name.pos()  
                    | sc_in_port_name.neg()
```

14.3.1.4 Reset Specification of SC_CTHREAD Processes

The *reset specification* of an **SC_CTHREAD** process consists of:

1. a set of *reset conditions*, and
2. the *reset behavior*

The *reset conditions* specify *when* reset takes place. The *reset behavior* specifies *what* actions take place upon reset.

A *reset condition of a thread process* can be either *synchronous* or *asynchronous*. Its specification consists of a reset signal and the active level of that signal.

A thread process may specify one or more reset conditions, but it can specify at most one asynchronous reset condition. For most thread processes, it is sufficient to specify a single reset condition.

Specifying more than one reset condition will result in more complex hardware and worse *quality of results* (QoR).

It is recommended that you specify a reset for each thread process, but it is allowable to specify none, if the process has only one control state.

The semantics of reset are as follows:

- A thread process with asynchronous reset is restarted any time the asynchronous reset signal transitions to the active level.
- A thread process is also restarted any time the clock event occurs and any of its reset conditions is active, that is, any time a clock event occurs and one or more of its reset signals is sampled active.

The *reset behavior* of a thread process is specified by the code between the start of the body of the function of the process and the first **wait** in the body. This is described in “[Function Associated with SC_CTHREAD Processes](#)” on page 14-14.

The *reset state* is the control state that the thread process reaches after a reset condition is asserted. This corresponds to the **wait** at which the process will be suspended after the process is restarted. CtoS requires that the reset state be unique.

Notes

- When viewing a process as a state transition system, reset is a specific transition defined between every state and the reset state.
- If reset is not specified, care should be taken in verification since internal variables will be initialized to **x** (unknown) in the synthesized model, while SystemC will initialize them to **zero** at the start of simulation.

The following subsections describe:

- “[Specifying a Synchronous Reset for SC_CTHREAD](#)” on page 14-13
- “[Specifying an Asynchronous Reset for SC_CTHREAD](#)” on page 14-14

Specifying a Synchronous Reset for SC_CTHREAD

You specify a *synchronous* reset by calling the function **reset_signal_is()** in the constructor of the module containing the process, right after calling the **SC_CTHREAD** macro.

```
reset_signal_is(reset_signal, reset_level);
```

where:

- *reset_signal* is the name of a module member of type **sc_in<bool>** or **sc_in<sc_logic>** that is the reset signal; if the reset signal is generated internally (see “[Internally Generated Reset Signals](#)” on page 14-36), it may be a module member of type **sc_signal<bool>** or **sc_signal<sc_logic>**.
- *reset_level* is either the constant **true** or **false**; it specifies the active level of the reset signal.

Specifying an Asynchronous Reset for SC_CTHREAD

You specify an *asynchronous* reset by calling the `async_reset_signal_is()` function in the constructor of the module containing the process, right after calling the `SC_CTHREAD` macro.

```
async_reset_signal_is(reset_signal, reset_level);
```

where:

- *reset_signal* is the name of a module member of type `sc_in<bool>` that is the reset signal; if the reset signal is generated internally (see [“Internally Generated Reset Signals” on page 14-36](#)), it may be a module member of type `sc_signal<bool>` or `sc_signal<sc_logic>`.
- *reset_level* is either the constant `true` or `false`; it specifies the active level of the reset signal.

14.3.1.5 Function Associated with SC_CTHREAD Processes

The `SC_CTHREAD` macro specifies the name of the function associated with the thread process. This function must be a member function of the module containing the process; it must have return type `void`; and it must not take any arguments.

The body of this function specifies the behavior of the process. Three parts can be distinguished:

- the *reset behavior*
- the *normal operation*
- the *optional initialization section*

The *reset behavior* is specified by the code from the first statement of the function body to the first `wait`. The control state associated with this first `wait` statement is the reset state of the process. In the reset behavior, the process cannot read primary inputs or `sc_signals` of the module containing the process. The only thing happening during the reset behavior is assigning compile-time constant values to variables (in particular, to the primary outputs and `sc_signals` of the module). If the process has multiple resets, it is allowed to read from the reset signals to decide which reset condition is active.

The *normal operation* of the process must be modeled by an infinite loop. This infinite loop must either call `wait` directly, or it must call a function that transitively calls `wait`.

The *optional initialization section* is specified by the code between the first `wait` and the infinite loop that describes normal operation. A common use of initialization code is a bounded loop initializing contents of an array that will be read (and possibly written) within the succeeding infinite loop.

Example: (1) Function Associated with SC_CTHREAD Process

```
void
module_name::sc_method_thread_function()
{
    [ reset_behavior ]
    wait_call
    [ initialization ]
    infinite_loop
}
```

where:

```
infinite_loop ::= infinite_loop_head { body }
               | do { body } while ( non_zero_constant );

infinite_loop_head ::= while (non_zero_constant)
| for ( reset_assignment ;[ non_zero_constant ]; expression )

non_zero_constant ::= 1 | true
```

Example: (2) Function Associated with SC_CTHREAD Process

An alternative form in which the first call to **wait** appears inside the infinite loop is described below. In this example, **reset_assignments** and **additional_reset_assignments** define the reset behavior.

```
void
module_name::sc_method_thread_function()
{
    [ reset_assignments ]
    infinite_loop_head {
        [ additional_reset_assignments ]
        wait_call
        [ body ]
    }
}
```

The **body** may contain declarations, loops, ifs, switches, assignments, and function calls.

Note See “Data Types” on page 14-53 and “Constructs” on page 14-85 for more legal data types for declarations/expressions and “Unsupported C/C++ Constructs” on page 14-93 for unsupported statements.

14.3.1.6 Calling wait Function in SC_CTHREAD Processes

In an **SC_CTHREAD** process, you can call the **wait** function with *no arguments* (which means it will wait until the next clock event) or with a *positive integer constant* (which means it will wait for that number of clock events).

```
wait_call ::= wait();  
           | wait(positive_integer_constant);
```

You cannot call other forms of the **wait** function from an **SC_CTHREAD** process.

14.3.1.7 Specifying Multiple Resets for SC_CTHREAD Processes

Multiple resets can be specified for an **SC_CTHREAD** process using the **reset_signal_is** and **async_reset_signal_is** functions (as described in [“Reset Specification of SC_CTHREAD Processes” on page 14-12](#)). When specifying multiple resets, keep the following in mind:

- No more than one asynchronous reset can be specified for a given process.
- The reset behavior must be consistent with reset priorities.
- If there is more than one synchronous reset, reset priorities are inferred from the first **if** statement in the body of the function of the process.
- If an asynchronous reset is specified, it always takes the highest priority.
- Multiple resets are an extension to SystemC 2.2 implemented in INCISIV, as of version 8.2. Other simulators may not implement this feature.
- A process with multiple resets may have worse QoR than a similar process with only a single reset, so specify multiple resets only when it is really needed.

The following sections clarify some additional topics concerning resets:

- [“Specifying Reset Priorities for SC_CTHREAD Processes” on page 14-17](#)
- [“Unspecified Reset Priorities” on page 14-18](#)
- [“Consistency of Reset Behavior with Reset Priorities” on page 14-18](#)

Specifying Reset Priorities for SC_CTHREAD Processes

The calls to `reset_signal_is` and `async_reset_signal_is` specify the reset conditions of a process, but they do not specify the *priorities* of these reset conditions. Asynchronous reset always takes the highest priority. If the process has more than one synchronous reset, the priorities of these resets are inferred from analyzing the first `if` statement in the body of the function associated with the process.

For a thread process with three reset conditions, the first `if` statement must take the following form:

```
if (reset_active) {  
    // Highest priority reset  
    ..  
} else if (reset_active) {  
    // Next to highest priority reset  
    ..  
} else {  
    // Lowest priority reset  
    ..  
}
```

where

```
reset_active ::= active_high_reset_condition  
              | active_low_reset_condition  
active_high_reset_condition ::= reset_signal  
                              | reset_signal == constant_1  
                              | reset_signal != constant_0  
active_low_reset_condition ::= ! reset_signal  
                              | ~ reset_signal  
                              | reset_signal == constant_0  
                              | reset_signal != constant_1  
constant_0 ::= 0 | 0x0 | sc_logic_0 | SC_LOGIC_0  
constant_1 ::= 1 | 0x1 | sc_logic_1 | SC_LOGIC_1  
reset_signal ::= the name of an sc_in<T> or sc_signal<T> module member,  
                 where T is bool or sc_logic
```

In this case, the highest priority reset condition is the one corresponding to the condition of the first `if`. The next highest priority reset condition is the one corresponding to the condition of the `if` in the `else` branch of the first `if`. The lowest priority reset should not be referred to explicitly.

There can be neither **wait** statements nor **control** statements, such as loops or **switch** statements, between the start of the body and the first `if` statement.

Unspecified Reset Priorities

A thread process may have multiple reset conditions for which the reset behavior is equivalent. In this case, the process is not required to specify the priorities of those resets.

Therefore, the body of the function associated with the process does not need to have an **if** statement between its start and the first **wait** or **control** statement.

Thus, for a thread process with more than one synchronous reset, CtoS issues a warning if no **if** statement is found, or if the chained **if** statement has fewer branches than there are reset conditions minus 1.

CtoS will arbitrarily assign priorities among those resets for which no priorities were specified.

If a thread process has two or more synchronous resets whose priorities are not specified, then the reset behavior of the process for each of those reset conditions must be equivalent.

For a given output, this means that either the output is not assigned if any of these reset conditions is asserted or else it is assigned the same constant value if any of those reset conditions is asserted.

In the following example, assume that the process has two synchronous resets, **rst1** and **rst2**. Assume that output **out1** is to be set to 0 if any of those resets is asserted.

The body of the function associated with the process would look as follows:

```
out1 = 0;
wait();
while (true) {
    ..
}
```

Consistency of Reset Behavior with Reset Priorities

The reset behavior can contain multiple assignments to a primary output of the module, and which of these assignments is active will typically depend on the reset condition asserted. However, the activation conditions for these assignments must be consistent with reset priorities; therefore, you should not test for a reset condition active unless you have already tested that higher priority resets are not active.

In the example on the following page, assume that there are two synchronous resets, **rst1** and **rst2**, which are both active high, and assume that **rst1** is the highest priority reset. The activation conditions for the reset assignments of output **out1** are consistent with these reset priorities.

The activation conditions for the reset assignments for **out2** are not consistent with reset priorities because the activation condition for the first assignment to **out2** tests for lower priority reset **rst2** active, regardless of whether the higher priority reset **rst1** is active.

Example: Consistency of Reset Behavior with Reset Priorities

```
if (rst1.read()) {
    out1 = 0x0;
} else {
    out1 = 0xFF;
}
if (rst2.read()) {
    // Activation condition is inconsistent with reset priorities
    out2 = 0x0;
} else {
    out2 = 0x10;
}
```

14.3.2 SC_THREAD Processes

An **SC_THREAD** process, whose static sensitivity consists of precisely the clock event, is an alternative to an **SC_CTHREAD** process. CtoS does not differentiate between these two alternatives, and all restrictions about **SC_CTHREAD** processes also apply to **SC_THREAD** processes.

```
sc_thread_instantiation ::=
SC_THREAD(module_member_function_name);
sensitive << clock_sensitivity;
[ async_reset_signal_is( reset_signal, reset_level_bool ); ]
[ reset_signal_is(reset_signal, reset_level_bool); ... ]
```

Note Support for resets for **SC_THREAD** processes is a Cadence extension to SystemC 2.2, which is implemented in INCISIV, as of version 8.2. Other simulators may not support this feature.

14.3.3 Combinational SC_METHOD Processes

The simplest process is a combinational **SC_METHOD** process. This process recomputes its outputs every time an input changes and is used to model combinational logic. The function associated with a combinational **SC_METHOD** must not call **wait**, and thus it must return in zero time. The function is called each time any input changes, as specified by the static sensitivity of the process.

This section has the following subsections:

- [“Example of Combinational SC_METHOD Process” on page 14-20](#)
- [“Specifying Combinational SC_METHOD Processes” on page 14-20](#)
- [“Static Sensitivity of Combinational SC_METHOD Processes” on page 14-21](#)
- [“Function Associated with Combinational SC_METHOD Processes” on page 14-21](#)
- [“Caveats when Using Combinational SC_METHOD Processes” on page 14-21](#)

14.3.3.1 Example of Combinational SC_METHOD Process

Here is a simple example of a combinational SC_METHOD process:

```
#include "systemc.h"

class mux2x8: public sc_module {
public:
    mux2x8(sc_module_name name);
    ~mux2x8();

    SC_HAS_PROCESS(mux2x8);

    sc_in< sc_uint<8> >    in0;
    sc_in< sc_uint<8> >    in1;
    sc_in<bool>            sel;
    sc_out< sc_uint<8> >   out;

private:
    void                proc();
};

mux2x8::mux2x8(sc_module_name name)
:   sc_module(name),
    in0("in0"),
    in1("in1"),
    sel("sel"),
    out("out")
{
    SC_METHOD(proc);
    sensitive << in0 << in1 << sel;
}

void
mux2x8::proc()
{
    out = sel ? in1 : in0;
}

SC_MODULE_EXPORT(mux2x8);
```

14.3.3.2 Specifying Combinational SC_METHOD Processes

A combinational SC_METHOD is specified in the module constructor as follows:

```
SC_METHOD(module_member_function);
sensitive << input_sensitivity [ << input_sensitivity ]* ;
```

14.3.3.3 Static Sensitivity of Combinational SC_METHOD Processes

The static sensitivity of a combinational **SC_METHOD** process must consist of the **value_changed_event** events of all inputs of the process. The **value_changed_event** is the default event for **sc_in** ports and **sc_signals**, so you can simply specify the name of the **sc_in** or **sc_signal** module member.

```
input_sensitivity ::= name_of_sc_in_port | name_of_sc_signal
```

The static sensitivity of a combinational **SC_METHOD** must not contain any edge events [**pos()**, **neg()**, **posedge_event()**, **negedge_event()**].

The static sensitivity specifies when the outputs of the process must be recomputed by calling the method associated with the process. In order for the simulation semantics in SystemC of such a process to match the behavior of the synthesized hardware you must ensure that:

- every input of the process is modeled as an **sc_signal** or an **sc_in** port
- every output of the process is modeled as an **sc_signal** or an **sc_out** port
- the static sensitivity of the process includes every input

In the previous example, the combinational process is called every time **in0**, **in1**, or **sel** changes.

14.3.3.4 Function Associated with Combinational SC_METHOD Processes

The **SC_METHOD** macro specifies the name of the function associated with the process.

This function must be a member function of the module containing the process; it must have return type **void**; and it must not take any arguments. The function must not call **wait**, and it must not call any other function that calls **wait**. The function must return in zero time; therefore, it cannot contain infinite loops.

14.3.3.5 Caveats when Using Combinational SC_METHOD Processes

Here are a few caveats to note when you are using combinational **SC_METHOD** processes in CtoS:

- [“Loops Must Be Fully Unrolled \(Comb\)” on page 14-22](#)
- [“Variables Must Be Assigned before Use \(Comb\)” on page 14-23](#)
- [“Writable Arrays Must Be Flattened \(Combinational\)” on page 14-24](#)
- [“SC_METHOD Should Not Call wait \(Combinational\)” on page 14-24](#)
- [“Multiple Combinational Processes Should Not “feed” Each Other \(Combinational\)” on page 14-25](#)

Loops Must Be Fully Unrolled (Comb)

Because **wait** statements are not legal in combinational **SC_METHOD** processes, all loops must be unrolled fully (see “[How Combinational Loops Are Determined in CtoS](#)” on page 8-17 and “[Unrolling Loops](#)” on page 8-17). This means CtoS must be able to determine the maximum number of iterations of a loop.

In the following example, the loop in **unrollableProc** can be unrolled because the loop body is executed at most eight times. The loop in **unsynthesizableProc** cannot be unrolled because the loop terminates only if there is a zero bit in the **in** value; otherwise it will index beyond the end of the input, and the behavior in CtoS will be undefined.

Example: Loops with Combinational SC_METHOD Processes

```
#include "systemc.h"

class findFirstOne: public sc_module {
public:
    findFirstOne(sc_module_name name);
    ~findFirstOne();

    SC_HAS_PROCESS(findFirstOne);

    sc_in< sc_uint<8> >    in;
    sc_out< sc_uint<4> >    out0;
    sc_out< sc_uint<4> >    out1;

private:
    void    unrollableProc();
    void    unsynthesizableProc();
};

findFirstOne::findFirstOne(sc_module_name name)
:    sc_module(name),
  in("in"),
  out0("out0"),
  out1("out1")
{
    SC_METHOD(unrollableProc);
    sensitive << in;
    SC_METHOD(unsynthesizableProc);
    sensitive << in;
}
```

Example: Loops with Combinational SC_METHOD Processes (Cont.)

```
void
findFirstOne::unrollableProc()
{
    unsigned int i;
    sc_uint<8>    input = in;
    for (i = 0; i < 8; i++) {
        if (input[i]) {
            out0.write(i);
            return;
        }
    }
    out0.write(8);
}

void
findFirstOne::unsynthesizableProc()
{
    unsigned int i = 0;
    sc_uint<8> input = in;

    while (!input[i]) {
        i = i + 1;
    }
    out1.write(i);
}

SC_MODULE_EXPORT(findFirstOne);
```

Variables Must Be Assigned before Use (Comb)

Local variables and member variables that are read and written by only a single **SC_METHOD** process must be assigned a value in the function call before the variable is used. If this restriction is not observed, then a latch is implied, which is not supported by CtoS, and a simulation-synthesis mismatch will occur.

The example on the following page will simulate, but will not synthesize correctly, because the process is not combinational. The previous value of **out** will be used, but the value was written by this process, which is unlocked.

Example: Variables with Combinational SC_METHOD Processes

```
#include "systemc.h"

class useError: public sc_module {
public:
    useError(sc_module_name name);
    ~useError();

    SC_HAS_PROCESS(useError);

    sc_in< sc_uint<8> >    increment;
    sc_in< sc_uint<8> >    clear;
    sc_out< sc_uint<8> >    out;

private:
    void    proc();
};

useError::useError(sc_module_name name)
:    sc_module(name),
  increment("increment"),
  clear("clear"),
  out("out")
{
    SC_METHOD(proc);
    sensitive << increment << clear << out;
}

void
useError::proc()
{
    if (clear) {
        out = 0;
    } else if (increment) {
        out = out + 1; // ILLEGAL use of out: out + 1
    }
}

SC_MODULE_EXPORT(useError);
```

Writable Arrays Must Be Flattened (Combinational)

A combinational **SC_METHOD** process cannot read or write memories, except read-only arrays; if such a process *does* access a writable array, the array must be flattened (see [“Flattening Arrays” on page 8-61](#)).

SC_METHOD Should Not Call wait (Combinational)

A combinational **SC_METHOD** process, and any functions called directly from the process or indirectly through other functions, should not call **wait**, it should be called only in a **SC_THREAD** or **SC_CTHREAD** process.

Multiple Combinational Processes Should Not “feed” Each Other (Combinational)

Care must be taken when using multiple combinational **SC_METHOD** processes that *feed* each other. An unintended combinational loop could result if the sensitivity lists of the two are inverted (the output of the first is in the sensitivity list of the second, and the output of the second is in the sensitivity list of the first). This problem could result even if the contents of the **SC_METHOD** process restricted a combinational feedback loop, so this structure is not recommended.

14.3.4 Clocked SC_METHOD Processes

A *clocked* **SC_METHOD** process is an **SC_METHOD** process whose static sensitivity consists of the clock event and, optionally, the asynchronous reset event. Consequently, the function associated with the process is called every clock cycle on the active edge of the clock and also on the active edge of the reset signal, if the process has asynchronous reset. The function associated with the process must return immediately – it cannot call **wait**.

A *clocked* **SC_METHOD** process is used for modeling simple state machines when you want to control the cycle in which every operation occurs, and you are willing to model the state machine explicitly.

A *clocked* **SC_METHOD** process is specified in the constructor of the module containing the process by calling the macro **SC_METHOD** and calling the `<<` operator on the **sensitive** member of the module. The **SC_METHOD** macro takes as its only argument the name of the function associated with the process. This function must be a member function of the module containing the process, its return type must be void; and it must not take any arguments. The `<<` operator of the sensitive module member takes as right-hand operand the clock or reset event of the process.

The function associated with the *clocked* **SC_METHOD** process implements the behavior of the process. If you want the tool to infer resets for the process and implement part of the behavior of the process by means of the reset functionality of resettable flip-flops, the body of the function must fit a very rigid structure – specifically, it should consist of a single if-then-else statement.

Optionally, this if-then-else statement may be preceded by declarations of local variables. The tool interprets the **then** branch as the reset behavior and the **else** branch as the normal (non-reset) behavior. The reset behavior is implemented by the reset functionality of resettable flip-flops in the synthesized design; therefore, only assignments of compile-time constant values are allowed in the reset behavior.

If the body of the function associated with the process does not fit this described structure, and the static sensitivity consists only of the clock event, then no resets will be inferred for the process.

A *clocked* **SC_METHOD** process may take multiple reset conditions, but only one asynchronous reset is allowed. Synchronous reset conditions are inferred purely from the form of the body of the function associated with the process.

A process with **N** reset conditions must consist of an **if-then-else** statement, possibly preceded by declaration, where the **else** branch consists of **N-1** nested **if-then-else** statements. The condition of each **if** statement must test for a reset condition being active.

Clocked **SC_METHOD** processes are further described in the following sections:

- [“Example of Clocked SC_METHOD Process” on page 14-26](#)
- [“Specifying Clocked SC_METHOD Processes” on page 14-27](#)
- [“Clock Specification of Clocked SC_METHOD Processes” on page 14-27](#)
- [“Reset Specification of Clocked SC_METHOD Processes” on page 14-28](#)
- [“Specifying Multiple Resets for Clocked SC_METHOD Processes” on page 14-28](#)
- [“Caveats when Using Clocked SC_METHOD Processes” on page 14-29](#)

14.3.4.1 Example of Clocked SC_METHOD Process

```
#include "systemc.h"
SC_MODULE(rtl_example1) {
public:
    sc_in_clk          clk;
    sc_in<bool>         rst;
    sc_in< sc_uint<8> > in;
    sc_out<bool>        overflow;

                                rtl_example1(sc_module_name name);
                                ~rtl_example1() {}

    SC_HAS_PROCESS(rtl_example1);

private:
    void                method();
};

rtl_example1::rtl_example1(sc_module_name name)
:    sc_module(name),
  clk("clk"),
  rst("rst"),
  in("in"),
  overflow("overflow")
{
    SC_METHOD(method);
    sensitive << clk.pos() << rst.pos();
}
// This clocked SC_METHOD is clocked on the rising edge of 'clk'
// and is asynchronously reset when rst is high (true).
void
rtl_example1::method()
{
    if (rst) {
        // Only constant assignments to the outputs of the
        // clocked SC_METHOD are allowed here.
        overflow.write(0);
    } else {
        // This is the main body of the clocked SC_METHOD.
        // It describes the non-reset behavior of the process.
        if (in.read() > 100) {
```

```
        overflow.write(1);
    }
}
SC_MODULE_EXPORT(rtl_example1);
```

14.3.4.2 Specifying Clocked SC_METHOD Processes

A clocked **SC_METHOD** is specified in the module constructor as follows:

```
SC_METHOD(sc_method_member_function);
sensitive << clock_sensitivity [ << reset_sensitivity ] ;
```

The requirements for the function associated with the clocked **SC_METHOD** process depend on whether reset is intended, as follows:

Case 1: Clocked SC_METHOD process with reset

In this case, the function associated with the clocked **SC_METHOD** must be of the form:

```
void
module_name::module_member_function()
{
    [ declarations_of_local_variables ]
    if ( reset_active_condition ) {
        reset_behavior
    } else {
        non_reset_behavior
    }
}
```

Case 2: Clocked SC_METHOD Process without Reset

In this case, there are no specific constraints on the form of the body of the function associated with the process; however, the body of the function must be distinguishable from the case in which reset is intended. This is easily done if the body of the function has any statements other than declarations and a single **if** statement.

14.3.4.3 Clock Specification of Clocked SC_METHOD Processes

The clock signal of a clocked **SC_METHOD** and its active edge are inferred from the static sensitivity.

If the process does not have an asynchronous reset, the static sensitivity must consist precisely of the clock event.

If the process has an asynchronous reset, its static sensitivity must consist of two events: the clock event and the asynchronous reset event. The event that is the actual clock event is inferred from analyzing the body of the function associated with the process. The asynchronous reset signal must appear as the (first) reset condition; therefore, the event whose signal does not appear as a reset condition is the clock event.

```
clock_sensitivity ::= sc_in_port.pos()
                  | sc_in_port.neg()
```

The signal from which the clock event is derived must be either an **sc_in<bool>** or **sc_in_clk** port of the module containing the process.

14.3.4.4 Reset Specification of Clocked SC_METHOD Processes

The reset condition of a clocked **SC_METHOD** is inferred from the condition of the **if** statement in the body of the method of the process. Reset is asynchronous if the event corresponding to the active transition of the reset signal is listed in the static sensitivity of the process; otherwise, it is synchronous.

```
reset_sensitivity ::= sc_in_port.pos()
                   | sc_in_port.neg()
                   | sc_signal_member.posedge_event()
                   | sc_signal_member.negedge_event()

reset_active_condition ::= active_high_reset_condition
                        | active_low_reset_condition
active_high_reset_condition ::= reset_signal
                                | reset_signal == constant_1
                                | reset_signal != constant_0
active_low_reset_condition ::= ! reset_signal
                                | ~ reset_signal
                                | reset_signal == constant_0
                                | reset_signal != constant_1
constant_0 ::= 0 | 0x0 | sc_logic_0 | SC_LOGIC_0
constant_1 ::= 1 | 0x1 | sc_logic_1 | SC_LOGIC_1
```

The final part of specifying reset is to describe the variables that need to be reset. Reset assignments are restricted to simple assignments – constant values to member variables.

```
reset_behavior ::= reset_assignment
                | reset_behavior reset_assignment
reset_assignment ::= module_member_variable = constant_value ;
```

Tip Most RTL coding conventions recommend the reset signal name be **rst** for an active high reset and **rst_n** for an active low reset.

14.3.4.5 Specifying Multiple Resets for Clocked SC_METHOD Processes

Multiple resets can be specified for a clocked **SC_METHOD** process by nesting additional **if-then-else** statements in the **else** branch of the **if-then-else** of the method of the process.

```
void
module_name::module_member_function()
{
    [ declarations_of_local_variables ]
    if ( reset_active_condition ) {
        // Highest-priority reset
```

```
        reset_behavior
    }
    [ else if ( reset_active_condition ) {
        // Lower priority reset
        reset_behavior
    } ]*
    else {
        non_reset_behavior
    }
}
```

When specifying multiple resets for a clocked **SC_METHOD** process, keep the following in mind:

- No more than one asynchronous reset can be specified for a given process.
- If there is more than one synchronous reset, reset priorities are inferred from the first **if** statement in the body of the function of the process.
- If an asynchronous reset is specified, it always takes the highest priority.
- A process with multiple resets may have worse QoR than a similar process with only a single reset, so specify multiple resets only when you really need to.

14.3.4.6 Caveats when Using Clocked SC_METHOD Processes

Here are a few caveats to note when you are using clocked **SC_METHOD** processes in CtoS:

- [“Loops Must Be Fully Unrolled \(Clocked\)” on page 14-29](#)
- [“Writable Arrays Must Be Flattened \(Clocked\)” on page 14-29](#)
- [“SC_METHOD Should Not Call wait \(Clocked\)” on page 14-30](#)
- [“Unintended Reset Conditions Must Be Avoided” on page 14-30](#)

Loops Must Be Fully Unrolled (Clocked)

Similar to combinational **SC_METHOD** processes, **wait** statements are not legal in clocked **SC_METHOD** processes, so all loops must be fully unrolled (see [“Loops Must Be Fully Unrolled \(Comb\)” on page 14-22](#) and [“How Combinational Loops Are Determined in CtoS” on page 8-17](#)).

Writable Arrays Must Be Flattened (Clocked)

Similar to combinational **SC_METHOD** processes, clocked **SC_METHOD** processes cannot read or write memories, except read-only arrays.

If a clocked **SC_METHOD** process accesses a writable array, the array must be flattened (see [“Writable Arrays Must Be Flattened \(Combinational\)” on page 14-24](#) and [“Flattening Arrays” on page 8-61](#)).

SC_METHOD Should Not Call wait (Clocked)

Similar to combinational **SC_METHOD** processes, clocked **SC_METHOD** processes, and any functions called directly from such processes or indirectly through other functions, should not call **wait**, which should be called only in an **SC_THREAD** or **SC_CTHREAD** process (see [“SC_METHOD Should Not Call wait \(Combinational\)” on page 14-24](#)).

Unintended Reset Conditions Must Be Avoided

Recognition of multiple reset conditions for clocked **SC_METHOD** processes is highly sensitive to syntax and, under certain circumstances, CtoS may try to infer reset conditions where no reset conditions are intended.

The following code, for example, contains a definition of the member function associated with a clocked **SC_METHOD** process.

The intent is that this process have a single reset condition; however, CtoS erroneously tries to infer a second reset condition from **expr2**:

```
void
module_name::module_member_function()
{
    if (expr1) {
        // Highest priority reset
        B1
    } else {
        // Non-reset behavior
        if (expr2) {
            B2
        } else {
            B3
        }
    }
}
```

As a workaround, you can add an unused declaration as the first statement in the **else** clause where no more resets are to be inferred:

```
void
module_name::module_member_function()
{
    if (expr1) {
        // Highest priority reset
        B1
    } else {
        int unused; // <- unused declaration to stop inferring reset conditions
        // Non-reset behavior
    }
}
```

```
    if (expr2) {  
        B2  
    } else {  
        B3  
    }  
}  
}
```

14.3.5 Resetting Fields of Modules

Fields of a module are often used to model the (data) state of a process contained in that module. In software, the fields of a module are usually initialized by the module constructor. However, such initialization is not appropriate for synthesis because the module constructor is run only at time zero and in hardware there is no concept of time zero. Consequently, CtoS ignores any initialization of module fields in the module constructor.

To ensure that fields are properly initialized before they are read, it is good practice to assign constant values to a module field in the reset behavior of the process that *owns* that field. If the field is accessed by multiple processes, it must be an **sc_signal** (see “[Interprocess Communication through Shared Variables](#)” on page 14-38). In this case, it is possible that the field is written by multiple processes. It is recommended that you put the assignments to initialize the field in the reset behavior of one of these processes.

There is a case, however, in which the process that owns the field is declared in a module outside the module that contains the field. This is typical for modules that implement interface functions that can then be called by processes outside that module. Such modules typically derive from **sc_channel**.

In this case, the function associated with the process is a member function of a different module from the module containing the field, and thus you cannot readily reset the field by means of a simple assignment in the reset section of the function because you cannot readily access the field from this function. Access to the field from the function is even more problematic if the field is a private member of the module.

A recommended approach is to define an interface function that does the reset of such member fields and let this function be called in the reset behavior of a process that accesses this module from the outside.

More specifically, the author of the channel module defines member functions that assign the member fields to compile-time constants. In general, you define multiples of these functions and partition the member fields, so each function resets a set of member fields that are logically related. Then, the author declares each of the functions as a member of an **sc_interface** that this module inherits. Typically, you would include each of these functions in an **sc_interface** class so this reset function and the other functions of the **sc_interface** are logically related, rather than creating new **sc_interface** classes only for these reset functions.

For example, if the author wants to let this module inherit an **sc_interface** FifoWriter and implement its function called *write*, then he would derive an **sc_interface** from FifoWriter and define a function in this **sc_interface** that resets all of (and only) the member fields of the module assigned by the body of *write* implemented in the module.

Then, when this module is instantiated and composed with another module instance, that is, the latter has a port bound to the instance of the channel module, when you are doing the instantiation, you are responsible for calling the reset functions of the **sc_interface** class used as the type of the port, in one of its processes.

14.3.5.1 Incomplete Reset of Fields of Modules

Reset of a module member variable is incomplete if that variable is not assigned for one or more of the reset conditions of the process that owns the variable, as follows:

- If the process has *only one* reset condition, this means the variable is not assigned during reset.
- If the process has *multiple* reset conditions, the variable may be assigned for some reset conditions, but not for others.

If a design has a module member variable that is not reset for reset condition **Ri** of the owning process, then that variable retains its value when **Ri** is asserted in the original SystemC model.

However, preserving this behavior in the synthesized design may significantly reduce the amount of optimization during synthesis. Furthermore, for many designs, the semantics of incomplete reset may be an artifact of the modeling language rather than design intent.

CtoS therefore makes the following assumption about module member variables with incomplete reset:

- If the module member variable is not an **sc_signal** or **sc_out**, then it does not preserve values when the owning process undergoes reset. (It is assumed that when reset is asserted, the value of the variable becomes unspecified. The synthesized model does not preserve the semantics of incomplete reset.)
- If the module member variable is an **sc_signal** or **sc_out**, and the variable is not reset for one or more of the reset conditions of the owning process, then it retains state when that reset condition is asserted. (The synthesized model preserves the semantics of incomplete reset of the original model.) Recall that consistency requires that the variable not be reset for all reset conditions that have lower priority than the given reset condition.

Finally, for processes with multiple reset conditions, if the process assigns a module member variable that is an **sc_signal** or **sc_out** for a reset condition of that process, then it also must assign that variable for all higher-priority reset conditions of the process.

14.3.6 Multiple Resets

Not all reset behaviors are synthesizable. Synthesizable reset behavior depends on whether the variable is an **sc_signal** or **sc_out**. It does not depend on whether the object is a module member.

- [“Reset Behavior for an sc_signal or sc_out Variable” on page 14-33](#)
- [“Reset Behavior for a non-\(sc_signal or sc_out\) Variable” on page 14-33](#)
- [“Tabular Representation of Reset Behavior” on page 14-34](#)

14.3.6.1 Reset Behavior for an sc_signal or sc_out Variable

For an **sc_signal** or **sc_out** variable, the following reset behaviors are supported:

- The variable may be set for *each* of the resets. The variable can be set to the same value for all resets, or it can be set to different values.
- The variable may *not* be set for *any* of the resets. However, although this is supported, it is *not recommended*.

Restriction: If the process has asynchronous reset, then the variable *must* be set on asynchronous reset.

- The variable may be set for *some* of the resets, while the state is kept for *all other* resets.

Restrictions:

- If the variable is *set* for reset condition R_i , then it must *also be set* for all higher-priority reset conditions.
- If the variable is *not* set for reset condition R_i , then it must also *not* be set for all lower-priority reset conditions.
- If the process has asynchronous reset, then the variable *must* be set on asynchronous reset.

14.3.6.2 Reset Behavior for a non-(sc_signal or sc_out) Variable

For a non-(**sc_signal** or **sc_out**) variable that is alive during reset, the following reset behaviors are supported:

- The variable may be set for *each* of the resets. The variable can be set to the same value for all resets, or it can be set to different values (RB1, RB2 in [Table 14-2](#) and [Table 14-3](#)).
- The value of the variable is don't care when any of the resets is active. This is specified in the SystemC source by not assigning the variable in the reset sections of any of the reset conditions (RB5 in [Table 14-2](#) and [Table 14-3](#)).

Note For a local variable that is not in the scope before the first **wait**, reset is not relevant.

14.3.6.3 Tabular Representation of Reset Behavior

[Table 14-1 on page 14-34](#), [Table 14-2 on page 14-34](#), and [Table 14-3 on page 14-35](#) show reset behavior information in tabular form.

Table 14-1 Reset Behavior

Code	Description
RB1	set to constant on rst1, rst2
RB2	set to constant1 on rst1, set to constant2 on rst2
RB3	set to constant on rst1, keep state on rst2
RB4	set to constant on rst1, don't-care on rst2
RB5	keep state on rst1, rst2 (not recommended)
RB6	don't-care on rst1, rst2 (not recommended)
RB7	keep state on rst1, set to constant on rst2
RB8	don't-care on rst1, set to constant on rst2

Table 14-2 Process with async reset rst1 and sync reset rst2 (rst1 has Highest Priority)

Reset behavior	sc_signal/sc_out	non-(sc_signal/sc_out)
RB1	supported	supported
RB2	supported	supported
RB3	supported	not supported

Table 14-2 Process with async reset rst1 and sync reset rst2 (rst1 has Highest Priority)

Reset behavior	sc_signal/sc_out	non-(sc_signal/sc_out)
RB4	not supported	not supported
RB5	not supported	not supported
RB6	not supported	supported
RB7	not supported	not supported
RB8	not supported	not supported

Table 14-3 Process with sync reset rst1 and sync reset rst2 (rst1 has Highest Priority)

Reset behavior	sc_signal/sc_out	non-(sc_signal/sc_out)
RB1	supported	supported
RB2	supported	supported
RB3	supported	not supported
RB4	not supported	not supported
RB5	supported	not supported
RB6	not supported	supported
RB7	not supported	not supported
RB8	not supported	not supported

Note To diagnose errors from the `create_initial_resources`, `schedule`, or `allocate_registers` commands, use the `check_design` command. This command tries to identify *all* synthesizability problems with a design, in contrast to the other commands, which report only the *first* problem preventing CtoS from synthesizing a design (see [Table 14-1 on page 14-34](#) & [Table 14-2 on page 14-34](#)).

14.3.7 Internally Generated Reset Signals

Reset is defined as *external* if the reset signal is driven (generated) outside the top-level module to be synthesized. This is the most common case. The module to be synthesized has an **sc_in** port for the reset, and processes refer to that port in their reset specifications. Sub-modules containing processes that use this reset also have an **sc_in** port connected to the reset input port of the parent module.

CtoS also supports *internally generated* reset signals. In this case, an **sc_signal** is used as a reset of some process underneath the synthesized module, and that signal is driven by a process also underneath the synthesized module.

An internally generated reset signal must be driven by one of the following:

- precisely one combinational **SC_METHOD** process
- precisely one clocked **SC_METHOD** process that has reset itself
- precisely one **SC_CTHREAD** process that has reset itself

Additionally, an internally generated reset cannot be used as the primary (the highest priority) reset condition of the process driving that internally generated reset.

14.3.8 Clocks with SC_METHOD and SC_CTHREAD

Clocks must be generated externally to a design. Thus, the input port used in the clock specification of a clocked **SC_METHOD** or **SC_CTHREAD** must be an input port either of the top-level module or of a submodule connected to an input port of the top-level module. Furthermore, if a design is built in hierarchical mode (see [“Module Hierarchy” on page 16-1](#)), the clocking of instances of a module must be consistent, that is, for a clock input port of a module built hierarchically, the corresponding ports of all instances of that module must be connected to the *same clock input port* of the top-level module.

For example, the following situation would *not* be allowed: a design in which the clock port of one instance of a module **M** is connected to clock input port **CLK1** of the top-level module, while the clock port of another instance of **M** is connected to clock input port **CLK2** of the top-level module.

14.3.9 dont_initialize() Function Calls

CtoS ignores **dont_initialize()** function calls, so designs should never rely on the behavior implied by this construct.

14.4 Mealy Communication

When the output values of a behavior of a process must be computed using its input values at the same clock cycle when the inputs are provided, this is known as a *Mealy* behavior. SystemC does not provide a way to describe a Mealy behavior of a process, unless the process is declared with **SC_METHOD** and its behavior is written by explicitly specifying the structure of the underlying state machine.

14.5 Inputs, Outputs, and Multiple Drivers

The inputs and outputs of a process must be declared with the semantics of primitive channels, that is, the value to be evaluated in an evaluation phase must be that given at the end of the immediately preceding update phase. This ensures that when one process assigns its output, while another process evaluates it in the same evaluation phase, the evaluated value is the same, regardless of the order in which the two processes are executed in simulation in that evaluation phase.

Since objects are referred to as the *inputs* of a process if the process evaluates them, and they are assigned by processes other than this process, the *outputs* are defined accordingly. The inputs or outputs may be ports of a module, and in that case, the objects bound to the ports must have this semantics. A SystemC class with this semantics is **sc_signal**, and this class is often used as an input or output of a process. The *LRM* states that it is illegal for an **sc_signal** to be assigned by multiple processes, but simulators typically allow writes from multiple processes. CtoS also allows **sc_signals** to be written from multiple processes; however, it assumes that **sc_signal** objects will never be assigned by multiple processes in any given clock cycle. Output models produced by CtoS include code to check this assumption and halt simulation in case the assumption is violated.

14.6 Variables

CtoS considers a *variable* to be an automatic variable declared in a function or a member variable of a SystemC module (**SC_MODULE**). A field of a variable of a struct type is not a variable; neither is an element of a variable of an array type. Variables play different roles in modeling a design.

The following sections describe, in greater detail, how to use variables in CtoS:

- [“Interprocess Communication through Shared Variables” on page 14-38](#)
- [“Array Variables” on page 14-38](#)
- [“Read-Only Non-Array Shared Data Variables” on page 14-41](#)
- [“Module Member Variables” on page 14-43](#)
- [“Global Variables” on page 14-44](#)
- [“Static Variables” on page 14-44](#)

14.6.1 Interprocess Communication through Shared Variables

CtoS supports interprocess communication modeled using **SC_MODULE** member variables of type **sc_signal**. The simulation semantics of **sc_signal** objects ensure that the behavior is deterministic for cases in which an **sc_signal** object is read by one process in the same delta cycle in which the object is written by another process; in such cases, the *old* value of the **sc_signal** object will be read.

In cases where *multiple* processes write an **sc_signal** object in the same delta cycle, the behavior is *non-deterministic*; therefore, you must ensure that an **sc_signal** variable is never written by multiple processes in the same delta cycle. CtoS-generated output models have *contention checkers* that stop simulation when they detect an **sc_signal** object being written by more than one process at a delta cycle.

If processes that access a shared variable are in an **SC_MODULE** different from the **SC_MODULE** in which the shared variable is declared, the variable must be made accessible via **sc_in** and **sc_out** ports.

All processes writing to a given **sc_signal** variable must have the same clock event, and only one of those processes should assign the **sc_signal** variable on reset. CtoS schedules read and write operations to an **sc_signal** object on the same edge as the edge these operations appear in the input SystemC model.

SC_MODULE member variables of type array of **sc_signal** are also supported. CtoS treats an array of **sc_signal** of N elements as N individual **sc_signal** variables. Interprocess communication via events is not supported in CtoS.

14.6.2 Array Variables

CtoS elaborates a local variable of an array type, and **SC_MODULE** members of an array type, where the element is not an **sc_signal**, as a memory.

This gives you the choice to implement the storage associated with the array variable as a built-in RAM, as a Vendor RAM, or as a set of discrete data registers. The latter option requires flattening the array (see [“Flattening Arrays” on page 8-61](#)).

CtoS decides whether to elaborate a data object as a memory, based on the type of the variable; therefore, a variable of type struct that has an array as one of its fields is not elaborated as a memory. Only variables that are elaborated into a memory can be mapped onto RAMs.

Initializations of arrays declared as module member variables are ignored by CtoS, unless the array variable can be recognized as a read-only array by CtoS (as described in [“Read-Only Arrays” on page 14-40](#)).

Initializations of arrays declared as local variables are elaborated as dictated by C++ semantics. If a local array variable is to be implemented by a RAM, then implicit initializations may be undesirable because they imply write access to the RAM that must be scheduled. In this case, consider declaring the array as a *module member variable*. CtoS will then ignore the initialization just as for any other module member variable that is not a read-only array.

An **SC_MODULE** member of type array of **sc_signal** with N elements is treated by CtoS as N individual **sc_signal** objects; it is not elaborated into a memory.

The following sections describe:

- [“Shared Arrays” on page 14-39](#)
- [“Non-Shared Arrays” on page 14-40](#)
- [“Read-Only Arrays” on page 14-40](#)
- [“Examples of Arrays” on page 14-40](#)

14.6.2.1 Shared Arrays

Storing data shared among multiple processes, whose synchronization is implemented by a protocol separate from the storage object, can be modeled as an **SC_MODULE** member variable of an array of a type that is not an **sc_signal**. CtoS elaborates such a variable into a memory.

Since it is accessed by multiple processes, the array must be implemented by a multi-ported built-in RAM or a multi-ported Vendor RAM. Each process that accesses the array will be assigned at least one port. In the design, you must implement a protocol that ensures that if in a given clock cycle one process is writing a given array element, no other process is accessing the same array element. Violation of this requirement results in a race condition. CtoS-scheduled models contain checkers that stop simulation if they detect race conditions on shared arrays.

By default, CtoS tries to schedule all memread and memwrite operations of a shared array on the edges on which these operations occur in the input model; however, you can relax this constraint by *floating the array accesses* (see [“Floating Array Accesses” on page 8-94](#), especially noting the requirements). In general, it is beneficial to allow array accesses to float within the largest subgraph of the CDFG possible, considering the protocol that synchronizes the processes accessing the array.

14.6.2.2 Non-Shared Arrays

Arrays accessed by a single process should be modeled as variables of an array type that are either members of an **SC_MODULE** or local variables of some function. In the former case, functions called by the owner process can readily access the array. In the latter case, access to the array from functions called by the function that declares the array variable requires passing a pointer to the array to the called function.

The use of pointers may add variability to the design, which may reduce the applicability of certain optimizations. Therefore, designs that *do not* use pointers may yield better QoR than functionally equivalent designs that *do* use pointers.

14.6.2.3 Read-Only Arrays

CtoS provides special support for implementing read-only arrays. CtoS recognizes an array variable as a read-only array variable only if it satisfies the following requirements:

- The variable is declared as:
 - a const static member of an **SC_MODULE** and is initialized at its definition, or
 - a local variable of a function and is initialized at its definition.
- The initializer expressions are compile-time constants.
- Except for the initialization, the array variable has only read accesses.

Array initialization uses curly-brace syntax. If the dimensions of the declared array are larger than the number of initializers, then the missing initializers are assumed to be 0.

14.6.2.4 Examples of Arrays

CtoS supports multi-dimensional read-only arrays and gives you a warning if any but the major dimension is not a power of 2. Upsizing the inner dimensions to a power of 2 improves QoR. Here are two examples of arrays.

Example: Array Example 1

```
SC_MODULE (Mod1) {  
    ...  
    int func1(int a);  
    static const int m_table1[8];  
    ..  
};  
  
const int Mod1::m_table1[8]  
    = {1,2,3,5,7,11,13,17};           // definition + initialization  
  
int Mod1::func1(int a) {
```



```
...
int val1 = m_table1[i];          // read access
...
}
```

Example: Array Example 2

```
struct RGB {
    int r,g,b;
};

SC_MODULE(Mod1) {
    ...
    int func1(int a);
    ..
};

int Mod1::func1(int a) {
    // Define a read-only array.
    RGB colors[]
    = { {255, 255, 255},
        {255, 250, 250},
        {248, 248, 255},
        {139, 62, 47}};          // declaration + initialization
    ...
    // Perform read access into the array.
    color = colors[i].r + colors[i].g + colors[i].b;
    ...
}
```

14.6.3 Read-Only Non-Array Shared Data Variables

Read-only non-array shared data to be accessed by only one process should be modeled as an **SC_MODULE** member variable of a non-**sc_signal** type. On reset, the given process should assign compile-time constant values to the variable. After reset, the process and any functions called from the process can read the variable.

Read-only non-array shared data that is read by multiple processes must be modeled by multiple **SC_MODULE** member variables of a non-**sc_signal** type. Each such variable holds a copy of the read-only data and is accessed by only one process. This usage then degenerates to the first usage. The following example illustrates two ways that you can use to specify a read-only array.

Example: A Read-Only Array

```
::::::::::::::
dut.h
::::::::::::
```

```
#include <systemc.h>

SC_MODULE(DUT) {
    sc_in<bool>      CLK;
    sc_in<bool>      RST;
    sc_in<int>       IN1;
    sc_out<int>      OUT1;
    sc_out<int>      OUT2;
    void             main();

    SC_CTOR(DUT) {
        SC_CTHREAD(main, CLK.pos());
        reset_signal_is(RST, true);
    }

    static const int codes1[4];
};

:::::::::::::
dut.cpp
:::::::::::::
#include "dut.h"

// This static const module member array is not written after
// initialization.
// It will be implemented by a lookup or a ROM.
const int DUT::codes1[4] = { 2, 3, 5, 7 };

void
DUT::main()
{
    // This locally declared array is not written after its initialization.
    // It will be implemented by a lookup or a ROM.
    const int codes2[4] = { 11, 13, 17, 19 };

    int sum1 = 0, sum2 = 0;

    OUT1.write(0);
    OUT2.write(0);

    while (true) {
        wait();
        sc_uint<2> ind = IN1.read();
        sum1 += codes1[ind];
        sum2 += codes2[ind];
        OUT1.write(sum1);
        OUT2.write(sum2);
    }
}
```


Determining when to Initialize Variables

In general, you must ensure that a variable is written before being read. In particular, when a variable is written and read by different process instances, you must ensure this order not only in the SystemC input description to CtoS, but also in the output models, whose timing of the write and read operations of multiple processes may differ from the input.

If a variable is declared with an **sc_signal**, the best practice is to initialize such a variable in the reset, specifically in one of the process instances that write to the variable in its core behavior; otherwise, the number of process instances that write to the variable will increase, which will introduce more hardware resources in the synthesized implementation.

If the variable is declared without **sc_signal**, then each process that accesses the variable should initialize the variable.

14.6.4.2 Module Member Variable Accessed by Single Process Instance

If a member variable is written or read by a single process instance, then you should *not* use **sc_signal** in its declaration. **sc_signal** imposes certain restrictions in the hardware implementation to guarantee its semantics, which tends to introduce more resources in the synthesized implementation.

14.6.5 Global Variables

CtoS does not support global variables, regardless of whether they are **const**.

14.6.6 Static Variables

Class or module member variables with storage class **static** are supported if the type of the variable is **const**-qualified. Global variables with storage class **static** are not supported.

Example: Static Variables

```
struct scratch_pad {
    static const unsigned int N = 16;
    int buf[N];
};

SC_MODULE(FILTER) {
    ..
    static const unsigned int N_TAPS = 7;
    static const int COEFFS[N_TAPS];
};

const int FILTER::COEFFS[N_TAPS] = { 1023, 874, 328, 349, -302, -258, 123 };
```

14.7 Object Models

The *object model* dictates how data objects in the SystemC source, such as local variables and data members of modules, are mapped on to data objects in the CtoS design database, such as values in the data flow, nets, and arrays.

CtoS uses three object models, the *field-fragmented object model*, the *packed object model*, and the *monolithic object model*. The former is the default as of the 10.10 release; the latter was the default in releases prior to 10.10.

This section describes these models and related commands, attributes, and pragmas:

- [“Packed Object Model” on page 14-45](#)
- [“Monolithic Object Model” on page 14-47](#)
- [“Field-Fragmented Object Model” on page 14-48](#)
- [“CtoS Assumptions about Data Access” on page 14-49](#)
- [“Using the merge_arrays Command with Object Models” on page 14-51](#)
- [“Choosing the Object Model for a Struct” on page 14-51](#)
- [“Using the build_monolithic_structs Design Attribute” on page 14-52](#)
- [“Known Problems and Limitations with Object Models” on page 14-52](#)

14.7.1 Packed Object Model

The *packed object model* is used for objects of structs that have the **#pragma ctos packed** directive.

In the *packed object model*, the way data objects are mapped to objects in the CtoS database is based on the type of the top-level variable of which they are a part. A top-level variable is either a local variable of a function or a data member of a module. The object mapping is as shown in [Table 14-4 on page 14-46](#).

Note When a variable is turned into data flow during elaboration, the most salient aspects of the variable that remain in the CtoS database are the so-called join muxes. These are multiplexers for passing the content of the variable across nodes in the control flow that have more than one in-edge.

For objects of structs, the fields are laid out one after the other without any padding or alignment. See also [“Data Layout of Packed Classes and Structs” on page 14-64](#).

Table 14-4 Object Mapping in Monolithic Object Model

Type of Top-Level Variable	Object in CtoS Database
sc_signal<T>	net
(multi-dimensional) array of sc_signal	set of nets
(multi-dimensional) array of T	(1-dimensional) array
T	data flow with join muxes that are as wide as the bit width of T
where <i>T</i> is: • a built-in C++ integer data type (bool, char, short, int, ...) • a SystemC integer data type (sc_uint<8>, ...) • a CtoS fixed-point data type (ctos_fx::sc_fixed<8,4>) • a user-defined class or struct	

Example: Object Mapping According to Monolithic Object Model

```
#pragma ctos packed
struct A {
    bool a1;
    short a2[4];
};
#pragma ctos packed
struct B {
    short b1;
    A      b2[3];
};
SC_MODULE(M) {
    ..
    sc_signal<A> m_A_sig;
    sc_signal<A> m_A_sig_arr5[5];
    A            m_A;
    A            m_A_arr5[5];
    B            m_B;
};
```

The inferred objects in the CtoS database are:

- one 65-bit wide net (**m_A_sig**)
- five 65-bit wide nets (**m_A_sig_arr5_0**, ...**m_A_sig_arr5_4**)
- data flow for **m_A** with 65-bit wide join muxes

- one 5x65-bit array for **m_A_arr5**
- data flow for **m_B** with 211-bit wide join muxes

14.7.2 Monolithic Object Model

The *monolithic object model* is used for objects of structs that have the **#pragma ctos monolithic** directive. In the **monolithic object model**, the mapping to objects in the CtoS database is the same as for the packed object model.

The difference with the packed object model is that fields of an object of type struct are laid out on byte boundaries and with alignment similar to what most C++ compilers use. See [“Data Layout of Packed Classes and Structs” on page 14-64](#).

Example: Object Mapping According to Monolithic Object Model

```
#pragma ctos monolithic
struct A {
    bool a1;
    short a2[4];
};

#pragma ctos monolithic
struct B {
    short b1;
    A      b2[3];
};

SC_MODULE(M) {
    ..
    sc_signal<A>    m_A_sig;
    sc_signal<A>    m_A_sig_arr5[5];
    A              m_A;
    A              m_A_arr5[5];
    B              m_B;
};
```

The inferred objects in the CtoS database are:

- one 65-bit wide net (**m_A_sig**)
- five 65-bit wide nets (**m_A_sig_arr5_0, ...m_A_sig_arr5_4**)
- data flow for **m_A** with 65-bit wide join muxes
- one 5x80-bit array for **m_A_arr5** (CtoS can usually trim the width of this array 65-bits)
- data flow for **m_B** with 256-bit wide join muxes (CtoS can usually trim the width to 211-bits)

14.7.3 Field-Fragmented Object Model

The *field-fragmented object model* is the default object model. In this object model, data objects are mapped to objects in the CtoS database, as follows:

- First, CtoS determines the top-level data object (a local variable or data member of a module) of which the object is part.
- Then, CtoS breaks the top-level data object into components, based on the type of the data object:
 - If the object is a built-in C++ integer data type or SystemC integer data type, it is not further split.
 - If the object is of type **sc_signal<T>**, it is not further split.
 - If the object is of type user-defined struct/class, it is split into components, one component for each data-field of the struct, unless the declaration of struct/class is labeled with the **ctos packed pragma** or **ctos monolithic pragma** (see “[Packing Struct](#)” on page 14-75 and “[Keeping Struct Monolithic](#)” on page 14-75). In the latter case, the object is not further split.
 - If the object is an array of type **T** or multi-dimensional array of type **T**, the object is split into a number of arrays, one for each of the components of type **T** (according to the previous rules).
 - The previous rules are applied recursively, so a top-level object is split into a set of leaf-components that can be of one of the following types:
 - **sc_signal<T>**
 - (multi-dimensional) array of **sc_signal<T>**
 - (multi-dimensional) array of **MT**
 - **MT**

where T is an arbitrary type, and MT is one of the following:

 - a built-in C++ integer data type (**bool, char, short, int, ...**), or
 - a SystemC integer data type (**sc_uint<8>, ...**)
 - a user-defined enumerated type
 - a user-defined class or struct that is labeled with the **ctos packed pragma**
 - a user-defined class or struct that is labeled with the **ctos monolithic pragma**
- After breaking a top-level data-object into its leaf-components, each component is mapped onto a database object, using the same rules as for the *monolithic object model* ([Table 14-4 on page 14-46](#)).

Example: Object Mapping According to Field-Fragmented Object Model

```
struct A {
    bool a1;
    short a2[4];
};

struct B {
    short b1;
    A      b2[3];
};

SC_MODULE(M) {
    ..
    sc_signal<A>    m_A_sig;
    sc_signal<A>    m_A_sig_arr5[5];
    A              m_A;
    A              m_A_arr5[5];
    B              m_B;
};
```

The inferred objects in the CtoS database are:

- one 65-bit wide net **m_A_sig**
- five 65-bit wide nets (**m_A_sig_arr5_0**, ... **m_A_sig_arr5_4**)
- data flow for component **m_A_a1** with 1-bit wide join muxes
- one 4x16-bit array for component **m_A_a2**
- one 5x1-bit array for component **m_A_arr5_a1**
- one 20x16-bit array for component **m_A_arr5_a2**
- data flow for component **m_B_b1** with 16-bit wide join muxes
- one 3x1-bit array for component **m_B_b2_a1**
- one 12x16-bit array for component **m_B_b2_a2**

14.7.4 CtoS Assumptions about Data Access

The programmers' memory model offered by most C++ compilers consists, simplistically, of a few different address spaces: the stack, the heap, and the global data. Given the address of a variable allocated on the stack, it is possible, in principal, to access other variables on the stack if the program has knowledge about data layout and variable allocation used by the targeted C++ compiler.

More realistically, given the address of a field of a SystemC module, it is possible, in principal, to access the next field, by using knowledge about the data layout of the C++ compiler and by using pointer arithmetic and C-style casts; nothing in the C++ language would prevent a program from doing so. Inferring efficient hardware from a C++ program in the presence of such accesses, however, is impractical. CtoS thus makes assumptions, related to the object model, on accessing data objects:

- [“Assumptions on Data Access in Field-Fragmented Object Model” on page 14-50](#)
- [“Assumptions on Data Access in Packed Object Model” on page 14-50](#)

- “Assumptions on Data Access in Monolithic Object Model” on page 14-50
- “Limited Static Checking of Assumptions” on page 14-51

14.7.4.1 Assumptions on Data Access in Field-Fragmented Object Model

The main assumption about data access in the field-fragmented object model is that an address taken from a component of a top-level data object can be used only to access objects nested within that component, if the given component is a leaf-component.

Referring to “Object Mapping According to Field-Fragmented Object Model” on page 14-49, CtoS allows access to **m_A.a1** using the address of **m_A**, that is, **m_A.a1** can be accessed via **(&m_A)->a1** because the component of which **m_A.a1** is a part is nested underneath the composite-component of **m_A**.

Similarly, **m_A.a2[2]** can be accessed using the address of **m_A.a2[0]** because these objects are part of the same leaf-component.

However, access to **m_A.a2[1]** via the address of **m_A.a1** is forbidden because these objects are part of disjoint components.

This is related to *strict addressing*, which forbids out-of-bounds array accesses and also forbids pointer arithmetic on addresses taken from objects that are not part of an array.

14.7.4.2 Assumptions on Data Access in Packed Object Model

The objects laid out according to the packed object model, CtoS enforces the same restrictions as for the field-fragmented object model.

14.7.4.3 Assumptions on Data Access in Monolithic Object Model

The main assumption about data access in the *monolithic object model* is that an address taken from (an object nested underneath) a top-level data object (a local variable or a data member of a module) can be used only to access objects also nested underneath that same top-level object.

Referring to “Object Mapping According to Monolithic Object Model” on page 14-47, and assuming this design is elaborated according to the *monolithic object model*, CtoS supports access to field **a2** of **m_A** using the address of field **a1** of **m_A**, and using the knowledge about data layout used by most Linux 32-bit C++ compilers: Object **m_A.a2[1]** can be accessed via **(2 + &(m_A.a1))**

Another example is to take the address of **m_A** and cast it to a **char*** pointer, and then to access the 10 bytes that make up **m_A** via pointer arithmetic on that pointer.

14.7.4.4 Limited Static Checking of Assumptions

In general, Ctos cannot fully check statically the assumptions in the three previous sections (“Assumptions on Data Access in Field-Fragmented Object Model” on page 14-50, “Assumptions on Data Access in Packed Object Model” on page 14-50, and “Assumptions on Data Access in Monolithic Object Model” on page 14-50).

CtoS tries to catch some obvious violations for the field-fragmented object model; however, failure to meet the assumptions defined for either of these models can lead to simulation mismatches between the original design and the synthesized design.

14.7.5 Using the merge_arrays Command with Object Models

Consider the following example:

```
struct Point { short x,y;};
SC_MODULE(M) {
  ..
  Point points256[256];
};
```

Under the field-fragmented object model, CtoS infers two 256x16 bit arrays (**points256_x** and **points256_y**), from the variable **points256**.

Under the *monolithic object model*, a single 256x32-bit array **points256** is inferred from the variable **points256**.

To implement **points256** with a single RAM in the RTL, you can use the **merge_arrays** command with the **-data** option (“merge_arrays” on page E-86) to merge arrays **points256_x** and **points256_y**, before allocating the RAM.

```
merge_arrays -data [find -array points256_x] [find -array points256_y]=
```

14.7.6 Choosing the Object Model for a Struct

In most situations a suitable implementation of the design can be obtained with the default object model. This object model leads to the most choices as to whether a (part of an) object is to be implemented as a RAM or as dataflow registers. After build, you have the opportunity to transform arrays using the **flatten_array**, **merge_arrays**, **restructure_array**, **split_array** commands.

However, the array transformation commands are not supported for external arrays. In such cases, you can use the **pragma ctos packed** pragma to control the layout of arrays.

The monolithic object model should be used only in if the design has code where an object of type struct is treated as an array of bytes or vice versa using pointer casts. If the **ctos monolithic pragma** is specified for such structs, CtoS may be able to handle such designs. The user needs to be aware that padding and alignment may lead to loss of QoR. The monolithic object model should be avoided for structs that have fields of type a SystemC data type.

14.7.7 Using the build_monolithic_structs Design Attribute

The preferred way to control the object model is by using the **ctos monolithic** pragma (see “[Keeping Instance Boundary](#)” on page 14-76).

However, to ease gradual migration of legacy designs and CtoS Tcl command scripts, you can use the **set_attr** command (“[set_attr](#)” on page E-140) to set the **build_monolithic_structs** design attribute (“[build_monolithic_structs](#)” on page D-12). By default, this attribute is *false*, and the design is elaborated using the *field-fragmented object model*.

If this attribute is set to *true* before executing the **build** command (“[build](#)” on page E-30), the design will be elaborated using the *monolithic object model*.

14.7.8 Known Problems and Limitations with Object Models

There are currently a few known problems and limitations for object models:

- CtoS Tcl command scripts developed for release 9.31 or earlier may need adjusting, because CtoS may now infer more arrays from the design.

The adjustments typically involve the following commands:

- The **find** command with the **-array** option (“[find](#)” on page E-60) for conveniently obtaining the object ids of arrays.
- The **flatten_array** command (“[flex_channels_nets](#)” on page E-69) to convert small arrays to data flow.
- The **merge_arrays** command with the **-data** option (“[merge_arrays](#)” on page E-86) to merge multiple arrays into a single array.
- Classes/structs with base classes (data inheritance) are currently not supported for the *field-fragmented object model*. Such classes/structs must be labeled with the **ctos monolithic** pragma.

14.8 Data Types

The following data types are defined in this section:

- “Primitive C/C++ Data Types” on page 14-53
- “SystemC Data Types” on page 14-54
- “User-Defined Data Types” on page 14-57
- “Pointers” on page 14-66

14.8.1 Primitive C/C++ Data Types

In Table 14-5 on page 14-53 are the primitive C/C++ data types, their size and alignment (in bytes), and their level of support in CtoS. Many C++ compilers have switches to control the size of these data types; for example, to set the size of type **int** to 4 bytes or 8 bytes. However, CtoS supports only the sizes and alignments shown in the table. Therefore, if your program depends on the sizes and alignments of these types, configure your C++ compiler to match the sizes and alignments used by CtoS.

For example, the following code relies on the size and alignment of **int** being 4 bytes:

```
struct {short a; int b;} s1; // s1 has a 2-byte hole between 'a' and 'b'
short *sp = &s1;
short b_low = s1[2];        // read least significant 2 bytes of s1.b
```

Table 14-5 Primitive C/C++ Data Types

Type	Size in Bytes	Alignment in Bytes	Supported in CtoS?
bool	1	1	yes, but represented by CtoS as a 1-bit entity ¹
unsigned char	1	1	yes
char, signed char	1	1	yes
unsigned short	2	2	yes
short	2	2	yes
unsigned int	4	4	yes
int	4	4	yes
unsigned long	4	4	yes
long	4	4	yes
unsigned long long	8	4	yes
long long	8	4	yes

Table 14-5 Primitive C/C++ Data Types

Type	Size in Bytes	Alignment in Bytes	Supported in CtoS?
float	4	4	no
double	8	4	yes, but with restrictions ²
long double	16	16	no
pointer	4	4	yes

1. It is recommended that you use the '!' operator, instead of the '~' operator if you are using an argument of type **bool**.
2. Variables and expressions of type **double** are supported, if CtoS can resolve them to constant values. Array of **double**, pointer to **double**, and class/struct containing **double** are not supported.

14.8.2 SystemC Data Types

SystemC data types fall into two categories:

- “Non-Channel Data Types” on page 14-54
- “Channel Data Types” on page 14-56

14.8.2.1 Non-Channel Data Types

The following table shows the SystemC non-channel data types, their current level of support in CtoS, and pertinent comments.

Table 14-6 Non-Channel Data Types

Type	Supported in CtoS?	Comments
sc_uint<N>	yes	
sc_int<N>	yes	
sc_biguint<N>	yes	
sc_bigint<N>	yes	
sc_bv<N>	yes	

Table 14-6 Non-Channel Data Types

Type	Supported in CtoS?	Comments
sc_lv<N>	yes	X and Z are not supported
sc_bit	yes	
sc_logic	yes	X and Z are not supported
sc_ufixed<..>	yes	supported via the CtoS Fixed-Point Library (see “CtoS Libraries” on page 15-1)
sc_fixed<..>	yes	supported via the CtoS Fixed-Point Library (see “CtoS Libraries” on page 15-1)
sc_ufixed_fast<..>	no	
sc_fixed_fast<..>	no	

Notes

- CtoS internally represents objects of data types using the minimum number of bits required, which is for templated types if using the template parameter.
- For a field of a struct, where the type of the field is a SystemC data type, CtoS uses an alignment of 1 byte if the bit width of the field type is less than or equal to 8 bits. If the bit width of the field type is between 9 and 16 bits, CtoS uses an alignment of 2 bytes for the field. Otherwise, an alignment of 4 bytes is used.

Bit Selects

CtoS supports bit selects of supported SystemC data types.

Part Selects

CtoS supports part selects of supported SystemC data types with constant indices.

CtoS also supports part selects of supported SystemC data types with *non-constant* indices, provided that:

- The right index expression is an integer expression that does not have any side effects and does not require the evaluation of functions, and

- The left index expression is of the form **REXPR + CONST**, where **REXPR** is the right index expression and **CONST** is a non-negative integer compile-time constant.

Example of part select usage:

```
sc_uint<48> val; int i,j;
..
val.range(i+3, i) = ..;    // supported
val.range(i,j) = ..;      // not supported
..
.. = val.range(i+3, i);    // supported
.. = val.range(i,i-3);    // not supported
```

Concatenations

CtoS supports concatenations of supported SystemC data types.

14.8.2.2 Channel Data Types

The following table shows the SystemC channel data types, their current level of support in CtoS, and pertinent comments.

Table 14-7 Channel data types

Type	Supported in CtoS?	Comments
sc_signal<T>	yes	
sc_in<T>	yes	
sc_out<T>	yes	
sc_inout<T>	no	
sc_port<T>	yes	supported for interfaces internal to the module to be synthesized
sc_buffer<T>	no	
sc_signal_resolved	no	
sc_inout_resolved	no	
sc_out_resolved	no	

Table 14-7 Channel data types

Type	Supported in CtoS?	Comments
sc_signal_rv<W>	no	
sc_inout_rv<W>	no	
sc_out_rv<W>	no	
sc_fifo<T>	no	
sc_mutex	no	
sc_semaphore	no	
sc_event_queue	no	
sc_in_clk	yes	
sc_out_clk	no	

14.8.3 User-Defined Data Types

The following table shows user-defined data types and how they are supported in CtoS.

Table 14-8 User-Defined Data Types

Construct	Supported in CtoS?
enum	yes
struct	yes
union	no
class	yes

This section has the following subsections:

- “Structures” on page 14-58
- “Classes” on page 14-65

14.8.3.1 Structures

Information about structures is presented in the following sections:

- “Declaration and Usage of Structures” on page 14-58
- “Initialization of Objects of a struct Type” on page 14-60
- “Access to Fields of Structs” on page 14-61
- “Member Functions” on page 14-61
- “Single Inheritance” on page 14-61
- “Multiple Inheritance” on page 14-61
- “Virtual Functions” on page 14-62
- “Virtual Inheritance” on page 14-63
- “Constructors” on page 14-63
- “Destructors” on page 14-63
- “Casts” on page 14-63
- “Data Layout of Integer Data Types” on page 14-64
- “Data Layout of Packed Classes and Structs” on page 14-64
- “Data Layout of Monolithic Classes and Structs” on page 14-64
- “sizeof” on page 14-65

Declaration and Usage of Structures

A structure is a data type defined by a designer. CtoS supports structures declared as follows:

```
struct struct_name {  
    type1  member1;  
    type2  member2;  
    ...  
    Type1  member_function1(signature);  
    Type2  member_function2(signature);  
    ...  
} objects;
```

where

- **struct** is a keyword.

- **struct_name** is the name given by a designer for the structure being defined.
- **type1** is a supported data type.¹
- **member1** is the name of an object or a pointer to an object for **type1**
- **member_function1** is the declaration of a member function of the structure
- **Type1** and **signature** are the return type and signature of **member_function1**, respectively
- **objects** is a set of names of objects or pointers to objects of this structure:
 - If more than one element is specified for **objects**, these elements must be separated by a comma.
 - Specifying **objects** is optional, but if it is specified, then **struct_name** may or may not be specified.
 - If **objects** is not specified, then **struct_name** must be specified.

1. The data types do not include channel classes, such as **sc_signal**, **sc_port**, **sc_clock**, or any data types that involve such classes.

Example: Defining a struct

This example defines a **struct** named *packet* that has three fields and two member functions:

```
struct packet {
    sc_uint<12>    id;
    sc_uint<8>     src;
    sc_uint<8>     dst;

    sc_uint<12>    getId() { return id; }
    bool          operator == (const packet &p) const;
};

bool packet::operator==(const packet &p) const {
    return id == p.id && src == p.src && dst == p.dst;
}
```

A declared structure may be used as a data type. Note that if a structure is used as a parameter of a template class, the template class may require the structure to include certain operators as its member functions. For example, in SystemC, if a structure is used as the data type of an **sc_signal** or **sc_port**, the declaration of the structure must have the equality operator `==` as its member function. This is illustrated in the following example, which builds on the previous example:

Example: Defining a struct (Continued)

```
SC_MODULE(M1) {
    sc_in< packet > packet_port;
    sc_signal< packet > first, last;
    ..
    void execute() {
        ..
        first.write( packet_port.read() );
        ..
    }
};
```

Initialization of Objects of a struct Type

Objects of structure types that do not define a constructor can be initialized using curly-brace syntax. CtoS supports the initialization of automatic variables of such structures using curly-brace syntax.

This is illustrated in the following example, which again builds on the previous example.

Example: Defining a struct (continued 2)

```
packet p1 = { 0x1, 0x8, 0xff };
```

Access to Fields of Structs

CtoS supports access to the fields of a struct using the dot operator ('.'). Fields may also be accessed using the arrow operator ('->').

Example: Access to Fields of structs Using Dot and Arrow

```
packet *findById(packet *packets, int num, sc_uint<12> id) {
    packet *p = packets;
    for (int i = 0; i < num; i++, p++) {
        if (p->id == id) {                                // indirect access
            return p;
        }
    }
    return NULL;
}

..
packet p1;
p1.id = 0x11;                                           // direct access
p1.src = 0x44;
p1.dst = 0x22;
packet arr[8];
arr[1] = p1;

..
packet *p2 = findById(arr, 8, 0x234);
```

Member Functions

CtoS supports member functions and member operators of structures. “[Defining a struct](#)” on page 14-60 illustrates:

- a member function **packet::getId()**
- a member operator **packet::operator==(const packet &p)**

Single Inheritance

Single inheritance is supported. This means that a struct or class can have one direct base class. SystemC data types, such as **sc_uint<24>**, are not supported as base classes. In other words, inheritance is supported only for base classes that you define yourself.

Multiple Inheritance

Multiple inheritance is not supported. This means that a struct or class cannot have more than one direct base class. Multiple inheritance is supported only for specifying TLM interfaces.

Virtual Functions

Virtual functions are supported. This means that a class can define a virtual function, and that virtual function can be overridden in a class derived from the given class.

Caveat Virtual functions in non-**sc_module**, non-**sc_interface** classes are supported only if the class that declares the virtual function, and all the classes derived from this class, use the *monolithic object model* (see “[Monolithic Object Model](#)” on page 14-47).

Virtual functions should be used with caution in modeling. Consider the following snippet of code:

```
struct A {
    virtual int f() { .. }
    ..
};

struct B1: A {
    virtual int f() { .. }
    ..
};

struct C1: B1 {
    ..
};

struct B2: A {
    virtual int f() { .. }
    ..
};

int g(A *a) {
    int tmp = a->f(); // virtual function call.
    ..
}
```

Consider a call to the virtual function **f**. This is elaborated as a set of conditional function calls, one function call for each override of that virtual function (in the example **A::f**, **B1::f**, **B2::f**). If CtoS can determine the type of the object on which the virtual function is called (in this case ***a**), only one of those override calls will remain. However, this may require loop unrolling and inlining. If the type of the object on which the virtual function is called *cannot* be determined statically, or if CtoS is unable to do that, all overrides will remain.

Also, objects of dynamic classes will contain one additional 32-bit field corresponding to the virtual table pointer. Again, CtoS may be able to optimize away this field if it is able to resolve all virtual function calls on the object statically.

Virtual Inheritance

Virtual inheritance is not supported. This means that a struct or class cannot have a virtual base class. Virtual inheritance is supported only for specifying TLM interfaces.

Constructors

CtoS supports the initialization of an automatic variable using a constructor of its struct or class type, for example:

```
class Point {
public:
    Point(): m_x(0), m_y(0) { }           // Default constructor
    Point(int x, int y) {m_x = x; m_y = y; } // Second constructor
    ..
private:
    int m_x, m_y;
};

void func() {
    Point origin; // initialization of 'origin' via default constructor
    Point x1(1,0); // initialization of 'x1' via 2nd constructor
    ..
}
```

Note Structures that declare a constructor cannot be initialized using curly-brace syntax.

Destructors

CtoS ignores any behavior associated with destructors.

Casts

CtoS does not support casts of objects of user-defined structs and classes. Pointer casts are supported and are described in [“Pointer Casts” on page 14-68](#).

Data Layout of Integer Data Types

CtoS follows a convention related to the SystemC convention of assigning bit 0 to the LSB; therefore, in a struct implemented as a bit vector, the LSB of the first field is assigned index 0 (LSB of the vector), and the MSB of the last field is assigned the highest index (MSB of the vector).

Data Layout of Packed Classes and Structs

Fields of a packed struct are laid out one after the other without any padding or alignment. In the example below.

Example: Data Layout of Packed Classes and Structs

```
#pragma ctos packed
struct S {
    bool          f1; // offset: bit 0
    short         f2; // offset: bit 1
    unsigned char f3; // offset: bit 17
}; // width of S = 25 bits
S buf[2];
```

If your design has packed structs, the code that manipulates object of such data structures must not rely on the actual data layout because the layouts used by CtoS and regular C++ compilers differs.

Data Layout of Monolithic Classes and Structs

Fields of a monolithic struct are aligned on boundaries that are a multiple of N bytes, where N is the alignment of the field type (see "Primitive C/C++ data types" on page 14-52). This leads to gaps between the fields. In the example on the following page, there is a so-called hole of 1 byte between field **f1** and field **f2**, because the type of field **f2** requires an alignment of 2 bytes. Also, there is a gap of 1 byte between **buf[0].f3** and **buf[1].f1** because the alignment of S is 2.

Example: Data Layout of Monolithic Classes and Structs

```
#pragma ctos monolithic
struct S {
    unsigned char f1; // offset: byte 0
    short        f2; // offset: byte 2
    unsigned char f3; // offset: byte 4
}; // sizeof(S) = 6 bytes, alignment(S) = 2 bytes.
S buf[2];
```

The internal representation that CtoS uses for SystemC types is not compatible with that used by C++ compilers. Therefore, objects of data structures built using SystemC data types should be handled by the design in a way that does not rely on the data layout of these objects used by a C++ compiler.

sizeof

CtoS supports the **sizeof()** operator on user-defined classes and structs. For primitive C++ types, CtoS evaluates the **sizeof()** operator as given in “[Primitive C/C++ Data Types](#)” on page 14-53, which is consistent with C++ compilers.

For structures built entirely of primitive C++ types, the interpretation of **sizeof()** by CtoS matches that of C++ compilers. For SystemC data types and structures that contain SystemC data types, CtoS evaluates the **sizeof()** operator in a manner consistent with its data representation for such types, but that does not match the behavior of C++ compilers.

14.8.3.2 Classes

Structures are simply classes for which all members have public access by default.

CtoS supports classes in the same manner as it supports structures.

14.8.4 Pointers

This section on pointers is organized as follows:

- “Terminology Used with Pointers” on page 14-66
- “Considering CtoS Memory Model when Using Pointers” on page 14-66
- “Pointer Usage Modes” on page 14-67
- “Potential Run-Time Errors with Pointers” on page 14-68
- “Pointer Casts” on page 14-68
- “Limitations when Using Pointers” on page 14-69

14.8.4.1 Terminology Used with Pointers

To clarify, the following terminology is used by CtoS around the topic of pointers:

- The term *design* is used to refer to the SystemC module to be synthesized by CtoS.
- The term *variable* is used to refer to an automatic variable declared in a function scope or a field of a SystemC module.
- A *pointer expression* is an expression of a pointer type.
- A *pointer object* is a variable, a field, or an element of a variable of type pointer.
- A *pointer value* is a value of a pointer type, which can be the address of some other object, a pointer arithmetic expression, NULL, or the result of casting some other value to a pointer type.

14.8.4.2 Considering CtoS Memory Model when Using Pointers

A C programmer’s view of memory is a large, monolithic, linearly addressed storage of bytes. In principle, a C programmer can take the address of one object allocated on the heap and use it to access any other object on the heap.

CtoS views memory as a collection of small linear memories of bytes – one associated with each variable of the design – and each of these memories has its own address space. Addresses belonging to different address spaces are incomparable, and it is impossible to access an object belonging to the memory of variable *v1* using an address taken from an object belonging to the memory of variable *v2*.

14.8.4.3 Pointer Usage Modes

CtoS supports two main usage modes for pointers:

- *External pointers* are pointers that do not point to any object known to CtoS. Consequently, such pointers cannot be dereferenced and are treated essentially like integers, except the rules of arithmetic are slightly different.
- *Internal pointers* are pointers that are dereferenced in the design; therefore, these pointers must also be assigned the address of some variable of the design.

Support for internal pointers requires CtoS to analyze the data flow between address-of expressions, for example, **&a**, and pointer dereference expressions, for example, ***prt**. To make such analysis practical, CtoS stops tracking the variables to which a pointer points when a pointer is cast to a non-pointer type. Similarly, when a non-pointer is cast to a pointer type, the resulting pointer is considered an external pointer by CtoS. Therefore, internal pointer usage must not be mixed with external pointer usage.

Internal pointer usage can be further classified as follows:

- *Local pointers* are pointers that act like references to variables and are typically used as formal arguments of functions that must return more than one value. For a pointer to be a local pointer, only the following uses are allowed:
 - dereference expression (star or arrow operator)
 - left-hand side of assignment, where the right-hand side is the address of a variable
 - actual argument of a function call for which the corresponding formal variable is also a local pointer
- An *array pointer* is simply an alternate mechanism for indexing into an array. In addition to the uses allowed by local pointers, array pointers also allow pointer arithmetic, comparison against other array pointers, and the index operator. Array pointers can be assigned the address of an array variable or the value of another array pointer. Array pointers are typically used for formal arguments of functions that need to access arrays.
- A *general internal pointer* is a pointer that may be used in comparisons against NULL and casts to another pointer type, in addition to the uses allowed by array pointers. General internal pointers can be assigned NULL, the address of a variable, or the value of another internal pointer. A typical example is the next pointer used in linked lists.

Pointers can increase the variability of the design, because they imply different behaviors depending on whether they are NULL or not NULL, and depending on the variable to which they point. Consequently, general internal pointers imply more variability than array pointers, and array pointers imply more variability than local pointers. Pointers are used judiciously if their use implies only as much variability as is really needed. Unneeded variability will lead to inferior QoR, compared to similar designs that do not use pointers.

14.8.4.4 Potential Run-Time Errors with Pointers

For any given pointer dereference expression, CtoS assumes that the pointer points to a valid object in a design. It is hard to check, in general, whether this assumption is valid, and CtoS does not make any attempt to check it.

In the following example, “[Potential Run-Time Error with Pointers](#)” on page 14-68, CtoS determines that pointer object **p** is dereferenced in the **return** statement. Since **p** is assigned the address of **a**, CtoS elaborates the dereference of **p** as a reference to **a**. However, if **func** is called with **c1** false and **c2** true, **p** is uninitialized at the time it is dereferenced. This would result in a run-time error (in the simulator). CtoS does not check for this type of problem.

Example: Potential Run-Time Error with Pointers

```
int func(bool c1, bool c2, int a)
{
    int *p;
    if (c1) {
        p = &a;
    }
    if (c2) {
        return *p;
    }
    return 0;
}
```

In general, CtoS *does not* check whether a design could suffer from run-time errors. It assumes that within the legal operating environment of the design, run-time errors will never happen. CtoS considers the behavior of the design for conditions where the input model would suffer a run-time error as a *don't care*. Run-time errors include the following:

- access memory beyond the memory associated with each variable
- index into an array with an index that is out of bounds
- dereference of a pointer that is NULL
- dereference of an external pointer
- dereference of an un-initialized pointer
- divide by 0

14.8.4.5 Pointer Casts

CtoS supports C-style pointer casts from type **T1*** to type **T2***, as long as types **T1** and **T2** are either supported primitive C/C++ types or built out of these types. Such pointer casts must be used with great care. The presence of holes in the data layout of either **T1** or **T2** could easily lead to unintended results.

Pointer casts should not be used when SystemC data types are involved because the internal representation of objects of such types is incompatible with the internal representation of such objects in a C/C++ compiler.

14.8.4.6 Limitations when Using Pointers

CtoS currently has the following limitations regarding the use of pointers:

- *CtoS does not track pointers across casts to non-pointer types.* This means that after you cast a pointer to a non-pointer type, CtoS is unable to resolve a dereference of the result of the cast.

```
short a, *p = &a, *q;
int u;
*p = 3;                      // OK
int u = (int) p;              // This cast causes CtoS to stop tracking p.
q = (int*) u;                  //
*q = 5;                        // Failure to resolve pointer dereference.
```

- *CtoS does not support pointer casts from type **T1*** to type **T2***, if*
 - the resulting pointer is dereferenced, *and*
 - the resulting pointer points into an array variable, *and*
 - *either* the alignment of **T2** is greater than the alignment of **T1**,
 - *or* the **sizeof(T2)** is greater than the **sizeof(T1)**.

```
typedef unsigned char UCHAR;
struct S { UCHAR c1,c2,c3};    // alignment(S) = 1, sizeof(S) = 3
S arr[5];
short *sp = (short *) arr;     // alignment(short)=2, sizeof(short)=2
                                // BAD cast: CtoS-11175
UCHAR *cp = (UCHAR*) arr;      // alignment(UCHAR)=1,
                                // sizeof(UCHAR)=1
                                // cast is OK
sp[i] = ...;                   // This may be partially writing 2 elements of arr!
cp[i] = ...;                   // OK because ts accesses only 1 element of arr
```

- *CtoS does not support bit selects and part selects on pointer dereferences.*

```
void func1(sc_uint<24> *a) {
    a->range(12,8) = 0;         // Not supported: use the workaround below:
    sc_uint<24> tmp = *a;
    tmp.range(12,8) = 0;
    *a = tmp;
}
```

- *CtoS does not support comparisons or pointer differences between pointers that point into different variables.* This follows from the memory model used by CtoS.

```
short a[10], b[10], *p = &a, *q, *r;
p = a;
q = &(a[3]);
r = b + 2;
int diff1 = q-p;               // OK: both point into 'a'
int diff2 = r-p;               // Not OK.
```

- *CtoS does not support pointer objects that point into more than 16 variables.*

```
short a0, a1, a2, ... a16, *p;
switch (i) {
    case 0: p = &a0; break;
    case 1: p = &a1; break;
    case 2: p = &a2; break;
    ...
    case 15: p = &a15; break;
    default: p = &a16; break;
}
*p = 12;    // Not supported because p points to more than 16 variables.
```

However, there are many workarounds for this situation. Here are two possibilities:

- If you have, for example, 17 3-element arrays of `sc_uint<1>`, you could change them to 17 `sc_uint<3>` variables.
- You could make the function whose formal argument refers to more than 16 variables a *templated function*, and then do the following:

```
    sc_uint<3> ary2uint3(sc_uint<1> []);
..
OUTA0 = ary2uint3(tmpa0);
OUTA1 = ary2uint3(tmpa1);
..
OUTA15 = ary2uint3(tmpa15);
OUTA16 = ary2uint3(tmpa16); // NG
..

=>
    template<int K>
    sc_uint<3> ary2uint3(sc_uint<1> []);

..
OUTA0 = ary2uint3<0>(tmpa0);
OUTA1 = ary2uint3<0>(tmpa1);
..
OUTA15 = ary2uint3<0>(tmpa15);
OUTA16 = ary2uint3<1>(tmpa16);
```

- *CtoS does not support pointer to SystemC channel/module types.*
- *CtoS does not support pointer to SystemC port/export types.*
- *CtoS does not support pointer to member types.*
- *CtoS does not support pointer to function types.*

14.9 Using the CtoS Pragmas

This section describes how to use CtoS pragmas in your SystemC code to improve the results:

- “Skipping Variable Initialization” on page 14-71
- “Keeping Signals” on page 14-72
- “Packing Struct” on page 14-75
- “Keeping Struct Monolithic” on page 14-75
- “Keeping Instance Boundary” on page 14-76
- “Annotating Function Calls” on page 14-76

14.9.1 Skipping Variable Initialization

CtoS initializes local variables at the point where they are declared in accordance with the C++ standard. If the type of the variable has a default constructor, the variable is initialized even if no initializer is specified at the declaration. This is the case for all SystemC integer datatypes. In the example below, no initializer is specified for variable **a**, but the default constructor of class **sc_uint<8>** sets the number to 0, and so variable **a** is actually initialized to 0.

```
sc_uint<8> a;
```

In some cases this behavior is not desirable. Consider the code fragment below:

```
sc_uint<8> table1[256];  
for (int i = 0; i < 256; i++) {  
    table1[i] = in.read()  
    wait();  
}  
// code that reads from table1 follows
```

According to C++ semantics, every element of variable **table1** is set to 0 at the declaration of **table1**. However, the for-loop immediately overwrites each element, and so the initialization to 0 is redundant. Nevertheless, if variable **table1** is implemented by a RAM and the loop is not unrolled, scheduling of the code fragment would fail, because there are insufficient states to schedule the 256 writes that set each element of **table1** to 0.

To address this problem, CtoS supports **ctos dont_initialize_variable** pragma. If the declaration of a local variable is directly preceeded by this pragma, CtoS will ignore any explicit or implicit initialization of that variable. However, in the example below, because of the pragma, the synthesized design will not set every element of **table1** to 0. Thus, if variable **table1** is implemented by a RAM and the loop is not unrolled, there would not be any problems scheduling the code fragment. For example:

```
#pragma ctos dont_initialize_variable
```

```
sc_uint<8> table1[256];
for (int i = 0; i < 256; i++) {
    table1[i] = in.read()
    wait();
}
// code that reads from table1 follows
```

Warning Using of the **ctos dont_initialize_variable pragma** changes the semantics of the code, and may result in simulation mismatches between the synthesized design and the original design. Therefore, make sure that the pragma is used only in situations where ignoring the initialization does not change the overall functionality of the design.

If the **-verbose** option of the **build** command is specified, CtoS issues an info message when a **ctos dont_initialize_variable pragma** is being processed. For example:

```
dut.cpp:43:16: Ignoring initialization of variable 'arr'
```

14.9.1.1 Declarations with Multiple Variables

When multiple variables are declared in the same statement, and the statement is preceeded by **ctos dont_initialize_variable pragma**, CtoS ignores the initialization only for the first variable.

In the example below, CtoS ignores the initialization of the elements of variable **table1**, but not the initialization of the elements of **table2**.

```
#pragma ctos dont_initialize_variable
sc_uint<8> table1[256], table2[256];
```

If the initialization of both variables is to be ignored, the code needs to be rewritten as follows:

```
#pragma ctos dont_initialize_variable
sc_uint<8> table1[256];
#pragma ctos dont_initialize_variable
sc_uint<8> table2[256];
```

14.9.2 Keeping Signals

White-box verification is a method for verifying that uses knowledge about the internal structure of the design to be verified. A white-box verification plan may include assertions and coverage points expressed on the internal state of the design.

A difficulty applying white box verification along with high-level synthesis is that there is no direct correspondence between the internal state of the synthesized design and that of the original design.

CtoS preserves the signal ports of the top-level module, and there is a one-to-one correspondence between the signal ports of the original design and the ports of the synthesized design. This is also true for **sc_signal** objects inside the design, except for that CtoS may optimize away **sc_signal** objects that have no readers. If design attribute **keep_all_signals** is set to true before build, all **sc_signal** objects are preserved through synthesis regardless of whether they are used.

One way to apply white-box verification along with high-level synthesis is to observe properties and coverage points on the internal state of the design using **sc_signals**. This consists of:

- adding **sc_signal** fields to the modules of the design.
- assigning those **sc_signals** during reset (optional).
- writing the outcome of an assertion or the value of a coverage point to the signal dedicated to observing that assertion/coverage point.

To make it easier to apply white box verification CtoS now provides:

- a new pragma for specifying that a particular **sc_signal** needs to be preserved through synthesis.
- a new attribute **keep_signal** of net

In the example below, the declaration **sc_signal assertion1** has been labeled with the **ctos keep_signal pragma**. Consequently, the net inferred from this **sc_signal** will be preserved through synthesis and the **keep_signal** attribute of this net is true.

```
SC_MODULE(DUT) {  
    ..  
    #pragma ctos keep_signal  
    sc_signal<bool> assertion1;  
    ..  
};
```

If the **-verbose** option of the **build** command is specified, CtoS issues an info message when the **keep_signal** attribute of a net is being set to true because of a **ctos keep_signal pragma**. For example:

```
..  
[SystemC Elaborator] Elaborating design in hierarchical mode ...  
[SystemC Elaborator] Processing function DUT::main  
dut.cpp:30:28: Setting attribute 'keep_signal' to true for net 'assertion1'
```

If the design has **sc_signals** that have the **ctos keep_signal pragma**, CtoS will not remove the nets inferred from these signals, even if they are not needed, and this may result in inferior QoR. To make you aware of this, CtoS now issues a warning message at the time that a module reaches the analyze micro-architecture step. For example:

```
WARNING (CTOS-17140): Removal of 1 redundant nets in module 'DUT' for a
total of 32 bits has been prevented by the 'keep_all_signals' design
attribute or the 'ctos keep_signal pragma'; this may result in inferior
QoR.
```

14.9.2.1 Finding the Nets Inferred from sc_signals with the Pragma in the RTL

To help you find the nets inferred from **sc_signals** that have the **ctos keep_signal pragma** in the RTL, the verilog models contain comments directly preceding a module declaration that explain the mapping. For example:

```
//
// Mapping of nets that have keep_signal=1:
//
// SystemC:sub1.my_assertions[0] maps to verilog:sub1_my_assertions_0_
// SystemC:sub1.my_assertions[1] maps to verilog:sub1_my_assertions_1_

module dut_rtl(...)
```

14.9.2.2 Declarations with Multiple Fields

If a **ctos keep_signal pragma** is specified right before a declaration of multiple fields of the same type (comma syntax), the pragma applies only to the first field.

In the example below, the pragma applies only to `assertion1`. Consequently, if there are no processes writing to `assertion2` it will be optimized out. Signal `assertion1` will be preserved through synthesis regardless of whether it is used.

```
SC_MODULE(DUT) {
    ..
    #pragma ctos keep_signal
    sc_signal<bool> assertion1, assertion2;
    ..
};
```

14.9.3 Packing Struct

You can specify that a struct is to be elaborated according to the packed object model with the **ctos packed pragma**. The pragma must occur immediately before the definition of the struct.

Note There is one restriction: If a class/struct **T1** has the **ctos packed pragma**, then a class/struct **T2** or array of **T2** that is instantiated as a field of **T1** or a base class of **T1** must also have the **ctos packed pragma**.

Example: Using ctos packed pragma

```
#pragma ctos packed
struct A {
    bool a1;
    short a2[4];
};
SC_MODULE(M) {
    ..
    A m_A;
    A m_A_arr5[5];
};
```

In the example above, **m_A** is elaborated as data flow with 65-bit join muxes. Similarly, **m_A_arr5** will be elaborated as a 5x65-bit array.

14.9.4 Keeping Struct Monolithic

For rare cases in which a design requires access to structs using a **char*** pointer, CtoS provides the **ctos monolithic pragma**. If the pragma immediately precedes the definition of a class/struct, that class/struct will be elaborated under the *monolithic object model*. (The remainder of the design will still use the *field-fragmented object model*.)

Note There is a restriction that, if a class/struct **T1** has the **ctos monolithic pragma**, then a class/struct **T2** or array of **T2** that is instantiated as a field of **T1** or a base class of **T1** must also have the **ctos monolithic pragma**.

Example: Using ctos monolithic pragma

```
#pragma ctos monolithic
struct A {
    bool a1;
    short a2[4];
};
SC_MODULE(M) {
    ..
    A m_A;
    A m_A_arr5[5];
};
```

```
};
```

Module member variable **m_A** is elaborated as 80-bit data flow with 80-bit join muxes. Similarly, **m_A_arr5** will be elaborated as a 5x80-bit array. Usually, CtoS can reduce the widths from 80-bits to 65-bits.

14.9.5 Keeping Instance Boundary

There are situations in which an **sc_module** meets one of the first three conditions mentioned in section “[Factors that Cause an sc_module to Be Collapsed](#)” on page 16-2, but the module is used in such a way that none of the member functions and none of the fields of the module are accessed from outside that module, with the exception of **sc_in** or **sc_out** ports. In this case, you can specify that an instance of that **sc_module** needs to be preserved using the **ctos keep_instance pragma**.

In the example below, **sc_module** M1 is instantiated in **sc_module** M2. Normally, M1 would be collapsed in M2, because it has an **sc_export** field. However, the instance of M1 has the **ctos keep_instance pragma** that directs CtoS to preserve M1.

```
SC_MODULE(M1) {  
    ..  
    sc_export<IF1> m_export;  
    ..  
};  
SC_MODULE(M2) {  
    SC_CTOR(M2);  
    #pragma ctos keep_instance  
    M1 IM1;  
};
```

14.9.6 Annotating Function Calls

The **ctos annotation pragma** allows you to specify the annotation of a function call or process in the CtoS SystemC source code. This helps to identify wait statements and function calls in the source code.

```
void DUT::main() {  
    ..  
    #pragma ctos annotation F_control  
    control();  
    ..  
}
```

The name of the function call annotation must be unique within the function in the SystemC source. CtoS issues a user error if 2 or more function call pragmas in a function definition have the same annotation name.

Example 1 Using the pragma on labeled wait(N)

For a call to `wait(int)`, multiple state nodes will result. If that wait statement is labeled with **L**, then the annotation for the first state is **L**, the annotation for the second state is **L_1**, and so on. The annotations of the resulting states from the following example is listed in [Table 14-9 on page 14-77](#).

```
void DUT::main() {
    ..
    INIT: wait(3);
    ..
}
```

Table 14-9 Annotations of a Labeled wait(N)

State Node	Annotation	Comment
INIT	INIT	First wait
INIT_1	INIT_1	Second wait
INIT_2	INIT_2	Third wait

Example 2 Using the Pragma for a Function Call on an Element of Array

For a function call on an element of an array of (pointer-to) module, port, and export, and the element is specified with a non-constant index, then multiple custom ops will result, one for each element of the array. The calls are conditional so that at most one will be active.

If **ctos annotation pragma** is associated with such a call, the annotation element for the resulting custom ops is not set to the name specified in the pragma, instead the annotation element is suffixed with the array index of the element. In the following example, three custom ops are elaborated and the annotation element of these ops are listed in [Table 14-10 on page 14-78](#):

```
SC_MODULE(DUT) {
    ..
    void main();
    SUB m_subs[3];
};

void DUT::main() {
    ..
    #pragma ctos annotation GET
    int value = m_subs[i].get();
    ..
}
```

Table 14-10 Annotations of an Array Element

Custom Op	Annotation	Comment
get_ln10	GET_0	This corresponds to m_subs[0].get()
get_0_ln10	GET_1	This corresponds to m_subs[1].get()
get_1_ln10	GET_2	This corresponds to m_subs[2].get()

In case the array of module is indexed with a number literal, only 1 custom op is elaborated, and the annotation element of that op is as specified in the pragma. For example:

```
void DUT::main() {  
..  
    #pragma ctos annotation GET  
    int value = m_subs[1].get();  
..  
}
```

Example 3 Using the Pragma on Nested Function Calls

Annotations will not be elaborated for function calls nested in expressions. For example, no annotations will be annotated for the following code fragment:

```
#pragma ctos annotation CALL1  
int value = 1 + func1(); // call is nested in arithmetic expression.
```

Annotations will be elaborated from annotation pragmas only if the function call appears in one of the following:

Expression Statement

```
#pragma ctos annotation CALL1  
func1();
```

Variable Declaration

```
#pragma ctos annotation CALL2  
int value = func1();
```

Return Statement

```
#pragma ctos annotation CALL3  
return func1();
```

14.10 Specifying Synthesis Directive

This section provides examples of how to specify a pragma for each source constructs and also lists the types of inferred objects:

- “Statements” on page 14-79
- “Declarations” on page 14-82

14.10.1 Statements

The section discusses about the following statements:

- “Block Statement” on page 14-79
- “The do, for, and while Statements” on page 14-81
- “Body Block of do, for, and while Statements” on page 14-81
- “Declarations” on page 14-82

14.10.1.1 Block Statement

Example

```
void Module1::process()
{
    // init
    ..
    #pragma ctos constrain_latency -max 32 -name body_latency
    {
        #pragma ctos unroll_loop
        for (..) { .. }
        #pragma ctos unroll_loop
        for (..) { .. }
    }
    ..
    // end
    ..
}
```

Compatible Directives	Inferred Object	Inferred Secondary Object
constrain_latency	the node associated with '{	the node associated with '}'
create_protocol_region		
float_array_accesses		
float_array_accesses		

Limitations:

If the block contains statements that cause a jump out of the block (break/continue/goto/return), the close curly brace may actually be unreachable, which will cause the directive to be deleted. Therefore, it is recommended to avoid directives on a block if the block has statements that jump out of the block.

Example

In the code below no directives are present after build because the close-brace of the block with the pragma is not reachable.

```
for (..) {
..
#pragma ctos constrain_latency -max 5
{
..
continue;
} // This line is not reachable
..
}
```

In the following code the **continue** statement has been moved outside the block with the pragma. In this case, the line with the close-brace is reachable, and the design has a directive after build.

```
for (..) {
..
#pragma ctos constrain_latency -max 5
{
..
} // This line is reachable
continue;
..
}
```


14.10.1.2 The do, for, and while Statements

Example

```
void Module1::process()
{
    #pragma ctos unroll_loop
    for (int i = 0; i < num_agents; i++) {
        out[i].write(0);
    }
    ..}
```

Compatible Directives	Inferred Object	Inferred Secondary Object
break_combinational_loop	loop join node	none
pipeline_loop	loop join node	none
unroll_loop	loop join node	none
create_protocol_region	loop join node	none
constrain_latency	loop join node	none
float_array_accesses	loop join node	end node of body of loop
float_io_accesses	loop join node	none

14.10.1.3 Body Block of do, for, and while Statements

If the body of a do, for, or while statement is a block statement, pragmas for that block are supported like they are supported for any other block statment.

Example

```
for (..)
#pragma ctos create_protocol_region
{
    ..
}
```

Compatible Directives	Inferred Object	Inferred Secondary Object
constrain_latency	the node associated with '{'	the node associated with '}'
create_protocol_region		
float_array_accesses		
float_array_accesses		

14.10.2 Declarations

This section covers the following:

- “Function Definitions” on page 14-82
- “Body Block of Functions” on page 14-83
- “Local Variables” on page 14-83
- “Declarations with Multiple Variables” on page 14-84
- “Static Member Variables” on page 14-84
- “Module Member Variables” on page 14-85
- “Class Member Variables where Class Is Not a Module” on page 14-85

14.10.2.1 Function Definitions

Example

```
#pragma ctos inline
int global_func1(int)
{ .. }

#pragma ctos create_protocol_region
int Module1::member_func1(int)
{ .. }

#pragma ctos pipeline_function
template<typename T>
int Module1<T>::member_func2(int)
{ .. }
```

Compatible Directives	Inferred Object	Secondary Object
convert_to_lookup	behavior	none
inline		
pipeline_function		
set_attr		
use_dsp		
use_ip		
create_protocol_region	behavior	none
float_io_accesses	origin node of behavior	end node of behavior

Notes

- Pragmas for creating behavior attributes must be specified on the function definition, and not on a function declaration without body.
- Differences between restrictions on specification of region among 4 region commands lead to irregularity.

14.10.2.2 Body Block of Functions

Pragmas on the body block of a function are ignored.

14.10.2.3 Local Variables

If multiple arrays are inferred from a local variable, array directives will be inferred for each of these arrays. In the following example, CtoS infers 2 arrays, **line_x** and **line_y**. Each array has an **flatten_array** directive.

Example

```
struct Point { int x,y;}
..
void Module::proc() {
    ..
    #pragma ctos flatten_array
    Point line[16];
```

Compatible Command	Inferred Object	Secondary Object	Description
flatten_array	array	none	1 directive for each array inferred from the variable
allocate_builtin_ram			
allocate_memory			
allocate_prototype_memory			

14.10.2.4 Declarations with Multiple Variables

If a declaration of multiple variables is preceded by a pragma containing an array command, directives are inferred only for the arrays associated with the first variable.

In the example below, only array **arr1** has a **flatten_array** directive. Array **arr2** does not have any directives.

Example

```
#pragma ctos flatten_array
int arr1[16], arr2[16];
```

14.10.2.5 Static Member Variables

Compatible Command	Inferred Object	Secondary Object	Description
flatten_array	array	none	1 directive for each array inferred from the variable
allocate_builtin_ram			
allocate_memory			
allocate_prototype_memory			

14.10.2.6 Module Member Variables

Compatible Command	Inferred Object	Secondary Object	Description
flatten_array	array	none	1 directive for each array inferred from the variable
allocate_builtin_ram			
allocate_memory			
allocate_prototype_memory			

14.10.2.7 Class Member Variables where Class Is Not a Module

Pragmas for member variables of classes/structs that do not derive from **sc_module** and that do not contain any **sc_in**, **sc_out**, and **sc_signal** are ignored.

14.11 Constructs

This section describes the CtoS support for the **goto** statement, as well as listing the unsupported constructs in CtoS:

- [“Support for goto Statements” on page 14-85](#)
- [“Unsupported C/C++ Constructs” on page 14-93](#)

14.11.1 Support for goto Statements

Support for goto statements is a preliminary feature.

The C++ standard [ISO/IEC 14882] specifies that the **goto** statement unconditionally transfers control to the statement labeled by the identifier. The identifier shall be a label located in the current function.

One way to implement these semantics is to elaborate an edge in the control flow graph from the node corresponding to the **goto** statement to the node corresponding to the target label. A problem with this approach is that it leads to control flow graphs in which loops have multiple entries, and graphs no longer have a proper loop nesting hierarchy. This is problematic for a high-level synthesis tool like CtoS because many of the transforms assume that loops are well defined and have only a single entry.

An alternative approach, which is adopted in CtoS, is to decompose the jump implied by a **goto** statement into a sequence of jumps so that no new entries into loops are required. If a **goto** statement requires a loop to be entered, the decomposition allows that loop to be entered via its proper entry node.

If a **goto** statement requires a backward jump, the decomposition allows you to perform this backward jump via the back edge of an existing loop. In simple cases, the decomposition consists of just a single edge from the node corresponding to the **goto** statement (the source node) to the node corresponding to the target label (the target node).

More information about goto statements is presented in the following sections:

- [“Simple Forward-Jumping goto Statement” on page 14-87](#)
- [“Backward-Jumping goto Statement” on page 14-88](#)
- [“Goto Statement that Jumps into a Loop” on page 14-90](#)
- [“Combinational Loops” on page 14-91](#)
- [“Coding Guidelines” on page 14-92](#)
- [“Final Notes on goto Statements” on page 14-93](#)

14.11.1.1 Simple Forward-Jumping goto Statement

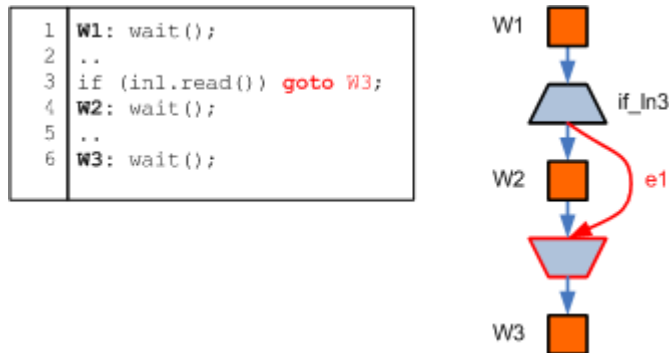
A **goto** statement is considered to be a simple forward-jumping **goto** statement if:

1. There is no path from the target label to the **goto** statement, excluding paths that go across loop iterations, and
2. Going from the **goto** statement to the target label does not require entering loops.

Such a **goto** statement is elaborated as a single edge from the source node to the target node.

Example: The code fragment in [Figure 14-1 on page 14-87](#) has a **goto** statement on line 3. This gives rise to edge **e1** (in red) in the corresponding control flow graph.

Figure 14-1 Simple Forward-Jumping goto Statement



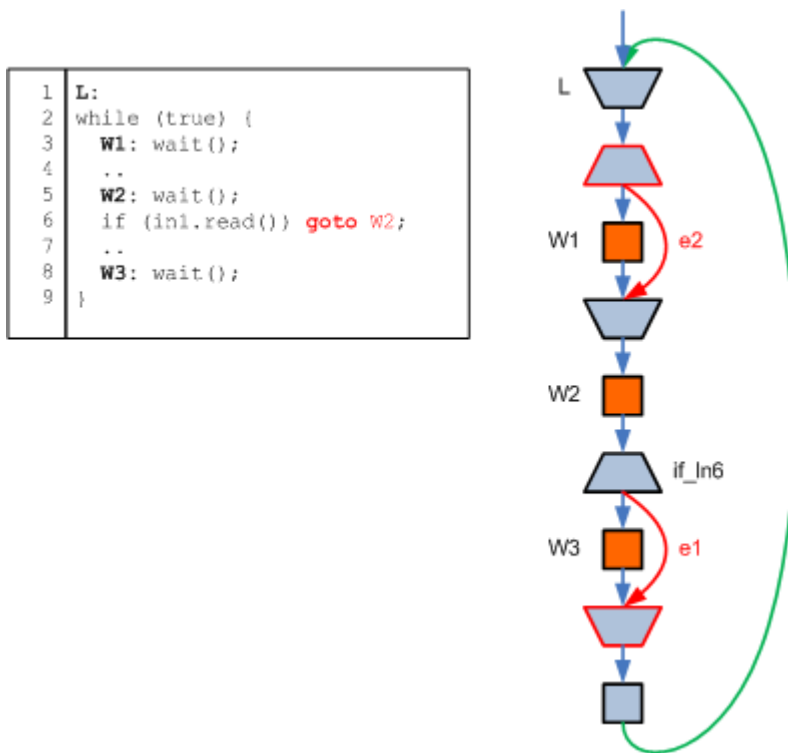
14.11.1.2 Backward-Jumping goto Statement

A **goto** statement is considered to be a backward-jumping **goto** statement if there is a path from the target label to the **goto** statement, excluding paths that go across loop iterations.

If the **goto** statement and the target label are enclosed in a syntactic loop, then such a **goto** statement is elaborated as a jump to the bottom of that loop, followed by a jump from the top of the loop to the target label.

Example: The code fragment in Figure 14-2 on page 14-88 has a backward-jumping **goto** statement on line 6. Both the **goto** statement and its target are enclosed in the while loop. The jump is decomposed in (1) a jump to the bottom of this loop (red edge **e1**), followed by (2) a jump along the back edge of the loop (green edge), followed by (3) a jump from the top of the loop to the final target (red edge **e2**).

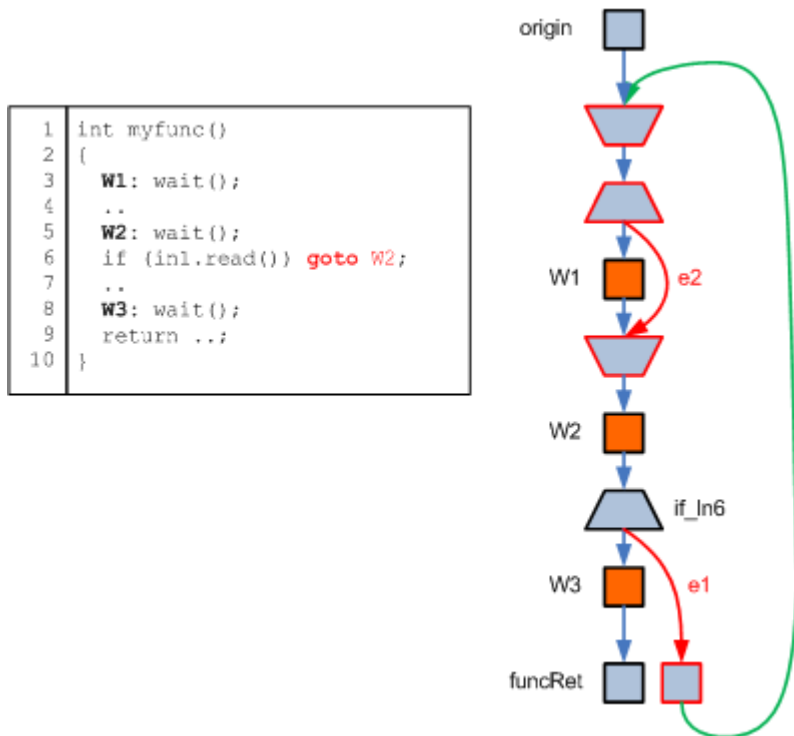
Figure 14-2 Backward-Jumping goto Statement, Enclosed in while Loop



If the **goto** statement and the target label are not enclosed in a syntactic loop, and this **goto** statement is in a function that is not the function associated with a process, CtoS creates a new loop that encompasses the whole function body. Then, the **goto** statement is elaborated as a jump to the bottom of that loop followed by a jump from the top of the loop to the target label. This is the only situation in which elaboration of a **goto** statement gives rise to a new loop. If the function is associated with a process, an error is issued, and the design is rejected because addition of a loop may interfere with the reset specification.

Example: The code fragment in Figure 14-3 on page 14-89 has a backward-jumping **goto** statement on line 6. However, this **goto** statement and its target are not enclosed in any syntactic loop. In this case, CtoS infers a new loop that encloses the body of the function. The jump implied by the **goto** statement is decomposed into (1) a jump to the bottom of the loop via edge **e1**, followed by (2) a jump along the back edge (in green) of the inferred loop, followed by (3) a jump from the top of the inferred loop to the final target (edge **e2**).

Figure 14-3 Backward-Jumping goto Statement, Not Enclosed in Syntactic Loop

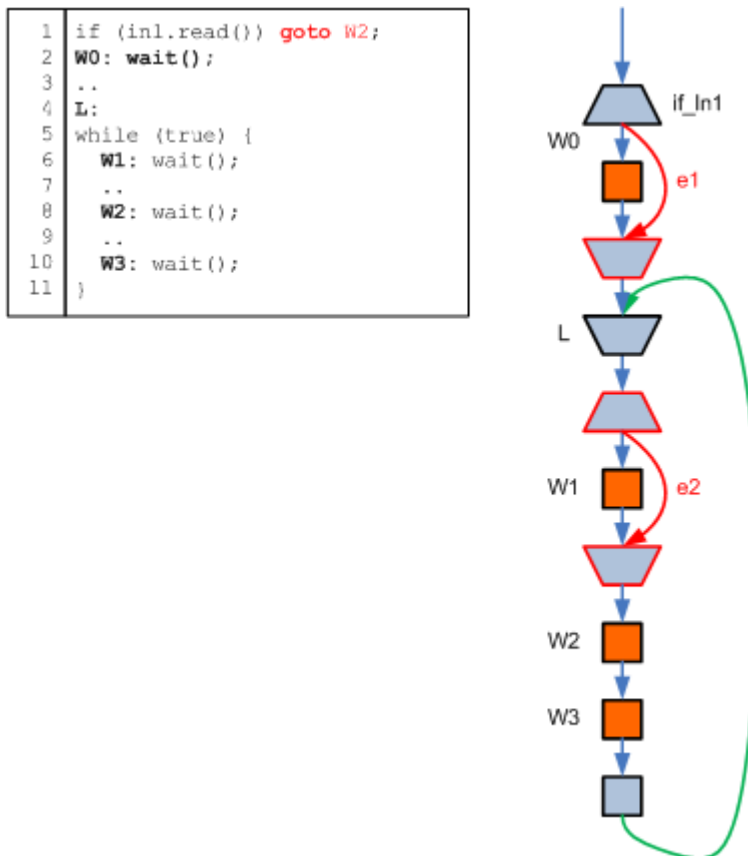


14.11.1.3 Goto Statement that Jumps into a Loop

CtoS elaborates a **goto** statement that requires entering loops as a sequence of jumps, one jump for each loop that needs to be entered.

Example: The code fragment in [Figure 14-4 on page 14-90](#) has a **goto** statement on line 1 that jumps into the **while** loop at line 5. This jump is decomposed into (1) a jump to the loop (red edge **e1**), followed by (2) a jump from the top of the loop to the final target (red edge **e2**).

Figure 14-4 Goto Statement that Jumps into a Loop

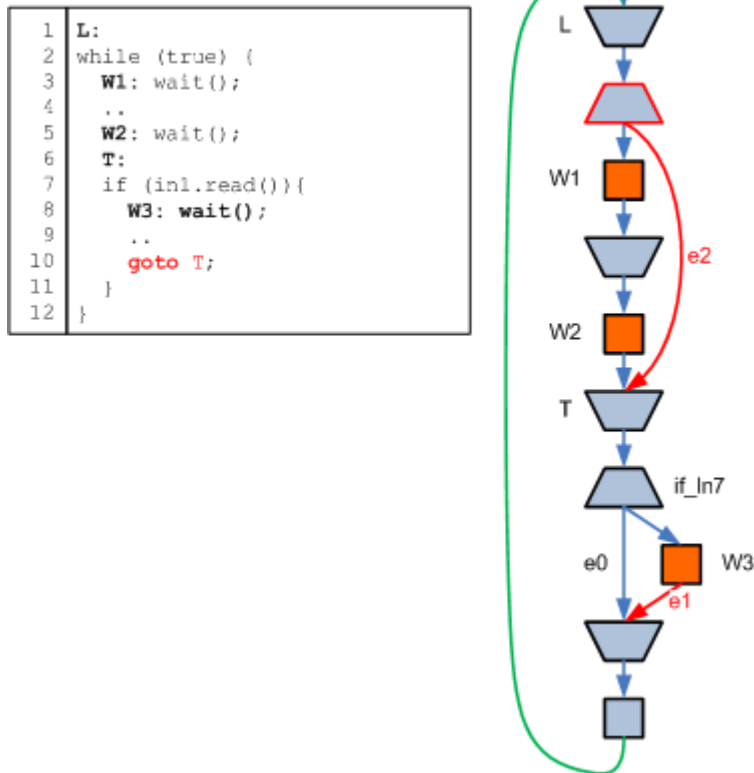


14.11.1.4 Combinational Loops

goto statements may introduce combinational loops.

Example: The code fragment in [Figure 14-5 on page 14-91](#) has a **goto** statement on line 10 that jumps backwards to the target on line 6. The jump implied by the **goto** statement is decomposed into (1) a jump to the bottom of the loop via edge **e1**, followed by (2) a jump along the back edge of the loop (green edge), and finally (3) a jump to the target via edge **e2**. The loop in the resulting control flow graph is a combinational loop because there is a combinational path through the body of this loop (via edges **e2** and **e0**). This combinational path is an artifact of the way that CtoS decomposes backward-jumping **goto** statements and it may seem un-intuitive at first. This combinational path would not have existed if the loop did not have a combinational path from the target of the **goto** statement (label **T**) to the bottom of the loop (via edge **e0**).

Figure 14-5 Combinational Loops



14.11.1.5 Coding Guidelines

Here are the coding guidelines for **goto** statements:

1. Use **goto** statements with great care and avoid them where possible.

Usage of the **goto** statement can quickly make source code very hard to follow; therefore, it should be avoided where possible.

Furthermore, high-level synthesis with CtoS requires that loops in control flow graphs have only one entry, and for this reason, CtoS elaborates the jump associated with a **goto** statement as a sequence of jumps, rather than as a single jump. This makes it harder to relate the resulting control flow graph to the source code.

2. Backward-jumping **goto** statements should be enclosed in a **do/for/while** loop.

CtoS does not directly infer any loops from **goto** statements. Only in the special case of a non-process function that has a **goto** statement where the **goto** statement and its target are not contained in a syntactic loop will CtoS infer a loop in the control flow graph.

The general approach for implementing a backward-jumping **goto** statement is to jump to the bottom of the loop that encloses both the **goto** statement and its target and to follow the back edge.

This approach gives you better control about the loops in the control flow graph. In general, loops are inferred only from **do/for/while** statements in the source code.

3. Make sure that no label that is the target of a backward-jumping **goto** statement has a combinational path to the bottom of the loop in which it is nested.

Combinational loops that are an artifact of the way CtoS implements backward-jumping **goto** statements can be avoided by ensuring that no target of a **goto** statement has a combinational path to the bottom of the loop in which the target is enclosed.

Consider for example the code fragment (given previously), in which label **T** has a combinational path to the bottom of the **while** loop. The loop in the resulting control flow graph has a combinational path through its body.

```
while (true) {
    W1: wait();
    ..
    W2: wait();
    T:
    if (in1.read()) {
        W3: wait();
        ..
        goto T;
    }
}
```

Here is a way to restructure this code to avoid combinational loops:

```
while (true) {
    W1: wait();
    ..
    W2: wait();
    T:
    while (in1.read()) {
        W3: wait();
        ..
    }
}
```

14.11.1.6 Final Notes on goto Statements

Here are a few final notes on the use of **goto** statements in CtoS:

- Labeled non-control statements in the source code are elaborated with the **preserve** attribute set to *true* (see [“Control Flow Node Preserve Attribute” on page 14-97](#)).
- If the **preserve** attribute of a node is set to *true*, that node will be preserved through transforms and optimizations.

This is useful when you want to refer to this node in a latency constraint.

- However, if a node is labeled in the source code simply because it is the target of a **goto** statement, and no latency constraints involving this node need to be specified, it is recommended that you set the **preserve** attribute of such nodes to *false* to give CtoS more freedom in optimizing the control flow.

14.11.2 Unsupported C/C++ Constructs

The following table shows unsupported C/C++ constructs in CtoS and provides some pertinent comments.

Table 14-11 Unsupported C/C++ Constructs

Construct	Comments
try/catch	
new[]	CtoS does not support dynamic memory allocation.
delete, delete[]	CtoS ignores destructors.

14.12 Using C++ Labels

When you are refining your design with CtoS, it is often necessary to refer to specific loops and nodes in the CFG.

Control flow nodes are identified by their names, which CtoS generates automatically.

However, you do have some control over how these nodes are named, as described in [“Design Property Dialog - Naming Tab” on page 6-24](#).

C++ labels provide an alternative way for controlling the names of control flow nodes. Control flow nodes inferred from labeled statements are named after the name of the label.

In addition, if the labeled statement is not a **control** statement, the control flow node inferred from the statement, and named after the label, has the **preserve** attribute set to *true*. The **preserve** attribute indicates that the node should be preserved through optimizations.

The naming of control flow nodes inferred from labeled statements (four cases are presented), as well as the **preserve** attribute, are explained in the following sections:

- [“Labeled Control Statements” on page 14-94](#)
- [“Labeled Block Statements” on page 14-96](#)
- [“Labeled wait Statements” on page 14-96](#)
- [“Labeled Simple Statements” on page 14-97](#)
- [“Control Flow Node Preserve Attribute” on page 14-97](#)

14.12.1 Labeled Control Statements

The following sections describe the two types of *labeled control statements*, as well as the way in which these statements are optimized.

- [“Labeled Conditional Statements” on page 14-95](#)
- [“Labeled Loop Statements” on page 14-95](#)
- [“Optimization of Labeled Control Statements” on page 14-96](#)

14.12.1.1 Labeled Conditional Statements

For *labeled conditional statements*, the control flow nodes that represent the beginning and end of the statements are named after the label, using the following scheme:

```
node_name           := label_name _ statement_type _ node_type
statement_type      := [ if | switch ]
node_type           := [ begin | end ]
```

Example: Labeled Conditional Statements

Code fragment:

```
L1:
    if (cond) {
        ..
    } else {
        ..
    }
```

Inferred control flow nodes named after the label:

```
L1_if_begin, L1_if_end.
```

14.12.1.2 Labeled Loop Statements

For *labeled loop statements*, up to five control flow nodes are named after the loop, using the following scheme:

```
node_name           := label_name _ statement_type _ node_type
statement_type      := [ do | for | while ]
node_type           := [ begin | end | exit | body_begin | body_end ]
```

Example: Labeled Loop Statements

Code fragment:

```
LOOP1:
    for (int i = 0; i < 32; i++) {
        ..
    }
```

Inferred control flow nodes named after the loop:

```
LOOP1_for_begin, LOOP1_for_end, LOOP1_for_exit.
```

14.12.1.3 Optimization of Labeled Control Statements

Control flow nodes inferred from labeled control statements do not have the **preserve** attribute set to *true*, so nodes named after the label of a labeled control statement may disappear during subsequent optimizations. However, the *do/while transform* does preserve the names of loop join (**begin**), **exit**, and bottom (**end**) nodes [be aware that user-directed transforms, such as the **unroll_loop** command, may *still* optimize them away]. The *if/while transform* also preserves the names of loop join nodes.

In this example, you can see the differences between these loop names with and without labels:

join node:	<code>label_name_while_begin</code>	vs.	<code>while_lnxx</code>
exit fork:	<code>label_name_while_exit</code>	vs.	<code>whileFork_lnxx</code>
bottom node:	<code>label_name_while_end</code>	vs.	<code>whileBot_lnxx</code>

This is also true for **for** loops: in this case **while** would be replaced by **for**.

14.12.2 Labeled Block Statements

For *labeled block statements*, two control flow nodes are inferred that mark the beginning and end of the statement, and the **preserve** attribute of those nodes is set to *true*.

Example: Labeled Block Statements

Code fragment:

```
COMPUTATION1:  
{  
    ..  
}
```

Inferred control flow nodes named after the label:

```
COMPUTATION1_begin, COMPUTATION1_end.
```

14.12.3 Labeled wait Statements

For a *labeled wait statement*, the state node inferred from that **wait** statement is named after the label, and the **preserve** attribute of that state is set to *true*.

Example: Labeled wait Statements

Code fragment:

```
STATE1: wait();
```

Inferred control flow nodes named after the label:

```
STATE1.
```


14.12.4 Labeled Simple Statements

For labeled statements not covered by any of the previous categories, one control flow node is inferred that marks the beginning of the labeled statement. It is named after the label, and its **preserve** attribute is set to *true*.

Example: Labeled Simple Statements (1)

Code fragment:

```
L1: m_x.write(x);
```

Inferred control flow nodes named after the label:

```
L1.
```

Example: Labeled Simple Statements (2)

Code fragment:

```
L1:;  
    while (condition) {  
        ..  
    }
```

Inferred control flow nodes named after the label:

```
L1.
```

Note In Example (2), note the semicolon (;). In this example, the labeled statement is an empty statement, and the while statement is not labeled, so this follows the rules for labeled simple statements.

14.12.5 Control Flow Node Preserve Attribute

As previously mentioned, for a labeled non-control statement, either one or two control flow nodes are inferred that have the **preserve** attribute set to *true* ("[preserve](#)" on page D-54).

This attribute indicates that the node should be preserved through optimizations of the control flow, and as such, it limits the amount of optimization.

If a given control flow node whose **preserve** attribute is *true* does not participate in any latency constraints, you should consider setting the **preserve** attribute of that node to *false* (using the **set_attr** command) to allow for further optimization of the design.

14.13 Coding Tips for High Quality Synthesis

This section contains helpful information for coding practices that will lead to high quality synthesis.

- “Controlling the Dynamics of Variables” on page 14-98
- “Optimizing the Control Flow” on page 14-99
- “Multi-Dimensional Array Access Management” on page 14-100

14.13.1 Controlling the Dynamics of Variables

CtoS supports both primitive C/C++ and SystemC data types. To produce optimal results, CtoS performs a trimming on variables; however, the analysis of variable dynamics used in algorithms can help to avoid overflow errors (which may occur due to undersized data types) and to precisely refine all variables used in an algorithm.

Dynamics analysis should be based on functional constraints on values of a variable, which in some cases cannot be expressed in the source code by simply constraining the width of the data types, as shown in the following example.

Example: Controlling the Dynamics of Variables

Consider a design that performs computations based on a formatted 5-bit data stream:

- The data stream protocol is based on an active high valid signal and a 5-bit input port declared as **sc_uint<5>**.
- The algorithm at some point of its execution performs a bit shift on an internal variable according to the 5-bit word read from the data stream.

The corresponding function may be implemented as follows:

```
unsigned int MyModule::ShiftOp(unsigned int data) {  
    // wait for next word of data stream  
    while (!stream_valid.read()) wait();  
    // return shifted value  
    return ( data << (32-stream_word.read()) );  
}
```

Because the protocol of the data stream restricts the value for bit-shift to a range of 1 to 16, and since CtoS does not have access to this information from the source code, it will implement a 32-bit variable bit-shift, while only a 16-bit variable bit-shift would have been necessary. This would lead to a sub-optimal synthesis.

To produce an optimal result, the code should be enhanced as shown in the example on the following page.

Example: Controlling the Dynamics of Variables - Enhanced

```
unsigned int MyModule::ShiftOp(unsigned int data) {  
    unsigned short data16 = data << 16;  
    // wait for next word of data stream  
    while (!stream_valid.read()) wait();  
    // return shifted value  
    return ( data16 << (16-stream_word.read()) );  
}
```

14.13.2 Optimizing the Control Flow

The complexity of the control flow of a design has a direct impact on its synthesizability and on the *quality of synthesis*.

CtoS has optimization techniques to produce a best quality of synthesis; however, designers can help the technology to produce the best result by writing efficient code.

The code prepared for synthesis may be directly extracted, or derived, from a functional implementation.

This implementation has most probably been written for algorithm validation or functional verification purposes, and not optimized for synthesis.

A few rules should be observed on the control flow of a design for synthesis:

- Use code coverage based on an exhaustive simulation suite to identify dead branches in the code. Some branches of the code may exist due to functional constraints on the primary ports of the design not expressed in the source model. As a consequence, CtoS will not have the information required to delete those dead branches.
- Compacting consecutive or nested **if** conditions in a smaller number of branches, when applicable, may improve the quality of synthesis.
- Transforming a cascaded **if** condition (priority decoder) into a switch statement (parallel decoder), when applicable, may also improve the quality of synthesis.

14.13.3 Multi-Dimensional Array Access Management

Functional models containing multi-dimensional arrays may have been optimized for low memory consumption. The resulting size of the inner dimensions in the model may not be a power of two.

An example of the layout of a three-dimensional array is shown in [Figure 14-6 on page 14-101](#).

Since multi-dimensional arrays are flattened during synthesis, the address computation in the flattened view may require additions and constant multiplications.

For example:

```
void MyModule::DecodeTable(int symbol, int *y, int *x) {
    const array LUT[3][9] = {
        { val00, val01, val02, val03, val04, val05, val06, val07, val08 },
        { val10, val11, val12, val13, val14, val15, val16, val17, val18 },
        { val20, val21, val22, val23, val24, val25, val26, val27, val28 }
    };
    for(int j=0 ; j<3 ; j++)
        for(int i=0 ; i<9 ; i++ )
            if (LUT[j][i] == symbol ) {
                *y = j;
                *x = i;
            }
    }
```

The inferred hardware for accessing the array will compute $j*9+i$, that is, a multiply by a constant and an adder.

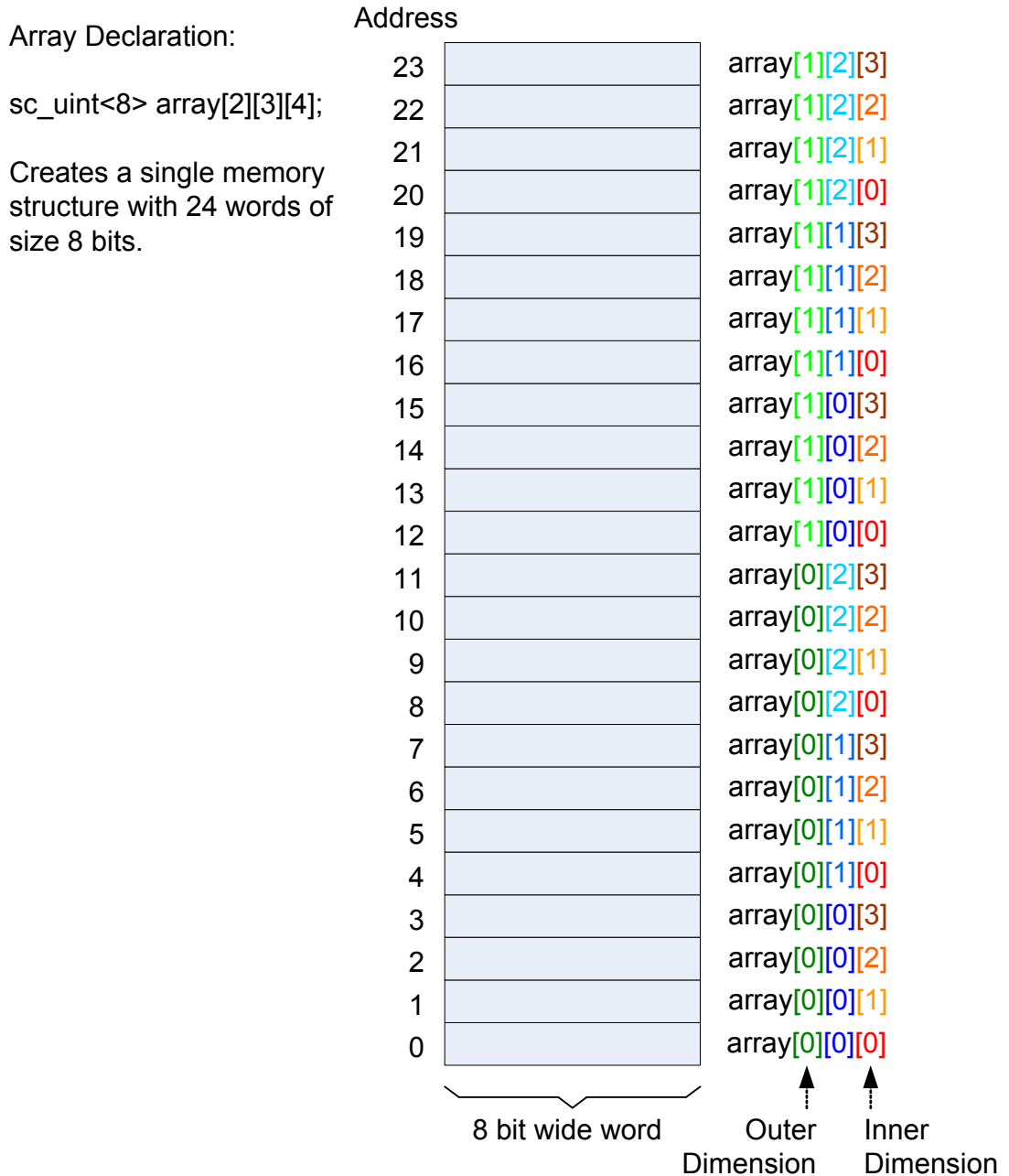
Padding the inner dimension of the array to a power of 2 will require only an index concatenation for accessing the array. Since the extraneous array elements will never be accessed, they should be deleted during optimization. This will lead to better QoR.

Since partial initialization is supported by CtoS, the previous piece of code will become:

```
void MyModule::DecodeTable(int symbol, int *y, int *x) {
    const array LUT[3][16] = {
        { val00, val01, val02, val03, val04, val05, val06, val07, val08 },
        { val10, val11, val12, val13, val14, val15, val16, val17, val18 },
        { val20, val21, val22, val23, val24, val25, val26, val27, val28 }
    };
    for(int j=0 ; j<3 ; j++)
        for(int i=0 ; i<9 ; i++ )
            if (LUT[j][i] == symbol ) {
                *y = j;
                *x = i;
            }
    }
```

The inferred hardware for accessing the array will concatenate j to i , that is, in Verilog { $j[1:0]$, $i[3:0]$ }.

Figure 14-6 Layout of a Three-Dimensional Array



14.14 Specifying Synthesis-Specific and Simulation-Specific Code (__CTOS__ macro)

Any statements and expressions in a process of a design, or a function called from a process of a design or a module constructor, should be synthesizable.

Similarly, any class instantiated in a design should be synthesizable.

However, for simulation it may be useful to include such non-synthesizable code in a design source.

You can use preprocessor directives to *exclude* chunks of code that a design sources when CtoS is processing it.

Similarly, you can use preprocessing directives to *include* chunks of code only when those design sources are processed by CtoS.

To help with this, CtoS pre-defines the macro `__CTOS__` (`CTOS` preceded by and followed by double underscore).

Here is how this macro is used.

- This macro can be used in preprocessor directives to identify chunks in source code that should be ignored by CtoS, for example:

```
#ifndef __CTOS__  
  
    sc_trace(tf, a, b, c);  
  
#endif
```

- This macro can be used in preprocessor directives to identify chunks of code that should be parsed only by CtoS, but ignored by other compilers, such as INCISIV, for example:

```
#ifdef __CTOS__  
  
    // Instantiate DUT for CtoS.  
  
    SC_MODULE_EXPORT(DUT);  
  
#endif
```

14.15 Specifying Clock Gate Integrated Cells (CGIC)

This section describes how to instantiate CGIC (clock gate integrated cells) in SystemC source and simulate the effects of user-controlled clock gating in your design.

CGIC cells are modeled in the SystemC source code using the following two components provided in the CtoS library:

- The **ctos_clock_gate_module** module. This is modeled after the 3-port model defined by RC. It is basically a CGIC cell with active high enable and is suitable for use by positive edge triggered flipflops.
- The **ctos_clock_gate_signal** class. This is similar to **sc_signal<bool>** except that writes are immediate, and the resulting value changes are notified in the same delta cycle of the write.

The definitions of both the components are included using the header file **ctos_clock_gating.h**, which is located in the directory `Install/share/ctos/include/ctos_clock_gating`. This directly is included in the standard include paths of CtoS.

For more information, see also [“Coarse Grain Clock Gating” on page 18-11](#).

14.15.1 Usage of Clock Gating

In SystemC, clocks are modeled as **sc_clock** objects. Class **sc_clock** is a **sc_signal<bool>** whose value is updated by the SystemC kernel. CtoS does not support synthesis of clock generators, so the **sc_clock** object must be instantiated outside the module to be synthesized. The module to be synthesized uses the clock using **sc_in<bool>** that is connected to the **sc_clock** in the testbench.

CtoS can synthesize gated clocks. A gated clock needs to be modeled in the SystemC code by an instance of the **ctos_clock_gate_module** and the **ctos_clock_gate_signal** object. The **ctos_clock_gate_module** is simplistically an and-gate of a clock and an enable signal. The output of the **ctos_clock_gate_module** object must be connected to the **ctos_clock_gate_signal** object. Processes and modules using the gated clocks must refer to the **ctos_clock_gate_signal** object.

Example

The code below illustrates a module DUT that contains a gated clock **clk2**. The gated clock is derived from primary clock **clk1** and enable signal **enable** using instance **clk2_mod** of **ctos_clock_gate_module**. The gated clock is used by **process main** in instance **sub** of module **SUB**.

```
#include <systemc.h>
#include "ctos_clock_gating.h"
SC_MODULE(SUB) {
    sc_in<bool> clk2;
    sc_in<bool> rst;
    SC_CTOR(DUT) {
        SC_CTHREAD(main, clk2.pos());
        reset_signal_is(rst, true);
        ..
    }
    void main();
}
```

```
..
};
SC_MODULE(DUT) {
    sc_in<bool>  clk1; // primary clock
    sc_in<bool>  rst;
    sc_in<bool>  enable;
    SUB sub;

    // Clock clk2 is derived from clk1
    ctos_clock_gate_module  clk2_mod;
    ctos_clock_gate_signal  clk2;
    SC_CTOR(DUT)
    , clk2_mod("clk2_mod")
    {
        clk2_mod.clk_in(clk1);
        clk2_mod.enable(enable);
        clk2_mod.clk_out(clk2);
        sub.clk2(clk2);
        sub.clk2(rst);
        ..
    }
    ..
};
```

Example

Gated clocks can be chained. In other words, one can derive a gated clock from another gated clock.

Primary clocks are used in the module to be synthesized indirectly by going through a **sc_in<bool>**. Gated clocks can be used in the same manner as illustrated in the earlier example. Gated clocks, however, can also be used directly. When gated clocks are used using a **sc_in<bool>**, the C++ syntax involved is identical to that for primary clocks. However, when they are used directly, the syntax is different, because in this case a **sc_signal<bool>** is accessed instead of a **sc_in<bool>**.

This example includes:

- a primary clock **clk1**.
- a gated clock **clk2**, which is derived from **clk1** through **clk2_mod**.
- a **SC_CTHREAD** process **cthread_proc1()**, whose clock is the rising edge of **clk1**.
- a **SC_CTHREAD** process **cthread_proc2()**, whose clock is the rising edge of **clk2**.

```
SC_CTHREAD(cthread_proc1, clk1.pos());
SC_CTHREAD(cthread_proc2, clk2);
```

Note the difference in the specification of the clock event of these processes:

- a **SC_THREAD** process **thread_proc1()**, whose clock is the rising edge of **clk1**.
- a **SC_THREAD** process **thread_proc2()**, whose clock is the rising edge of **clk2**.

```
SC_THREAD(thread_proc1)
sensitive << clk1.pos();
..
SC_THREAD(thread_proc2);
sensitive << clk2.posedge_event();
```

While comparing the clock specification of **cthread_proc2()** and **thread_proc2()**, note that:

- If a **SC_CTHREAD** process is clocked by a gated clock without **sc_in<bool>**, you cannot choose the edge that is active. The active edge is always the rising edge.
- If a **SC_THREAD** process clocked by a gated clock without **sc_in<bool>**, you do have a choice which edge is the active edge.

```
SC_MODULE(DUT) {
    sc_in<bool>  clk1;
    sc_in<bool>  rst;
    sc_in<bool>  enable;
    // Clock clk2 is derived from clk1
    ctos_clock_gate_module  clk2_mod;
    ctos_clock_gate_signal  clk2;
    void cthread_proc1();
    void cthread_proc2();
    void thread_proc1();
    void thread_proc2();
    SC_CTOR(DUT)
    , clk2_mod("clk2_mod")
    {
        clk2_mod.clk_in(clk1);
        clk2_mod.enable(enable);
        clk2_mod.clk_out(clk2);
        SC_CTHREAD(cthread_proc1, clk1.pos());
        reset_signal_is(rst, true);
        SC_CTHREAD(cthread_proc2, clk2); // uses clk2 instead of clk2.pos()
        reset_signal_is(rst, true);
        SC_THREAD(thread_proc1)
        sensitive << clk1.pos();
        reset_signal_is(rst, true);
        SC_THREAD(thread_proc2);
        sensitive << clk2.posedge_event(); // uses clk2.posedge_event() instead
                                           //of clk2.pos()
        reset_signal_is(rst, true);
    }
    ..
}
```

14.16 SystemC Standard Language Restrictions

The *LRM* defines the execution of an application program as a sequence of phases, as follows:

- a. Elaboration – Construction of the module hierarchy
- b. Elaboration – Callbacks to the function **before_end_of_elaboration()**
- c. Elaboration – Callbacks to the function **end_of_elaboration()**
- d. Simulation – Callbacks to the function **start_of_simulation()**
- e. Simulation – Initialization
- f. Simulation – Evaluation, Update, Delta notification, Timed Notification (repeated)
- g. Simulation – Callbacks to the function **end_of_simulation()**
- h. Simulation – Destruction of the module hierarchy

When CtoS analyzes an application program to identify the behavior to be implemented by the resulting hardware, it considers only phases a and f. Therefore, CtoS ignores the language constructs relevant only to the other phases. Furthermore, CtoS imposes restrictions on the use of some of the constructs, because either there is no corresponding notion in hardware to implement it or it is expensive to analyze the precise behavior to be implemented.

Note CtoS supports a process as the target of synthesis only if it is statically declared and its sensitivity list is also static; therefore, CtoS does not support constructs, or usage of constructs, for dynamic processes or dynamic sensitivity lists. If the *LRM* defines constructs for dynamic processes or dynamic sensitivity, the restrictions for these constructs and their usage is not described in this user guide.

Here are SystemC Standard Language Restrictions for CtoS, based on the definitions given in the *LRM*:

- [“Language Class Restrictions” on page 14-107](#)
- [“Data Type Restrictions” on page 14-108](#)
- [“Utility Class Restrictions” on page 14-108](#)
- [“Struct Endianness and Byte Alignment Restrictions” on page 14-109](#)
- [“Pointer Restrictions” on page 14-110](#)
- [“Array Restrictions” on page 14-110](#)
- [“Known Discrepancies between OSCI and IEEE” on page 14-110](#)
- [“Restrictions Relating to SystemC 2.2” on page 14-111](#)

14.16.1 Language Class Restrictions

Among the core language classes described in Section 5, and the channel classes described in Section 6, of the *LRM*, the following declarations are *not* supported for synthesis, so you may not include them in a design:

- **get_child_objects** in **sc_module**
- **sc_process_handle**
- **sc_event_and_list**
- **sc_event_or_list**
- **sc_event**
- **sc_time**
- **register_port** in **sc_interface**
- **default_event** in **sc_interface**
- all member functions of **sc_object**
- member functions **update**, **print**, **dump** in **sc_signal**
- **sc_fifo**
- **sc_buffer**
- **sc_mutex**
- **sc_semaphore**
- **sc_event_queue**
- **sc_clock**

Notes

- You may include these declarations in a testbench.
- Streams are generally not supported; however, for your convenience, CtoS will ignore simple uses of **cout** and **cerr**.

14.16.2 Data Type Restrictions

As common characteristics for all data types, CtoS does not support the notion of setting a word length of an instance of a certain type using the context in which the instantiation is made.

In other words, CtoS ignores **sc_length_context** and **sc_length_parameter**.

This means that the instantiation must be made with a specific length as a parameter of the constructor.

Ignored constructs are listed as follows:

- **get_child_objects** in **sc_module**
- member functions with stream inputs or stream outputs, such as **print**, **scan**, or **dump**
- **sc_generic_base**
- **sc_numrep** and member functions that take it as a parameter, such as *to_string*

14.16.3 Utility Class Restrictions

Among the utility classes described in Section 8 of the *LRM*, CtoS ignores those not intended for specifying behavior to be implemented as hardware:

- **sc_trace**
- **sc_report** and **sc_report_handler**
- **sc_exception**
- **sc_copyright**
- **sc_version**
- **sc_release**

Again, these constructs may appear in the application program, but CtoS does not interpret them or reflect their semantics in the resulting hardware.

Note Similarly, CtoS also ignores the C++ **cout** statement.

14.16.4 Struct Endianness and Byte Alignment Restrictions

Here is an explanation of how the elaborator handles endianness and byte alignment for structs.

1. The elaborator follows a convention related to the SystemC convention of assigning bit 0 to the LSB; therefore:
 - In a struct implemented as a bit vector, the LSB of the first field is assigned index 0 (LSB of the vector), and the MSB of the last field is assigned the highest index (MSB of the vector).
 - In a memory with words smaller than a scalar datum, the datum is laid out with the LSB in the lowest address memory location, and the MSB in the highest address memory location.
2. Alignment is computed following C/C++ rules, using a default word size of 8 bits. Alignment is significant only in the case of arrays mapped to memories, because in other cases, the bit positions corresponding to the *alignment holes* are neither written nor read.
 - Scalars are aligned to the number of words that are evenly divided by the number of words they require.
 - Structs, classes, and unions are aligned to the most restrictive alignment of their members.
 - Arrays are aligned to the alignment of their elements.
3. Pointers are incremented by the size of the object to which they point, according to the default word size, meaning that pointers into arrays mapped to RAMs are usable as C pointers.
4. Pointers are sized to 32 bits by default.

14.16.5 Pointer Restrictions

CtoS-supported pointer usage is described in “[Pointers](#)” on page 14-66. Here are a few additional restrictions and examples of unsupported usage:

- Pointers to bit selects, part selects, and concatenations of SystemC data types are not supported, for example, the following code is not supported:

```
sc_uint<4> b = 9;
sc_uint<4> *p;
p = (sc_uint<4> *) &((sc_uint<4>)b.range(2,1));
*p += 1;
```

- Casts to and from pointers to SystemC data types are not supported.

14.16.6 Array Restrictions

CtoS maps array variables declared at the module or function-block level onto memories. You then have a choice of implementing them as RAMs or flattening them to bit vectors.

Arrays that are members of structs or non-module classes are mapped onto bit vectors; therefore, they cannot be implemented by RAMs.

14.16.7 Known Discrepancies between OSCI and IEEE

Some behaviors of the *Open SystemC Initiative (OSCI)* reference simulator (and of INCISIV) are known to be different from the *IEEE Standard 1666-2005 for SystemC Language Reference Manual*.

In these situations, the CtoS synthesizer follows the OSCI simulator.

- The size of operands of operations with **sc_bv** as the left operand and a non-bit-vector type (native C++ types, **sc_int**, etc.) as the right operand are adjusted to the size of the smaller operand, for example:

```
sc_bv<7> b7("1110111");
sc_bv<8> b8("10001111");

sc_bv<8> b7or8 = b7 | b8;
sc_bv<8> b8or7 = b8 | b7;
```

The *LRM* (page 252) specifies that binary bitwise operators return a result with a word length equal to the word length of the longest operand; thus, the value of both **b7or8** and **b8or7** should be 11111111, but the simulator reports **b7or8** = 01111111 and **b8or7** = 11111111, as **b7** | **b8** has size 7 bits.

- The size of operands of operations involving **sc_lv** as one operand and a non-bit-vector type (native C++ type, **sc_int**, etc.) as the other operand is adjusted to the size of the smaller operand (this is the same as the previous situation, except the bug in the simulator is symmetric).

14.16.8 Restrictions Relating to SystemC 2.2

1. The "... " argument (for functions with a variable number of arguments) is not supported for synthesized functions (it can be used, for example, for ignore file I/O functions).
2. GNU extensions to the C++ language (**min**, **max**, **expression** statements) are not supported.
 - The **asm** directive is not supported.
 - Dynamically allocated arrays are not supported.
3. Variable-length arrays are not supported (this is a C99 feature and a GNU extension).
4. The argument of **sizeof** must be statically determined (runtime **sizeof** is a C99 feature and a GNU extension).
5. The number of cycles passed to the **wait** method of an **SC_CTHREAD** must be a compile-time numeric constant.
6. Only module-level globals are supported, file-level global variables (**static** and **extern**) are not.
7. Floating-point variables, constants, and literal values are not supported (they must be replaced with integer approximations or emulated via synthesized integer arithmetics).
8. The scan method of SystemC data types cannot be used in synthesized code.
9. **wait_until** is not supported.
10. Nested side effects (for example, **m[i++]**— or ***(a = &b)++**) are not supported.

In other words, an expression that updates a variable cannot be lexically contained within an expression that updates another (or the same) variable.

11. Part selects, as expressed via the **range** and **operator()** methods of SystemC data types where the indices are *not* constant, are supported, provided that:
 - The right index expression is an integer expression that does not have any side effects and does not require the evaluation of functions, and
 - The left index expression is of the form **REXPR + CONST**, where **REXPR** is the right index expression and **CONST** is a non-negative integer compile-time constant.

An example of this is shown on the next page.

Here is an example of part select usage:

```
sc_uint<48> val; int i,j;
..
val.range(i+3, i) = ..;    // supported
val.range(i,j) = ..;      // not supported
..
.. = val.range(i+3, i);    // supported
.. = val.range(i,i-3);    // not supported
```

12. Ranges or indexed bits cannot be further ranged or indexed. For example, **a.range(3,1).range(2,1)** is illegal and must be replaced with **a.range(3,2)**.
13. Multiports are not supported.
14. **sc_inout** and the **Z** value are not supported.
15. Exception handling (including **try-catch** statements, even if exceptions are not actually thrown in the code) is not supported.
16. Unions are not supported.
17. Pointer-to-member types are not supported.
18. Sign-and-magnitude and canonical signed digit representations for bit vector initializers are not supported (they must be replaced with the corresponding two's complement notation).
19. Pointers to pointer and arrays of pointers are not supported.
20. Classes **sc_sensitive_pos** and **sc_sensitive_neg** and the corresponding data members of class **sc_module** are not supported. (Use the event finders **pos** and **neg** instead.)
21. Explicit references are not supported, for example, **int &r = i**. Note that references as function arguments are supported: **void f(int &r)**.
22. Taking the address of a formal argument of type **reference** is not supported.
23. The **build** command may fail on designs that have internal communication via a user-defined **sc_port<IF>** and for which the channel methods and the methods that invoke the port method are defined in different compile units (different **cpp** files).

To preclude this potential problem, you can simply combine the compile units into a single compile unit.

Note See “[Compiling Multi-Source Designs through Single Combined Source](#)” on page 6-29 for more information.

15 CtoS Libraries

This chapter documents about the following *CtoS Libraries*:

- [“Fixed-Point Library” on page 15-1](#)
- [“Flex Channels Library” on page 15-33](#)
- [“TLM Library” on page 15-65](#)

15.1 Fixed-Point Library

The intent of this library is to provide a synthesizable version of the **sc_fixed** and **sc_ufixed** classes, with most of the interface as specified in the *LRM*.

As mentioned in [“Prerequisites for CtoS and for this User Guide” on page 3-1](#), the term *LRM* refers to the *IEEE Standard 1666-2005 for SystemC Language Reference Manual*, which describes the SystemC language.

This section covers following:

- [“Overview of Fixed-Point Data Types” on page 15-1](#)
- [“CtoS Fixed-Point Library” on page 15-5](#)
- [“API of CtoS Fixed-Point Library” on page 15-10](#)

15.1.1 Overview of Fixed-Point Data Types

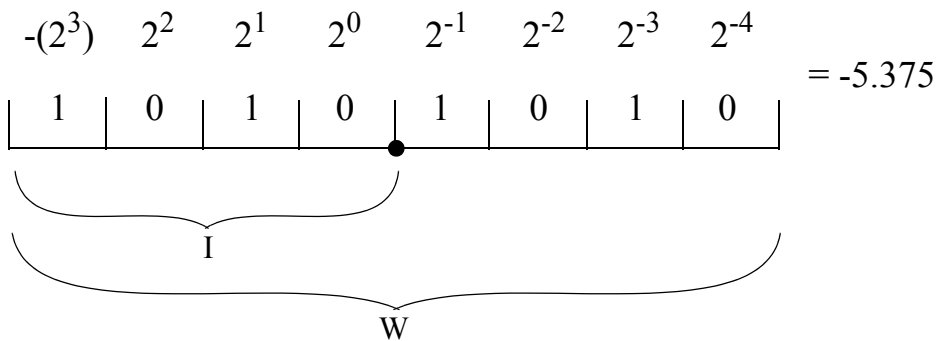
The following sections provide an introduction to SystemC fixed-point data types:

- “Fixed-Point Data Types” on page 15-2
- “Finite-Precision Fixed-Point Types” on page 15-4

15.1.1.1 Fixed-Point Data Types

SystemC fixed-point data types represent a number as a sequence of bits with a specified position for the binary point. Bits at the left of the binary point represent the integer part of the number, while bits at the right of the binary point represent the fractional part.

Figure 15-1 Fixed-Point Data Type Representation



A SystemC fixed-point type is characterized by:

- the *word length* (**W**), which represents the total number of bits in the number representation,
- the *integer word length* (**I**), which represents the number of bits in the integer part, and
- the *bit encoding*, which is either signed, two’s complement, or unsigned.

In the SystemC library, fixed-point type conversion is performed whenever a value is assigned to a fixed-point type variable, including at initialization.

For example, [Figure 15-1 on page 15-2](#) depicts a fixed-point number with $W = 8$ and $I = 4$, and a value of:

$$(-1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (0 * 2^0) + (1 * 2^{-1}) + (0 * 2^{-2}) + (1 * 2^{-3}) + (0 * 2^{-4}) = -5.375$$

If the magnitude of the value is outside the range of the fixed-point representation, or if the value has greater precision than provided by the fixed-point representation, it is mapped to a value that can be represented.

Fixed-point type conversion is performed by means of:

- *quantization (Q)*, if the value is within the range, but has greater precision, or
- *overflow (O)*, if the magnitude of the value is outside the range.

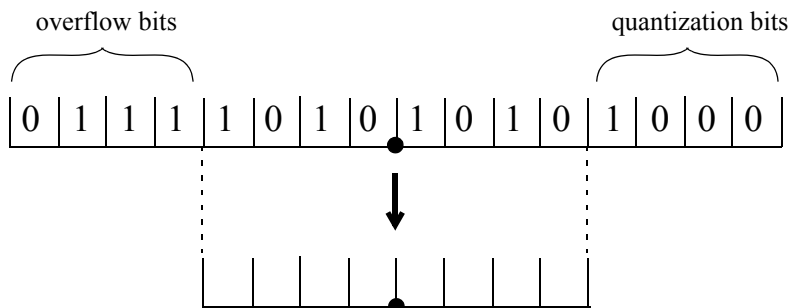
Figure 15-2 on page 15-3 shows an example of a value that is assigned to a variable that cannot store this value.

The range of the destination variable, depicted on the lower part, is $-8 \dots 7.9375$.

However, the value of the source variable, depicted on the upper part, is 122.65625, which is clearly out of range.

This is an example of a scenario in which overflow and quantization must be done.

Figure 15-2 Fixed-Point Type Conversion



15.1.1.2 Finite-Precision Fixed-Point Types

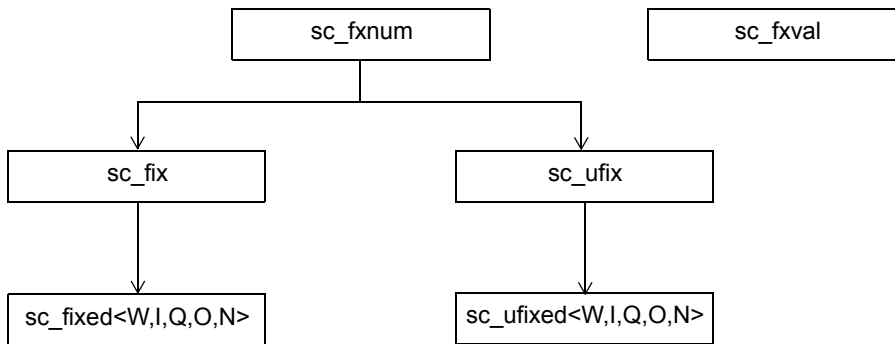
In the following sections, the OSCI specification for the SystemC fixed-point data types will be described, and an appropriate subset of synthesis will be identified.

- “OSCI Class Hierarchy” on page 15-4
- “Synthesizable Subset” on page 15-5

OSCI Class Hierarchy

In the *LRM*, section 7.10, the OSCI finite-precision fixed-point types are defined through the class hierarchy depicted in [Figure 15-3 on page 15-4](#).

Figure 15-3 OSCI Finite-Precision Fixed-Point Types Class Hierarchy



Class **sc_fxnum** is a common base class for all finite-precision fixed-point types. A pointer to this class can be used to refer to an object of any finite-precision fixed-point data type.

Class **sc_fxval** is a variable precision type that stores a fixed-point value of arbitrary width and binary point location and is used internally by the other classes.

For **sc_fixed** and **sc_ufixed**, fixed-point parameters are defined at compile time through template parameters. These parameters are defined, as follows:

- **W** represents the word length
- **I** represents the integer word length
- **Q** represents the quantization mode
- **O** represents the overflow mode
- **N** represents the number of saturated bits

Synthesizable Subset

Synthesis requires that the bit widths of expressions be determined by static analysis. Consequently, classes **sc_fix** and **sc_ufix** are not supported, because the fixed-point parameters are specified as constructor arguments. Similarly, class **sc_fxval** is not supported because this data type has variable precision. Also, class **sc_fxnum** is not supported – again due to the fixed-point parameters. Lastly, the limited precision fixed-point types (whose class names have suffix **fast**) are not supported by CtoS.

The synthesizable subset of the OSCI fixed-point library consists of the classes **sc_fixed** and **sc_ufixed**. CtoS provides a library with synthesizable implementations of these classes. A detailed list of supported features is presented in “API of CtoS Fixed-Point Library” on page 15-10.

15.1.2 CtoS Fixed-Point Library

The CtoS fixed-point library is defined as an application library along with the CtoS package. The CtoS fixed-point library provides the **sc_fixed** and **sc_ufixed** data types.

This section has the following subsections:

- “Bit Lengths” on page 15-5
- “Type Conversions” on page 15-6
- “Expressions and Operations” on page 15-6
- “Library Structure” on page 15-6
- “Important Notes” on page 15-6

15.1.2.1 Bit Lengths

The CtoS fixed-point library supports all three cases defined by the *LRM* as follows:

$$\begin{aligned} 0 < W &\leq I \\ 0 &\leq I < W \\ I < 0 &< W \end{aligned}$$

The minimum word length is 1 bit, and the maximum word length is restricted to 64 bits. It is strongly advised to always use: $0 < W \leq 64$.

15.1.2.2 Type Conversions

Fixed-point type conversion is performed by the CtoS application library for both quantization and overflow modes. All quantization and overflow modes are supported. Values of type **int**, **unsigned int**, **long**, **unsigned long**, **int64**, **uint64**, **sc_int<W1>**, **sc_uint<W1>**, **sc_bigint<W1>**, **sc_bignint<W1>**, **sc_fixed<W1, I1, Q1, O1, N1>** and **sc_ufixed<W1, I1, Q1, O1, N1>** can be assigned to a fixed-point type variable (including at initialization). For type **double**, only constant values can be assigned.

15.1.2.3 Expressions and Operations

The CtoS application library performs fixed-point operations with no loss of precision or magnitude. The right-hand side of a fixed-point assignment is evaluated as a variable-precision-like value and is then converted to the fixed-point representation specified by the target. However, since CtoS does not support variable-precision data types, there are restrictions on division and shift operations, as specified in “API of CtoS Fixed-Point Library” on page 15-10.

15.1.2.4 Library Structure

The API of the CtoS fixed-point library is almost identical to that defined in the *LRM*, with the following differences (see “API of CtoS Fixed-Point Library” on page 15-10):

- File names are prefixed with **ctos_** to distinguish them from OSCI fixed-point files.
- The library is divided among many files, and the top-level file is **ctos_fx.h**.
- Classes are encapsulated in the namespace **ctos_sc_dt** instead of **sc_dt**.

The library can be found in your CtoS installation area, at the following location:

```
install_directory/share/ctos/include/ctos_fx/
```

An example of a design using the CtoS fixed-point library can be found in the following location:

```
install_directory/share/ctos/examples/libraries/fixed_point
```

Note Before running any CtoS examples, first review “Setup for Examples” on page F-2.

15.1.2.5 Important Notes

Here are a few important notes, on the following subjects, when using the CtoS fixed-point library.

- “Automatic Micro-Architectural Configuration” on page 15-7
- “Overflow and Quantization Flags” on page 15-7
- “Syntactic Differences” on page 15-8
- “Simulation Speed” on page 15-9
- “Avoiding Compilation Problems with Ambiguous Operators” on page 15-9

- [“Alternative Implementation of Divide Operators” on page 15-10](#)

Automatic Micro-Architectural Configuration

A design using the CtoS fixed-point library can have a large number of small functions that originate from the library.

After successfully building a design, CtoS will automatically inline all behaviors that originate from the CtoS fixed-point library. Inlining these behaviors will usually have a positive impact on QoR, as it will reduce the number of behaviors and enable CtoS to further optimize the logic.

To disable the automatic configuration, add the following Tcl command at the beginning of your configuration script:

```
set ctos::enable_config_lib_ctos_fx 0
```

Overflow and Quantization Flags

Classes **sc_fixed<W, I>** and **sc_ufixed<W, I>** have the following fields, internally:

- a bit vector of length **W** for the data,
- a Boolean variable for the quantization flag, and
- a Boolean variable for the overflow flag

In the OSCI specification for **sc_fixed<W, I>** and **sc_ufixed<W, I>**, two method objects are used to determine if a quantization or overflow was performed during the last assignment:

- **bool quantization_flag()**
- **bool overflow_flags()**

To support these methods, it is necessary to keep two extra state bits in the classes. However, these flags are seldom used, and removing them from the classes will remove them from both the logic and the interfaces, typically resulting in a QoR improvement of a few percentage points.

Therefore, these methods are not enabled by default in the CtoS version of the fixed-point classes. In the library, a macro with **#ifdef CTOS_FX_WITH_FLAG** surrounds all the code pertaining to these methods, and by default the macro is undefined, that is, the methods are not in the class definition.

For more information, see the following sections:

- [“Defining the Quantization /Overflow Flag Methods” on page 15-8](#)
- [“Bit Widths of Fixed-Point Data Types with Flags” on page 15-8](#)

Defining the Quantization /Overflow Flag Methods

To include the quantization and overflow flag methods, you may do either of the following:

- Add the following macro definition immediately before the inclusion of header file **ctos_fx.h**:

```
#define CTOS_FX_WITH_FLAGS  
#include "ctos_fx/ctos_fx.h"
```

- Add the following to the compiler flags:

```
-DCTOS_FX_WITH_FLAGS=1
```

Bit Widths of Fixed-Point Data Types with Flags

With the use of quantization and overflow flags, the interfaces parameterized with **sc_fixed<W, I>** and **sc_ufixed<W, I>** will be of bit width **WL+2**. If you want to use the flags, but have interfaces of bit width **WL**, you should parameterize the interfaces with bit vectors, for example:

```
sc_in<sc_bv<WL> >  
sc_out<sc_bv<WL> >  
sc_signal<sc_bv<WL> >
```

Then, you can easily convert the fixed-point format to/from bit vectors. For example, assume you want to design a module that receives fixed-point data of format **<16,8>**, inverse the numbers, and write the result to an output. You can use 16-bit vectors as input and output ports, as follows:

```
sc_in<sc_bv<16> > m_x;  
sc_out<sc_bv<16> > m_y;
```

Then, the process will read a value from port **m_x**, inverse the sign, and write the result to port **m_y** with the following code:

```
sc_fixed<16,8> f;  
CTOS_FX_ASSIGN_RANGE(f, m_x.read())  
f = ~f;  
m_y.write(sc_bigint<16>(f.range()));
```

Ports **m_x** and **m_y** will be of width 16 bits, whereas if you had used **sc_fixed<16,8>** as the data type, the ports would have been of width 18 bits.

Syntactic Differences

There are a few syntactic differences between the OSCI and the CtoS syntax of the library due to restrictions imposed by the synthesizable subset of SystemC and by the CtoS parsing engine.

The main differences are due to the limitations on reference handling, which are used in the methods for writing to bit ranges or to individual bits.

These differences and workarounds are described in [“Bit and Part Selections” on page 15-25](#).

Simulation Speed

The CtoS fixed-point library is designed for optimal QoR with synthesis, whereas the OCSI fixed-point library is designed for fast simulation speed.

Simulating a design using the CtoS fixed-point library will take more time than simulating with the OSCI or INCISIV library. Therefore, when possible, it is recommended that you use the OSCI or INCISIV simulation library in order to achieve fast simulation speeds.

Avoiding Compilation Problems with Ambiguous Operators

If you are using the IUS fixed-point library with SystemC integer data types (**sc_int**, **sc_uint**, **sc_bigint**, and **sc_biguint** data types), you may encounter compilation errors when comparing an integer data type to an integer literal. For example, consider the following code:

```
sc_uint<4> m_internal = input_a;
if (m_internal != 5) {
    wait();
    m_internal = 3;
}
```

In this case, you would get a compilation error stating that the comparison operator is ambiguous:

```
"$TESTDIR/src/n_sqr_avg.cpp", line 43: error:
more than one operator "!=" matches these operands:
    built-in operator "arithmetic != arithmetic"
    function "sc_dt::operator!=(const sc_dt::sc_uint_base &, const
        sc_dt::sc_fxval &)"
    function "sc_dt::operator!=(const sc_dt::sc_uint_base &, const
        sc_dt::sc_fxval_fast &)"
operand types are: sc_dt::sc_uint<4> != unsigned int
if (m_internal != 5)
```

There are two ways to solve this problem. You can either convert the SystemC integer data type to an integer, with:

```
if (m_internal.to_int() != 5) {
```

Or, you can define the macro **SC_FX_EXCLUDE_OTHER**, which avoids including a number of type conversion with fixed-point data types and integer data types, avoiding ambiguous definitions. To use the macro, whenever you:

```
#define SC_INCLUDE_FX
```

you must also immediately:

```
#define SC_FX_EXCLUDE_OTHER 1
```

before including "systemc.h".

Alternative Implementation of Divide Operators

CtoS has added an alternative implementation for the divide operators of the **ctos_fx** fixed-point library.

The alternative implementation is based on iterative computation of quotient and remainder in terms of subtractions and shifts in the **ctos_sc_dt::full_udiv()** function, and the **/** and **%** operators natively supported by CtoS are not used.

This lets you adjust the micro-architecture of division (by unrolling or breaking the combinational loop in the iterative divide algorithm) in order to meet timing. The resulting micro-architecture does not require divide and modulo resources.

Two new macros have been added to support this new implementation:

- If you define the **CTOS_FX_USE_FULL_UDIV_FUNC** macro, the divide operators are implemented using the **ctos_sc_dt::full_udiv()** function, in the **ctos_fx_full_udiv.h** file, of the **ctos_fx** fixed-point library.

However, by default, this macro is not defined, and the divide operators are implemented in terms of the **/** and **%** operators on finite-precision integers natively supported by CtoS.

- If you define the **CTOS_FX_DONT_TOUCH_FULL_UDIV_FUNC** macro, the **ctos_dont_touch** pragma is specified for the **ctos_sc_dt::full_udiv()** function, in the **ctos_fx_full_udiv.h** file. of the **ctos_fx** fixed-point library.

This is appropriate if you are planning to implement **full_udiv** via an RTL IP.

However, by default, the **ctos_dont_touch** pragma is not specified for the **full_udiv** function.

Note Currently, all behaviors inferred from the **ctos_fx** fixed-point library are automatically inlined. For the **ctos_sc_dt::full_udiv()** function, you may want to implement it as an RTL IP. Therefore, CtoS will not automatically inline this behavior when the macro is defined.

Note **ctos_sc_dt::full_udiv()** is an internal integer division function as a part of **ctos_fx** fixed point library, do not directly use it out of the fixed point library in your designs.

15.1.3 API of CtoS Fixed-Point Library

The following sections define the subset of SystemC fixed-point data types currently supported in the CtoS fixed-point library and indicate the level of support for each, as follows:

- **SYN** (supported for synthesis)
- **SIM** (supported *only* for simulation)
- **NS** (not supported) - this feature or construct is not supported in the current version of the CtoS fixed-point library

- [“Representation” on page 15-11](#)
- [“Constructors and Destructor” on page 15-14](#)
- [“Assignment Operators” on page 15-17](#)
- [“Expressions and Operations” on page 15-19](#)
- [“Relational \(Including Equality\) Operators” on page 15-22](#)
- [“Auto-Increment and Auto-Decrement Operators” on page 15-24](#)
- [“Unary Operators” on page 15-24](#)
- [“Shift Operators” on page 15-24](#)
- [“Bit and Part Selections” on page 15-25](#)
- [“Parameter Functions” on page 15-29](#)
- [“Explicit Conversions to Character Strings” on page 15-29](#)
- [“Query Values” on page 15-30](#)
- [“Explicit/Implicit Conversions to Primitive Types” on page 15-30](#)
- [“Methods Inherited from Base Classes” on page 15-32](#)

15.1.3.1 Representation

This section is divided into three subsections:

- [“Bit Widths” on page 15-11](#)
- [“Classes” on page 15-12](#)
- [“Overflow and Quantization Modes” on page 15-13](#)
- [“Default Values for Template Parameters” on page 15-14](#)

Bit Widths

The level of support for fixed-point bit widths is shown in [Table 15-1](#).

Table 15-1 Bit Widths

Feature	Support
Fixed-point representation:	
$0 < W \leq I$	SYN
$0 \leq I < W$	SYN
$I < 0 < W$	SYN
Bit encoding:	
two's complement	SYN
unsigned	SYN

Classes

The level of support for fixed-point classes is shown in [Table 15-2](#).

Table 15-2 Classes

Feature	Support
Finite precision fixed-point types:	
<code>sc_fixed<W, I, Q, O, N></code>	SYN
<code>sc_ufixed<W, I, Q, O, N></code>	SYN
<code>sc_fxval</code>	NS
<code>sc_fxnum</code>	NS
<code>sc_fix</code>	NS
<code>sc_ufix</code>	NS
Limited precision fixed-point types:	
<code>sc_fxnum_fast</code>	NS
<code>sc_fix_fast</code>	NS

Table 15-2 Classes

Feature	Support
sc_ufix_fast	NS
sc_fixed_fast	NS
sc_ufixed_fast	NS
sc_fxval_fast	NS

Overflow and Quantization Modes

The level of support for overflow and quantization modes is shown in [Table 15-3](#).

Table 15-3 Overflow and Quantization Modes

Feature	Name	Support
Quantization mode (Q):		
rounding to plus infinity	SC_RND	SYN
rounding to zero	SC_RND_ZERO	SYN
rounding to minus infinity	SC_RND_MIN_INF	SYN
rounding to infinity	SC_RND_INF	SYN
convergent rounding	SC_RND_CONV	SYN
truncation	SC_TRN	SYN
truncation to zero	SC_TRN_ZERO	SYN
Overflow mode (O):		
saturation	SC_SAT	SYN
saturation to zero	SC_SAT_ZERO	SYN
symmetrical saturation	SC_SAT_SYM	SYN
wrap-around	SC_WRAP	SYN with saturated bits N
sign magnitude wrap-around	SC_WRAP_SM	SYN with saturated bits N

Default Values for Template Parameters

The level of support for default values for template parameters, defined according to the SystemC fixed-point standard, is shown in [Table 15-4](#).

Table 15-4 Default Values for Template Parameters

Feature	Value	Support
W	32	SYN
I	32	SYN
Q	SC_TRN	SYN
O	SC_WRAP	SYN
N	0	SYN

15.1.3.2 Constructors and Destructor

This section is divided into two subsections:

- “[sc_fixed Constructors/Destructor](#)” on page 15-14
- “[sc_ufixed Constructors/Destructor](#)” on page 15-16

sc_fixed Constructors/Destructor

The level of support for constructors and destructor for **sc_fixed** is shown in [Table 15-5](#).

Table 15-5 `sc_fixed` Constructors and Destructor

Type	Support
<code>sc_fixed<W,I,Q,O,N>()</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(int)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(unsigned int)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(long)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(unsigned long)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(double)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(int64)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(uint64)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(const sc_int<W1>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(const sc_uint<W1>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(const sc_bigint<W1>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(const sc_biguint<W1>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(const sc_fixed<W,I,Q,O,N>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(const sc_fixed<W1,I1,Q1,O1,N1>&)</code>	SYN
<code>~sc_fixed<W,I,Q,O,N>()</code>	SYN
<code>sc_fixed<W,I,Q,O,N>(const char*)</code>	NS
<code>sc_fixed<W,I,Q,O,N>(const sc_fxval&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>(const sc_fxval_fast&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>(const sc_fxnum&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>(const sc_fxnum_fast&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>(const sc_int_base&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>(const sc_uint_base&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>(const sc_signed&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>(const sc_unsigned&)</code>	NS

sc_ufixed Constructors/Destructor

The level of support for constructors and destructor for **sc_ufixed** is shown in [Table 15-6](#).

Table 15-6 sc_ufixed Constructors and Destructor

Type	Support
<code>sc_ufixed<W,I,Q,O,N>()</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(int)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(unsigned int)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(long)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(unsigned long)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(double)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(int64)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(uint64)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(const sc_int<W1>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(const sc_uint<W1>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(const sc_bigint<W1>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(const sc_biguint<W1>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(const sc_fixed<W,I,Q,O,N>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(const sc_fixed<W1,I1,Q1,O1,N1>&)</code>	SYN
<code>~sc_ufixed<W,I,Q,O,N>()</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>(const char*)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>(const sc_fxval&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>(const sc_fxval_fast&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>(const sc_fxnum&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>(const sc_fxnum_fast&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>(const sc_int_base&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>(const sc_uint_base&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>(const sc_signed&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>(const sc_unsigned&)</code>	NS

15.1.3.3 Assignment Operators

This section is divided into two subsections:

- “[sc_fixed Assignment Operators](#)” on page 15-17
- “[sc_ufixed Assignment Operators](#)” on page 15-18

sc_fixed Assignment Operators

The level of support for assignment operators for **sc_fixed** is shown in [Table 15-7](#).

Table 15-7 Assignment Operators for sc_fixed

Type	Support
<code>sc_fixed<W,I,Q,O,N>& operator = (int)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (unsigned int)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (long)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (unsigned long)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (double)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (int64)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (uint64)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_int<W1>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_uint<W1>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_bigint<W1>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_biguint<W1>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_fixed<W,I,Q,O,N>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_fixed<W1,I1,Q1,O1,N1>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_ufixed<W1,I1,Q1,O1,N1>&)</code>	SYN
<code>sc_fixed<W,I,Q,O,N>& operator = (const char*)</code>	NS
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_fxval&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_fxval_fast&)</code>	NS

Table 15-7 Assignment Operators for `sc_fixed`

Type	Support
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_fxnum&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_fxnum_fast&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_int_base&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_uint_base&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_signed&)</code>	NS
<code>sc_fixed<W,I,Q,O,N>& operator = (const sc_unsigned&)</code>	NS

sc_ufixed Assignment Operators

The level of support for assignment operators for `sc_ufixed` is shown in [Table 15-8](#).

Table 15-8 Assignment Operators for `sc_ufixed`

Type	Support
<code>sc_ufixed<W,I,Q,O,N>& operator = (int)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (unsigned int)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (long)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (unsigned long)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (double)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (int64)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (uint64)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_int<W1>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_uint<W1>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_bigint<W1>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_biguint<W1>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_fixed<W,I,Q,O,N>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_fixed<W1,I1,Q1,O1,N1>&)</code>	SYN

Table 15-8 Assignment Operators for `sc_ufixed`

Type	Support
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_ufixed<W1,I1,Q1,O1,N1>&)</code>	SYN
<code>sc_ufixed<W,I,Q,O,N>& operator = (const char*)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_fxval&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_fxval_fast&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_fxnum&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_fxnum_fast&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_int_base&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_uint_base&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_signed&)</code>	NS
<code>sc_ufixed<W,I,Q,O,N>& operator = (const sc_unsigned&)</code>	NS

15.1.3.4 Expressions and Operations

The level of support for expressions and operations is shown in the following tables:

- [Table 15-9 Bitwise Functions on page 15-20](#)
- [Table 15-10 Finite-Precision Addition Operations on page 15-21](#)
- [Table 15-11 Finite-Precision Subtraction Operations on page 15-21](#)
- [Table 15-12 Finite-Precision Multiplication Operations on page 15-22](#)
- [Table 15-13 Finite-Precision Division Operations on page 15-22](#)

Note All expressions and operations apply to both classes: `sc_fixed` and `sc_ufixed`.

In [Table 15-9](#), the following notation is used:

- **s1, s2, s3** represent signed finite-precision fixed-point objects of class `sc_fixed` or `sc_ufixed`.
- **u1, u2, u3** represent unsigned finite-precision fixed-point objects

Table 15-9 Bitwise Functions

Expression	Operation	Support
<code>s1 = s2 & s3;</code>	Bitwise-and with assignment for signed operands	SYN
<code>s1 = s2 s3;</code>	Bitwise-or with assignment for signed operands	SYN
<code>s1 = s2 ^ s3;</code>	Bitwise-exclusive-or with assignment for signed operands	SYN
<code>u1 = u2 & u3;</code>	Bitwise-and with assignment for unsigned operands	SYN
<code>u1 = u2 u3;</code>	Bitwise-or with assignment for unsigned operands	SYN
<code>u3 = u2 ^ u3;</code>	Bitwise-exclusive-or with assignment for unsigned operands	SYN

For [Table 15-10](#), [Table 15-11](#), [Table 15-12](#), and [Table 15-13](#), note that the specified arithmetic operations are permitted for fixed-point objects. The following applies:

- **F**, **F1**, **F2** represent objects of type **sc_fixed**<W1,I1,Q1,O1,N1>, **sc_fixed**<W2,I2,Q2,O2,N2>, **sc_ufixed**<W1,I1,Q1,O1,N1> or **sc_ufixed**<W2,I2,Q2,O2,N2>
- **n** represents an object of supported numeric type **int**, **long**, **unsigned int**, **unsigned long**, **double**, **sc_int**<W1>, **sc_uint**<W1>, **sc_bigint**<W1>, **sc_biguint**<W1>,
- The following types are not supported: **sc_signed**, **sc_unsigned**, **sc_int_base**, **sc_uint_base**, **sc_fxval**, **sc_fxval_fast**, **sc_fxnum**, **sc_fix**, **sc_ufix**, **sc_ufixed**, **sc_fxnum_fast** and derived classes.

Table 15-10 Finite-Precision Addition Operations

Expression	Operation	Support	Limitations
F = F1 + F2;	addition, assign	SYN	
F1 += F2;	addition, assign	SYN	
F += n;	addition, assign	SYN	not supported for type double
F1 = F2 + n;	addition, assign	SYN	not supported for type double
F1 = n + F2;	addition, assign	SYN	not supported for type double

Table 15-11 Finite-Precision Subtraction Operations

Expression	Operation	Support	Limitations
F = F1 - F2;	subtraction, assign	SYN	
F1 -= F2;	subtraction, assign	SYN	
F -= n;	subtraction, assign	SYN	not supported for type double
F1 = F2 - n;	subtraction, assign	SYN	not supported for type double
F1 = n - F2;	subtraction, assign	SYN	not supported for type double

Table 15-12 Finite-Precision Multiplication Operations

Expression	Operation	Support	Limitations
<code>F = F1 * F2;</code>	multiplication, assign	SYN	
<code>F1 *= F2;</code>	multiplication, assign	SYN	
<code>F *= n;</code>	multiplication, assign	SYN	not supported for type double
<code>F1 = F2 * n;</code>	multiplication, assign	SYN	not supported for type double
<code>F1 = n * F2;</code>	multiplication, assign	SYN	not supported for type double

Table 15-13 Finite-Precision Division Operations

Expression	Operation	Support	Limitations
<code>F1 /= F2;</code>	division, assign	SYN	
<code>F /= n;</code>	division, assign	SYN	not supported for type double
<code>F = F1 / F2;</code>	division, assign	NS	
<code>F1 = F2 / n;</code>	division, assign	NS	
<code>F1 = n / F2;</code>	division, assign	NS	

15.1.3.5 Relational (Including Equality) Operators

The level of support for relational (including equality) operators, which are permitted for fixed-point objects, is shown in [Table 15-14](#).

For [Table 15-14](#), note the following:

- **F, F1, F2** represent objects of type **sc_fixed**<W1,I1,Q1,O1,N1>, **sc_fixed**<W2,I2,Q2,O2,N2>, **sc_ufixed**<W1,I1,Q1,O1,N1> or **sc_ufixed**<W2,I2,Q2,O2,N2>
- **n** represents an object of supported numeric type **int**, **long**, **unsigned int**, **unsigned long**, **sc_int**<W1>, **sc_uint**<W1>, **sc_bigint**<W1>, **sc_biguint**<W1>
- The following types are not supported: **double**, **sc_signed**, **sc_unsigned**, **sc_int_base**, **sc_uint_base**, **sc_fxval**, **sc_fxval_fast**, **sc_fxnum**, **sc_fix**, **sc_ufix**, **sc_ufixed**, **sc_fxnum_fast** and derived classes.

Table 15-14 Relational (Including Equality) Operators

Expression	Support
Smaller than operator:	
$F1 < F2$	SYN
$F < n$	SYN
$n < F$	SYN
Smaller than equal operator:	
$F1 \leq F2$	SYN
$F \leq n$	SYN
$n \leq F$	SYN
Greater than operator:	
$F1 > F2$	SYN
$F > n$	SYN
$n > F$	SYN
Greater than equal operator:	
$F1 \geq F2$	SYN
$F \geq n$	SYN
$n \geq F$	SYN
Equal operator:	
$F1 == F2$	SYN
$F1 == n$	SYN
$n == F$	SYN
Not equal operator:	
$F1 != F2$	SYN
$F != n$	SYN
$n != F$	SYN

15.1.3.6 Auto-Increment and Auto-Decrement Operators

The level of support for auto-increment and auto-decrement operators is shown in [Table 15-15](#).

Note These operators are defined for both `sc_fixed` and `sc_ufixed`.

Table 15-15 Auto-Increment/Auto-Decrement Operators

Expression	Support
<code>operator++(int)</code>	SYN
<code>operator--(int)</code>	SYN
<code>operator++()</code>	SYN
<code>operator--()</code>	SYN

15.1.3.7 Unary Operators

The level of support for unary operators is shown in [Table 15-16](#).

Note These operators are defined for both `sc_fixed` and `sc_ufixed`.

Table 15-16 Unary Operators

Expression	Operation	Support
<code>operator-()</code>	unary minus operator	SYN
<code>operator+()</code>	unary plus operator	SYN
<code>operator~()</code>	unary bitwise operator	SYN

15.1.3.8 Shift Operators

The level of support for shift operators is shown in [Table 15-17](#).

Notes

- These operators are defined for both `sc_fixed` and `sc_ufixed`.

- To support the overflow/quantization functionality of a fixed-point variable of size **W** with shift operations, the return type of the operation is of bit width **W+64**. The constant propagation will typically eliminate the extra bits.

Table 15-17 Shift operators

Expression	Operation	Support	Limitations
<code>operator<=(int)</code>	shift, assign left	SYN	
<code>operator>=(int)</code>	shift, assign right	SYN	
<code>operator<<(int)</code>	shift left	SYN	Overflow and quantization supported for up to 64-bit shifts only
<code>operator>>(int)</code>	shift right	SYN	Overflow and quantization supported for up to 64-bit shifts only

15.1.3.9 Bit and Part Selections

The syntax for bit and part selection does not fully match the syntax defined in the SystemC LRM.⁶

Table 15-18 shows the APIs provided by the LRM for selecting bits and ranges from fixed point numbers. We distinguish cases where the bit/part select appears in the left-hand side of an assignment from those where it appears in the right-hand side. The second column shows whether `ctos_fx` supports the feature with the same syntax as the LRM API.

Table 15-18 LRM API for Selecting Bits and Ranges

Feature		Support	Comments
Right-hand side bit select			
<code>const sc_fxnum_bitref operator [] (int) const</code>		SYNS	
<code>const sc_fxnum_bitref bit(int) const;</code>		SYN	
Left-hand side bit select			
<code>sc_fxnum_bitref</code>	<code>operator [] (int);</code>	No	Supported using <code>assign_bit</code>
<code>sc_fxnum_bitref</code>	<code>bit(int);</code>	No	

Table 15-18 LRM API for Selecting Bits and Ranges

Feature	Support	Comments
Right-hand side part select		
const sc_fxnum_subref operator () (int , int) const	No	Supported using template range
const sc_fxnum_subref range (int, int) const;	No	
Left-hand side part select		
sc_fxnum_subref operator () (int , int);	No	Supported using assign_range
sc_fxnum_subref range(int , int);	No	
Right-hand side full part select		
const sc_fxnum_subref operator () () const	SYN	Supported using template range
const sc_fxnum_subref range () const;	SYN	
Left-hand side full part select		
sc_fxnum_subref operator () ();	No	Supported using assign_range
sc_fxnum_subref range();	No	

Table 15-19 shows the APIs provided by `ctos_fx` for selecting and assigning bits and ranges from fixed point numbers. Note that even for cases where the syntax is the same in LRM, the argument types and the return types are still different.

Table 15-19 LRM API for Selecting Bits and Ranges Provided by `ctos_fx`

Feature	Comments
Right-hand side bit select	
<code>bool operator [] (int) const</code>	Same syntax as LRM
<code>bool bit(int) const;</code>	Same syntax as LRM
Left-hand side bit select	
<code>void assign_bit (int , bool);</code>	
Right-hand side part select	
<code>sc_bv<I+1-J> range<I,J> () const</code>	Bit Indices are template parameters

Table 15-19 LRM API for Selecting Bits and Ranges Provided by ctoS_fx

Feature	Comments
Left-hand side part select	
<code>void assign_range<I,J,WR> (sc_bv<WR>) const</code>	Bit Indices are template parameters
Right-hand side full part select	
<code>sc_bv<W> range () const</code>	Same syntax as LRM, but different return type.
Left-hand side full part select	
<code>void assign_full_range<WR>(const sc_dt::sc_bv<WR>& v)</code>	
<code>void assign_full_range(int v)</code>	

The following are some usage examples:

```

sc_ufixed<16,8> f;
..

// **** RHS bit select:
boolb1 = f[3];
boolb2 = f.bit(3);

// **** RHS full part select:
sc_bv<16> bv1 = f.range();
// Equivalent OSCI full part select:
// sc_bv<16> bv1 = f.range().to_uint();

// **** RHS part select:
sc_bv<8> bv2 = f.range<11,4>();
// Equivalent OSCI RHS part select:
// sc_bv<8> bv2 = f.range(11,4).to_uint();

// **** LHS bit select:
f.assign_bit(3, true);
// Equivalent OSCI LHS bit select:
// f.bit(3) = true;

sc_bv<8> bv3;
..
// **** LHS part select:

```

```
f.assign_range<11,4,8>(bv3);  
// Equivalent OSCI LHS part select:  
// f.range(11,4) = bv3.to_uint();  
  
sc_bv<16> bv4;  
..  
// **** LHS full part select:  
f.assign_full_range<16>(bv4);  
// Equivalent OSCI LHS full part select:  
// f.range() = bv4.to_uint();
```

Macros to Encapsulate Syntactic Differences

To minimize inconvenience, CtoS provides the following two macros that encapsulate syntactic differences:

`CTOS_FX_ASSIGN_RANGE(var, val)` encapsulates the operations for full-range assignment, that is, `"var.range() = val;"` for the OSCI SystemC syntax.

`CTOS_FX_ASSIGN_BIT(var, index, val)` encapsulates the operations for setting a bit, that is, `"var[index] = val;"` for the OSCI SystemC syntax.

These macros are available in the file **ctos_fx_macros.h**.

Notes

- Macros **CTOS_FX_ASSIGN_RANGE** and **CTOS_FX_ASSIGN_BIT** are dependent on:

```
#define __CTOS__
```

Do *not* define **__CTOS__** if you are simulating with the OSCI library, but *do* define it if your are simulating or synthesizing with the CtoS library.

- The directive

```
"#include "ctos_fx/ctos_fx_macros.h"
```

must be present in the code even if you are linking with the OSCI or INCISIV SystemC library.

15.1.3.10 Parameter Functions

The level of support for parameter functions is shown in [Table 15-20](#).

Table 15-20 Parameter Functions

Function	Parameter	Support
<code>int wl() const;</code>	W	SYN
<code>int iwl() const;</code>	I	SYN
<code>int n_bits() const;</code>	N	SYN
<code>sc_q_mode q_mode() const;</code>	Q	SYN
<code>sc_o_mode o_mode() const;</code>	O	SYN
<code>const sc_fxcast_switch& cast_switch() const;</code>	sc_switch	NS
<code>const sc_fxtype_params& type_params() const;</code>	sc_fxtype_params	NS

15.1.3.11 Explicit Conversions to Character Strings

The level of support for explicit conversions to character strings is shown in [Table 15-21](#).

Table 15-21 Explicit Conversions to Character Strings

Shortcut Method	Number Representation	Support	Limitations
<code>to_string()</code>	SC_F	SIM	SC_E is not supported
<code>to_hex()</code>	SC_HEX	SIM	
<code>to_dec()</code>	SC_DEC	SIM	
<code>to_bin()</code>	SC_BIN	SIM	
<code>to_oct()</code>	SC_OCT	SIM	

15.1.3.12 Query Values

The level of support for query values is shown in [Table 15-22](#).

Table 15-22 Query Values

Method	Support	Comment
<code>bool is_neg() const;</code>	SYN	
<code>bool is_zero() const;</code>	SYN	
<code>bool quantization_flag() const;</code>	SYN	By default, not in the class definition. Use <code>#define CTOS_FX_WITH_FLAGS</code> to include in the class definition.
<code>bool overflow_flag() const;</code>	SYN	By default, not in the class definition. Use <code>#define CTOS_FX_WITH_FLAGS</code> to include in the class definition.
<code>const sc_fxval value() const;</code>	NS	

Note See “[Overflow and Quantization Flags](#)” on [page 15-7](#) for more information about **`CTOS_FX_WITH_FLAGS`**.

15.1.3.13 Explicit/Implicit Conversions to Primitive Types

The level of support for explicit/implicit conversions to primitive types is shown in [Table 15-23](#).

Table 15-23 Explicit/Implicit Conversions to Primitive Types

Method	Support
Explicit Conversion	
<code>to_double()</code>	SIM
<code>to_short()</code>	SIM
<code>to_ushort()</code>	SIM

Table 15-23 Explicit/Implicit Conversions to Primitive Types

Method	Support
<code>to_int()</code>	SIM
<code>to_uint()</code>	SIM
<code>to_long()</code>	SIM
<code>to_ulong()</code>	SIM
<code>to_int64()</code>	SIM
<code>to_uint64()</code>	SIM
<code>to_float()</code>	SIM
Implicit Conversion	
<code>double()</code>	NS

15.1.3.14 Methods Inherited from Base Classes

The level of support for methods inherited from base classes is shown in [Table 15-24](#).

Table 15-24 Methods Inherited from Base Classes

Base Class	Method	Support
sc_fxnum	<code>void neg (sc_fxval&, const sc_fxnum&);</code>	NS
	<code>void neg (sc_fxnum&, const sc_fxnum&);</code>	NS
	<code>void lshift (sc_fxval&, const sc_fxnum&, int);</code>	NS
	<code>void rshift (sc_fxval&, const sc_fxnum&, int);</code>	NS
	<code>void lshift (sc_fxnum&, const sc_fxnum&, int);</code>	NS
	<code>void rshift (sc_fxnum&, const sc_fxnum&, int);</code>	NS
	<code>void mult (sc_fxval&, const sc_fxnum&, const sc_fxnum&);</code>	NS
	<code>void div (sc_fxval&, const sc_fxnum&, const sc_fxnum&);</code>	NS
	<code>void add (sc_fxval&, const sc_fxnum&, const sc_fxnum&);</code>	NS
	<code>void sub (sc_fxval&, const sc_fxnum&, const sc_fxnum&);</code>	NS
	<code>const sc_fxnum_bitref bit (int) const;</code>	NS
	<code>sc_fxnum_bitref bit (int);</code>	NS

Table 15-24 Methods Inherited from Base Classes

Base Class	Method	Support
sc_fix	<code>void b_not (sc_fix&, const sc_fix&);</code>	NS
	<code>void b_and (sc_fix&, const sc_fix&, const sc_fix&);</code>	NS
	<code>void b_and (sc_fix&, const sc_fix&, const sc_fix_fast&);</code>	NS
	<code>void b_and (sc_fix&, const sc_fix_fast&, const sc_fix&);</code>	NS
	<code>void b_or (sc_fix&, const sc_fix&, const sc_fix&);</code>	NS
	<code>void b_or (sc_fix&, const sc_fix&, const sc_fix_fast&);</code>	NS
	<code>void b_or (sc_fix&, const sc_fix_fast&, const sc_fix&);</code>	NS
	<code>void b_xor (sc_fix&, const sc_fix&, const sc_fix&);</code>	NS
	<code>void b_xor (sc_fix&, const sc_fix&, const sc_fix_fast&);</code>	NS
	<code>void b_xor (sc_fix&, const sc_fix_fast&, const sc_fix&);</code>	NS

15.2 Flex Channels Library

The Cadence *Flex Channels Library* contains a set of synthesizable and highly reusable basic blocks for point-to-point communication between design components. These building blocks are constructed to be easy to use and provide good quality of results (QoR) with high-level synthesis.

Using Flex Channels, you can construct your own synthesizable models, which are:

- simpler and easier to understand than corresponding signal-level models
- easier to reuse than corresponding signal-level models
- easier to verify

The following are some of the benefits of using Flex Channel:

- *Higher abstraction:* Code is written at a high level of abstraction, abstracting out many of the communication details, while keeping the structures necessary for high QoR with high-level synthesis (HLS), ensuring good throughput, latency, timing, and area.
- *Enhanced productivity:* You can begin verification earlier in your design process, overlapping the design and verification cycles for major productivity gains.

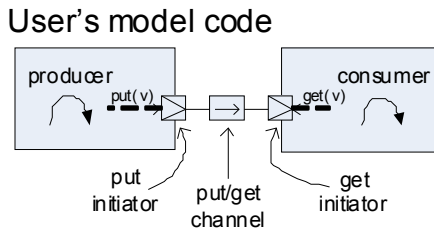
The basic components of the library are:

- the channels, which carry data between design components, and
- the initiators, which provides the means to initiate transactions on the channels.

The main building blocks in the library is the put/get channels and initiators.

The put/get initiators provide standard TLM put and get interfaces to send and receive data through the channel. A data producer can generate data and initiate a put transaction by calling the put() function of the initiator. A consumer gets the data by initiating get transactions on the channel; this is done by calling the get() function of the initiator. The initiators are connected to a channel, which can hold the data until the data consumer picks it up, effectively acting as a point-to-point message-passing protocol.

Figure 15-4 User Model: Blocks Communicate through Initiators and Channel



The Flex Channels Library supports simulation both at the transaction level, and at the signal level, using the same code. The initiators and channels seamlessly define themselves into either standard SystemC TLM or signal-level constructs, depending on the context. The components in the Flex Channels library can be configured using traits. The traits enable you to select if the data signals are reset, which clock edge and reset polarities are used, and if the channels should add wait-statements to avoid combinational cycles.

This lets you create a single model for both the transaction and signal-levels of configurations. This helps to make your code look clean by encapsulating the signal-level protocols. When you change the level of abstraction at which you are working on your models, your user code and structure can stay the same.

The Flex Channels Library can be found in the CtoS install tree at the following location:

Directory	Description
<code>install_directory/share/ctos/include/ctos_flex_channels</code>	Header files for Flex Channels library. This location is added to include paths for CtoS build command and generated makefiles.
<code>install_directory/share/ctos/examples/flex_channels/</code>	Examples for put/get channels.

Flex Channels are further described in the following sections:

- [“Illustrative Example” on page 15-35](#)
- [“Flex Channels Terminology” on page 15-39](#)
- [“Put/Get Channels Protocol” on page 15-40](#)
- [“Parameterizing Structures with Traits” on page 15-41](#)
- [“Initiators and Interfaces” on page 15-44](#)
- [“Pipelined Design with Put/Get Channels” on page 15-50](#)
- [“Hierarchical Initiators” on page 15-54](#)
- [“Custom Traits” on page 15-56](#)
- [“Automatic Configuration of Micro-Architecture” on page 15-61](#)
- [“Simulation and Debugging of Flex Channels” on page 15-62](#)

15.2.1 Illustrative Example

In this section, an illustrative example “Hello World” is presented using the put/get Flex Channels. The code for this example can be found in the CtoS install at this location:

```
install_directory/share/ctos/examples/libraries/flex_channels/hello_world
```

This example illustrates a design pattern typical of put/get channel usage. The process gets some data, computes something, and puts the response on the response channel. Below is the code for the DUT:

```
SC_MODULE(DUT) {
    sc_in<bool>          clk;
    sc_in<bool>          nrst;
    get_initiator<char>  din;
    put_initiator<char>  dout;

    SC_CTOR(DUT)
    : clk("clk")
    , nrst("nrst")
    , din("din")
    , dout("dout")
    {
        SC_THREAD(process);
        sensitive << clk.pos();
        reset_signal_is(nrst,false);
        // Bind clock and reset signal to put/get channel internal logic.
        din .clk_rst(clk,nrst);
    }
}
```

```
dout.clk_rst(clk,nrst);
}

void process() {
    din.reset_get();    // Put/get initiators need to be reset.
    dout.reset_put();
    wait();
    while (1) {
        // Get a character (data element); this call will block if
        // the input channel is empty.
        char c = din.get()
        c = c - ('a' - 'A'); // Convert from lower to uppercase.
        wait();
        // This call will block only if the output channel is full.
        dout.put(c);
        // We need a wait here in case we did both get() and put()
        // in the current cycle (to avoid combinational loop).
    }
}
};
```

The following code shows how the DUT is wired to a simple testbench with stimuli and monitor components through the put/get channels:

```
SC_MODULE(TB) {
    sc_clock          clk;
    sc_signal<bool>    nrst;
#ifdef CTOS_MODEL
    DUT_ctos_wrapper  dut;
#else
    DUT                dut;
#endif
    Stimuli            stimuli;
    Monitor            monitor;

    put_get_channel<char> in_chan;
    put_get_channel<char> out_chan;

    SC_CTOR(TB)
    : clk("clk")      // Clk arguments.
      , nrst("nrst")
#ifdef CTOS_MODEL
      , dut("dut", CTOS_TARGET_SUFFIX(CTOS_MODEL))
#else
      , dut("dut")
#endif
      , stimuli("stimuli")
      , monitor("monitor")
}
```

```
, in_chan("in_chan")
, out_chan("out_chan")
{
    SC_THREAD(reset_thread);
    sensitive << clk.posedge_event();

    dut.clk (clk);
    dut.nrst(nrst);
    dut.din (in_chan); // Connect initiators to channels
    dut.dout(out_chan);

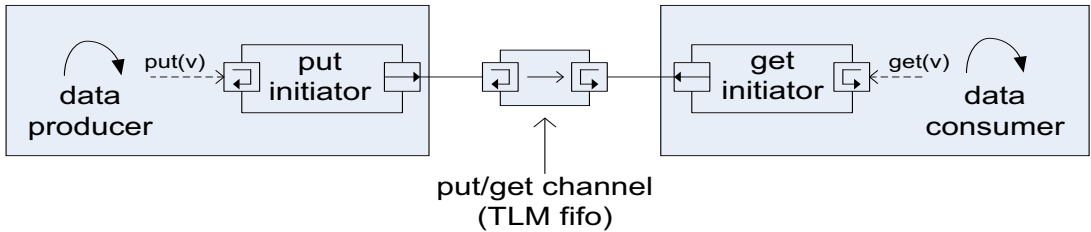
    stimuli.clk (clk);
    stimuli.nrst(nrst);
    stimuli.dout(in_chan); // Connect initiator to channel

    monitor.clk (clk);
    monitor.nrst(nrst);
    monitor.din(out_chan); // Connect initiator to channel
}
...
};
```

As you can see, the usage of the put/get channels is very similar to the usage of TLM FIFOs and signals: one has to reset the initiators, call put/get to read or write data, and the initiators are bound to the channels with syntax similar to binding signals.

The put/get channel can be implemented either at the TLM level or at the signal-level, and this is controlled by a configuration parameter. [Figure 15-5 on page 15-38](#) shows the initiators are specialized for TLM and signal-level implementation by using partial template specialization. The initiator interface to the user code stays the same so that the same design code can be used for both TLM and signal-level simulation.

**Figure 15-5 Flex Channels Initiators are Switch between TLM and Signal-Level
Structure at the TLM-level**



Structure at the signal-level

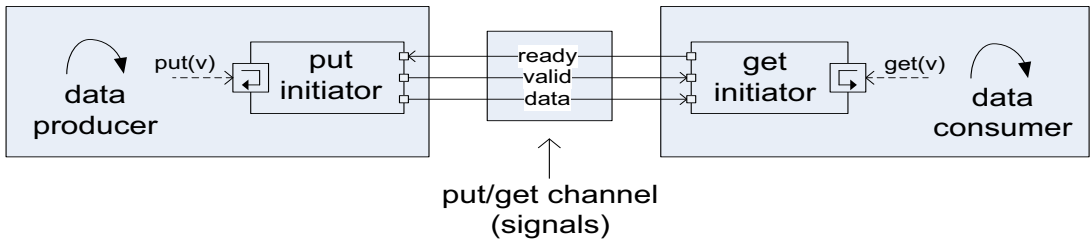


Figure 15-6 on page 15-39 shows the wave diagram for the simulation at the TLM level, where the blue part of the wave diagram is the transactions during delta cycle where the FIFO is filled up by the test bench. Then, the DUT reads one data element every cycle and waits for one cycle (the wait statement at line 52 in the DUT process).

Figure 15-6 Wave Diagram for Simulation at the TLM Level

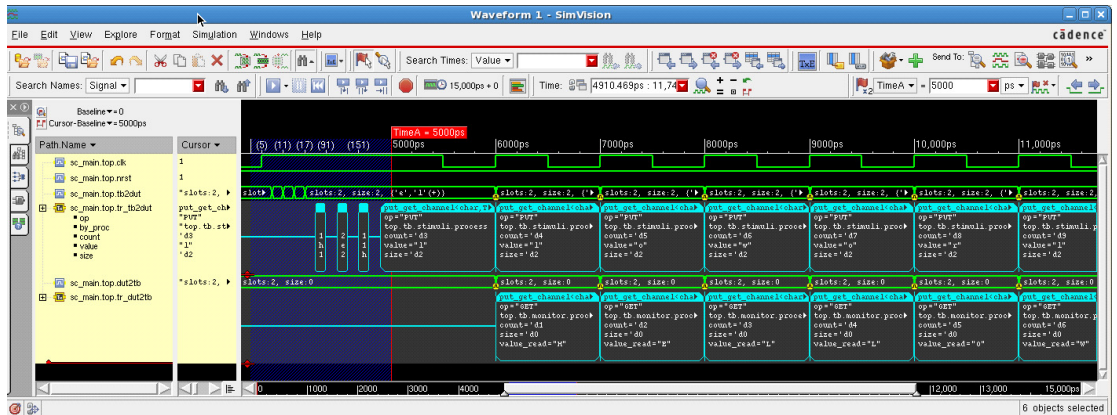
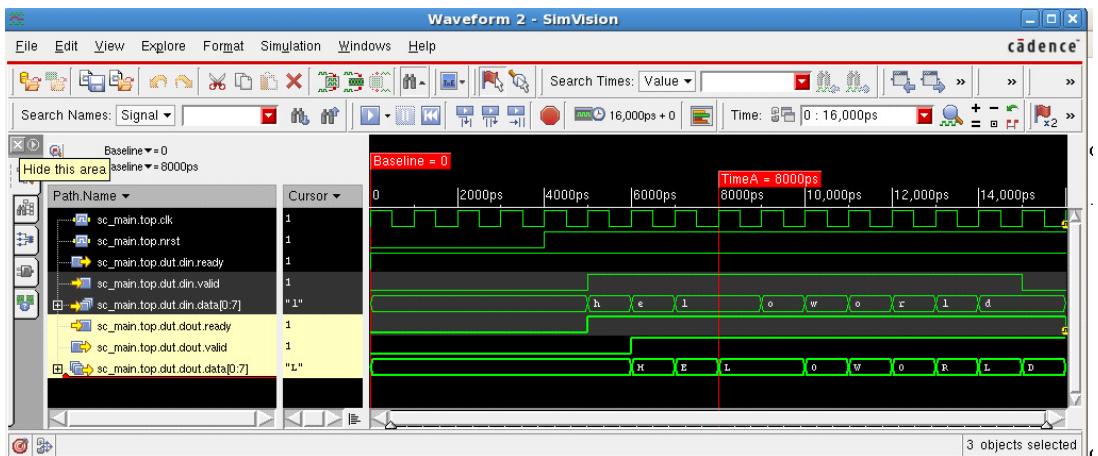


Figure 15-7 on page 15-39 shows the wave diagram for the simulation at the signal-level, where we can see the ready/valid handshakes and data transmission.

Figure 15-7 Wave Diagram for Simulation at the Signal-Level



15.2.2 Flex Channels Terminology

The following is the list of definitions of the key terms used with Flex Channels:

- **Channel:** A channel implements a communication pathway. At the RTL, a channel is implemented with signals. At the transaction level, a channel is implemented with SystemC TLM constructs, such as `tlm_fifo`'s, or ports and exports.
- **Interface:** An interface is a set of functions (called methods) that can be called on an initiator.
- **Initiator:** An initiator is a communication interface on a block that begins (or initiates) transaction communication. Initiators are specialized using traits and partial template specialization for both TML and signal-level implementations. At the transaction-level, an initiator connects directly to the channel and the channel implements the initiator interface. At the signal level, an initiator acts as a transactor to convert the function call into a signal-level handshake.
- **(Always-)blocking:** A blocking initiator has functions that will call `wait()`, and will consume time.
- **May-block:** A may-block initiator has functions that may call `wait()` and may consume time.
- **Non-blocking:** A non-blocking initiator has functions that can never call `wait()`.
- **Peek:** An initiator can provide peek interfaces, which allows to peek at a valid data value in the channel without consuming the data.
- **Traits:** A traits class is a template parameter which contains a set of configuration parameters.

Note The blocking and may-block initiators that have the same blocking put/get interface behave the same at the TLM level, but their behavior differ at the signal level. Functions implementing the always-blocking initiator will always take at least one cycle. Functions implementing the may-block initiators may return in the same cycle in which they are called.

15.2.3 Put/Get Channels Protocol

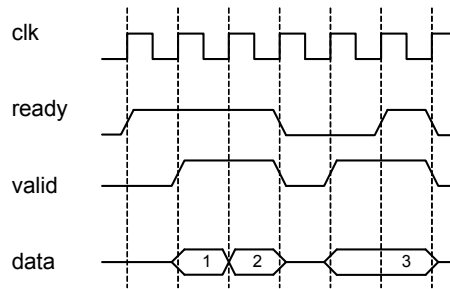
The Flex Channels Library currently contains only the put/get channels library, which is a library for point-to-point communication between a producer process and a consumer process.

A put/get channel has the following characteristics:

- At the transaction-level, the channel is implemented with a FIFO of depth 2.
- At the signal-level, the channel is a set of ready, valid, and data signals.
- The channel is parameterized by the data type of the data item to be transferred.

At the TLM level, the initiators are bound to a FIFO, and at the signal-level, the initiators are bound to the ready, valid, and data signals and the communication is the common ready/valid handshake. [Figure 15-8 on page 15-41](#) shows the wave diagram for this handshake: a data item is transferred every time both the ready and valid signals are true. The figure depicts three data item being transferred.

Figure 15-8 Ready/Valid Protocol used by Put/Get Channel



The put/get channels and initiators provided in the Flex Channels library have the following characteristics:

- Data items are guaranteed to always be reliably delivered between the producer and consumer.
- The initiators are optimized to provide good QoR with HLS.
- The initiators support full throughput: one new data item can be transferred on each clock cycle.
- The channel supports optimal latency consistent with the requirement that no combinational paths stretch over multiple modules: a new data item is available to the consumer one clock cycle after the producer produces it.

15.2.4 Parameterizing Structures with Traits

The selection of the ‘level of abstraction’, TLM or signal-level, is made with a ‘traits’ template parameters. The initiators and channels are parameterized with the traits class and partial template specialization is used to select the configuration. Below is the description of how this is achieved for the get-initiator.

```
template<typename T, typename TRAITS, bool LEVEL>
struct get_initiator_imp
{ };
```

This class is empty because it will be specialized for the values of the template parameter LEVEL. When the LEVEL argument is 0, the following TLM implementation is selected:

```
template<typename T, typename TRAITS>
struct get_initiator_imp<T, TRAITS, 0> // Specialization of LEVEL to 0 for
    TLM
    : sc_module
    , sc_interface
{
    get_initiator_imp(sc_module_name);
```

```

template<typename CHAN>
void operator() (CHAN& chan);
template<typename CHAN>
void bind(CHAN& chan);
template<typename CLK, typename RST>
void clk_rst(CLK&, RST&);

virtual void get(T& v);
virtual T get(tlm_tag<T>* t=0);
virtual void reset_get(tlm_tag<T>* t=0);
virtual void reset();

private:
    sc_port<tlm_blocking_get_if<T> > p;
};

```

The TLM version contains an **sc_port** implementing the **blocking-get** tlm interface. This **sc_port** is connected to a FIFO and each of the `get()` function call is forwarded to the FIFO through this port.

Setting the LEVEL parameter to 1 selects the signal-level implementation. Below is the partial specialization of the for the implementation of the get initiator:

```

template<typename T, typename TRAITS>
struct get_initiator_imp<T, TRAITS, 1> // Specialization of LEVEL to 1 for
    signal-level
    : sc_module
    , sc_interface
{
    get_initiator_imp(sc_module_name);

    template<typename CHAN>
    void operator() (CHAN& chan);
    template<typename CHAN>
    void bind(CHAN& chan);
    template<typename CLK, typename RST>
    void clk_rst(CLK&, RST&);

    virtual void get(T& v);
    virtual T get(tlm_tag<T>* t=0);
    virtual void reset_get(tlm_tag<T>* t=0);
    virtual void reset();

public:
    sc_in<bool> clk;
    sc_in<bool> rst;
    sc_in<bool> valid;
    sc_in<T> data;
    sc_out<bool> ready;

```

```
    ...  
};
```

You can see the methods are the same, but now this time the class contains signal-level input and output ports. These ports are connected to the channel which contains the ready and valid signals. The class for the get initiator is defined as below. It derives from the implementation class and has two template parameters. The value for selecting the TLM or the signal-level is taken from the traits class:

```
template <typename T, typename TRAITS=DEFAULT_TRAITS>  
struct get_initiator : get_initiator_imp<T,TRAITS,TRAITS::level> {  
    get_initiator(sc_module_name n = "get_initiator")  
        : get_initiator_imp<T,TRAITS,TRAITS::level>(n)  
    { }  
};
```

We now explain how the traits are structured. The base class for TLM traits defines the level variable to hold value 0:

```
struct TLM_TRAITS_BASE {  
    // The variable "level" is used to select between the TLM and the  
    // signal-level configuration of initiators and channels.  
    //      - level = 0 : TLM channel and initiators,  
    //      - level = 1 : signal-level channel and initiators.  
    static const bool level = 0;  
};
```

As illustrated above, this variable is what is used to select the partial template specialization for the TLM implementation. The traits class for selecting TLM implementation can be defined from the base TLM traits class as follows:

```
struct TLM_TRAITS : TLM_TRAITS_BASE { };
```

The traits class for selecting the signal-level implementation defines the level variable to hold value 1:

```
struct SIG_TRAITS_BASE {  
    static const bool level = 1;  
    // The variable "Allow_Multiple_Calls_Per_Cycle" is used to insert a  
    wait()  
    // in a may-block put() or get(). This wait is executed when the function  
    // is called more once in a cycle. Note that this additional wait() will  
    // not be in the RTL.  
    static const bool Allow_Multiple_Calls_Per_Cycle = 0;  
    // Setting variable "Allow_Multiple_Calls_Per_Cycle_RTL" will insert  
    the  
    // additional wait statement into the code parsed by CtoS. If this  
    variable  
    // is not set, a bit "multiple_calls_in_rtl" is set in the RTL when there  
    // is more than one call per cycle.
```

```
static const bool Allow_Multiple_Calls_Per_Cycle_RTL = 0;
};
```

The base signal-traits class also contains other variables. For information on how to use these variables, see [“Waits in May-Block Initiator for TLM Simulation” on page 15-58](#). These are used to select whether states are added to the processes to avoid combinational cycle. Following is the definition of the signal-level for processes sensitive to positive clock edge, negative reset and whether to reset the data signals flops or not:

```
struct SIG_TRAITS_pCLK_nRST : SIG_TRAITS_BASE
{
    static const bool PosEdgeClk = 1;
    static const bool ResetLevel = 0;
    static const bool ResetData = 0;
};
```

Default traits are defined as follows. First, we declare the configuration for the signal-level traits. In the Flex Channel library, the default configuration is sensitivity to positive clock edge, negative reset, and not to reset the data signals:

```
typedef SIG_TRAITS_pCLK_nRST SIG_TRAITS;
```

Then, the default traits are defined around the TLM_SIM macro flag:

```
#ifdef TLM_SIM
typedef TLM_TRAITS DEFAULT_TRAITS;
#else
typedef SIG_TRAITS DEFAULT_TRAITS; // posedge clock, negative reset
#endif
```

When the TLM_SIM macro-flag is defined, the TLM configuration is selected for all initiators and channels that are using the default traits. If the TLM_SIM flag is not defined, all initiators and channels with the default configuration are at the signal level.

For more information on how to change the default configuration or use different configurations for different initiators, see [“Custom Traits” on page 15-56](#).

15.2.5 Initiators and Interfaces

This section describes the interface that the put/get initiators can provide to a process. The following are the eight kinds of initiators:

- blocking put
- blocking get

- may-block put
- may-block get
- may-block get-peek
- non-blocking put
- non-blocking get
- non-blocking get-peek

The following methods are provided by all the initiators:

```
void          operator()(CHAN& chan);          // bind to channel
void          clk_rst (CLK&, RST&);           // bind clock and
void          clk_rst (CLK&, RST&, SRTS&);    // reset signals
virtual void  reset();                          // reset the initiator
```

The following sections describes about the initiators and how to choose an initiator:

- [“Blocking put Initiators” on page 15-45](#)
- [“Blocking get Initiators” on page 15-46](#)
- [“May-block put Initiators” on page 15-46](#)
- [“May-block get Initiators” on page 15-46](#)
- [“May-Block Get-Peek Initiators” on page 15-47](#)
- [“Non-Blocking Put Initiators” on page 15-47](#)
- [“Non-Blocking Get Initiators” on page 15-48](#)
- [“Non-Blocking Get-Peek Initiators” on page 15-48](#)
- [“Choosing Appropriate Initiator” on page 15-48](#)
- [“Combinational Cycles with May/Non-Blocking Initiators” on page 15-49](#)

15.2.5.1 Blocking put Initiators

The blocking put initiator is declared with the following syntax:

```
b_put_initiator<type,traits> name;
```

Blocking put initiators provides the following methods:

```
virtual void reset_put();          // reset the put side of the channel
virtual void put(const T &v);     // put item, waits until successful
```

The put function will always take one cycle to execute.

Note The blocking put initiators cannot be used in a pipelined loop.

15.2.5.2 Blocking get Initiators

The blocking get-initiator is declared with the following syntax:

```
b_get_initiator<type,traits> name;
```

Blocking get-initiators provide the following methods:

```
virtual void reset_get();    // reset the get side of the channel
virtual void get(T &t);      // get item, waits until successful
virtual T    get();          // get item, waits until successful
```

The get() method of the blocking-get initiator will always take one cycle, but it does not have an input buffer.

Note The blocking get-initiators cannot be used in a pipelined loop.

15.2.5.3 May-block put Initiators

The may-block put-initiator is declared with the following syntax:

```
put_initiator<type> name;
```

May-block put-initiators provide the following methods:

```
virtual void reset_put();    // reset the put side of the channel
virtual void put(const T &v); // put item, waits until
```

Syntactically, these are the same as the ones provided for the blocking-get initiators. Semantically, the difference is that the methods may or may not consume a clock cycle. This initiator can be used in a pipeline.

15.2.5.4 May-block get Initiators

The may-block get-initiator is declared with the following syntax:

```
get_initiator<type> name;
```

May-block get-initiators provide the following methods:

```
virtual void reset_get();    // reset the get side of the channel
virtual void get(T &t);      // get item, waits until successful
```

```
virtual T    get();           // get item, waits until successful
```

Syntactically, these are the same as the ones provided for the blocking-put initiators. Semantically, the difference is that methods may or may not consume a clock cycle. There is an input buffer that will buffer the incoming data when the process is not ready to consume it immediately. This buffer is necessary to maintain full throughput with one data item transferred per cycle.

15.2.5.5 May-Block Get-Peek Initiators

The may-block get-peek-initiator is declared with the following syntax:

```
get_peek_initiator<type> name;
```

May-block get-peek-initiators provide the following methods:

```
virtual void reset_get() ; // reset the get side of the channel
virtual void get(T& v);    // get item, waits until successful
virtual T    get();        // get item, waits until successful
virtual void peek(T &v) const; // read without consuming the data item,
    // wait until successful
virtual T    peek() const; // read without consuming the data item,
    // wait until successful
```

The may-block get-peek-initiator is the same as the may-block get-initiator, with the addition of the peek methods. A peek allows to read an incoming data value without taking it off the channel. The peek methods will block if the channel is empty, until a new valid value is on the channel.

15.2.5.6 Non-Blocking Put Initiators

The non-blocking put-initiator is declared with the following syntax:

```
nb_put_initiator<type> name;
```

Non-blocking put-initiators provide the following methods:

```
virtual void reset_put() ; // reset the put side of the channel
virtual bool nb_put(const T &v) ; // put item, waits until successful
virtual bool nb_can_put() const; // returns true if there is the
    // channel is not full
```

These methods never call wait (will never cost a cycle). The Boolean value returned by **nb_put()** is true if the data was put on the channel. And, the Boolean value returned by **nb_can_put()** is true if calling **nb_put()** in this cycle will succeed.

15.2.5.7 Non-Blocking Get Initiators

The non-blocking get -initiator is declared with the following syntax:

```
nb_get_initiator<type> name;
```

The non-blocking get-initiators provide the following methods:

```
virtual void reset_get() ;    // reset the get side of the channel
virtual bool nb_get(T &t) ;    // get item, returns true if successful
virtual bool nb_can_get() const; // returns true if the channel is not empty
```

These methods will never consume a clock cycle. There is an input buffer that will buffer the incoming data when the process is not ready to consume it immediately. This buffer is necessary to maintain full throughput with one data item transferred per cycle.

15.2.5.8 Non-Blocking Get-Peek Initiators

The may-block get-peek-initiator is declared with the following syntax:

```
nb_get_peek_initiator<type> name;
```

May-block get-peek-initiators provide the following methods:

```
virtual void reset_get ();      // reset the get side of the channel
virtual bool nb_get(T& v);      // get item, returns true if successful
virtual bool nb_can_get() const; // returns true if the channel is not empty
virtual bool nb_peek(T& v) const; // read without consuming the data item
virtual bool nb_can_peek() const; // returns true if the channel is not
//empty
```

The non-blocking get-peek-initiator is the same as the non-blocking get-initiator, with the addition of the peek methods. A peek allows to read an incoming data value without taking it off the channel. The peek methods will not block if the channel is empty, but it will return false.

15.2.5.9 Choosing Appropriate Initiator

When choosing an appropriate initiator for your application you should consider the following:

- A loop that contains calls to put() and get() functions, which are implemented by always-block initiator cannot be pipelined. Functions implemented by non-blocking and may-block initiators can be pipelined.

Note Make sure that both may-block put() and get() contain an internal stall loop.

- Synthesizing always-blocking initiators generally produces simpler hardware than non-blocking and may-block initiators. This is because blocking initiators do not have the input buffer and the internal logic to store the incoming data into the buffer when the process is not picking up the data in the same cycle.
- The recommended use of non-blocking initiators is only for designs that require features that cannot be nicely coded with the other initiators. The typical example is one where we are processing data from multiple channels, as shown in the below:

```
SC_MODULE(DUT) {
    sc_in<bool>  clk;
    sc_in<bool>  nrst;
    nb_get_initiator<int> req1;
    nb_get_initiator<int> req2;
    b_put_initiator <int>rsp;

    ...

    void process() {
        req1.reset_get();
        req2.reset_get();
        rsp.reset_put();
        wait();
        while (true) {
            int req = 0;
            while (!req1.nb_can_get() || !req2.nb_can_get()) wait();
            if (req1.nb.can_get()) {
                req1.nb_get(req);
            } else if (req2.nb.can_get()) {
                req2.nb_get(req);
            }
            rsp.put(req);
        }
    }
};
```

15.2.5.10 Combinational Cycles with May/Non-Blocking Initiators

At the signal level, the methods of the may-block and non-blocking initiators (**put()**, **get()**, **nb_put()**, and **nb_get()**) can only be called once in a given clock cycle. The reason for that is that the logic inside the channel is registered and the changes in the channel will only be visible at the next cycle.

If you try to call one of these functions a second time in a cycle, an error is reported during the simulation of the input SystemC.

Note At the transaction level, there is no such limitation, and you can call these functions twice in the same clock cycle.

15.2.6 Pipelined Design with Put/Get Channels

In this section we show the design pattern to use the put/get channels to pipeline a design. The code for this example can be found in the CtoS install at:

```
install_directory/share/ctos/examples/libraries/flex_channels/pipeline
```

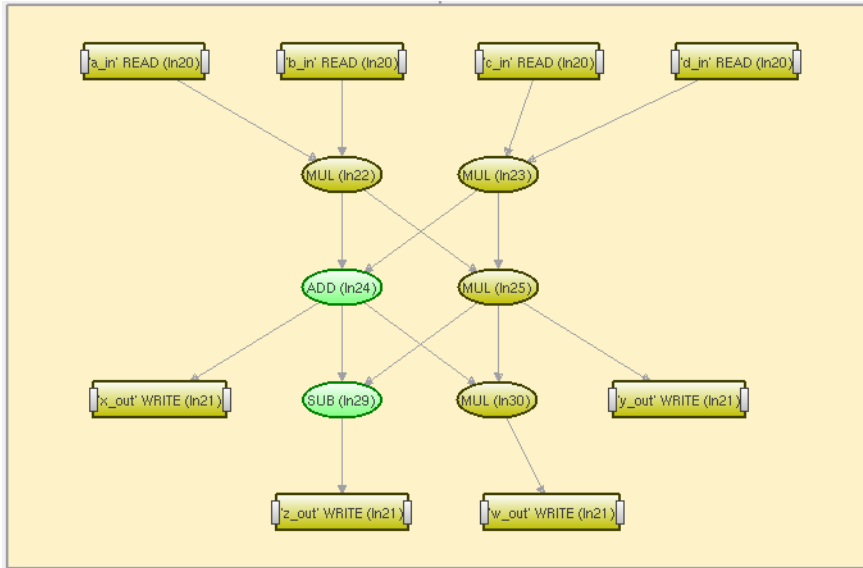
We start with a C function which is modeling arithmetic datapath computation:

```
void fun(int  a, int  b, int  c, int  d,
        int* x, int* y, int* z, int* w ) {
    int v1 = a * b;
    int v2 = c * d;
    int v3 = v1 + v2;
    int v4 = v1 * v2;

    *x = v3;
    *y = (v1 * v2) >> 3;
    *z = v4 - v3;
    *w = v4 * v3;
}
```

This C function has four inputs and four outputs. The data-flow graph for the function is showed in [Figure 15-9 on page 15-51](#).

Figure 15-9 C Function to Pipeline



You can see the chain of multipliers which is what we will want the scheduler to spread over multiple pipeline stages. We will wrap this function in a process which will get the input data and put the output data and we will pipeline that process. The resulting circuit is a pipeline which can have bubbles and back pressure. Assuming that the four inputs are sent at the same time from one process in a source module, and that the four outputs are send at the same time to one process in a sink module, the code to pipeline this function is as follows:

```

// Declare a bit vector typedef to carry all input and output data
typedef sc_biguint<128> DT;

// -----
// SystemC DUT module wrapping the C function.
//
SC_MODULE(DUT) {
    sc_in<bool>      clk;
    sc_in<bool>      nrst;
    get_initiator<DT> din;// data input initiator.
    put_initiator<DT> dout;// data output initiator.

    SC_CTOR(DUT)
    : clk("clk")
    , nrst("nrst")
    , din("din")
    , dout("dout")

```

```
{
    SC_THREAD(process); // One process to read inputs,
    sensitive << clk.pos(); // call function and write outputs.
    reset_signal_is(nrst, false);

    din.clk_rst(clk, nrst);
    dout.clk_rst(clk, nrst);
}

void process() {{
    din.reset_get();
    dout.reset_put();
    wait();

    PROCESS_LOOP:
    while (1) {
        DT v = din.get(); // read inputs

        // Convert types for data inputs:
        int x_i = v.range( 31, 0).to_int();
        int y_i = v.range( 63,32).to_int();
        int z_i = v.range( 95,64).to_int();
        int w_i = v.range(127,96).to_int();

        int x_o,y_o,z_o,w_o; // for fun outputs

        // Call the C function here (compute).
        fun( x_i,  y_i,  z_i,  w_i,
            &x_o, &y_o, &z_o, &w_o);

        // Convert types for data outputs:
        v.range( 31, 0) = x_o;
        v.range( 63,32) = y_o;
        v.range( 95,64) = z_o;
        v.range(127,96) = w_o;

        dout.put(v); // write outputs
        wait();
    }
}
};
```

The code template is as follow: get the data, convert the types if needed, compute, convert outputs if needed, and send outputs. In the Tcl script for this example, we inline the function and then use the following command to pipeline this loop:

```
pipeline_loop \
    -init_interval 1 -min_lat_interval 2 -max_lat_interval 100 \
    -expand_before [flex_channels_nets dout] \
    [find -node PROCESS_LOOP_while_begin]
```

Figure 15-10 on page 15-53 shows the effect of the pipeline loop command. The pipeline expansion point, where the additional stages will be added, is specified using the **flex_channels_nets** command. This command returns all the signals used by an initiator, with the effect that the extra stages will be added before access to any of these signals. You can see the stall loop for the get-initiator in the first stage, and the stall loop and the other signals for the put-initiator have been pushed to the second stage. Additional pipeline stages will be inserted between the first and the second stages.

For more information about the **flex_channels_nets** command, see “[flex_channels_nets](#)” on page E-69.

Figure 15-10 Pre-Schedule Pipeline

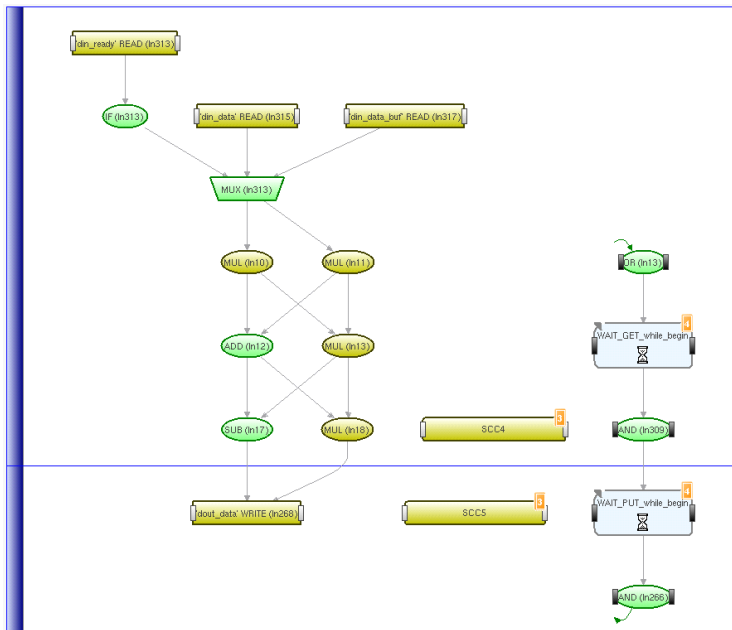


Figure 15-11 Post-Scheduled Pipelined

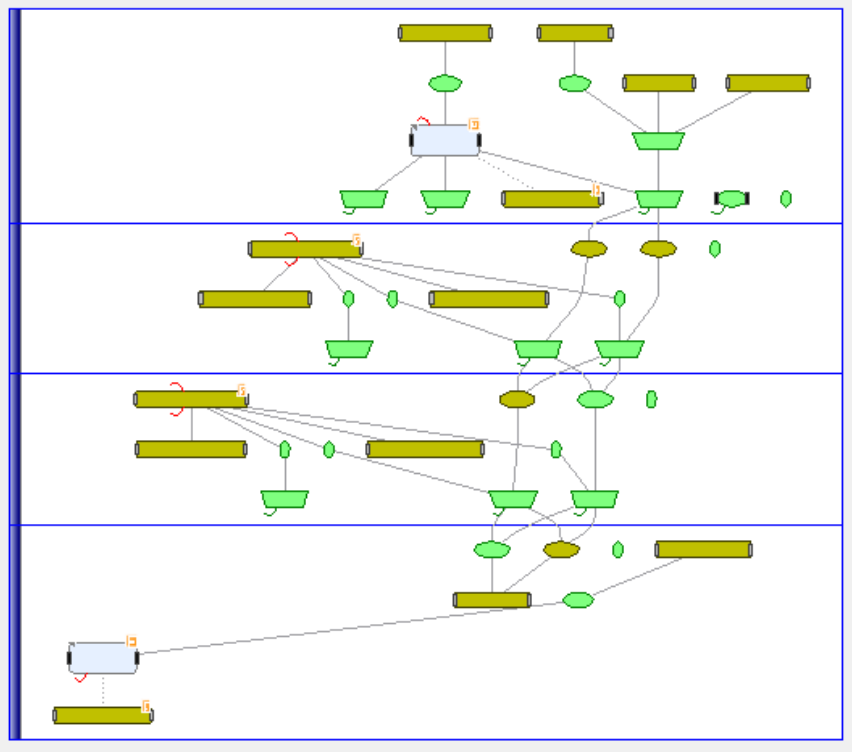


Figure 15-11 on page 15-54 shows how the pipeline was built by CtoS. One can observe that the scheduler added two extra stages in the pipeline. This is because the timing for a multiplier is about the same as the clock period. The four multipliers have been spread over the last three stages in the pipeline. Consequently, the stall loop for the get initiator is in the first stage, and the stall loop and output logic for the put initiator is in the fourth stage.

The other ops in the diagram are for the pipeline control logic that is added in the structural model.

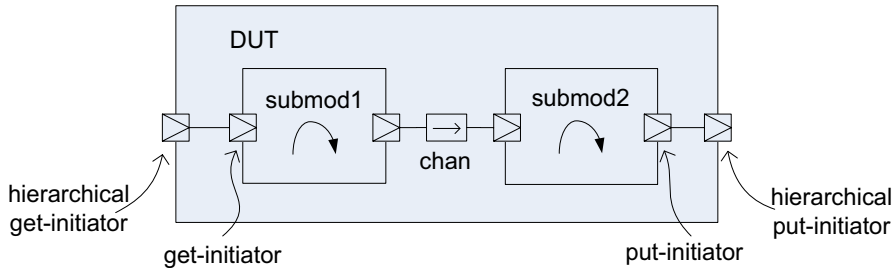
Note that one can get the same effect by adding label in the code right before the put() and specify the pipeline stage insertion point to be at that label.

15.2.7 Hierarchical Initiators

This example shows how to use put/get channels in hierarchical designs. You can find the source code for this example in the following subdirectory of the CtoS install:

```
install_directory/share/ctos/examples/libraries/flex_channels/hierarchy
```

Figure 15-12 Hierarchical Design with Put/Get Channels



Hierarchical design are like the one illustrated in [Figure 15-12 on page 15-55](#), where the parent module instantiates two instances of the child module, and the parent uses hierarchical initiator to connect the initiators of the child module to the DUT boundary. A put/get channel is used to connect the output of submod1 to the input of submod2. This code for the design is encoded as follows:

```
SC_MODULE(SubMod) {
    sc_in<bool>          clk;
    sc_in<bool>          nrst;
    b_get_initiator <int> din;
    nb_put_initiator<int> dout;

    SC_CTOR(SubMod) {
        SC_THREAD(process);
        sensitive << clk.pos();
        reset_signal_is(nrst, false);

        din.clk_rst(clk, nrst);
        dout.clk_rst(clk, nrst);
    }

    void process() {
        din.reset_get();
        dout.reset_put();
        wait();
        while(1) {
            int v = din.get();
            while (!dout.nb_put(v)) wait();
        }
    }
};

SC_MODULE(DUT) {
    sc_in<bool>          clk;
    sc_in<bool>          nrst;
```

```
hier_get_initiator<int> din; // hierarchical initiators to be
hier_put_initiator<int> dout; // connected to leaf modules.

SubMod                                submod1;
SubMod                                submod2;

put_get_channel<int>    chan; // internal channel.

SC_CTOR(DUT)
{
    : clk("clk")
    , nrst("nrst")
    , din("din")
    , dout("dout")
    , submod1("submod1")
    , submod2("submod2")
    , chan("chan")
{
    submod1.clk(clk);
    submod1.nrst(nrst);
    submod1.din(din); // connect leaf module to DUT boundary.
    submod1.dout(chan); // connect to internal channel.

    submod2.clk(clk);
    submod2.nrst(nrst);
    submod2.din(chan); // connect to internal channel.
    submod2.dout(dout); // connect leaf module to DUT boundary.
}
};
```

Note The hierarchical initiator (**hier_put_initiator** and **hier_get_initiator**) can be connected to any kind of initiators (blocking, non-blocking, and so on).

15.2.8 Custom Traits

This section describes how to use traits to configure the initiators.

- “Process Sensitivity and Reset” on page 15-57
- “Waits in May-Block Initiator for TLM Simulation” on page 15-58

15.2.8.1 Process Sensitivity and Reset

For the signal-level configuration, initiators can be configured for clock sensitivity, reset level and whether to reset the data signals or not. [Table 15-25 Traits Class for Process Characteristics on page 15-57](#) lists the traits classes that are defined to allow for all combinations.

Table 15-25 Traits Class for Process Characteristics

Traits Class	Clock Edge	Reset Level	Reset Data Signals
SIG_TRAITS_pCLK_nRST	posedge	negative	No
SIG_TRAITS_pCLK_nRST_ResetData	posedge	negative	Yes
SIG_TRAITS_pCLK_pRST	posedge	positive	No
SIG_TRAITS_pCLK_pRST_ResetData	posedge	positive	Yes
SIG_TRAITS_nCLK_nRST	negedge	negative	No
SIG_TRAITS_nCLK_nRST_ResetData	negedge	negative	Yes
SIG_TRAITS_nCLK_pRST	negedge	positive	No
SIG_TRAITS_nCLK_pRST_ResetData	negedge	positive	Yes

Redefining Default Traits

Redefining default traits will set the traits class for all initiators used in SystemC program. For instance, if we want to define signal-level traits for posedge clock, positive reset, and reset of data signals, we first define a typedef enclosed in the **TLM_SIM** directive as follows:

```
#include <ctos_flex_channels_traits.h>
#define FLEX_CHANNELS_OVERRIDE_DEFAULT_TRAITS
#ifdef TLM_SIM
typedef TLM_TRAITS  DEFAULT_TRAITS;
#else
typedef SIG_TRAITS_pCLK_pRST_ResetData  DEFAULT_TRAITS;
#endif
#include <ctos_flex_channels.h>
```

First, we include the Flex Channels traits file which has the structures for the signal-level traits. Then, we set the macro **FLEX_CHANNELS_OVERRIDE_DEFAULT_TRAITS** to override the default traits. After that, we define the default traits to be the sensitivity and reset characteristics we want. This defines the default configuration for the traits. Provided this is done in the header file that is included in all the other files in the design, then the traits will all be for the newly defined default value:

```
SC_MODULE(DUT) {
```

```
    sc_in<bool>          clk;
    sc_in<bool>          rst;
    get_initiator<char>  din;
    put_initiator<char>  dout;
    ...
};
```

You can find the source code for this example in the following subdirectory of the CtoS install:

```
install_directory/share/ctos/examples/libraries/flex_channels/traits_default
```

Defining Traits Specific to Initiator

For a given initiator, the traits can be specified as the second template argument. For instance, if we want to define signal-level traits for posedge clock, positive reset and reset of data signals, we first define a typedef enclosed in the TLM_SIM directive as follows:

```
#ifdef TLM_SIM
typedef TLM_TRAITS                                level;
#else
typedef SIG_TRAITS_pCLK_pRST_ResetData  level;
#endif
```

Then, we specify the newly defined level to be the traits parameter for the initiators which we want to have those traits. This is done as follows:

```
SC_MODULE(DUT) {
    sc_in<bool>          clk;
    sc_in<bool>          rst;
    get_initiator<char, level>  din; // use defined traits
    put_initiator<char, level>  dout; // use defined traits
    ...
};
```

You can find the source code for this example in the following subdirectory of the CtoS install:

```
install_directory/share/ctos/examples/libraries/flex_channels/traits_param
```

15.2.8.2 Waits in May-Block Initiator for TLM Simulation

Fast TLM simulation requires avoiding the use of wait statements and event driven synchronization. You can find the source code of this example in the following subdirectory of the CtoS install:

`install_directory/share/ctos/examples/libraries/flex_channels/multi_waits_in_cycle`

The DUT process in the example has the following structure:

```
SC_MODULE(DUT) {
    sc_in<bool>          clk;
    sc_in<bool>          nrst;

    get_initiator<REQ>    din;
    put_initiator<RSP>    dout;

    SC_CTOR(DUT) {
        SC_CTHREAD(process, clk.pos());
        reset_signal_is(nrst, false);
        din.clk_rst(clk, nrst);
        dout.clk_rst(clk, nrst);
    }

    void process() {
        din.reset_get();
        dout.reset_put();
        wait();
        while (1) {
            REQ req = din.get();

            COMPUTATION_START: ;

            sc_uint<32> v1(req.range(31, 0));
            sc_uint<32> v2(req.range(63, 32));
            sc_uint<32> v3(req.range(95, 64));
            sc_uint<32> v4(req.range(127, 96));
            RSP rsp1 = (v1*v2) / 8;
            RSP rsp2 = (v3*v4) * rsp1;
            RSP rsp = rsp1 + rsp2 / 16;

            COMPUTATION_END: ;

            dout.put(rsp);
        }
    }
};
```

Recall that the put and get functions of may-block initiators cannot be called more than once per cycle. In order to use may-block initiators without waits in a TLM simulation and without modifying the code of the DUT, there are two extra flags in the signal-level traits class:

```
static const bool    Allow_Multiple_Calls_Per_Cycle      = 0;
static const bool    Allow_Multiple_Calls_Per_Cycle_RTL = 0;
```

The variable **Allow_Multiple_Calls_Per_Cycle** is used to insert a **wait()** in a may-block **put()** or **get()** when it is called more than once in a clock cycle. This additional **wait()** will be executed in SystemC simulation, and not be in the RTL. Setting the variable **Allow_Multiple_Calls_Per_Cycle_RTL** will insert the additional wait statement into the code parsed by CtoS. A new traits class with these attributes on can be globally defined as follows:

```
#include <ctos_flex_channels_traits.h>
#define FLEX_CHANNELS_OVERRIDE_DEFAULT_TRAITS

#ifdef TLM_SIM
typedef TLM_TRAITS  DEFAULT_TRAITS;
#else
struct MyTraits : SIG_TRAITS {
    static const bool Allow_Multiple_Calls_Per_Cycle = 1;
    static const bool Allow_Multiple_Calls_Per_Cycle_RTL = 1;
};
typedef MyTraits    DEFAULT_TRAITS;
#endif

#include <ctos_flex_channels.h>
```

The synthesis of the DUT will require to break the combinational cycle. In the example, this is done by creating a state. The example is then scheduled with relaxed latency, and a few states are added to get positive slack.

Note that using these traits will cause a few additional flops in the RTL, which should get optimized out by the logic synthesis tool.

Important The post-build simulation model is going to fail when you use those traits because it will have a combinational loop.

15.2.8.3 Asynchronous and Multiple Resets

The default configuration of FlexCahnnels assumes a single, asynchronous reset signal. It is possible to configure this primary reset to be asynchronous, and also to optionally add one more synchronous reset signal.

The following trait variable controls if the primary, "hard" reset signal is synchronous or not. The default value is false, indicating that the hard reset is asynchronous by default:

```
static const bool ResetSync ;
```

The active level of the hard reset is controlled by the following variable with default value `false`, indicating active-low:

```
static const bool ResetLevel;
```

And, the following trait variable controls if there is another, "soft" reset. The default value is `false`, indicating that there is a single hard reset:

```
static const bool HasSoftReset;
```

The active level of the soft reset is controlled by the following variable with default value `false`, indicating active-low:

```
static const bool SoftResetLevel;
```

The soft reset can only be synchronous.

If you specify:

```
static const bool HasSoftReset=1;
```

then you must call:

```
void clk_rst(CLK&, RST&, SRST&);
```

to connect the clock and reset signals. Otherwise, you must use:

```
void clk_rst(CLK& , RST&); .
```

The configuration of resets described here is only used for for resetting the side-process inside FlexChannels initiators. You still must use:

```
reset_signal_is(...)
```

and/or

```
async_reset_signal_is(...)
```

to properly configure resets of your SC_CTHREAD's. It is also your responsibility to ensure that the two configurations are consistent.

15.2.9 Automatic Configuration of Micro-Architecture

A design using the CtoS Flex Channel Library can have a large number of small functions that originate from the library.

After successfully building a design, CtoS will automatically inline all behaviors that originate from the CtoS Flex Channel Library. Inlining these behaviors will usually have a positive impact on QoR, as it will reduce the number of behaviors and enable CtoS to further optimize the logic.

In addition, CtoS will automatically create protocol regions inside the put/get channel. Protocol regions will ensure that state insertion by scheduler will not break communication protocols.

To disable the automatic configuration, add the following Tcl command at the beginning of your configuration script:

```
set ::ctos::enable_config_lib_ctos_flex_channels 0
```

15.2.10 Simulation and Debugging of Flex Channels

Flex Channels enable you to create a single model and a single testbench that work at multiple levels of abstraction.

This section describes the following:

- [“TLM and Signal-Level Simulation” on page 15-62](#)
- [“Transaction Recording” on page 15-63](#)

15.2.10.1 TLM and Signal-Level Simulation

The use of the Flex Channels Library enables switching between TLM and signal-level interfaces. The advantages and disadvantages of each type of interface are described in this section.

TLM interfaces:

- enables fast simulation
- includes simulation safety checks against multiple put calls in the same cycle
- are, however, not synthesizable

A TLM interface where the channel is implemented as a two-place FIFO is enabled by defining the TLM_SIM macro, as follows:

```
#define TLM_SIM
```

If you are using a generated Makefile, you can simulate (at transaction level) with this command:

```
make USER_ARGS=-DTLM_SIM orig_sim
```

Signal-level interfaces, where the channel is a set of signals and a transactor that implements a read/valid protocol:

- enables good QoR synthesis
- includes resetting of data and simulation checks to ensure that all initiators are reset

15.2.10.2 Transaction Recording

Flex Channels support the writing of transaction records that can be browsed later using the Cadence SimVision product. The recording mechanism is built into the Flex Channels Library and you do not need to instrument the code.

To enable transaction recording, define the CDS_TR macro, as follows:

```
#define CDS_TR
```

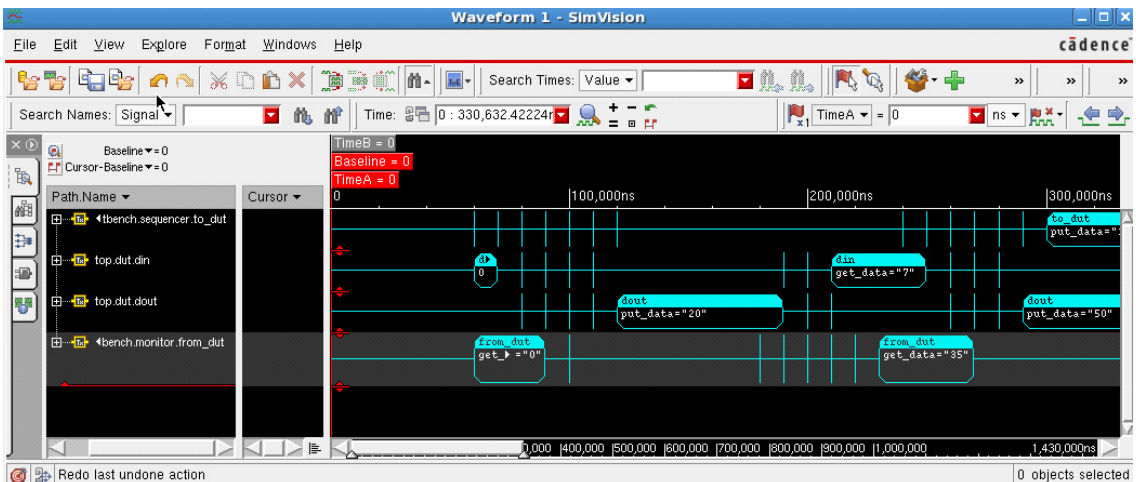
Note CDS_TR is consistent with other uses of transaction recording in Cadence, if you turn on transaction recording with a single macro instead of with a Flex Channels-specific macro.

Figure 15-3 on page 15-4 shows a screen shot of Cadence SimVision with transactions recorded using this capability, for the following example (which is included in the CtoS release):

```
install_directory/share/ctos/examples/flex_channels/may_block
```

Note For details on how to run the simulation and how to start the viewer for the transactions, see the README included with the example.

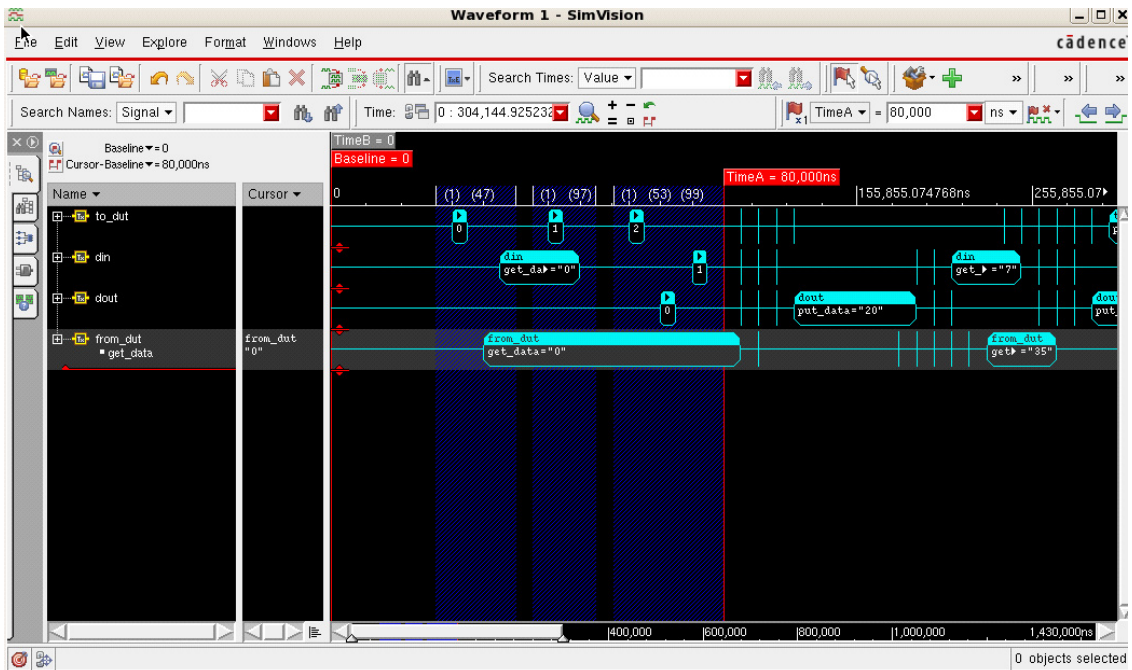
Figure 15-13 Transactions through Flex Channels



Here is a description of the SimVision display:

- Each row shows a sequence of transactions that go through an initiator of the Flex Channel. The name of the initiator is shown in the first column.
- A transaction is defined as a communication of a single data through the initiator.
- The beginning of a transaction is the clock cycle in which the SystemC process calls the **put()** or **get()** function to initiate the data transfer.
- The end of a transaction is the clock cycle in which the called function returns to the calling process. In the shortest case, the Flex Channel interface functions - **put()** or **get()** - **return** at the same clock cycle in which they are called. In this case, the beginning and the end of a transaction are the same. Each vertical line shows a transaction of this case; otherwise, a single transaction could take multiple clock cycles, which are in the figure with the rest of the rectangles.
- If you click the plus sign (+) next to an initiator name in the first column, the individual transactions for that initiator are displayed separately, one for each row.
- If a transaction is shown as a single line - because its begin and end times are the same. In this case, to visualize data values, you can expand the time axis to show the delta cycles executed within each clock cycle, by selecting **View -> Expand Sequence Time -> All Time**, as illustrated in [Figure 15-14 on page 15-65](#).
- By scaling the time axis, you can change the duration of time displayed in the window. The value and data of the transactions, as well as other attributes, are displayed, even for those transactions that happen in a single clock cycle. You can collapse the delta cycles again, by choosing **View -> Collapse Sequence Time -> All Time**.

Figure 15-14 Transactions with Expanded Delta Cycles



15.3 TLM Library

The OSCI TLM 1.0 standard defines a set of interfaces and channels to support TLM. In this section, the following aspects are reviewed:

- “Infrastructure” on page 15-65
- “Considerations for Synthesis” on page 15-68
- “Core Interfaces” on page 15-68
- “Core Channels” on page 15-72

15.3.1 Infrastructure

The TLM modeling framework is based on the following infrastructure classes provided by SystemC:

- **sc_interface**: This class serves as a base class of all interfaces. An interface is defined by a class that derives from **sc_interface** and specifies a set of virtual functions, which any channel that implements this interface must implement.

- **sc_module**: This is the base class for modules. Modules are the principle structural building blocks of SystemC. A module that implements a given interface is specified by a class that derives from **sc_module** and from that given interface, and defines bodies for all methods in the interface.

Note **sc_channel** is a **typedef** for **sc_module**.

- **sc_export**: This class allows a module to explicitly *provide* a given interface. An export forwards each IMC to the module to which it is bound. The export has a template parameter to describe the interface provided by the export. A module with an export can bind the export to itself, or to one of its member variables.
- **sc_port**: This class forwards each IMC to the module to which it is bound. The port takes a template parameter that specifies the interface *required* by this port. An **sc_port** must be bound to an **sc_export** (or directly to a module) that provides the specified interface.

These are the basic TLM constructs supported by CtoS. With those, you can build custom TLM interfaces or use the core TLM interfaces from the TLM 1.0 library.

Figure 15-15 on page 15-67 shows the class diagram for an example with custom interfaces. The class **Filter_Interface** derives from **sc_interface** to establish the TLM interface for a filter: method **analyze**, which takes a matrix as input, is meant to compute a threshold on the matrix, and returns a result as an integer value.

The class **Filter** defines an **sc_export**, parameterized with the **Filter_Interface**, to export its TLM interface through the **sc_export** named **pv_port**. Class **Master** has an **sc_port** named **slave** that is parameterized with the **Filter_Interface**.

Figure 15-16 on page 15-67 shows a structural view of this example. The **sc_port** of the master is bound to the **sc_export** of the filter. The master component has its own thread, and calls the **analyze()** function from its port. Through the **sc_port/sc_export** binding, the call on the filter method on the port of the master will execute the **analyze()** method directly on the filter, and return the result to the master.

Figure 15-15 Simple TLM Master/Slave Design of a Filter

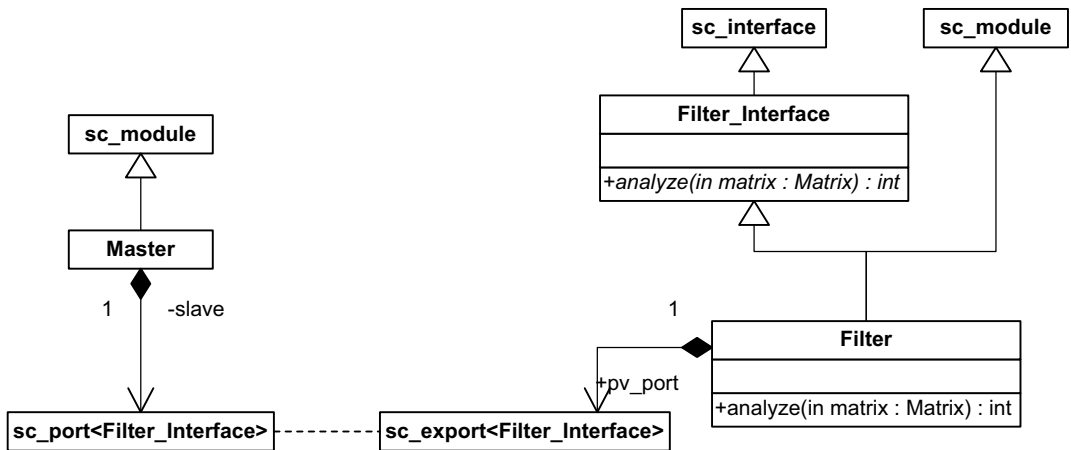
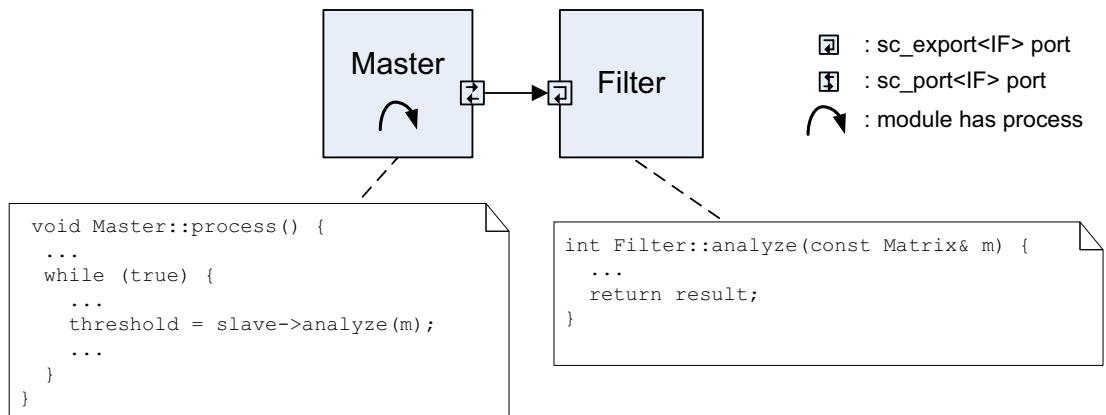


Figure 15-16 Structural Diagram for Simple TLM Filter Example



15.3.2 Considerations for Synthesis

The OSCI TLM 1.0 library was originally developed for simulation; therefore, the OSCI reference implementation is not synthesizable. Furthermore, synthesis requires the explicit modeling of the initializations associated with reset. Thus, to support synthesis, CtoS provides a modified version of the OSCI TLM 1.0 library, which is referred to as the *CtoS TLM Library*.

The main differences with the OSCI TLM 1.0 library are as follows:

- a reset interface has been added to all core interfaces
- all channels are replaced by synthesizable versions of these channels
- all channel configurations are done via template parameters

The CtoS TLM Library can be found in the CtoS installation directory in the following location:

```
install_directory/share/ctos/include/ctos_tlm/
```

Note All modules that are implementations of class **sc_interface** are deresolved during synthesis.

To distinguish them from OSCI TLM, the names of all files in the CtoS TLM Library are prefixed with **ctos_**, and all constructs are defined in the namespace **ctos_tlm::** instead of namespace **tlm::**.

Here are the requirements and limitations for TLM synthesis:

- A channel must derive from class **sc_module** (or **sc_channel**). This is in line with other limitations, such as, the limitation that ports can be instantiated only in classes that derive from **sc_module**.
- Synthesis with CtoS does not support asynchronous communication; therefore, methods that return **sc_event** references are not supported.
- CtoS does not support custom ports (a custom port is a port that derives from **sc_port** or **sc_export**). The recommended coding style is to implement such additional functionality in a module.
- In the TLM 1.0 specification, there is no restriction for the number of processes that can access a given **sc_port** or **sc_export**. As defined in [“Variables” on page 14-37](#), access by multiple processes will require arbitration among processes on the access of the variables of the channel. Therefore, these situations should typically be avoided.

The use of tlm constructs such as **sc_port** and **sc_export** affects the module hierarchy of the synthesized design. For more information, see [“Understanding the Module Hierarchy” on page 16-2](#).

15.3.3 Core Interfaces

The core TLM 1.0 interfaces, and methods defined in them, are described in [Table 15-26 Overview of Synthesizable Core TLM 1.0 Interfaces on page 15-69](#).

Table 15-26 Overview of Synthesizable Core TLM 1.0 Interfaces

Bidirectional blocking interface		
tlm_transport_if<typename REQ, typename RSP>		
RSP transport(const REQ&)	Transports a request, returns a response	
Bidirectional reset interface		
tlm_reset_transport_if		
void reset_transport()	Resets the transport channel or transport slave	Only in CtoS TLM Library
Unidirectional blocking interfaces		
tlm_blocking_put_if<typename T>		
put(const T& value)	Appends a new data item, blocks if the channel is full	
tlm_blocking_get_if<typename T>		
T get(tlm_tag<T> *t=0)	Consuming read, blocks if no data is available	
Unidirectional non-blocking interfaces		
tlm_nonblocking_put_if<typename T>		
bool nb_put(const T& value)	Appends a new data item, returns false if no space is available	
bool nb_can_put(tlm_tag<T> *t=0)	Checks if space is free in the channel	
sc_event& ok_to_put(tlm_tag<T> *t=0)		Not supported by CtoS
tlm_nonblocking_get_if<typename T>		
bool nb_get(T& value)	Gets a new data item, returns false if no data is available	
bool nb_can_get(tlm_tag<T> *t=0)	Checks if data is available in the channel	
sc_event& ok_to_get(tlm_tag<T> *t=0)		Not supported by CtoS

Table 15-26 Overview of Synthesizable Core TLM 1.0 Interfaces

Bidirectional blocking interface		
Unidirectional reset interfaces		
tlm_reset_put_if<typename T>		
void reset_put(tlm_tag<T> *t=0)	Resets the put side of the channel or module	
tlm_reset_get_if<typename T>		
void reset_get(tlm_tag<T> *t=0)	Resets the get side of the channel or module	

The characteristics of the core interfaces are as follows:

- An interface is *bidirectional* if data flows both ways between the master and slave during a transaction; a request from master to slave and a response from slave to master.
- An interface is *unidirectional* if data flows only in one direction.
- An interfaces is *blocking* if the method may call the **wait()** function.
- An interface is *non-blocking* if the method will never call **wait()**.

The transport interface is a bidirectional interface. The master calls the transport function with a request; the slave executes its transport function with that request, and the response is returned to the master.

The transport interface has two template parameters: one for the request data type and one for the response data type.

The slave component must implement the transport method to execute the transaction.

Synthesis methodology recommends that you always reset your output and state variables. A channel will typically have such variables; therefore, it must be reset.

The reset transport method is to be implemented if the slave has state variables that need to be re-initialized in a reset phase. The transport interface is very useful to model bus-based communication and to model bus slaves that execute requests in a blocking fashion.

The unidirectional interfaces define put and get transactions. These interfaces are used to decouple request from response – a request can be sent on a channel and the response received asynchronously on a different channel.

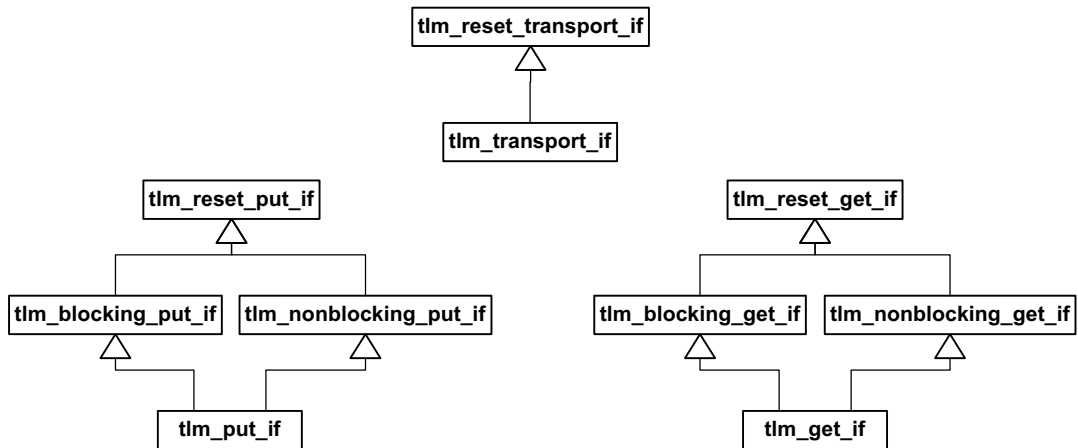
This is very useful for high-performance design, as well as to explore data flow architectures. When a global reset occurs, a put/get channel must reset from both the master and the slave sides of the communication.

Figure 15-17 on page 15-71 shows the inheritance relationships between the core interfaces.

For example, interface **tlm_transport_if** inherits from **tlm_reset_transport_if**.

This means that **tlm_transport_if** has both its own methods, as well as the methods defined in the parent class **tlm_reset_transport_if**.

Figure 15-17 Inheritance Relationship between Core Interfaces



15.3.3.1 Inlining Reset Behaviors

CtoS requires that all method calls in the reset path be inlined. Generally, it is recommended to inline all TLM functions, because these methods are typically small and contain only synchronization code. The following Tcl listing shows how to inline all TLM functions for a given module:

```

set_attr name_use_class_name true [get_design]
build
set top_module [ get_attr top_module [get_design] ]
foreach behavior [find -root $top_module -behavior tlm_blocking_*] {
    inline $behavior
}

```

The first line instructs CtoS to use class names when naming the database objects. The second line is the **build** command. Then, the **top_module** variable is set to the name of the top module in the current design. Finally, the **foreach** loop iterates over all behaviors in the design, matching those whose names begin with **tlm_blocking_**, and inlines all matching behaviors.

15.3.4 Core Channels

The TLM 1.0 library has a set of core channels that can be used in TLM designs, described as follows:

- “TLM FIFOs” on page 15-72
- “Other Channels” on page 15-73

15.3.4.1 TLM FIFOs

The most widely used channel is the FIFO (first-in-first-out) channel. The FIFO channel has a put interface for the sender and a get interface for the receiver. By default, the FIFO channel can store one element. The CtoS TLM Library has four synthesizable versions of the FIFO channel:

Note As you will see below, the behavior of the **1t** (one-time) and **reg** (registered) FIFOs differ from the behavior of FIFOs in the OSCI TLM library.

- **tlm_fifo** uses an internal array to be implemented by a RAM. It allows for multiple puts and multiple gets per clock cycle.
- **tlm_fifo_1t** uses an internal array to be implemented by a RAM. It allows only one get per delta cycle and one put per clock cycle.
- **tlm_fifo_reg** uses an array of **sc_signal** for storing data, which is intended for small FIFOs to be implemented by registers. It allows for multiple puts and multiple gets per clock cycle.
- **tlm_fifo_reg_1t** is similar to **tlm_fifo_reg**, but allows only one get per delta cycle and one put per clock cycle.

Using TLM FIFOs

- Because of access restrictions, **1t** (one-time) FIFOs have fewer states than their counterparts and yield better QoR.
- To reset a FIFO, the producer must call method **reset_put()**, and the consumer must call method **reset_get()**. This will empty the FIFO.
- The CtoS TLM_FIFOs support probing just like the OSCI **tlm_fifo** shipped with INCISIV. This allows the internal state of the FIFO to be displayed in a waveform with INCISIV/Simvision. It also allows transactions on the **tlm** ports connected to the **fifo** to be displayed.

- All four FIFOs implement the **tlm_fifo_status_if** interface, which provides status information about the fifo. The three functions provided reflect that the state of the fifo at the beginning of the current delta cycle. If the process calling the function is clocked, then the status reflects the state of the fifo at the beginning of the clock cycle. The functions are synthesizable and they can be called from the get -process, the put -process, or an unrelated process. The 3 functions provided are as follows:

Signature	Description
bool is_empty() const	Returns true if the fifo is empty at the beginning of the current delta cycle.
bool is_full() const	Returns true if the fifo is full at the beginning of the current delta cycle.
int num_items() const	Returns the number of items in the fifo at the beginning of the current delta cycle.

Other Differences in TLM FIFOs

- TLM FIFOs derive from **sc_module** instead of **sc_prim_channel**.
- Their constructors do not take size as an argument, but have one extra template parameter for specifying size.

Unsupported FIFO Functionalities

- event-returning methods of non-blocking interfaces
- resize interface
- debug interface
- rendezvous protocol (SIZE=0)
- infinite size (SIZE<0)

15.3.4.2 Other Channels

The CtoS TLM Library also includes synthesizable versions of the **tlm_req_rsp_channel** and **tlm_transport_channel** channels. The **tlm_req_rsp_channel** is used to group request and response FIFOs in one channel; the **tlm_transport_channel** is used to convert a bidirectional interface into two unidirectional transactions, one for request and one for response. The use of these channels is much less common than the use of FIFO channels; therefore, for more detail on the internals of these channels, see the OSCI TLM 1.0 documentation and the header files in the directory for the CtoS TLM Library.

15.3.5 Automatic Configuration of Micro-Architecture

After successfully building a design, CtoS will automatically inline all behaviors that originate from the CtoS TLM Library. Inlining these behaviors will usually have a positive impact on QoR, as it will reduce the number of behaviors and enable CtoS to further optimize the logic.

To disable the automatic configuration, add the following Tcl command at the beginning of your configuration script:

```
set ::ctos::enable_config_lib_ctos_tlm 0
```

16 Module Hierarchy

Structural hierarchy is an important feature of SystemC that lets you decompose your design into a set of modules that communicate through ports. CtoS supports this feature by representing the design in the CtoS database also in a form that has structural hierarchy. However, not every SC_MODULE has a one-to-one correspondence to a module in the CtoS database.

In addition, each of these submodules can be synthesized by one or more team members in more than one session of CtoS. The resulting Verilog RTL files can be assembled together for simulation and/or logic synthesis of the overall design. During this integration, you could possibly have name conflicts of module-level names, for example global functions. CtoS supports various naming policies to avoid such name conflicts as described in [“Integrating Multiple CtoS Designs” on page 16-5](#).

This chapter has the following sections:

- [“Understanding the Module Hierarchy” on page 16-2](#)
- [“Commands, Options, Attributes in Hierarchical Mode” on page 16-3](#)
- [“Naming Differences between Hierarchical and Flat Modes” on page 16-4](#)
- [“Integrating Multiple CtoS Designs” on page 16-5](#)

16.1 Understanding the Module Hierarchy

This section describes the elaboration requirements for Hierarchical Synthesis:

- “Preserved and Collapsed `sc_modules`” on page 16-2
- “Factors that Cause an `sc_module` to Be Collapsed” on page 16-2
- “Port Bundles” on page 16-3
- “Uniquified `sc_modules`” on page 16-3

16.1.1 Preserved and Collapsed `sc_modules`

By default, CtoS tries to preserve the structural hierarchy of your design. So, you can expect that for each **`sc_module`** that is instantiated as or somewhere underneath the top-level module of your design, CtoS infers a module that will show up in the design database of CtoS. The **`sc_modules`** are preserved by CtoS. There are instances when CtoS cannot preserve an **`sc_module`**. Suppose that **`sc_module`** M1 instantiates **`sc_module`** M2, and suppose that module M2 cannot be preserved. In this case, the content of M2 is considered to be part of M1. That is, M2 is collapsed into M1, the parent of M2. Consequently, M2 will not show up as a module in the database of CtoS. However, all the processes of M2 will show up as processes of the database module inferred from M1.

The main criterion for deciding whether an **`sc_module`** needs to be collapsed into its parent is—An **`sc_module`** needs to be collapsed into its parent if one of its member functions is called from outside of that module, or if one of its fields that is not an **`sc_in`** or **`sc_out`** port is accessed from outside of that **`sc_module`**. This is done because in the synthesized design, modules can communicate only through signal-level ports.

A problem with this criterion is that it requires a global analysis of the design, and for this reason, CtoS uses a different set of criteria that can be evaluated based on an analysis of the definition of that **`sc_module`** and based on the port connectivity of the whole design.

16.1.2 Factors that Cause an `sc_module` to Be Collapsed

CtoS collapses an **`sc_module`** M1 into its parent module in one of the following conditions:

- M1 has an **`sc_export`** as a field or an object dynamically allocated in its constructor.
- M1 has **`sc_interface`** as a (indirect) base class.
- M1 has an **`sc_port`** as a field or an object dynamically allocated in its constructor, and the **`sc_port`** is connected to an object exterior to M1.
- M1 is a base class of class M2.

- Design attribute **build_flat** is true (by default it is false).

If none of these conditions hold, and there are calls to member functions of M1 from outside of M1, or there are accesses to fields of M1 that are not **sc_in** or **sc_out** ports, the design will be rejected during by the build command. As a workaround you can derive M1 from **sc_interface**, or alternatively set design attribute **build_flat** to true.

You can also specify that an instance of an **sc_module** needs to be preserved using the **ctos keep_instance pragma**. For more information, see [“Keeping Instance Boundary” on page 14-76](#).

16.1.3 Port Bundles

A port bundle is a class or struct that is not an **sc_module** (it does not derive from **sc_module**), but it contains an **sc_in**, **sc_out**, **sc_signal**, **sc_port**, or **sc_export** field. Port bundles are always collapsed and the ports that they contain are considered to be port of the parent module.

16.1.4 Uniquified sc_modules

If there are multiple instances of an **sc_module** in a design and the **sc_module** can be preserved, it is expected that CtoS infers a single module for that **sc_module** in the CtoS database. However, if that **sc_module** has multiple constructors, or if the constructor of the **sc_module** takes arguments other than the name of the module, CtoS may infer several modules from that **sc_module**. Instances of the **sc_module** that are constructed with the same constructor and with the same constructor arguments (but ignoring the module name argument) will be represented by instances in the database that use the same master module.

16.2 Commands, Options, Attributes in Hierarchical Mode

Several CtoS commands, options, and attributes may behave differently when used in Hierarchical Synthesis.

16.2.1 The build Command and build_flat Attribute

The **build** command tells CtoS to preserve the design hierarchy, by default.

If you want to flatten the **SC_MODULE** hierarchy into a single database module, you must set the **build_flat** design attribute during design setup (see [“Building a Design” on page 6-28](#) and [“build” on page E-30](#) for more detail).

Important Versions of CtoS prior to 2.0.0 flatten the **SC_MODULE** hierarchy by default and do not support preservation of the **SC_MODULE** hierarchy.

16.3 Naming Differences between Hierarchical and Flat Modes

There are some naming differences between hierarchical and flat modes for process behaviors:

- In flat mode, the name of a process behavior in the database consists of the name of the corresponding function in the SystemC source, prefixed with the names of all module instances in the instantiation chain, separated with underscores, from the top-module instance to the leaf-module instance.
- In hierarchical mode, the name of a process behavior in the database consists of the name of the corresponding function in the SystemC source, prefixed with just the owner module's name (not the instance name). However, if this owner module is collapsed during elaboration, then the behavior's name will be prefixed with the first non-collapsed module's name, followed by the names of all of the module instances in the collapsed chain.

For example, suppose a design consists of the top instance **top** of a module **M0** with subinstances **m1** of **M1** and **m2** of **M2**. Each of the three has a process **process** and a method **function**.

In this case, the **build** command (with the **build_flat** design attribute having been set during design setup) would produce the following behaviors:

```
top_process
top_m1_process
top_m2_process
function
function_0
function_1
```

For the same design, assuming also that in hierarchical mode, the instance **m1** is preserved, while the instance **m2** gets collapsed into its parent module, the **build** command (without the **build_flat** design attribute having been set during design setup, so the hierarchy is preserved) would produce the following behaviors:

```
M0_process
M1_process
M0_m2_process
function
function_0
function_1
```

Note Naming of *non-process* behaviors is not dependent on flat vs. hierarchical elaboration. The name of a non-process behavior in the database is the name of the corresponding function or method in the SystemC source, possibly suffixed with a numeric index for uniquification.

16.4 Integrating Multiple CtoS Designs

CtoS supports synthesis of a hierarchical design by decomposing them into a set of modules each synthesized in one or more CtoS sessions. The resulting Verilog RTL files can be assembled together for simulation and/or logic synthesis of the overall design. During this integration, you could possibly have name conflicts of module-level names, for example global functions. CtoS supports various naming policies to avoid such name conflicts.

Module name conflicts or duplicate definitions can occur because names are chosen based on the declarations in a design. For example when modules **M1** and **M2** both call a combinational function **myfunc ()**, the Verilog generated in session **M1** will have module **myfunc**, which conflicts with module **myfunc** generated in session **M2**.

The following sections describe ways to avoid these module name conflicts. The approach described in “[Prefix by SC Module](#)” is recommended because it is easier to use and it does not affect names of references to common sub modules.

- “[Prefix by SC Module](#)” on page 16-5
- “[Name Module Prefix](#)” on page 16-6

16.4.1 Prefix by SC Module

The conflict described in the above example and other possible conflicts can be avoided by selecting the **Prefix by SC module** option for the **Verilog Module Naming Policy** section on the **Naming** tab of the **Design Property** dialog (See [Figure 6-22 on page 6-24](#)). This is the same as setting the **name_module_prefix_policy** attribute to **by_sc_module** (See “[name_module_prefix_policy](#)” on page D-19).

This affects names in the resulting RTL in the following ways:

- The names of RTL Verilog modules for SystemC modules have the same name as the SystemC module.
- The names of RTL Verilog modules that are generated by CtoS and that do not correspond to SystemC modules are prefixed with the name of the SystemC module that references these modules. These modules include:
 - Modules for combinational functions, combinational processes and clocked SC_METHOD processes
 - Modules that define memory bridges, memory stall buffers and memory interface muxes, built-in RAM and prototype memories
 - Modules for pipeline functions
 - Modules for CGIC

- Modules for DSP
- The names of RTL Verilog modules provided by user remain unchanged. These modules are:
 - The name entry of **rtl_ip_def** definitions for RTL IP modules in IP definition files
 - The name entry of RAMDef or ROMDef definitions for wrapper modules of RAM or ROM definitions with wrapper filenames

Note The value of the **name_module_prefix** design attribute is not used in this situation where the **name_module_prefix_policy** attribute is set to **by_sc_module**.

Consider the following example:

- A top module **Top** has instances of module **M1** and module **M2**.
- Module **M1** and module **M2** reference global function **func** ().
- Modules **M1** and **M2** are synthesized independently.
- All resulting Verilog files for **Top**, **M1**, and **M2** are used together in simulation or logic synthesis.

This results in multiple definitions of the Verilog module **func** for the global SystemC function **func** if **No Prefix** is specified as naming policy. Using **Prefix by SC module** results in:

- Synthesis of module **M1**: RTL Verilog modules **M1** for SystemC module **M1** and **M1_func** for function **func**
- Synthesis of module **M2**: RTL Verilog modules **M2** for SystemC module **M2** and **M2_func** for function **func**
- Synthesis of module **Top**: RTL Verilog module **Top**

All Verilog RTL files can be used together without redefinition or name conflict.

16.4.2 Name Module Prefix

Name conflicts can also be avoided by selecting the **Prefix by** option for the **Verilog Module Naming Policy** section on the **Naming** tab of the **Design Property** dialog (See [Figure 6-22 on page 6-24](#)). This is the same as setting the **name_module_prefix_policy** attribute to **by_name_module_prefix_attr** and setting the **name_module_prefix** attribute to a non-empty string (See [“name_module_prefix_policy” on page D-19](#)). The value of the prefix should be the name of the module that is being synthesized, followed by an underscore '_'.

This prefix affects the names of the following modules:

- The RTL Verilog modules for SystemC modules
- The RTL Verilog modules that are generated by CtoS and that do not correspond to SystemC modules. These modules include:

- Modules for combinational functions, combinational processes and clocked SC_METHOD processes
- Modules that define memory bridges, memory stall buffers and memory interface muxes, built-in RAM and prototype memories
- Modules for pipeline functions
- Modules for CGIC
- Modules for DSP

The names of RTL Verilog modules provided by user are not prefixed:

- The **name** entry of **rtl_ip_def** definitions for RTL IP modules in IP definition files
- The **name** entry of **RAMDef** or **ROMDef** definitions for wrapper modules of RAM or ROM definitions with wrapper file names

The name of a modules is not prefixed with this prefix if the module starts with the prefix. For example, if you specify prefix **M2_**, module **M2_mod** remains unchanged and is not renamed to **M2_M2_mod**. The name of a module is also not prefixed if the prefix starts with the module name. Module **M2** remains unchanged and is not renamed to **M2_M2** if the prefix is **M2_**.

Consider the same example of the previous section:

- A top module **Top** has instances of module **M1** and module **M2**.
- Module **M1** and module **M2** reference global function **func** ().
- Modules **M1** and **M2** are synthesized independently.
- All resulting Verilog files for **Top**, **M1**, and **M2** are used together in simulation or logic synthesis.

Separate synthesis of **M1** and **M2** results in multiple definitions of the Verilog module **func** for the global SystemC function **func** if **No Prefix** is specified as naming policy. Conflicts can be avoided through the **Name Module Prefix** policy:

- Synthesize **M1** with **name_module_prefix** = **M1_**. The RTL Verilog module for **func** is named **M1_func** and the module for **M1** is named **M1**.
- Synthesize **M2** with **name_module_prefix** = **M2_**. The RTL Verilog module for **func** is named **M2_func** and the module for **M2** is named **M2**.
- Synthesis of module **Top** with an empty **name_module_prefix** design attribute: RTL Verilog module **Top**.

All Verilog RTL files can be used together without redefinition or name conflict.

Note If you have set design attribute **name_use_class_name** to **true** to prepend module member names with the module name, and you then specify **M2_** with **name_module_prefix**, the member function **func** of module **M2** would be named **M2_func**, but not **M2_M2_func**.

17 Incremental Synthesis

The *Incremental Synthesis* feature of CtoS provides an efficient way to synthesize a design to which you have made certain types of changes.

By using this feature, you save time and resources because you are not performing a complete re-synthesis each time you make a relatively small change to a design.

The CtoS *Incremental Synthesis* feature enables you to:

- minimize the difference between the results of synthesis after certain types of changes in your source code.

These results do not include log files, but do include the following:

- textual reports
 - output simulation files
 - output RTL files
 - minimize the difference in generated names (for downstream synthesis scripts and constraints)
 - receive early warnings about possible *non-incrementality* and its causes
 - generate output *change reports* documenting changes with respect to a previous run
- Note** Currently, the *change reports* feature is not available.
- minimize the QoR impact from incremental scheduling and binding decisions
 - reduce CtoS execution times by re-using results from a previous run, to the fullest extent possible

This chapter has the following subsections:

- “Uses for Incremental Synthesis” on page 17-2
- “Incremental Synthesis Design Flow” on page 17-4
- “Sample Incremental Synthesis Scripts” on page 17-8
- “Incremental Synthesis Commands and Options” on page 17-9
- “Changes Handled, and Not Handled, by Incremental Synthesis” on page 17-11

17.1 Uses for Incremental Synthesis

Incremental Synthesis is especially useful in the following situations:

- “Design Space Exploration” on page 17-2
- “Design Flow Bug Fixing (ECO)” on page 17-3

In Design Space Exploration, the primary emphasis is on ensuring that textual similarity is maintained between reports and that the QoR is significantly affected by incrementality.

In Design Flow Bug Fixing (both top-down and bottom-up), in addition to minimizing changes to the reports, the stability (defined as *small input changes produce small output changes*) of the generated RTL is also very important. The stability of names is also extremely important to ensure minimal changes to downstream scripts and constraints.

17.1.1 Design Space Exploration

Design Space Exploration can be a useful practice in the early part of CtoS synthesis – when you want to focus attention on a specific aspect of the source code (for example, loops or sequences of array accesses) that could potentially cause a performance bottleneck or an excessive area cost. The goal of the exploration is to try different micro-architectural choices, or perform different code optimizations, for this specific aspect, while leaving the rest of the input specification unchanged.

In situations such as these, performing a full scheduling and register binding might be excessively time consuming, and analyzing the results might require an excessive effort due to the non-local rippling effects of even localized source or constraint changes.

CtoS Incremental Synthesis is advantageous for such exploration because it preserves the resource information, resource binding, register allocation and register binding decisions performed by CtoS during a previous run, and reuses them *as much as possible* during new scheduling and register allocation runs.

The caveat, *as much as possible*, means some of those previous decisions could lead to illegal schedules and thus could not be copied *as is* to the new design. In such cases, CtoS Incremental Synthesis silently discards (and allows you to browse afterwards, thanks to incrementality reports) those decisions that would lead, for example, to violations of cycle time constraints, latency constraints, resource constraints, etc., in the new design. However, these preserved decisions usually remain valid for areas of the design not affected by the currently explored modifications.

For example, assume that a CTHREAD or THREAD in the source SystemC code consists of a loop reading input data, two loops performing the main computation, and a loop writing output data (this type of behavior is common in bidimensional signal processing applications, such as image compression and decompression).

When focusing on choosing an unrolling level for one of the two main loops, or deciding whether to flatten an array used between the first and the second loop, it is quite advantageous to use Incremental Synthesis in order to substantially reduce synthesis time (re-executing past decisions is much faster than making new ones) and to avoid being confused by excessive differences in output files and reports by keeping the resources and scheduling decisions for the unmodified loops.

17.1.2 Design Flow Bug Fixing (ECO)

Design Flow Bug Fixing, also known as *ECOs* (*engineering change orders*), occurs when a bug is found, either

- in the CtoS input files (“[Top-Down ECO](#)” on page 17-3)
- in a downstream file (“[Bottom-Up ECO](#)” on page 17-4)

The two cases are sufficiently different to deserve separate treatment.

For an example showing how CtoS preserves design information during an ECO flow, look in:

```
install_directory/share/ctos/examples/features/eco
```

Note Before running any CtoS examples, first review “[Setup for Examples](#)” on page F-2.

17.1.2.1 Top-Down ECO

In top-down ECO, a bug is found, for example, when new test cases become available from a customer after a significant amount of downstream work has already been performed, such as in logic synthesis. In this case, a fix to the bug might involve only a very localized change in the source (for example, using a + sign instead of a - sign, or using saturation arithmetic instead of overflow for one specific operator).

It is thus very beneficial to significantly limit changes to the generated RTL, to minimize the effort required to understand the new output, and to minimize the effect on the generated logic, assuming RTL similarity implies similarity after RTL and logic synthesis (logic synthesis also supports top-down ECO). Encounter Conformal ECO is an example of a mechanism that ensures such similarity after RTL.

17.1.2.2 Bottom-Up ECO

In bottom-up ECO, a bug is found and fixed in a low-level model, for example, at the gate level. In this case, making a change to the source that results in an equivalent gate-level netlist would be most efficient, allowing you to re-run extensive system tests using faster high-level models, rather than gate-level ones.

Also in this case, the similarity of the generated RTL before and after the ECO (especially in terms of register allocation, since bottom-up ECO relies heavily on equivalence checking to prove that the low-level and the high-level model changes preserve equality of behavior) helps reduce both equivalence checking tool run time and designer efforts.

17.2 Incremental Synthesis Design Flow

When using the Incremental Synthesis feature, you always have two designs open:

- your previous version, which is referred to as your *baseline design*
- your *current design*, which you are in the process of synthesizing and analyzing

These two designs can be the result of command executions in a current session (two **builds** (“[build](#)” on [page E-30](#)), two micro-architectural selections, a **schedule** (“[schedule](#)” on [page E-138](#)) and an **allocate_registers** (“[allocate_registers](#)” on [page E-18](#)) for the *baseline design* only), or the *baseline design* can be the result of opening an already scheduled and register-allocated design that has been saved in a previous CtoS session, the latter being the more likely scenario.

The goal of Incremental Synthesis is to make sure the *current design* has scheduling and register allocation decisions as similar as possible to those made earlier for the *baseline design*.

Important It is up to you, the designer, to assure that micro-architectural decisions are as similar as possible, in order for Incremental Synthesis to be considered truly *incremental*. Match percentage is reported at each step in the incremental matching flow. You can set a *required match percentage* using the **set_attr** command (“[set_attr](#)” on [page E-140](#)) on the **required_match_percentage** design attribute (“[required_match_percentage](#)” on [page D-22](#)). CtoS will issue an error if this percentage is not met.

Incremental Synthesis essentially tries to preserve as many operation/resource and value/register bindings as possible, while keeping the order of text lines in output files (reports, simulation, and RTL files) as similar as possible, in order to make textual comparison as easy as possible.

Names from the *baseline design* are also preserved as much as possible, again in order to maximize textual similarity between the output files and to maximize the re-use of scripts (both for CtoS and for downstream tools) initially written for the *baseline design*. Be aware, however, that CtoS is able to copy information *only* up to the point where the design was last saved.

There are two flows for incremental synthesis:

- “[Automated Incremental Synthesis Flow](#)” on [page 17-5](#)
- “[Manual Incremental Synthesis Flow](#)” on [page 17-6](#)

Important It is highly recommended that you use the *automated flow*, since it automatically runs *save*’s and *copy*’s at appropriate stages, unless something in your particular situation makes this flow unusable.

17.2.1 Automated Incremental Synthesis Flow

In *automated* mode, you must define only the names of the directories in which the baseline design is to be saved, and from which it is opened, while working on the new design.

Here are the exact steps you must follow in automated mode:

1. To use a design as a *baseline* for an Incremental Synthesis run, *before running the **build** command* (“[build](#)” on page E-30), you must set the **auto_save_dir** design attribute (“[auto_save_dir](#)” on page D-11) to the directory in which CtoS will save this baseline design at two places during the flow:
 - after the **build** command (in a subdirectory **elaborated**).
 - after registers have been allocated for every behavior in the design (in a subdirectory **rtl_ready**).
2. To synthesize a design in Incremental Synthesis mode, *before running the **build** command*, you must set the **baseline_dir** design attribute (“[baseline_dir](#)” on page D-12) to the directory in which the design was saved during the baseline run.

CtoS will open the elaborated baseline design immediately *after* the **build** command, renaming the new design so names of CtoS objects (nodes, edges, ops, etc.) match the baseline names as much as possible.

CtoS will also open the **rtl_ready** baseline design just *before* the **schedule** command (“[schedule](#)” on page E-138) simply to copy states, resources, and bindings to minimize report and RTL file differences (however, there is no matching or renaming at this point).

For example, in the baseline design, the script initial fragment, immediately before the **build** command:

```
set_attr source_files    xbus_hw_idct.cc [get_design]
build
```

should become:

```
set_attr source_files    xbus_hw_idct.cc [get_design]
set_attr auto_save_dir   /home/mydir/designs/ctos_demo [get_design]
build
```

In the new design, the same script fragment should become:

```
set_attr source_files    xbus_hw_idct.cc [get_design]
set_attr baseline_dir    /home/mydir/designs/ctos_demo [get_design]
build
```

To also use this new design as a new baseline, both attributes can be specified (*with a different directory name*; otherwise the baseline design will be overwritten):

```
set_attr source_files    xbus_hw_idct.cc [get_design]
set_attr baseline_dir    /home/mydir/designs/ctos_demo [get_design]
set_attr auto_save_dir   /home/mydir/designs/ctos_demo_v2 [get_design]
build
```

17.2.2 Manual Incremental Synthesis Flow

In *manual* mode, you must explicitly save your baseline design and copy its information to the new design, using the **set_baseline** command ([“set_baseline Command” on page 17-10](#))

For the *baseline design*:

1. Start, using the **new_design** command ([“new_design” on page E-88](#)), or **open_design** command ([“open_design” on page E-89](#)) for an existing design (which could be, itself, the result of another Incremental Synthesis), and **build** ([“build” on page E-30](#)) an initial version of your design. This is referred to as the *baseline design* from which all changes will be determined.

Note See [“Starting, Setting Up, and Building a Design” on page 6-1](#).

2. Optionally, at this point, you can use the **save_design** command ([“save_design” on page E-137](#)) to save the *baseline design*, using the directory (**-dir**) option to specify a directory name that will indicate that the design was saved after elaboration, but before synthesis, for example, *BuildDirectory*.

```
save_design -dir BuildDirectory
```

Saving the design at this point, and consequently using this version with the **set_baseline** command (in [Step 7](#), below) minimizes changes to scripts that use names to identify nodes and ops, such as loops, arrays, etc. However, you are required to save the design only after register allocation, so you do not have to save it here.

3. Perform the necessary micro-architecture commands (see [“Specifying Micro-Architecture” on page 8-1](#)). Schedule and allocate registers for the *baseline design* (see [“Scheduling and Managing Registers” on page -1](#)).
4. Use the **save_design** command ([“save_design” on page E-137](#)) to save the *baseline design* (this save is mandatory).

If you optionally saved the *baseline design* in [Step 2](#) (after build), be sure to use the directory (**-dir**) option to specify a directory name that will indicate that the design is now being saved after register allocation, for example, *ScheduleDirectory*.

```
save_design -dir ScheduleDirectory
```

5. Use the **close_design** command ([“close_design” on page E-33](#)) to close the *baseline design*. You have now established the baseline for future changes to your design.

When you have made changes to the initial version of your design (previously saved in CtoS as the *baseline design*), and these changes seem to fall within the CtoS guidelines for Incremental Synthesis (found in [“Changes Handled, and Not Handled, by Incremental Synthesis” on page 17-11](#)), you are then ready to continue with the design flow.

6. Start, using the **new_design** command ("[new_design](#)" on page E-88), the modified design, which is referred to in CtoS as your *current design*.
7. If you performed [Step 2](#), that is, if you saved the *baseline design* after build (if you did *not* perform [Step 2](#), skip to [Step 8](#)), open that version of it now using the **open_design** command ("[open_design](#)" on page E-89) using the **-not_current** option to make sure it does not become the current design and the **-rename** option to give it a unique name. Also, remember to use the *BuildDirectory* name, or the name you gave the directory in which you saved it.

```
open_design -not_current -rename unique_name BuildDirectory
```

Run the **set_baseline** command ("[set_baseline Command](#)" on page 17-10) using the name to which you renamed the design in the previous step. At this point, the **set_baseline** command checks if the current design already has a baseline and performs renaming only if there is no baseline:

```
set_baseline unique_name
```

8. Perform the necessary micro-architecture commands for the *current design* (see "[Specifying Micro-Architecture](#)" on page 8-1).
9. Open using the **open_design** command ("[open_design](#)" on page E-89) the version of the baseline design that you saved in [Step 4](#), using the **-not_current** option to make sure it does not become the current design and the **-rename** option to give it a unique name. Also, remember to use the *ScheduleDirectory* name, or the name you gave the directory in which you saved it.

```
open_design -not_current -rename unique_name ScheduleDirectory
```

Run the **set_baseline** command, using the ID of the *baseline design* saved after register allocation.

```
set_baseline ID_from_ScheduleDirectory
```

10. Schedule and allocate registers for the *current design* (see "[Scheduling and Managing Registers](#)" on page -1).

Here are the resulting actions taken by the CtoS Incremental Synthesis feature to implement these steps:

1. *Just after elaboration*, when you run the (optional) **set_baseline** command, CtoS opens the post-elaboration *baseline design* and performs a matching with the *current design*. It then uses the matching information to rename database nodes and ops using the old design names and appending the (incremented) *ECO version number* to the names of the newly added nodes and ops.

In this manner, scripts can be re-used between the old and new design, and report and RTL file comparison is made easier.

2. *Just before scheduling* (that is, after micro-architectural selections, in particular optimization, pipelining and array constraining), when you run the (mandatory) **set_baseline** command, CtoS opens the post-RTL *baseline design* to perform a matching with the *current design* and to rename its nodes and ops as aforementioned.
3. *During the following processes*, CtoS uses the restored information to set *soft constraints* for:
 - scheduling (performing both resource allocation and op binding)
 - register allocation (performing both register allocation and value binding)
 - RTL generation
4. *After design matching*, CtoS reports the percentage of matched nodes and ops (a first indication of possible non-incrementality), and *after scheduling and register allocation*, the percentage of successful bindings.
5. *At the end of the Incremental Synthesis run*, you can run the **report_incremental** command (“[report_incremental](#)” on page E-107) to get a detailed report of which bindings could and could not be copied to the new design, and why.

17.3 Sample Incremental Synthesis Scripts

Here are examples of scripts that could be used for Incremental Synthesis:

- “[Sample Baseline Creation Script](#)” on page 17-8
- “[Sample Incremental Synthesis Script](#)” on page 17-9

17.3.1 Sample Baseline Creation Script

Here is a sample baseline creation script:

```
new_design version0
... [set attributes for the baseline design]
build
save_design v0_build
source microarchitecture.tcl
schedule ...
allocate_registers
save_design v0_sched
write_rtl ...
```

17.3.2 Sample Incremental Synthesis Script

Here is a sample Incremental Synthesis script using the “[Sample Baseline Creation Script](#)” on page 17-8:

```
new_design version1
... [set attributes for the modified (current) design]
build
open_design -not_current -rename v0_build v0_build
                                # the -not_current option is used to
                                # read a baseline design, preventing
                                # it from becoming the current design

set_baseline /designs/v0_build
save_design v1_build             # only after set_baseline
source microarchitecture.tcl     # same script as before
open_design -not_current -rename v0_sched v0_sched
set_baseline /designs/v0_sched
schedule ...
allocate_registers
save_design v1_sched
write_rtl ...
close_design
```

Tip You may want to give CtoS a longer clock cycle constraint than that used in the baseline, in order to prevent negative slack and the ensuing rippling effects if the changed source code was part of a path that CtoS considered critical. Often, the cycle time is not affected by these minor changes after logic synthesis.

In general, a negative slack may force the CtoS scheduler to work harder, and thus drop several copied op bindings, before it realizes that negative slack is unavoidable and those copied bindings could have been preserved.

17.4 Incremental Synthesis Commands and Options

The following commands and command options have been developed for the Incremental Synthesis feature:

- “[set_baseline Command](#)” on page 17-10
- “[unbind_baseline Command](#)” on page 17-11
- “[report_incremental Command](#)” on page 17-11

17.4.1 set_baseline Command

The preservation of names and of resource and value bindings is performed by the **set_baseline** command ([“set_baseline” on page E-141](#)) which lies at the heart of Incremental Synthesis.

This command performs three main actions:

1. It compares the *baseline design* and *current design* to identify portions that were not changed.

Note This process is heuristic in nature and is similar to the UNIX *diff* command; thus it has no guarantee of identifying the same similarities as a human designer. However, if *diff* finds few differences, this matching is very likely to be highly successful.

2. For all nodes and ops of the *current design* for which it could find a correspondence to the *baseline design*, it renames those nodes and ops to be *identical* to the corresponding names in the *baseline design*. Conversely, the names of any nodes and ops of the *current design* that could not be matched are appended with **_v** and the **eco_version** (*ECO version number*) to clearly identify them.

Note The **eco_version** is an integer attribute of a design, which is automatically initialized to zero when a design is created, and is set to one higher than the corresponding attribute of the most recently used baseline in a **set_baseline** command¹. Thus, nodes and ops of the *current design* used in scripts remain identical. This is helpful in reusing scripts that were initially defined for the *baseline design*.

3. It flags, in the baseline design (which must remain open in its scheduled version throughout the scheduling, register allocation and RTL generation steps of the new design), the following:
 - set of resources
 - created states
 - operation bindings
 - value bindings
 - RTL instance order

Therefore, the **create_initial_resources** command ([“create_initial_resources” on page E-42](#)), **schedule** command ([“schedule” on page E-138](#)), **allocate_registers** command ([“allocate_registers” on page E-18](#)), and **write_rtl** command ([“write_rtl” on page E-161](#)) can reuse them as much as possible later on.

1. Manipulation of this attribute via **set_attr**, although possible, is not recommended. The ECO version number of zero is not appended, so **_v** and **eco_version** are appended only after matching and renaming occurs.

Note the following about the **set_baseline** command:

- Since its performance is based on the current form of the control flow graph, for maximum effectiveness, you should run the **set_baseline** command just after the initial **build**, to enable the reuse of scripts for micro-architectural selection.
- Because it sets baseline information for the current design, if you close the current design and open a new one, the baseline information for the new design must be set again. Also, if the baseline design is closed, the **schedule**, **allocate_registers** and **write_rtl** commands behave non-incrementally.

17.4.2 unbind_baseline Command

You can use the **unbind_baseline** command ("["](#) on page E-148) any number of times, *after* you have used the **set_baseline** command and *before* you have used the **schedule** command, to prevent copying of resource and register bindings for the specified region in the source.

This can be useful if you want the CtoS scheduler to perform its job completely anew in a specific section of the code that you have modified (or constrained differently), while copying the bindings for the rest.

17.4.3 report_incremental Command

You can use the **report_incremental** command ("[report_incremental](#)" on page E-107) to see how much information was copied successfully from the baseline design.

17.5 Changes Handled, and Not Handled, by Incremental Synthesis

CtoS Incremental Synthesis is appropriate for only certain types of changes. This section defines:

- "[Changes Handled by Incremental Synthesis](#)" on page 17-12
- "[Changes Not Handled by Incremental Synthesis](#)" on page 17-12

17.5.1 Changes Handled by Incremental Synthesis

Incremental Synthesis correctly handles (in decreasing order of expected stability) these changes:

- changes of names of variables that do not result in module nets, module I/O, or memories
- changes of a small percentage of the operator types in the source (for example, + to -)
- changes of the bit width of a few or of several variables in the source code
- changes in the control flow (for example, insertion of an **else** branch or of a **break** or **continue**)
- code motions (for example, moving an assignment outside or inside a loop).
- changes in latency constraints, unrolling or inlining that affects a small number of operations. In the case of latency or unrolling changes for a loop, then only that loop must be affected.

17.5.2 Changes Not Handled by Incremental Synthesis

Incremental Synthesis is not designed to handle (and may fail to preserve similarity in) these changes:

- *Library changes*. If a changed library leads to substantially different area or timing results, the incremental flow may produce radically different or grossly suboptimal results. However, if the library is strictly *better* (for example, a technology generation change), you will get about the same results with the incremental flow as you would with a remapping of the generated RTL.
- *Global constraint changes*, such as a reduction of the clock cycle. Incrementality in this case would be preserved only if the change in clock cycle affects, for example, only a minority of the states.
- Small changes in functionality that derive from *major rewritings of the source code*.
- Changes of the partition into *non-inlined functions*.
- Changes in some *optimization directives*, with a global effect, such as:
 - loop pipelining and loop unrolling
 - function inlining
 - array constraining or flattening
- Changes to the *names* of:
 - **SC_THREADS**, **SC_CTHREADs** and **SC_METHODs** and other *non-inlined methods*. In this case, incrementality is preserved only if the changes affect a minor part of the specification. A specification in which a single loop does most of the computation, and is unrolled a different number of times, or it is split, will not be treated as incremental.
 - *externally visible variables*, that is **sc_module** members that become nets in the synthesized module (I/Os and **sc_signals** shared among threads) and arrays that become memories (non-flattened arrays). Normally, an ECO would not require changing a variable name. Variables eliminated in the data flow (all function locals, except for unflattened arrays) *are handled incrementally*, because they completely disappear by the time scheduling occurs.

18 Low Power Estimation Using CtoS

Power Estimation is a preliminary feature.

CtoS provides a way to estimate the power consumed by a design and to analyze, for example, the correlation between the power consumed by hardware resources and the source code they implement.

This capability depends on activity estimation for signals, which can be derived from:

- simulation and analysis tools (for example, the INCISIV tool, **ncsim**) generating *TCF (toggle count format)* files,
- propagation of activities through combinational logic, or
- default activity levels.

This chapter has the following subsections:

- [“RTL Power Estimation Flow” on page 18-1](#)
- [“Commands/Options/Attributes for Power Estimation” on page 18-5](#)
- [“Considerations when Using INCISIV dumptcf” on page 18-9](#)
- [“Using Clock Gating” on page 18-10](#)

Note For power reporting in CtoS, see [“Reporting Power and Area Using the Tree Map” on page 13-29](#)

18.1 RTL Power Estimation Flow

As is well known, *active power* in a CMOS circuit depends on switching activity, while *leakage power* depends on the logic level of the signal.

Tools can be used to record both kinds information (which will collectively be referred to here as *activity*) – for example using files in the TCF format for a design under a set of typical operating conditions represented as a testbench. If there is no realistic testbench, default activity values can be used for a preliminary analysis of trade-offs.

If the TCF file provides activity information for only a subset of the signals, or if it is not available, CtoS can use the switching and level probability propagation capabilities of the RTL Compiler (RC) to propagate the information from a subset of the signals, with expectably reduced precision.

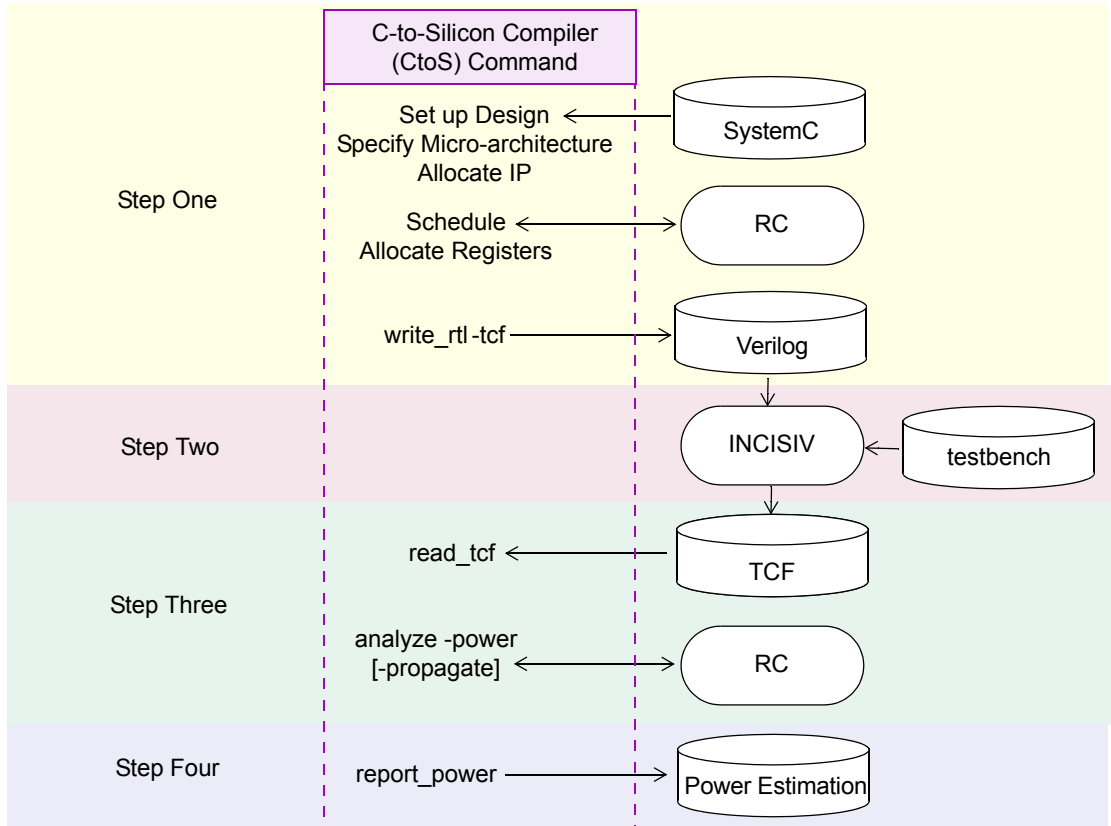
RTL Power Estimation uses RC and library power models, similar to the way timing and area estimation are done; however, since CtoS uses pre-synthesized implementations of the resources, it is also considerably faster than power analysis in RC and provides powerful back-annotation and cross-referencing with source code and the CDFG.

Figure 18-1 on page 18-3 presents a graphical depiction of the RTL Power Estimation flow.

Note Although RTL Power Estimation in CtoS uses RC, and is thus expected to have similar precision levels, CtoS ignores *global optimizations* (especially with respect to area/timing trade-offs), which RC is able to perform.

Hence, the estimation error can be significant. The expected estimation error can be considered comparable to the error that RC makes when performing RTL estimation, that is, before a detailed gate-level netlist is synthesized.

Figure 18-1 RTL Power Estimation Flow



Here is the step-by-step RTL Power Estimation flow:

1. Set up a design, specify the micro-architecture, allocate IP, schedule and allocate registers, that is, take the design to the Analyze Micro-architecture step (see [“Setting Default Toggle Probability” on page 18-5](#)).

Now, run the **write_rtl** command with the **-tcf** option (see [“Using the -tcf Option of write_rtl and write_sim Commands” on page 18-6](#)).

Important Steps 2 through 4 should be invoked on the same design on which the simulation model was generated; therefore, do not make any changes to your design in between these steps.

2. Run INCISIV (specifically, run **dumptcf** command from **ncsim**) to save a record of the toggling activity and of the probability of being at logic level 1 for each net.

Note Consult the Cadence INCISIV documentation for more on **dumptcf** and see [“Considerations when Using INCISIV dumptcf” on page 18-9](#).

3. Run RC power analysis, using CtoS [that is, run the **read_tcf** command (see [“Using the read_tcf Command” on page 18-7](#)) and run the **analyze** command with the **-power** option (see [“Using the analyze Command” on page 18-6](#))], to request that RC estimate the power, given the switching activity at its inputs.

If you skipped step 2, defaults set with Design Properties are used.

Notes

- The **analyze** command with the **-power** option will use the last set values of the [“Setting Default Toggle Probability” on page 18-5](#).
 - The **-propagate** option of the **analyze** command ([“Using the analyze Command” on page 18-6](#)) may be useful at this stage if either (1) the percentage of asserted probabilities for nets, as reported by the **read_tcf** command ([“Using the read_tcf Command” on page 18-7](#)), is low, or (2) the percentage of nets with default probabilities, as reported by the **analyze** command, is high.
4. Run CtoS reports [that is, run the **report_power** command (see [“Using the report_power Command” on page 18-7](#))] to browse the power of various resources and to explore the impact of various design decisions, similar to using the **report_area** command ([“report_area” on page E-103](#)) and **report_timing** command ([“report_timing” on page E-130](#)).

Here is a typical sequence of commands that would comprise an RTL Power Estimation flow:

```
# ctos
% new_design
% ... (other ctos commands)
% schedule
% allocate_registers
% write_rtl -tcf -o mymodule.v /designs/top/modules/mymod
% save_design -dir rtl_ready /designs/top
% exit
# ncsim ... (other ncsim arguments) mymodule.v +access+r -TCL
ncsim> dumptcf -scope sc_main.wrapper.m_dut_vlog -internal -output file.tcf
ncsim> run
ncsim> exit
# ctos
% open_design rtl_ready
% read_tcf file.tcf /designs/top/modules/mymod
Asserted 120 nets (15 with 0 toggling) out of 134 in module and 215 in TCF file.
% analyze -propagate
% report_power
```

18.2 Commands/Options/Attributes for Power Estimation

The following commands and options have been developed for both of the Power Estimation flows:

- “Setting Default Toggle Probability” on page 18-5
- “Using the analyze Command” on page 18-6
- “Using the analyze Command” on page 18-6
- “Using the -tcf Option of write_rtl and write_sim Commands” on page 18-6
- “Using the report_power Command” on page 18-7
- “Using the read_tcf Command” on page 18-7

18.2.1 Setting Default Toggle Probability

Two design attributes, **default_toggle_probability** and **default_value_1_probability**, are used in power estimation:

- **default_toggle_probability** (“[default_toggle_probability](#)” on page D-14) is the *default toggling probability* for nets and values, that is, the probability of a net or value to change its value between one clock cycle and the next (or within a clock cycle, if the TCF were generated from a back-annotated gate-level simulation including glitching effects).

Its value type is double, and values can range from 0.0 to 1.0 (it can be more than 1.0 if glitches are considered), with a default of 0.02.

Note Be aware that **ncsim** generates two transitions even for a glitch within a delta cycle.

- **default_value_1_probability** (“[default_value_1_probability](#)” on page D-15) is the *default value 1 probability* for nets and values, that is, the probability of a net or value to stabilize to logic value 1 at the end of a clock cycle.

Its value type is double, and values can range from 0.0 to 1.0 (it can be more than 1.0 if glitches are considered), with a default of 0.5.

18.2.2 Using the analyze Command

The **analyze** command (“[analyze](#)” on page E-20), with the **-power** option, chooses the best speed grade for every resource (as implemented by the **analyze** command with the **-area** option), in order to improve the accuracy of Power Estimation, by ensuring that all resources are properly sized according to their timing criticality.

This is a potentially expensive operation, since it may require synthesizing using several speed grades for each resource, unless they are already in the RC cache.

The **-propagate** option of the **analyze** command propagates toggle and value probabilities across instances. It can be used whenever the coverage of the TCF file is low (as reported at the end of the **analyze -power** command), since it can propagate toggling probabilities across combinational logic, in order to set those (and only those) that were not derived from the TCF file.

18.2.3 Using the -tcf Option of write_rtl and write_sim Commands

The INCISIV tool, **ncsim**, is currently limited to collecting activity for only *one* Verilog scope (module or **always** block) at a time, using the **dumptcf** command from the simulation user interface.

To deal with this limitation, the **-tcf** option of the **write_rtl** command (“[write_rtl](#)” on page E-161) and the **write_sim** command (“[write_sim](#)” on page E-165) collects *all* signal declarations at the module level, instead of keeping them inside the **always** block where they are used.

18.2.4 Using the report_power Command

The **report_power** command (“[report_power](#)” on page E-116) reports the estimated power consumption of all resources in the specified design, module, or behavior – automatically calling power analysis [that is, the **analyze** command (“[analyze](#)” on page E-20) with the **-power** option], if needed.

18.2.5 Using the read_tcf Command

The **read_tcf** command (“[read_tcf](#)” on page E-98) reads in switching activity described in Toggle Count Format (TCF). It has the following options to further refine this process:

- “[-weight option of read_tcf Command](#)” on page 18-7
- “[-accumulate Option of read_tcf Command](#)” on page 18-8
- “[-scale Option of read_tcf Command](#)” on page 18-8

Note For complex hierarchical designs, you may need to perform multiple simulation runs, collecting activity for each module, and then using multiple **read_tcf** commands to annotate the CtoS database.

18.2.5.1 -weight option of read_tcf Command

Multiple simulation runs are frequently executed with the same scope, to model different scenarios. In these cases, the **read_tcf** command provides a way to properly accumulate transition and level probabilities.

Executing the **read_tcf** command, with the **-accumulate** and/or **-weight** options (the default of the latter is 1.0), multiple times for the same module (or for one of its sub-modules) accumulates, with different user-specifiable relative weights, the activities of nets that have been already stored.

For example:

```
read_tcf -weight 2.0 f1.tcf /designs/top/modules/mymod
read_tcf -accumulate -weight 0.5 f2.tcf /designs/top/modules/mymod
read_tcf -accumulate -weight 0.5 f3.tcf /designs/top/modules/mymod
read_tcf -accumulate f4.tcf /designs/top/modules/mymod
```

The formula for accumulation at step *n* is:

$$\text{total\_prob} = (\text{total\_prob} * (n - 1) + \text{new\_prob} * \text{weight}) / n$$

The argument to the **-weight** option is a floating point number, which can be used to provide relative importance to the various scenarios, and which is used to multiply both transition and level probabilities read from the TCF file.

The overall sum of the **-weight** values must be the same as the total number of TCF files that are read (four in the previous example); otherwise, Power Estimation will be misreported.

In the previous example, the scenario simulated by **f1.tcf** is four times more likely to occur than the scenarios simulated by **f2.tcf** and **f3.tcf**, and two times more likely to occur than the scenario specified by **f4.tcf**.

18.2.5.2 -accumulate Option of `read_tcf` Command

The **-accumulate** option of the `read_tcf` command is used to accumulate activities, rather than to overwrite them.

If this option is specified, probabilities are accumulated with a weight incremented every time a new probability is read for a net (that is, past and current values are weighed evenly), as described in the previous section, [“-weight option of read_tcf Command” on page 18-7](#).

18.2.5.3 -scale Option of `read_tcf` Command

The argument of the **-scale** option of the `read_tcf` command is a positive floating point number that *multiplies* the clock frequency of the module by the specified scaling factor without re-running the simulation.

For example, the following command specifies that power should be estimated with a frequency that is *twice* the frequency used for the original simulation that generated the TCF file.

```
read_tcf -scale 2 file.tcf /designs/top/modules/mymod
```

18.3 Considerations when Using INCISIV dumpptcf

Here are a few considerations when using the INCISIV **dumpptcf** command:

- “Scope Used in dumpptcf Command” on page 18-9
- “+access+r and -afile Options of dumpptcf” on page 18-9
- “Module Name Must Correspond to dumpptcf Scope” on page 18-9

18.3.1 Scope Used in dumpptcf Command

The scope to be used in the **dumpptcf** command depends on design hierarchy and can also be identified using the SimVision simulator GUI.

Tcl input can also be provided in a file, with the **-INPUT** option.

18.3.2 +access+r and -afile Options of dumpptcf

The **+access+r** option, which may have a negative impact on simulation performance, is required for proper execution of the **dumpptcf** command in the simulator (otherwise, many activities will be incorrectly reported as zero).

The **-afile** option can be passed to the simulator in order to refine the sub-design; this option must be applied to properly generate the TCF file with minimum overhead.

Note Consult the Cadence INCISIV documentation for more information.

18.3.3 Module Name Must Correspond to dumpptcf Scope

The module name passed as an argument to the **read_tcf** command must correspond exactly to the scope of the **dumpptcf** command.

Due to the way SystemC and RTL modules are instantiated in the simulation hierarchy, it is impossible for CtoS to automatically verify the correspondence

Therefore, you must carefully check the reported number of nets for which switching activities and level activities were read, in order to identify possible errors.

A low percentage of asserted nets reported by the **read_tcf** command is an indication that either:

- the wrong module path has been used, or
- the **-tcf** option of the **write_sim** or **write_rtl** command has not been used.

18.4 Using Clock Gating

The following section describes about how to use clock gating:

- “Fine Grain Clock Gating” on page 18-10
- “Coarse Grain Clock Gating” on page 18-11

18.4.1 Fine Grain Clock Gating

If your technology and design flow meet the criteria described in “Clock Gating Requirements for ASIC Designs” on page 18-10 and “Clock Gating Requirements for FPGA Designs” on page 18-11, you can direct CtoS to optimize its output to maximize potential clock gating usage. This feature is controlled by the `low_power_clock_gating` design attribute (“`low_power_clock_gating`” on page D-17).

By setting this attribute to `true` before scheduling, CtoS will choose signals to use as clock enables. CtoS is able to significantly exploit clock gating, because CtoS has a precise notion of the conditions under which registers in a design are used. CtoS uses this knowledge to select ideal nets in a design as enables for clocking, which maximally disables each register according to its particular dynamic usage.

In addition, CtoS groups registers together according to similarity of inputs, enables, and resets, ensuring that a single clock enable can be used for the largest number of registers with common enables. As an output to the synthesis step, CtoS generates conditional update statements in the RTL, which other tools can then use to infer clock enables.

Note If the `low_power_clock_gating` design attribute is set to `true`, when you write an RC script, the following line will be added to your script (see “Understanding RC Run Scripts” on page 13-51):

```
set_attr lp_insert_clock_gating true
```

18.4.1.1 Clock Gating Requirements for ASIC Designs

To use clock gating in an ASIC design, you must make sure your post-RTL design flow supports the use of gated clocks. This typically requires the following features:

- A synthesis tool with clock gating insertion support. This feature is well-supported in RC.
- A technology library with clock gating-specific cells. The vendor for your tech library, along with your foundry, affects this requirement.
- A place-and-route tool with support for gated clocking. The Cadence *SoC Encounter RTL-to-GDSII System* supports this feature.

18.4.1.2 Clock Gating Requirements for FPGA Designs

FPGAs do not directly support the concept of clock gating. However, all modern FPGAs have a similar capability – through the use of an Enable pin on the internal registers. FPGA synthesis tools can map the enables that CtoS generates to drive the Enable pin on the internal registers, thereby reducing some switching frequency in a design. The effectiveness of this strategy strongly depends on the vendor’s FPGA technology, so the choice of whether to use the CtoS clock gating feature with FPGA designs is left to the designer.

18.4.2 Coarse Grain Clock Gating

Coarse grain clock gating enables a user to model gated clocks in the SystemC source of your design and implement them with CGIC (clock gate integrated cell) technology library cells in the synthesized design. This allows you to specify specific regions of their design, at the process level, whose clocks can be gated for power reduction. This feature helps you implement your clock gating strategy as a synthesizable flow. Using coarse grain clock gating in your design involves the following elements:

- Modeling coarse grain clock gating in the input SystemC source. See “[Specifying Clock Gate Integrated Cells \(CGIC\)](#)” on page 14-102. Once a design has been built, the `ctos_clock_gate_module` and all of its instances will show up in the hierarchy browser.
- Writing an XML file that describes the CGIC cell. See “[cgic_ip_def XML Elements](#)” on page H-6
- Using the `read_ip_defs` command to load a `cgic_ip_def` from an XML file. See “[read_ip_defs](#)” on page E-97.
- Using the `use_ip` command to select `cgic_ip_def` to implement the `ctos_clock_gate_module` instances in the synthesized design.

The simulation models generated by the `write_sim` command use a functional model for the clock gating cells. The RTL generated by the `write_rtl` command can be configured to use the same functional model for clock gating cells by defining the `CTOS_SIM_CGIC` verilog macro. Otherwise, the technology-dependent model of the CGIC cell that you provide is used.

Restrictions:

The following are the restrictions on coarse grain clock gating:

- A process cannot use the negative edge of a gated clock as its clock event. If the clock is gated, then only the posedge can be used as clock event.
- A process that accesses an external array cannot use a gated clock.
- A memory that is exported cannot be accessed by a process that uses a gated clock.
- Gated clocks are not supported for FPGA designs.
- CGIC cells with reset are not supported.

A CtoS Tutorial for ASIC designs

This appendix guides you through a CtoS GUI-based tutorial for ASIC designs. In this tutorial, an implementation of a JPEG IDCT (*Joint Photographic Experts Group Inverse Discrete Cosine Transform*) decoder, originally written in C, is transformed into a synthesizable SystemC model.¹

This tutorial assumes that several design steps have already been performed – specifically that:

- The design started with a sequential algorithmic model describing the functionality to be implemented.
- The algorithmic model was partitioned into concurrent SystemC modules. This partitioning reflects the architecture of the intended implementation. The communication between the modules is at the level of transactions.
- The communication of the module (to which CtoS is to be applied) has been refined to a cycle-accurate, signal-based level. However, the core of the module remains at the algorithmic level.

At this point in a design flow, you can use CtoS both to transform a design into synthesizable RTL and to verify that the CtoS-generated RTL has the same functionality as the original source.

This tutorial is organized as follows:

- “Starting, Setting Up and Building the Design” on page A-2
- “Specifying Micro-Architecture” on page A-11
- “Scheduling and Allocating Registers” on page A-14
- “Analyzing the Design” on page A-16
- “Implementing the Design” on page A-20

1. This software is based in part on the work of the Independent JPEG Group.

A.1 Starting, Setting Up and Building the Design

To get started with this tutorial, you will perform the following tasks:

- “Starting the CtoS GUI” on page A-2
- “Starting and Setting Up the Design” on page A-4
- “Building the Design” on page A-8

A.1.1 Starting the CtoS GUI

You can start CtoS in either a graphical user interface (GUI) or a Tcl command-line environment.

The CtoS GUI is highly recommended, as you are provided with very helpful menus, dialogs, viewers, and other navigational aids. The CtoS GUI also lets you type any CtoS or native Tcl command at its command line.

For this tutorial, you will use the CtoS GUI.

Step One

From your CtoS installation area, change to the following directory

```
install_directory/share/ctos/examples/flows/eco/jpeg_idct/base
```

Note Before running any CtoS examples, first review “Setup for Examples” on page F-2.

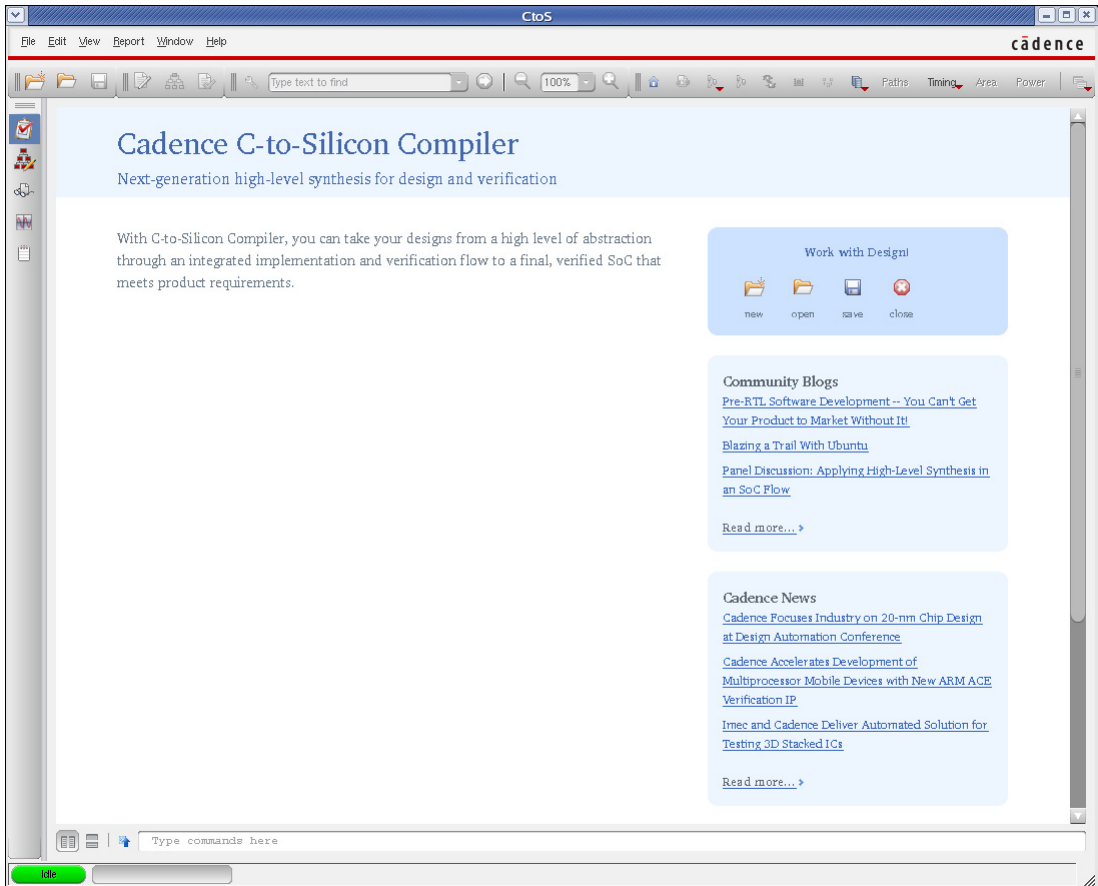
Type at the command line:

```
ctosgui
```

You should see a window similar to the one shown in [Figure A-1 on page A-3](#).

Note See also “Starting the CtoS GUI” on page 6-2.

Figure A-1 Main View of the CtoS GUI



A.1.2 Starting and Setting Up the Design

CtoS uses the concept of *designs* to manage a synthesis environment setup, so your first step is always to start a new *design property file*, which contains information about your design that is very stable. This may include the filename of the source and the names and characteristics of the clocks.

After a design property file has been created, you can load it during subsequent runs of CtoS so you do not have to re-enter all of this information.

Step Two

From the *Tool Bar*, select the **New Design** icon 

You should see the **Create New Design Wizard**, as shown in [Figure A-2 on page A-4](#).

Tip There are several ways to select a command in the CtoS GUI. In addition to the colorful icons, there are pull-down menus:

File -> New Design

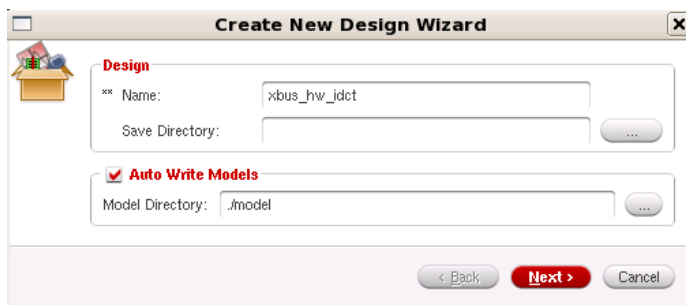
You can also type CtoS Tcl commands in the input area (blinking >) of the *Command Window*:

new_design [design_name]

However, note that the **new_design** command will not bring up the **Create New Design Wizard**, as do the other two methods. The **new_design** command will simply start a design and give it a name. You would then go directly to editing the design properties using the **Design Property** dialog, as shown in [Figure A-7 on page A-8](#).

See “[CtoS Command Reference](#)” on [page E-1](#) for descriptions of all of the Tcl commands.

Figure A-2 Create New Design Wizard (Page One)



In this page, you are selecting names and directories for your design:

- **Name:** You can enter a name for the design. For this example, specify the design name as **xbus_hw_idct**. The name cannot have any embedded spaces.

- **Save Directory:** Use the ... button to scroll to the directory in which to save the design.
- **Auto Write Models:** Uncheck this box to *not* have CtoS automatically write simulation models after a successful build (**post_build**) and a successful allocate registers (**final**).
- **Model Directory:** Use the ... button to scroll to the directory in which to store simulation models.

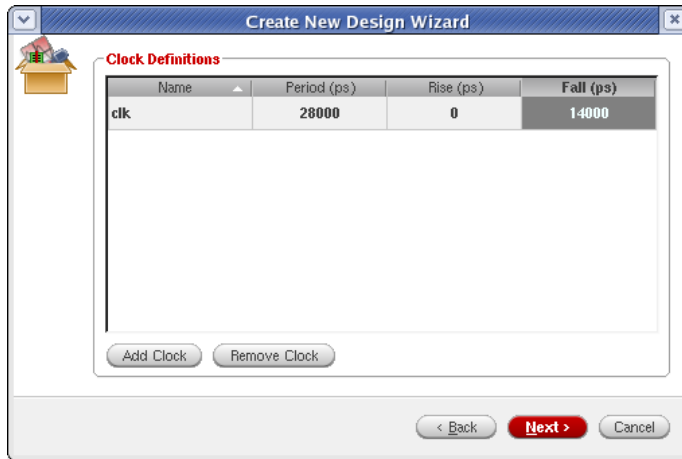
Figure A-3 Create New Design Wizard (Page Two)



For this page:

- **Source Files:** Scroll to `src/xbus_hw_idct.cc` using the **Add** button
- **Top Module:** Scroll to `xbus_hw_idct` using the ... button
- Leave the defaults for all other fields.
- Click **Next**. You will see the next page, as shown in [Figure A-4 on page A-6](#).

Figure A-4 Create New Design Wizard (Page Three)



For this page:

- Click the **Add Clock** button. The **clock_0 (20000,0,10000)** should be displayed.
- Type **clk** over **clock_0** and change **Period** to **28000**; **Fall** should automatically change to **14000**. Leave **Rise** with its default value of **0**.
- Click **Next**. You will see the next page, as shown in [Figure A-5 on page A-7](#).

Figure A-5 Create New Design Wizard (Page Four)

Create New Design Wizard

☒ **Implementation Aware**

Implementation Target: ASIC

Default Scheduling Effort: high

Setup RC

*** Technology Library: Add

RC Startup Script: ...

Working Directory: rc_work ...

Prototype Memory Timing Attributes

Setup Delay: 20% Launch Delay: 80%

Export and External Memories

Default Export: ☐

Top Wrapper Suffix: _top_wrapper

Reset Registers: Only Required

Maximize Clock Gating: ☐

< Back Next > Cancel

For this page:

- **Technology Library:** Type in **tutorial.lib** (using basic technology library that is shipped with the installation).
- **RC Working Directory:** add **../** in front of **rc_work** to share it between Incremental Synthesis runs.
- **Reset Registers:** select the **Internal** option.
- Leave the defaults for all other fields.
- Click **Next**. You will see the next page, as shown in [Figure A-6 on page A-7](#).

Figure A-6 Create New Design Wizard (Page Six)

Create New Design Wizard

We have all the information needed to create your design. Click on **Finish** to create the design.

☒ Build now

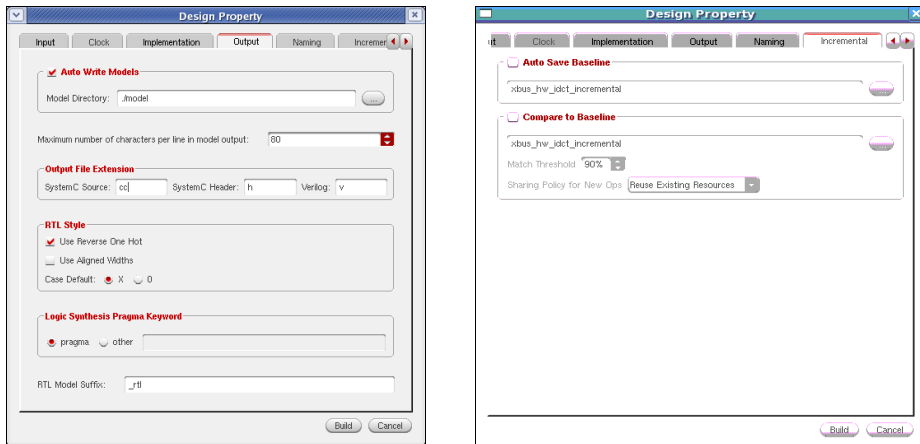
☐ Edit design properties before build

< Back Finish Cancel

For this page:

- You need to set additional Design Properties before building, so change the selection to to:
 - **Edit design properties before build**
- Click **Finish**. You will see the **Design Property** dialog, as shown in [Figure A-7 on page A-8](#).

Figure A-7 Design Property Dialog (Output and Incremental Tabs)



Select the **Output** tab:

- Change **SystemC Source** to **cc**. Do not select Build yet.

Select the **Incremental** tab:

- Select the **Auto Save Baseline** checkbox
- Add **../** in front of **xbus_hw_idct_incremental** to write it to the directory one level above. CtoS automatically saves the design datapoints for [“Rerunning with Incremental Synthesis” on page A-26](#).

You do not need to change anything in the **Input**, **Clock**, or **Naming** tabs, so select the **Build** button.

A.1.3 Building the Design

During the **build** process, CtoS reads input files, elaborates them, and creates data structures representing the many modules and behaviors that make up a design in CtoS memory. When the **build** process has completed, you will see the **Input Source** viewer, as shown in [Figure A-8 on page A-9](#).

Figure A-8 Results of Building the Design - Input Source Viewer



```

1
2
3
4  The following code is derived, directly or indirectly, from the SystemC
5  source code Copyright (c) 1996-2004 by all Contributors.
6  All Rights reserved.
7
8  The contents of this file are subject to the restrictions and limitations
9  set forth in the SystemC Open Source License Version 2.4 (the "License");
10 You may not use this file except in compliance with such restrictions and
11 limitations. You may obtain instructions on how to receive a copy of the
12 License at http://www.systemc.org/. Software distributed by Contributors
13 under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF
14 ANY KIND, either express or implied. See the License for the specific
15 language governing rights and limitations under the License.
16
17
18
19
20 #include "xbus_hw_idct.h"
21
22 #ifdef __CTOS__
23 SC_MODULE_EXPORT(xbus_hw_idct);
24 #endif
25
26 xbus_hw_idct::xbus_hw_idct(sc_module_name name)
27 : sc_module(name),
28   clk("clk"),
29   reset("reset"),
30   ms("ms"),
31   read("read"),
32   size("size"),
33   adr("adr"),
34   data_in("data_in"),
35   data_out("data_out")
36 {
37     SC_CTHREAD(run, clk.pos());
38     reset_signal_is(reset, false);
39 }
40
41 void
42 xbus_hw_idct::bus_if()
43 {

```

Tip Look at the **Design Property** dialog (**Edit -> Design Properties**) again, as shown in [Figure A-9](#) on [page A-10](#). Some fields are now *locked* to prevent you from inadvertently changing an attribute that should not be changed after you have built a design, to preserve the integrity of the synthesis process. However, you can use the **Change Setup** button if you find you *must* change an attribute.

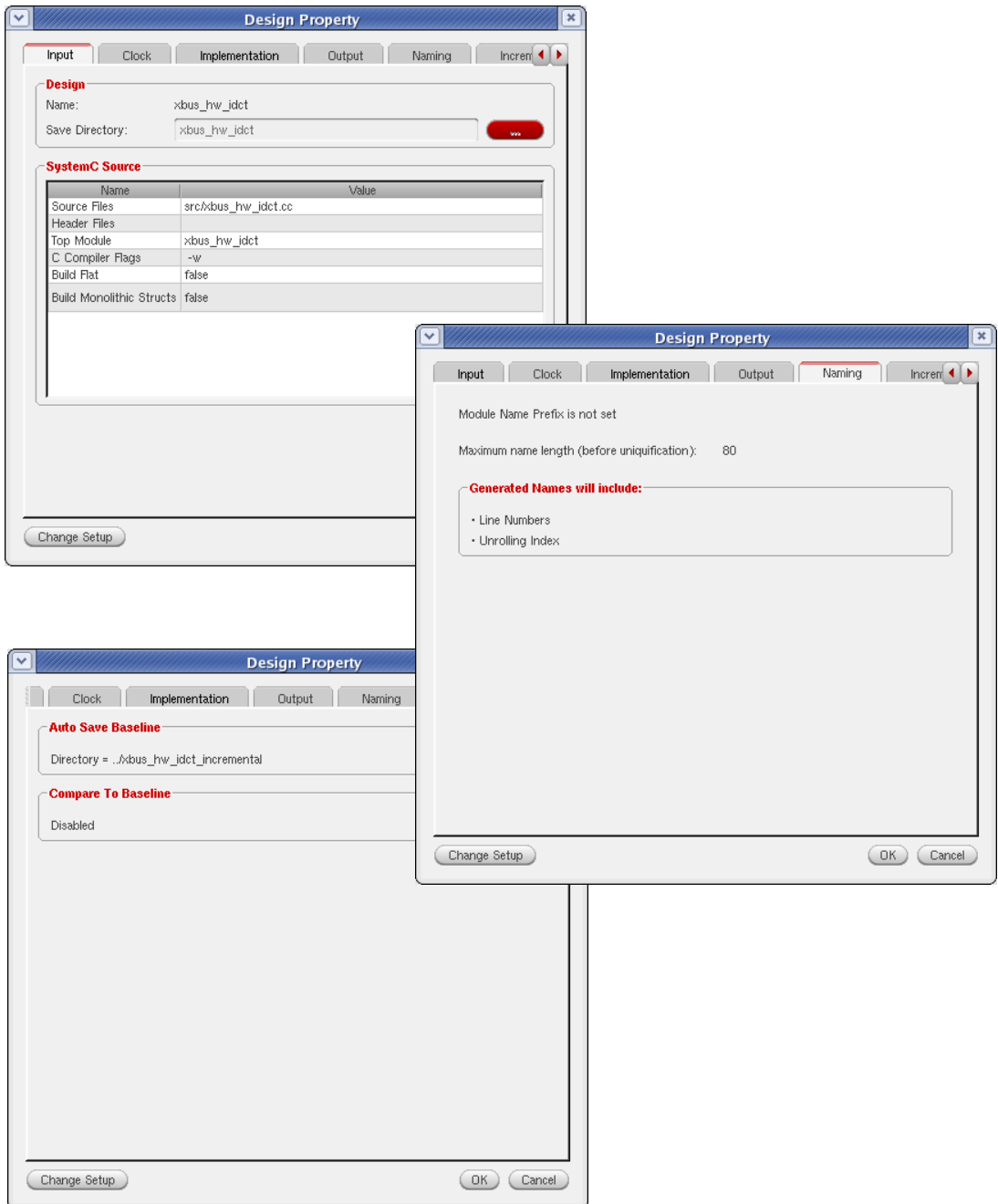
While there is not *one* equivalent Tcl command for this dialog, you can set individual attributes with the **set_attr** command ("**set_attr**" on [page E-140](#)); for example, to change the **rc_work_dir** design attribute, you could type in the input area (blinking >) of the *Command Window*:

(note – do NOT actually do this here in the tutorial)

set_attr rc_work_dir "../rc_work_dir2" /designs/xbus_hw_idct

For a complete list of all design attributes and when you must set them, see "[Design Object Attributes \(Designs\)](#)" on [page D-9](#).

Figure A-9 Design Property Dialog (Three Tabs Shown, some Fields Locked after Build)



A.2 Specifying Micro-Architecture

Returning to the CtoS GUI, information about the design is now displayed in both the **Task** and **Hierarchy Windows**.

The main difference between the **Task** and **Hierarchy Windows** is the **Hierarchy Window** shows *all* of your design objects, while the **Task Window** focuses attention only on those you must resolve before being able to schedule your design.

In either window, if an object is highlighted in red, you must resolve that object before you can continue.

For example, combinational loops must be eliminated, because they cannot be implemented in hardware; similarly, you must specify how arrays should be implemented.

To resolve an object, you simply click on the object in the **Task Window**, and you will be presented with various options for each object.

When none of the objects is highlighted in red, and there are no yellow flags in the **Task Window**, you are ready to schedule the design.

This section has the following subsections:

- “Inlining Functions” on page A-11
- “Breaking Combinational Loops” on page A-12
- “Allocating IP” on page A-12
- “Creating Latency Constraints” on page A-13
- “Generating a Verilog Simulation Model (_current)” on page A-14

A.2.1 Inlining Functions

Inlining of functions can be a useful technique to control the implementation of a design in order to deliver the desired performance or required area.

Note See also “Inlining Functions” on page 8-3.

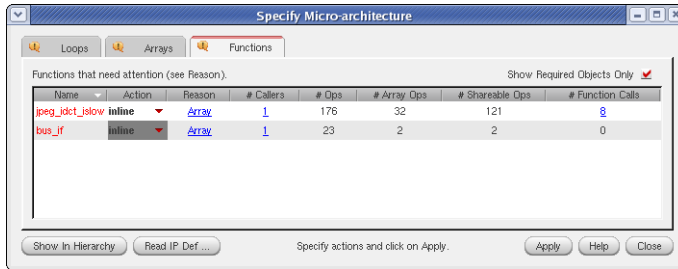
Step Seven

In the **Task Window**, click **Specify Micro-architecture** (if necessary); then double-click **Functions**. You will see the Specify Micro-architecture dialog, as shown in [Figure A-10 on page A-12](#).

Select **inline** as the **Action** for each of the functions, then click **Apply**. Do not close this dialog yet.

Tip You can use the standard keyboard *Ctrl-C* and *Ctrl-V* to copy the same action for each function.

Figure A-10 Specify Micro-Architecture Dialog



A.2.2 Breaking Combinational Loops

Combinational loops must be eliminated before scheduling, because they cannot be implemented in hardware. A loop is *combinational* if at least one program execution from the top to the bottom of the loop does not include waiting for a clock edge.

In some cases, a loop may not have such an execution, but CtoS will consider it combinational, because it cannot prove at compile time that such an execution does not exist.

Note See also “How Combinational Loops Are Determined in CtoS” on page 8-17.

Step Eight

Select the **Loops** tab of the **Specify Micro-architecture** dialog (previously shown in [Figure A-10 on page A-12](#)). Select **break** as the **Action** for each loop and click **Apply**. The dialog will close automatically.

A.2.3 Allocating IP

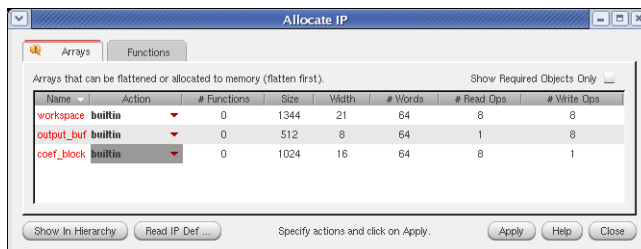
You must also allocate IP for some arrays (memories).

Note See also “Resolving Arrays (Memories)” on page 8-61.

Step Nine

In the **Task Window**, in the **Allocate IP** pane, double-click **Arrays**. In the **Allocate IP** dialog, as shown in [Figure A-11 on page A-12](#). Select **builtin** as the **Action** for each of the arrays, then click **Apply**.

Figure A-11 Allocate IP Dialog



A.2.4 Creating Latency Constraints

You have broken the two combinational loops (**Pass1_for_begin** and **Pass2_for_begin**), by adding a state to each, but more states are required in each loop iteration for memory reads and writes. To do this, you will set a latency constraint for each of these loops.

Note See also “[Constraining Latency](#)” on page 11-12.

Step Ten

In the **Hierarchy Window**, right-click on **Pass1_for_begin** (Design -> Modules -> xbus_hw_idct -> Processes -> xbus_hw_idct_run -> Inlined Function Calls -> jpeg_idct_islow_ln81 -> Loops) and select **Constrain Latency -> Set Region Start Node**.

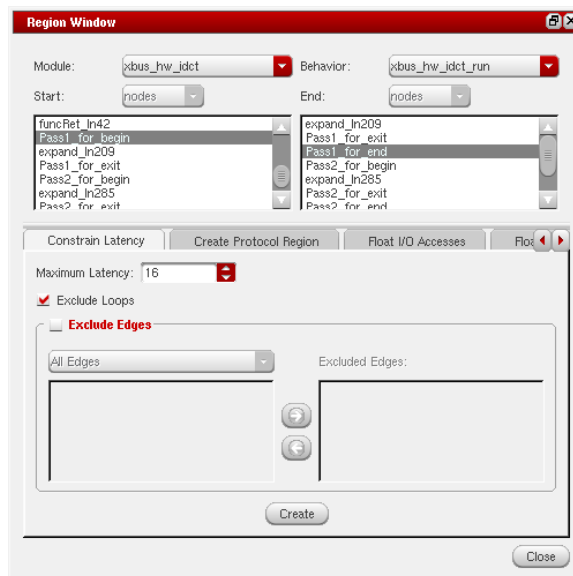
From the *Tool Bar*, select the **Region Editor** icon .

Note If this is not already displayed, select **Window -> Toolbars -> Region**.

In the **Region Window** dialog, as shown in [Figure A-11 on page A-12](#), **Pass1_for_begin** and **Pass1_for_end** should already be selected. Change **Maximum Latency** to **16**, and click **Create**, but do not close the dialog yet.

Still in the **Region Window** dialog, select **Pass2_for_begin**, and **Pass2_for_end** should automatically be selected for the region end. Again, change **Maximum Latency** to **16**, and click **Create** and **Close**

Figure A-12 Region Window Dialog



A.2.5 Generating a Verilog Simulation Model (_current)

Just before scheduling your design, you will generate another Verilog simulation model.

Step Eleven

Select:

File -> Generate -> Verilog Behavioral Model

In the Generate **Verilog Behavioral Model** dialog, just leave all the defaults, and click **OK**.

A.3 Scheduling and Allocating Registers

Now, you are ready to schedule your design.

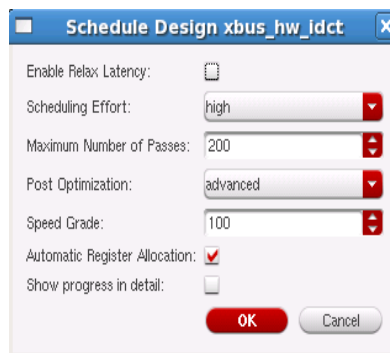
Scheduling consists of a sequence of steps that produce specified output models.

Note See also “Scheduling” on page -2.

Step Twelve

Right-click **Design (xbus_hw_idct)** under **Schedule** in the **Task Window** and select **Schedule**. The **Schedule Design** dialog, as shown in [Figure A-13 on page A-14](#), is displayed.

Figure A-13 Schedule Design Dialog

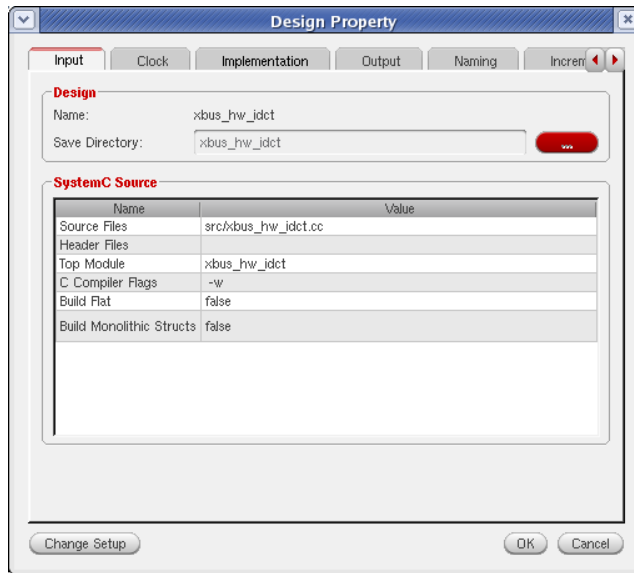


For this tutorial, you can simply leave the defaults, and click **OK**.

CtoS first uses Encounter RTL Compiler (RC) to characterize the resources from the given library that it needs. This will take several minutes.

CtoS next tries to find a feasible schedule for all of the operations.

Figure A-14 Design Property Dialog (One Tab Shown, Which Is Locked after Schedule)



If it cannot find a solution with the current set of resources in the currently allowed number of cycles, it will add more resources or allow more cycles (up to the bound specified by latency constraints) and try another pass.

This process continues until CtoS finds a solution, reaches a predetermined number of passes, or cannot find any action that would improve schedulability.

Since you left **Automatic Register Allocation** checked in the **Schedule** dialog, if scheduling is successful, CtoS will automatically run the **allocate_registers** command.

Tip Look at the **Design Property** dialog (**Edit -> Design Properties**) again, as shown in [Figure A-14 on page A-15](#).

Additional fields are now locked after scheduling. Again, this is to prevent you from inadvertently changing an attribute that should not be changed after you have scheduled your design, in order to preserve the integrity of your synthesis process.

For a complete list of all design attributes and when you must set them, see [“Design Object Attributes \(Designs\)” on page D-9](#).

A.4 Analyzing the Design

When synthesis is complete, you can analyze the synthesized design using the features of CtoS described in the following sections:

- “Understanding Scheduling Actions” on page A-16
- “Exploring Registers” on page A-17
- “Exploring Resources” on page A-19

A.4.1 Understanding Scheduling Actions

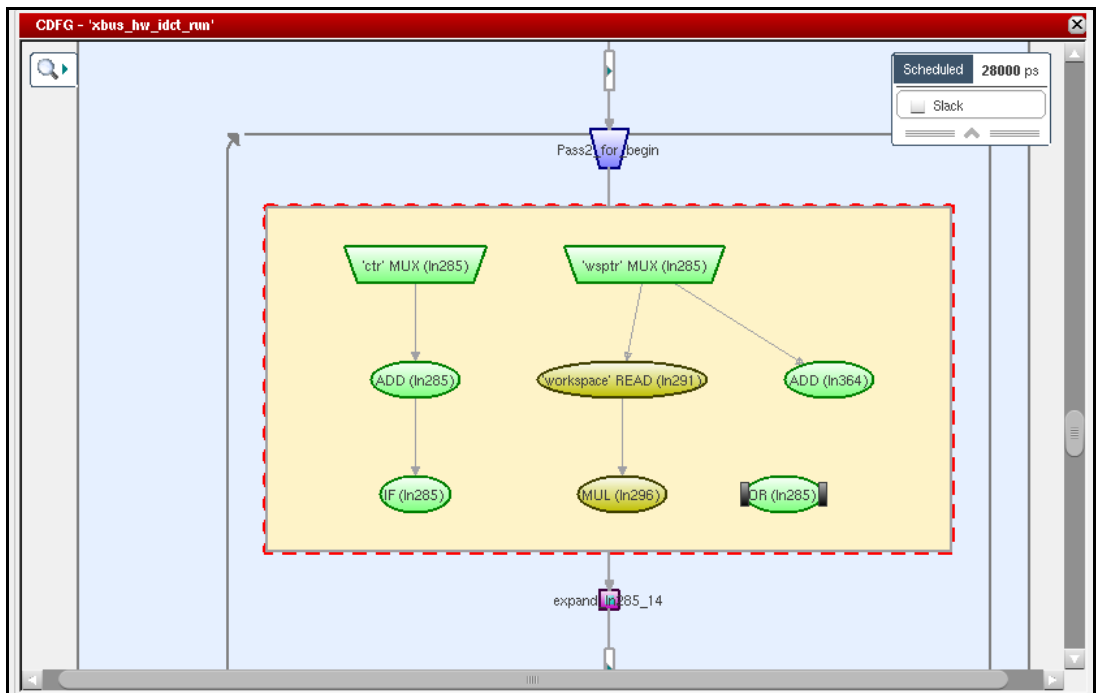
To understand scheduling actions, you can view, in the CDFG, one of the loops in this design.

Step Thirteen

In the **Hierarchy Window**, right-click on **Pass2_for_begin** (**Design** -> **Modules** -> **xbus_hw_idct** -> **Processes** -> **xbus_hw_idct_run** -> **Inlined Function Calls** -> **jpeg_idct_islow_ln81** -> **Loops**), and select **Show in CDFG**. You should see the **CDFG** viewer, as shown in [Figure A-15 on page A-16](#). You can right-click on one of the edges in the loop to use the Expand/Collapse feature.

Tip You can use the zooming features of the **Tool Bar** to change the display of the CDFG. You can also zoom using the **Ctrl** button with the mouse wheel.

Figure A-15 CDFG Viewer (Pass2_for_begin) - Edge Expanded



A.4.2 Exploring Registers

To understand how CtoS selects and uses registers, you can use the CtoS reporting and viewing features.

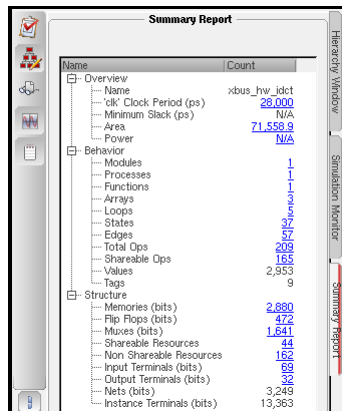
Step Fourteen

From the icons along the left side of the main CtoS GUI window, select the **Summary Report** icon (the bottom-most icon, which looks like a notepad), or select:

Window -> Summary Report

You should see the **Summary Report**, as shown in [Figure A-16 on page A-17](#).

Figure A-16 Summary Report for Complete Design



This report gives you a sense of the design complexity, both in terms of the number and kind of behavioral objects appearing in the source code and in terms of the number and kind of structural elements CtoS uses to implement that behavior.

Step Fifteen

To better understand why CtoS selected a certain number of registers and how they are used, from the *Menu Bar*, select:

View -> Registers

You should see the **Register Bindings** viewer, as shown in [Figure A-17 on page A-17](#).

Figure A-17 Register Bindings Viewer

Register Bindings						
xbus_hw_idct_run						
	Wait_In78_reg	ctrlOr_In79_2_0	ctrlOr_In209_2_0	ctrlOr_In209_2_0_expand_In209_0	ctrlOr_In209_2_0_expand_In209_1	ctrlOr_In209_2
origin	0	0	0	0	0	0
Wait_In78						
Wait_In50						
Wait_In53						
Wait_In60						
Wait_In70						
expand_In209_14						
expand_In209_13						
expand_In209_6						

The *columns* of the **Register Bindings** viewer represent the registers used in the resulting RTL, and the *rows* represent the control states of the process.

Right-clicking on a *state* lets you display that state in the **CDFG** and also, if that state corresponds to a **wait** statement in the input code, in the **Input Source**.

Right-clicking on a *value* lets you display the **receiver** (that is, the consumer) or the **driver** (that is, the producer) of that value.

Step Sixteen

Right-click any state, and select **Show in CDFG** or **Show Input Source**, to see an example of this.

Right-click a value, and select **Show receiver** (that is, the consumer) or **Show driver** (that is, the producer) again, just to see an example of this. (You may have to click on several to see this viewer; for this figure, the value for **expand_In209_9** and **mul_In254_Z_0** was used). You should see the **Resource Bindings** viewer, as shown in [Figure A-17 on page A-18](#). Resources will be discussed in the next section.

Figure A-18 Resource Bindings Viewer (expand_In209_9, mul_In254_Z_0)

Resource Bindings Viewer										
xibus_hw_idct_run										
Resource Bindings	Resource Type	smul				add				
	Expand/Collapse	Collapse				Collapse				
Shareable Ops	Widths	32x22x16	32x18x16	28x23x16	28x17x11	32	(*)_15	(*)_19	(*)_26	(*)_31
Instance Name	(*)	(*)	(*)	(*)	(*)	(*)	(*)_15	(*)_19	(*)_26	(*)_31
mul_In325	bound									
mul_In322				bound						
mul_In295				bound						
mul_In324	bound									
mul_In254					bound					
mul_In247				bound						
mul_In296				bound						
mul_In321				bound						
mul_In328				bound						
mul_In323				bound						
add_ops										
add_In256										
add_In260										
add_In260_0										
add_In261									bound	
add_In266									bound	
add_In268										
add_In270								bound		
add_In272						bound				
add_In259										
add_In262										

A.4.3 Exploring Resources

The **Resource Bindings** viewer displays the resource bindings for an operation at different states.

Step Seventeen

At the input prompt (blinking >) of the *Command Window*, type:

```
report_resources
```

You should see a report similar to the following, in the **Command Output** window:

```
% report_resources
```

```
Resources allocated to module xbus_hw_idct, behavior xbus_hw_idct_run:
```

Count	ModuleType	Master
----	-----	-----
1	ram	ram_64x21_1ar_1w_r
1	ram	ram_64x8_1ar_1w_r
1	ram	ram_64x16_1ar_1w_r
4	custom	range_limit
3	addsub	addsub_32
1	smul	smul_28x17x11
1	smul	smul_32x18x16
1	addsub	addsub_28
1	smul	smul_28x23x16
7	add	add_32
1	add	add_17
1	add	add_3x2
2	add	add_3x1
1	smul	smul_32x22x16
2	add	add_3
2	add	add_2x1
2	add	add_2
2	add	add_31
5	add	add_22x1
2	sub	sub_32
4	add	add_11x1
1	sub	sub_3x1
1	add	add_30

Step Eighteen

Observe the number of multipliers – could fewer have been used? To explore this possibility, in the **Input Source** viewer (**View -> Input**) for a multiply operator (for example, at line 247, hover on **MULTIPLY**, select the arrow, and right-click on **mul_In247**), select:

```
Show Resource Bindings
```

In the **Resource Bindings** viewer, as shown in [Figure A-19 on page A-20](#), you can see all of the multiplier resources.

Right-click on one of the **mul_** operators, and select **Show Op Span**. This will display the **CDFG** with the op span highlighted. Observe that the span includes several edges, so there is indeed potential for the operation to be rescheduled. If you would like to continue this analysis, check for when the operand is scheduled to be computed.

Figure A-19 Resource Bindings Viewer - Multiplier Resources Expanded

Resource Bindings	Resource Type	Widths	Instance Name
mul ops			
mul_in245	bound	32x22x16	
mul_in252	bound	32x18x16	
mul_in253	bound	32x18x16	
mul_in248	bound	32x18x16	
mul_in217	bound	32x18x16	
mul_in251	bound	32x18x16	
mul_in218	bound	32x18x16	
mul_in250	bound	32x18x16	
mul_in249	bound	32x18x16	
mul_in219	bound	32x18x16	
mul_in319	bound	28x23x16	
mul_in326	bound	28x23x16	
mul_in327	bound	28x23x16	
mul_in294	bound	28x23x16	
mul_in325	bound	28x23x16	
mul_in322	bound	28x23x16	
mul_in295	bound	28x23x16	
mul_in324	bound	28x23x16	
mul_in254	bound	28x17x11	
mul_in247	bound	28x17x11	
mul_in296	bound	28x17x11	
mul_in321	bound	28x17x11	
mul_in328	bound	28x17x11	
mul_in323	bound	28x17x11	

A.5 Implementing the Design

At this point, CtoS can generate several models of your design, including a synthesizable RTL Verilog model and a Verilog model equivalent to the RTL model, but not synthesizable. The Verilog model is constructed to simulate much faster than the RTL model and is intended to be used in verification. For more about each of these models, see the following sections:

- “Generating a Synthesizable RTL Model” on page A-21
- “Generating a Verilog Behavioral Model (_final)” on page A-22
- “Comparing Simulation Results” on page A-22
- “Rerunning with Incremental Synthesis” on page A-26

A.5.1 Generating a Synthesizable RTL Model

In this section, you will generate a synthesizable RTL model.

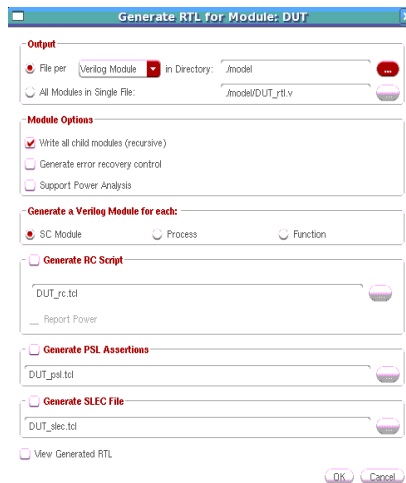
Step Nineteen

To generate the synthesizable RTL model, from the *Menu Bar*, select:

File -> Generate -> RTL

In the **Generate RTL dialog**, as shown in [Figure A-20 on page A-21](#), leave the defaults, and click **OK**.

Figure A-20 Generate RTL Dialog



Step Twenty

To relate the generated RTL to the original source code, CtoS creates links from objects in the RTL to the corresponding objects in the source code whenever possible.

From the *Menu Bar*, select:

View -> Generated Models -> RTL

In the RTL viewer, locate **case (1'b1)** on line 1160.

Tip Type this in the *Find* box (to the right of the **Check Design** icon in the *Menu Bar*) and then click the arrow to the right of it. Keep clicking the *Find Next* arrow to get to line 1217.

Click **case (1'b1)**, click the corresponding arrow, right click the instance, and select the following to observe the loop in the source code:

mux_ctr_ln285 INSTANCE -> Show Ops -> in Input

A.5.2 Generating a Verilog Behavioral Model (_final)

In this section, you will generate a Verilog behavioral model, equivalent to the generated RTL.

Step Twenty-one

From the *Menu Bar*, select:

File -> Generate -> Verilog Behavioral Model

In the **Generate Verilog Behavioral Model** dialog:

- Change the **Module Suffix** to **_final**
- Change **Output file name** to **./model/xbus_hw_idct_final.v** *(just change the suffix)*
- Leave all the other defaults, and click **OK**.

Step Twenty-two

You are now ready to exit the CtoS GUI, so from the *Menu Bar*, select:

File -> Exit

A.5.3 Comparing Simulation Results

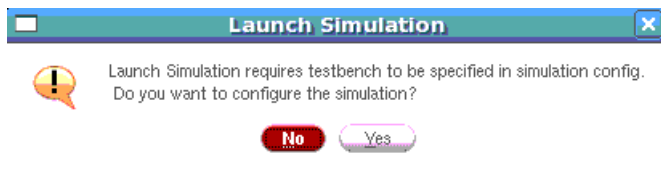
You can now compare simulation results.

Step Twenty-three

Select the **Simulation Monitor** icon (Waveform icon) from the tool list on the left-hand side of main window.

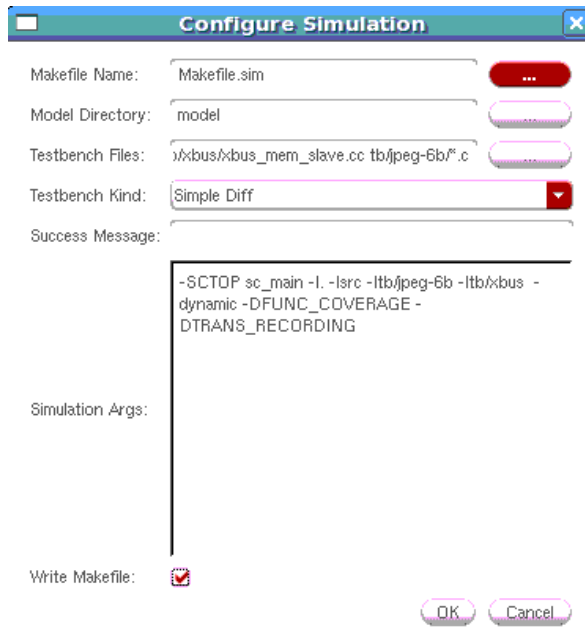
Then, click the **Launch** button (green arrow) on top edge of the **Simulation Monitor** pane. The Launch Simulation dialog is displayed, as shown in [Figure A-21 on page A-22](#).

Figure A-21 The Launch Simulation Dialog



The Click **Yes** on the **Launch Simulation** dialog to configure it. The **Configure Simulation** dialog is displayed as shown in [Figure A-22 on page A-23](#).

Figure A-22 The Configure Simulation Dialog

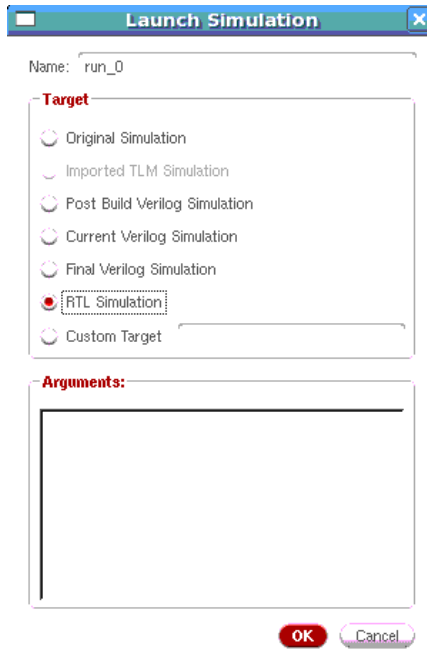


On the **Configure Simulation** dialog, do the following:

- **Makefile Name:** Type in **Makefile.sim**
- **Testbench Files:** Select the following files from **tb**, **tb/xbus** and **tb/jpeg-6b** directories:
Tip Use the ... button to browse to these files and add more files. You can also add files by typing in the **Testbench Files** field.
 - From the **tb** directory, select all files and click **Open**.
 - From the **tb/xbus** directory, select the **xbus_master.cc** and **xbus_mem_slace.cc** files and click **Open**.
 - From **tb/jpeg-6b** directory, add all the **.c** files by typing **tb/jpeg-6b/*.c** in the **Testbench Files** field.
- **Simulation Args:** Type in **-SCTOP sc_main -I. -Isrc -Itb/jpeg-6b -Itb/xbus -dynamic -DFUNC_COVERAGE -DTRANS_RECORDING**.
- **Write Makefile:** Select the checkbox (if not already checked).
- Leave the defaults for all other fields.
- Click **OK**.

On the **Launch Simulation** dialog, select the **RTL Simulation** radio button and click **OK**.

Figure A-23 The Launch Simulation Dialog



The **Simulation Monitor** shows the progress of the simulation.

An entry is added to the list of simulation runs (blue showing simulation in progress). And the lower half shows the running log of the simulator. When simulation is successful, the entry will change to green color. Since simulation may take significant run time, you can continue to use CtoS to analyze this design while simulation is running.

As a result of clicking **OK** on the **Configure Simulation** dialog, a makefile named **Makefile.sim** is generated. You can run simulations using the generated makefile.

Note Make sure your path includes the **ncsc_run**, and **ctos** executables.

To simulate the original design, type the following:

```
make -f Makefile.sim orig_sim
```

To simulate and compare the results of simulation with the CtoS-generated model against the reference model, type one of the following for the simulation or RTL model, respectively:

```
make -f Makefile.sim final_sim
```

```
make -f Makefile.sim rtl_sim
```

A.5.4 Generating Gates

You can now generate gates.

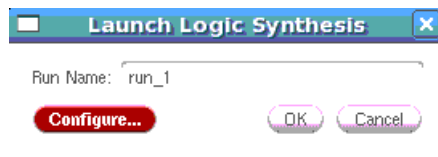
Step Twenty-four

Select the **Synthesis Monitor** icon (gate icon) from the tool list on the left-hand side of main window.

Then, click the **Launch** button (green arrow) on top edge of the **Synthesis Monitor** pane.

The **Launch Logic Synthesis** dialog is displayed as shown in [Figure A-24 on page A-25](#).

Figure A-24 The Launch Logic Synthesis Dialog



Click **OK** on the **Launch Logic Synthesis** dialog.

The **Synthesis Monitor** shows progress of the synthesis run.

An entry is added to the list of synthesis runs (blue showing synthesis in progress). And, the lower half shows the running log of the synthesis run. When synthesis is successful, the entry will change to green color and the generated gates file is named **xbus_hw_idct_rtl.vg** in the **run_synth_gates** directory.

Since logic synthesis make take significant run time, you can continue to use CtoS to analyze this design while synthesis is running.

As a result of clicking **OK** on the **Launch Logic Synthesis** dialog, the **rc_run.tcl** script is generated in the **run_synth_gates** directory to direct RC to generate gate-level netlist from the CtoS generate RTL.

Also, you can generate a makefile to run this script, by typing:

```
write_synth_makefile -name run_synth_gates/Makefile.synth
```

You can then run logic synthesis using the generated makefile.

Note Make sure your path includes the **rc** and **ctos** executables.

To generate gate-level netlist from the CtoS generated RTL model, type:

```
cd run_synth_gates;  
make -f Makefile.synth synth_gates
```

A.5.5 Rerunning with Incremental Synthesis

When a required specification change occurs late in your design schedule, you can use the *Incremental Synthesis* feature of CtoS.

Continuing with the same *baseline* design, you will now use *Incremental Synthesis* mode to read in a new source file. When the two files are compared, CtoS tries to retain as much of the original design as possible.

Note See also “Incremental Synthesis” on page 17-1.

For these steps, you will use the supplied **ctos.tcl** script in the **V1** directory and run CtoS in batch mode.

Step Twenty-four

Change to the following directory:

```
cd ../V1
```

In this directory, the **src/xbus_hw_idct.cc** file has been changed.

Look at the **ctos.tcl** script (in **/V1**), and notice this line (**you do not have to type this in**):

```
set_attr baseline_dir ../xbus_hw_idct_incremental /designs/xbus_hw_idct
```

This tells CtoS to look in **../xbus_hw_idct_incremental /designs/xbus_hw_idct** for a baseline for which to compare your design.

CtoS will automatically perform this matching after it runs the **build** command and before it schedules.

Step Twenty-five

In the **/V1** directory, run CtoS in batch mode using the command:

```
ctos ctos.tcl
```

This will go through all the same steps as with the baseline design, except now using Incremental Synthesis.

The newly generated RTL will be written in the **model** directory.

Step Twenty-six

Compare the newly generated RTL file from this Incremental Synthesis run to that of the baseline synthesis run to see that the changes are minimal:

```
vi -d model/xbus_hw_idct_rtl.v ../Baseline/model/xbus_hw_idct_rtl.v
```

B CtoS Tutorial for FPGA designs

Support for FPGA designs is a preliminary feature.

This appendix provides a brief overview of the CtoS *FPGA prototype flow* and also guides you through a CtoS GUI-based tutorial for FPGA designs.

FPGA prototype flow

Many designs are validated with the help of FPGAs. For this purpose, CtoS provides a simple flow to support RTL generation for FPGAs. However, this flow is focused on *prototype building*, as many advanced FPGA synthesis capabilities are not exploited during the high-level synthesis run.

The CtoS *FPGA prototype flow* focuses on two major components: (1) CtoS will utilize actual timing and area information of the specified FPGA, which ensures that the generated schedule will actually be synthesizable onto the FPGA, with the given constraints. (2) The CtoS RTL is specifically modified to work as input for the chosen FPGA tool.

This flow should be considered only for *prototype* development, as CtoS is not taking advantage of the utilization information of the chosen FPGA. CtoS assumes at all points that the requested resource is available in the FPGA. In cases where the actual FPGA will have to implement resources in an alternative approach during RTL synthesis, the generated RTL might not result in a feasible schedule.

Similarly, the routing delay within the FPGA can have significant impact on the result. However, in the CtoS flow, this delay is neglected to the extent that potential adjustments to target clock frequency might be necessary to actually create a schedule satisfying the prototype requirements.

In addition, CtoS works under the assumption that all basic resources are supported by the implementation tool. You will find a table of basic resources in the description of the **create_resource** command ([“Resource Types and Required Widths for Creating Resources” on page 11-22](#)). If an unsupported resource is encountered, a simple rewrite of the SystemC input can often eliminate the restriction of the implementation tool.

FPGA memories require special attention during high-level synthesis. FPGA tools map memories to specially characterized memory blocks within the FPGA. The FPGA synthesis tool auto-detects memories, based on the provided RTL description, and maps them directly. As a result, during the high-level synthesis approach, CtoS is challenged to provide a detectable description to the characterization step through the FPGA tool. The CtoS synchronized built-in memories are usually correctly identified as memories and handled accurately by the FPGA tool. Conversely, arrays mapped to asynchronous memory might not be recognized or supported.

Some Notes on the Tutorial

In this tutorial, an implementation of a JPEG IDCT decoder, originally written in C, is transformed into a synthesizable SystemC model.¹

For most steps in the tutorial, cross-references to other parts of the user guide are provided so you can get more information about a particular topic.

This tutorial assumes that several design steps have already been performed – specifically that:

- The design started with a sequential algorithmic model describing the functionality to be implemented.
- The algorithmic model was partitioned into concurrent SystemC modules. This partitioning reflects the architecture of the intended implementation. The communication between the modules is at the level of transactions.
- The communication of the module (to which CtoS is to be applied) has been refined to a cycle-accurate, signal-based level. However, the core of the module remains at the algorithmic level.

If a design has gone through such a series of steps, CtoS can be used both to transform it into synthesizable RTL and to verify that the CtoS-generated RTL module has the same functionality as the original source.

This tutorial is organized as follows:

- [“Starting, Setting Up and Building the Design” on page B-3](#)
- [“Specifying Micro-Architecture” on page B-10](#)
- [“Scheduling and Allocating Registers” on page B-15](#)
- [“Analyzing and Implementing FPGA Designs” on page B-16](#)

1. This software is based in part on the work of the Independent JPEG Group.

B.1 Starting, Setting Up and Building the Design

To get started with this tutorial, you will perform the following tasks:

- “Starting the CtoS GUI” on page B-3
- “Starting and Setting Up the Design” on page B-4
- “Building the Design” on page B-9

B.1.1 Starting the CtoS GUI

You can start CtoS either in a graphical user interface or a Tcl command-line environment.

The CtoS GUI is highly recommended, as you are provided very helpful menus, dialogs, viewers, and other navigation aids. The CtoS GUI also lets you type any CtoS or native Tcl command at its command line.

For this tutorial, you will use the CtoS GUI.

Step One

From your CtoS installation area, change to the following directory:

`install_directory/share/ctos/examples/flows/fpga/xilinx`

Note Before running any CtoS examples, first review “Setup for Examples” on page F-2.

There is an additional example for Altera-based FPGA designs, located in:

`install_directory/share/ctos/examples/flows/fpga/altera`

Step Two

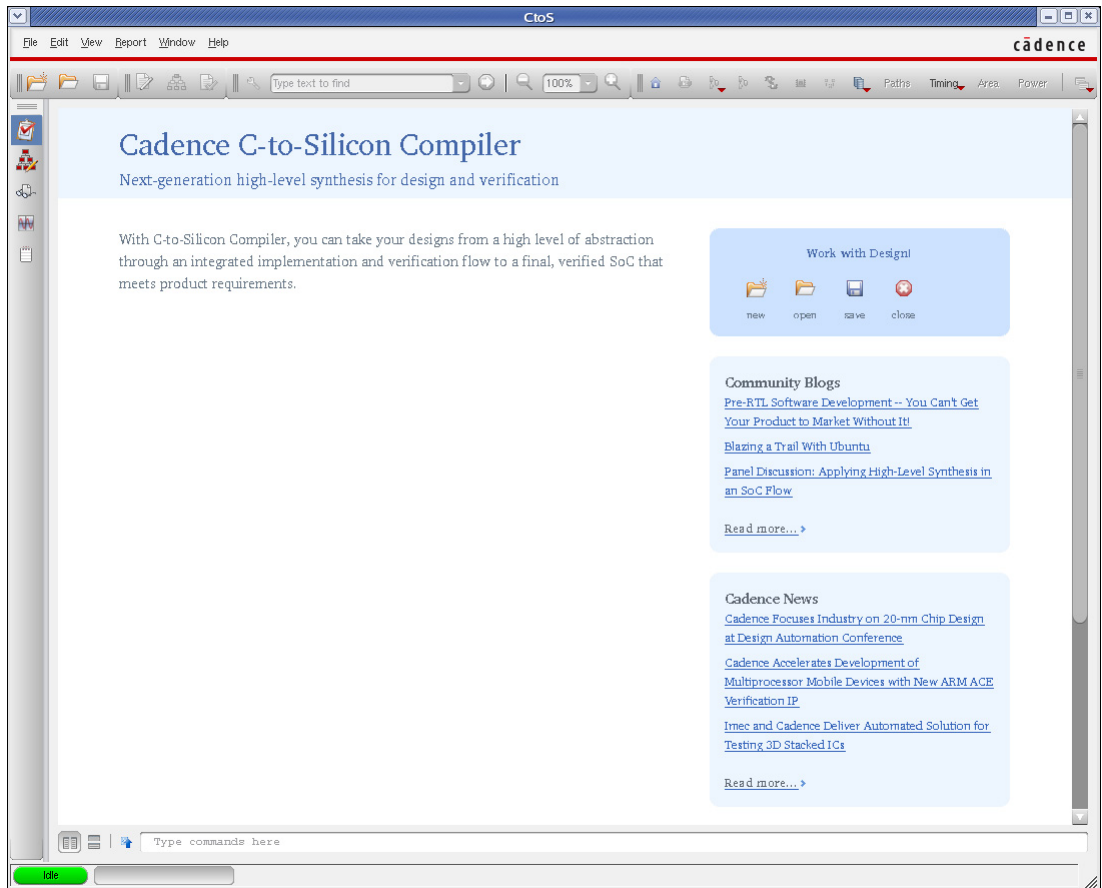
Type at the command line:

`ctosgui`

You should see a window similar to the one shown in [Figure B-1 on page B-4](#).

For more detail, see “Starting the CtoS GUI” on page 6-2.

Figure B-1 Main View of the CtoS GUI



B.1.2 Starting and Setting Up the Design

CtoS uses the concept of *designs* to manage a synthesis environment setup, so your first step is always to start a new *design property file*, which contains information about your design that is very stable. This may include the filename of the source and the names and characteristics of the clocks.

After a design property file has been created, you can load it during subsequent runs of CtoS so you do not have to re-enter all of this information.

Step Three

From the *Tool Bar*, select the **New Design** icon 

You should see the **Create New Design Wizard**, as shown in [Figure B-2 on page B-5](#).

Tip There are several ways to select a command in the CtoS GUI. In addition to the colorful icons, there are pull-down menus:

File -> New Design

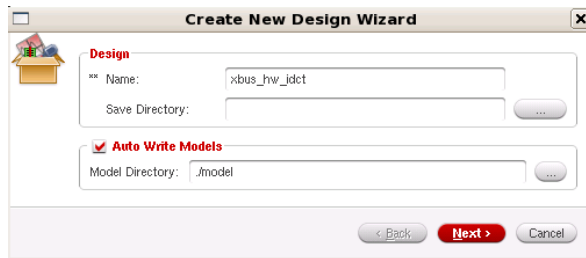
You can also type CtoS Tcl commands in the input area (blinking >) of the *Command Window*:

new_design [design_name]

However, note that the **new_design** command will not bring up the **Create New Design Wizard**, as do the other two methods. The **new_design** command will simply start a design and give it a name. You would then go directly to editing the design properties using the **Design Property** dialog, as shown in [Figure B-7 on page B-8](#).

See “[CtoS Command Reference](#)” on [page E-1](#) for descriptions of all of the Tcl commands.

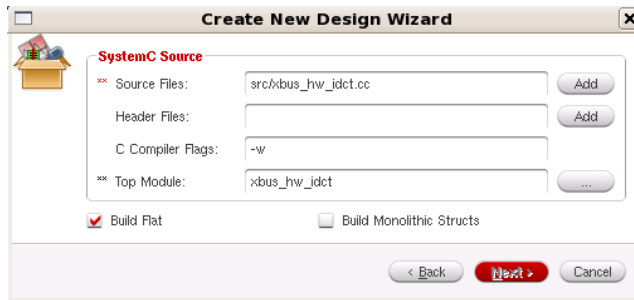
Figure B-2 Create New Design Wizard (Page One)



In this page, you are selecting names and directories for your design:

- **Name:** You can enter a name for the design.
- **Save Directory:** Use the ... button to scroll to the directory in which to save the design.
- **Auto Write Models:** Uncheck this box to *not* have CtoS automatically write simulation models after a successful build (**post_build**) and a successful allocate registers (**final**).
- **Model Directory:** Use the ... button to scroll to the directory in which to store simulation models.

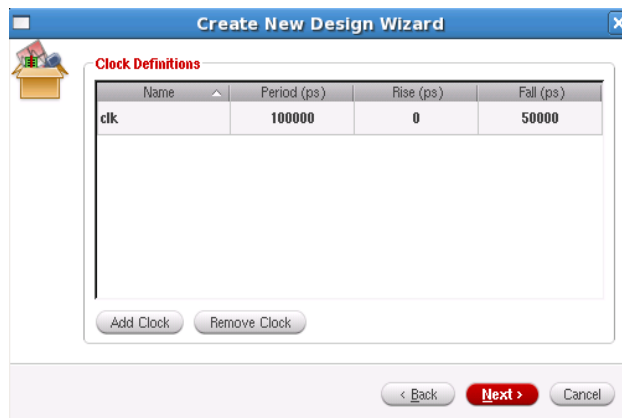
Figure B-3 Create New Design Wizard (Page Two)



For this page:

- **Source Files:** scroll to **src/xbus_hw_idct.cc** using the **Add** button
- **Top Module:** scroll to **xbus_hw_idct** using the **...** button
- **Build Flat:** select the checkbox.
- Leave the defaults for all other fields.
- Click **Next**. You will see the next page, as shown in [Figure B-4 on page B-6](#).

Figure B-4 Create New Design Wizard (Page Three)



For this page:

- Click the **Add Clock** button. The **clock_0 (20000,0,10000)** should be displayed.
- Type **clk** over **clock_0**, type **100000** in **Period**, leave **Rise** at 0. **Fall** automatically changes to **50000**.
- Click **Next**. You will see the next page, as shown in [Figure B-5 on page B-7](#).

Figure B-5 Create New Design Wizard (Page Four)

Create New Design Wizard

☒ **Implementation Aware**

Implementation Target:

Default Scheduling Effort:

- Setup FPGA

FPGA Vendor:

** Install Path:

** Family Name:

** Part Name:

Working Directory:

DSP Cost: (in terms of LUTs)

- Prototype Memory Timing Attributes

Setup Delay: Launch Delay:

Reset Registers

Maximize Clock Gating ☐

Back **Next** Cancel

For this page:

- **Implementation Target:** change to **FPGA**
- **Install Path:** type in, or scroll to, the installation path to your Xilinx executable.
- **Family Name:** type in **virtex4**
- **Part Name:** type in **xc4vlx40-12-FF668**
- Leave the defaults for all other fields.
- Click **Next**. You will see the next page, as shown in [Figure B-6 on page B-8](#).

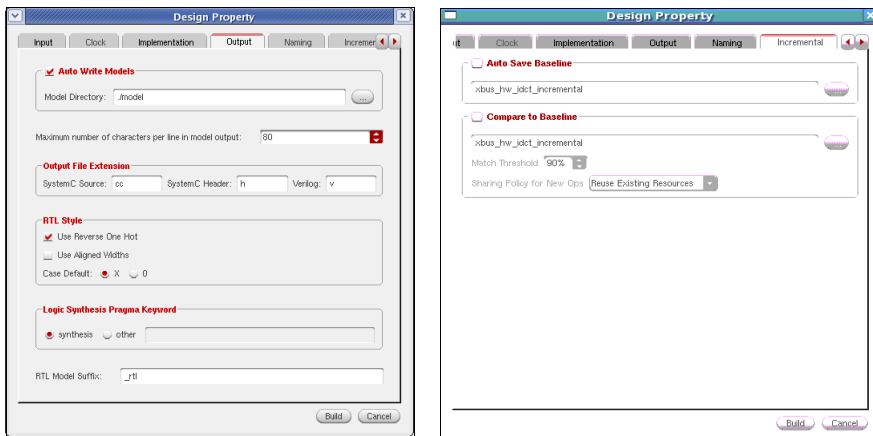
Figure B-6 Create New Design Wizard (Page Six)



For this page:

- You need to set additional Design Properties before building, so change the selection to:
 - **Edit design properties before build**
- Click **Finish**. You will see the **Design Property** dialog, as shown in [Figure B-7 on page B-8](#).

Figure B-7 Design Property Dialog (Output and Incremental Tabs)



Select the **Output** tab:

- Change **SystemC Source** to **cc**. Do not select Build yet.

Select the **Incremental** tab:

- Select the **Auto Save Baseline** checkbox
- Add **../** in front of **xbus_hw_idct_incremental** to write it to the directory one level above. CtoS automatically saves the design datapoints for [“Rerunning with Incremental Synthesis” on page A-26](#).

You do not need to change anything in the **Input**, **Clock**, or **Naming** tabs, so select the **Build** button.

Tip To restart your session quickly, CtoS provides a **ctos_setup.tcl** file that includes all setup steps, but you must replace the **get_xilinx_install_path** with the installation path to your Xilinx executable.

B.1.3 Building the Design

In the **build** step, CtoS reads input files, elaborates them, and creates data structures representing the many modules and behaviors that make up a design in CtoS memory. When the build has completed, you will see information about this design, as shown in [Figure B-8](#) on [page B-9](#).

Figure B-8 Results of Building the Design (Input Source)

```

1
2  /*****
3
4  The following code is derived, directly or indirectly, from the SystemC
5  source code Copyright (c) 1996-2004 by all Contributors.
6  All Rights reserved.
7
8  The contents of this file are subject to the restrictions and limitations
9  set forth in the SystemC Open Source License Version 2.4 (the "License");
10 You may not use this file except in compliance with such restrictions and
11 limitations. You may obtain instructions on how to receive a copy of the
12 License at http://www.systemc.org/. Software distributed by Contributors
13 under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF
14 ANY KIND, either express or implied. See the License for the specific
15 language governing rights and limitations under the License.
16
17 *****/
18
19
20 #include "xbus_hw_idct.h"
21
22 #ifdef __CTOS__
23 SC_MODULE_EXPORT(xbus_hw_idct);
24 #endif
25
26 xbus_hw_idct::xbus_hw_idct(sc_module_name name)
27 : sc_module(name),
28   clk{"clk"},
29   reset{"reset"},
30   ms{"ms"},
31   read{"read"},
32   size{"size"},
33   adr{"adr"},
34   data_in{"data_in"},
35   data_out{"data_out"}
36 {
37   SC_CTHREAD(run, clk.pos());
38   reset_signal_is(reset, false);
39 }
40
41 void
42 xbus_hw_idct::bus_if()
43 {

```

Step Four

You also need to set two additional design attributes, so just type in the input area (blinking >) of the *Command Window*:

```

set_attr verilog_use_indexed_part_select false /designs/xbus_hw_idct
set_attr verilog_use_wire_array false /designs/xbus_hw_idct

```

B.2 Specifying Micro-Architecture

Information about the design is now displayed in both the **Task** and **Hierarchy Windows**.

The main difference between the **Task** and **Hierarchy Windows** is the **Hierarchy Window** shows *all* of your design objects, while the **Task Window** focuses attention only on those you must resolve before being able to schedule your design.

In either window, if an object is highlighted in red, you must resolve that object before you can continue. For example, combinational loops must be eliminated, because they cannot be implemented in hardware; similarly, you must specify how arrays should be implemented.

To resolve an object, you simply click on the object in the **Task Window**, and you will be presented with various options for each object. When none of the objects is highlighted in red, and there are no yellow flags in the **Task Window**, you are ready to schedule the design.

This section has the following subsections:

- “Inlining Functions” on page B-10
- “Breaking Combinational Loops” on page B-11
- “Allocating IP” on page B-11\
- “Creating Latency Constraints” on page B-14

B.2.1 Inlining Functions

Inlining of functions can be a useful technique to control the implementation of a design in order to deliver the desired performance or required area.

Note See also “Inlining Functions” on page 8-3.

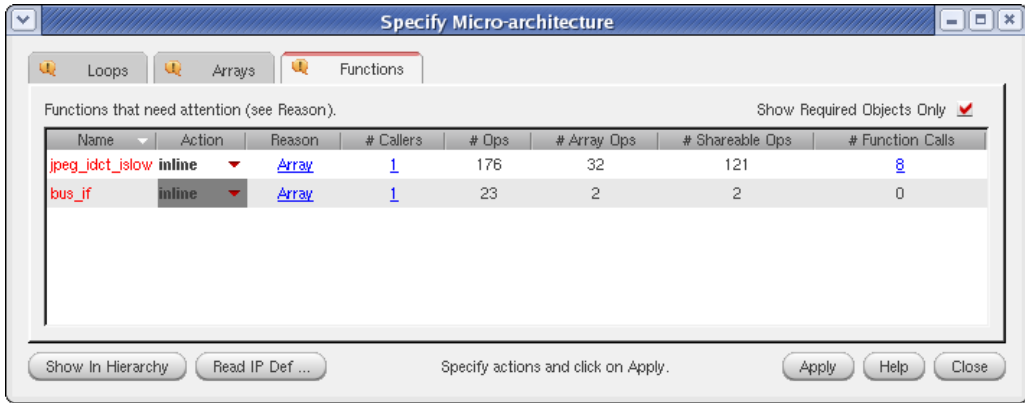
Step Five

In the **Task Window**, click **Specify Micro-architecture** (if necessary); then double-click **Functions**. You will see the Specify Micro-architecture dialog, as shown in [Figure B-9 on page B-11](#).

Select **inline** as the **Action** for each of the functions, then click **Apply**. Do not close this dialog yet.

Tip You can use the standard keyboard *Ctrl-C* and *Ctrl-V* to copy the same action for each function.

Figure B-9 Specify Micro-Architecture Dialog



B.2.2 Breaking Combinational Loops

Combinational loops must be eliminated before synthesis because they cannot be implemented in hardware.

A loop is *combinational* if at least one program execution from the top to the bottom of the loop does not include waiting for a clock edge. In some cases, a loop may not have such an execution, but CtoS will consider it combinational because it cannot prove at compile time that such an execution does not exist.

Note See also “How Combinational Loops Are Determined in CtoS” on page 8-17.

Step Six

Now, select the **Loops** tab of the **Specify Micro-architecture** dialog.

Select **break** as the **Action** for each of the loops, then click **Apply**.

B.2.3 Allocating IP

You must also allocate IP for some arrays (memories).

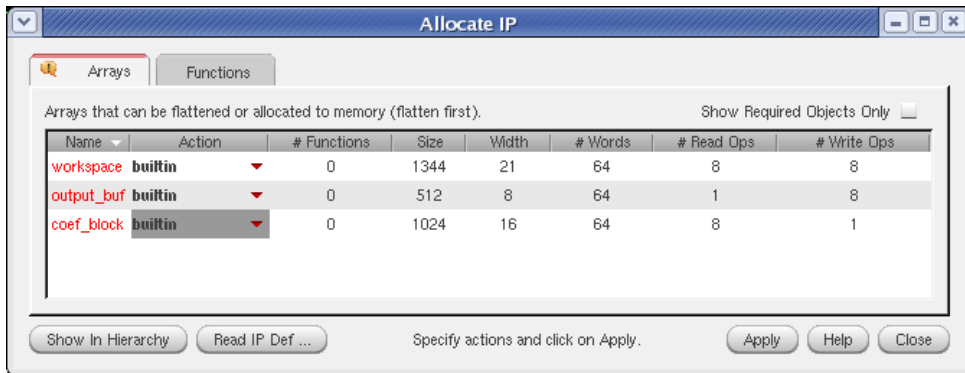
For more detail, see “Resolving Arrays (Memories)” on page 8-61.

Step Seven

In the **Task Window**, in the **Allocate IP** pane, double-click **Arrays**. In the **Allocate IP** dialog, as shown in Figure B-10 on page B-12.

Select **builtin** as the **Action** for each of the arrays, then click **Apply**.

Figure B-10 Allocate IP Dialog



Built-in RAMs with a single synchronous read port and a single write port are implemented by FPGA tools with a RAM that is available on the selected part.

There are situations in which the CtoS Verilog generated for the RAM is inadequate.

In these situations, the RAM for the FPGA design can be specified by using the allocating Vendor RAM feature of CtoS.

For FPGA designs, the **liberty_filename** element is ignored and the **verilog_filename** element is used to determine the Verilog file that contains the RAM.

Here is an example of the IP definition file for such a scenario:

```
<?xml version="1.0"?>
<ctos_ip_definitions>
  <RAMDef>
    <name>wrap64x16ram</name>
    <wrapper_filename>wrap64x16ram.v</wrapper_filename>
    <verilog_filename>my64x16ram.v</verilog_filename>
    <num_words>64</num_words>
    <width>16</width>
    <min_write_width>0</min_write_width>
    <interface_types>
      <elem>async_read</elem>
      <elem>sync_write</elem>
    </interface_types>
    <clock_posedge>true</clock_posedge>
  </RAMDef>
</ctos_ip_definitions>
```

The specified Verilog file contains the definition of the memory with the style for the FPGA vendor.

The wrapper file contains the Verilog definition, with ports that match the RAM resource in CtoS, and connects an instance of the FPGA style memory to the ports of the wrapper.

Extending the example of the previous XML file, the wrapper file is:

```
`ifndef IN_NC
`include "my64x16ram.v"
`endif

module wrap64x16ram(CLK, CE0, A0, CE1, A1, D1, WE1, Q0);
    input CLK;
    input CE0;
    input [5 : 0] A0;
    input CE1;
    input [5 : 0] A1;
    input [15 : 0] D1;
    input WE1;
    output [15 : 0] Q0;

    my64x16ram ram (.CLK(CLK), .CE0(CE0), .A0(A0), .A1(A1), .D1(D1),
        .WE1(WE1), .Q0(Q0));
endmodule
```

The **my64x16ram** module is defined in the file **my64x16ram.v** (specified with **verilog_filename**):

```
module my64x16ram(CLK, CE0, A0, CE1, A1, D1, WE1, Q0);
    input CLK;
    input CE0;
    input [5 : 0] A0;
    input CE1;
    input [5 : 0] A1;
    input [15 : 0] D1;
    input WE1;
    output [15 : 0] Q0;

    reg      [15 : 0] M [0 : 63];

    assign Q0 = {16{CE0}} & M[A0];

    always @(posedge CLK) begin
        if (CE1 & WE1)
            M[A1] <= D1;
    end
endmodule
```

The array **coeff_block** can now be allocated using the **Allocate Memory** feature, instead of **Allocate Builtin RAM**.

B.2.4 Creating Latency Constraints

You have broken the two combinational loops (**Pass1_for_begin** and **Pass2_for_begin**), by adding a state to each, but more states are required in each loop iteration for memory reads and writes. To do this, you will set a latency constraint for each of these loops.

Note See also “[Constraining Latency](#)” on page 11-12.

Step Eight

In the **Hierarchy Window**, right-click on **Pass1_for_begin** (Design -> Modules -> xbus_hw_idct -> Processes -> xbus_hw_idct_run -> Inlined Function Calls -> jpeg_idct_islow_ln81 -> Loops) and select **Constrain Latency -> Set Region Start Node**.

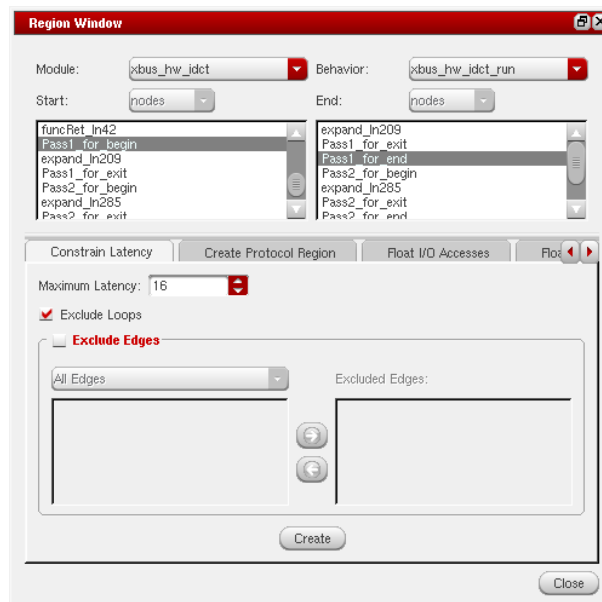
From the *Tool Bar*, select the **Region Editor** icon .

Note If this is not already displayed, select **Window -> Toolbars -> Region**.

In the **Region Window** dialog, as shown in [Figure B-10 on page B-12](#), **Pass1_for_begin** and **Pass1_for_end** should already be selected. Change **Maximum Latency** to 16, and click **Create**, but do not close the dialog yet.

Still in the **Region Window** dialog, select **Pass2_for_begin**, and **Pass2_for_end** should automatically be selected for the region end. Again, change **Maximum Latency** to 16, and click **Create** and **Close**.

Figure B-11 Region Window Dialog



B.3 Scheduling and Allocating Registers

Now, you are ready to schedule your design.

Scheduling consists of a sequence of steps that produce specified output models.

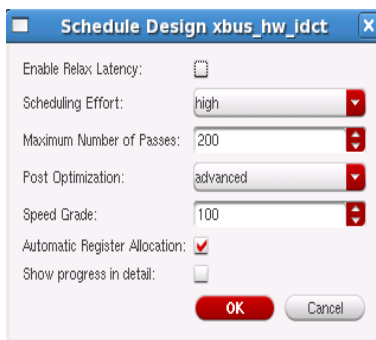
Note See also “[Scheduling](#)” on page -2.

Step Nine

Right-click on **Design (xbus_hw_idct)** under **Schedule** in the **Task Window** and select **Schedule**.

The **Schedule Design** dialog, as shown in [Figure B-12 on page B-15](#), is displayed.

Figure B-12 Schedule Design Dialog



For this tutorial, you can simply leave the defaults, and click **OK**.

CtoS first uses Encounter RTL Compiler (RC) to characterize the resources from the given library that it needs. This will take several minutes.

CtoS next tries to find a feasible schedule for all of the operations.

If it cannot find a solution with the current set of resources in the currently allowed number of cycles, it will add more resources or allow more cycles (up to the bound specified by latency constraints) and try another pass.

This process continues until CtoS finds a solution, reaches a predetermined number of passes, or cannot find any action that would improve schedulability.

Since you left **Automatic Register Allocation** checked in the **Schedule** dialog, if scheduling is successful, CtoS will automatically run the **allocate_registers** command.

B.4 Analyzing and Implementing FPGA Designs

To now analyze and implement the design, see [“Analyzing the Design” on page A-16](#) and [“Implementing the Design” on page A-20](#), noting that actual numbers will differ based on implementation target.

For FPGA designs, the area of a resource is approximated with the number of LUTs that are reported in the Xilinx device utilization summary or the ALUT number in the Altera analysis and synthesis.

Most resources are supported by the FPGA implementation tool.

You will find a table of basic resources in the description of the **create_resource** command ([“Resource Types and Required Widths for Creating Resources” on page 11-22](#)).

CtoS issues an error if an unsupported resource is encountered, and a simple rewrite of the SystemC input can often eliminate the restriction of the implementation tool.

To synthesize the design with Xilinx, type the following:

```
make -f Makefile.synth synth_xilinx_rtl XST=<xilinx_path>
```

Important You must specify the installation path to your Xilinx executable.

C Micro-Architectural Exploration Example

This appendix steps you through a simple example to illustrate a possible design flow for *exploring the design space* when you are implementing a loop.

Based on the design requirements for the throughput of the input and output operations, as well as the clock speed, implementation options with different pipelining and array structures are explored.

In addition, the feasibility and effectiveness of those options are evaluated using the comprehensive analysis capabilities of CtoS.

This appendix is organized as follows:

- [“Starting, Setting Up and Building the Design” on page C-1](#)
- [“Considering Loop Implementation” on page C-4](#)
- [“Examining Implementation Options” on page C-9](#)

C.1 Starting, Setting Up and Building the Design

To set up the design for this exercise, you will perform the following steps:

- [“Starting the CtoS GUI” on page C-2](#)
- [“Setting Up the Design” on page C-3](#)
- [“Building the Design \(Flat\)” on page C-4](#)

C.1.1 Starting the CtoS GUI

From your CtoS installation area, change to the following directory:

```
install_directory/share/ctos/examples/features/loops/pipelining
```

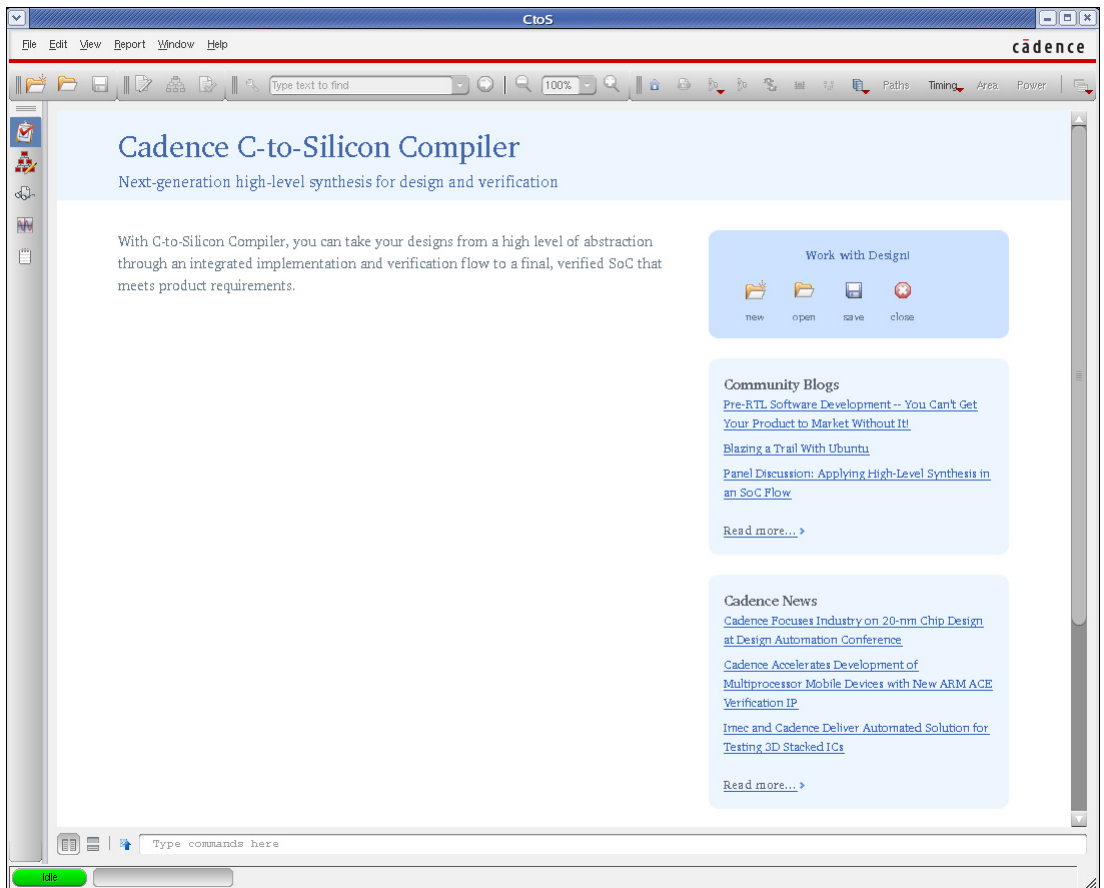
Note Before running any CtoS examples, first review “Setup for Examples” on page F-2.

Type at the command line:

```
ctosgui
```

You should see a window similar to the one shown in Figure C-1 on page C-2.

Figure C-1 Main View of the CtoS GUI



Note For more detail, see “Starting the CtoS GUI” on page 6-2.

C.1.2 Setting Up the Design

To set up a new design with the attributes necessary for the example, in the input area (blinking >) of the *Command Window*, type:

```
source ctos_setup.tcl
```

Here is a listing of the **ctos_setup.tcl** file:

```
*****
**
#                               Cadence C-to-Silicon Compiler
#
# Copyright notice: Copyright 2006-2012 Cadence Design Systems, Inc. All
# rights reserved worldwide.
#
# The code contained herein is provided to Cadence's customer and may be used
# in accordance with a previously executed license agreement between Cadence
# and that customer. This code is provided as an example for educational
# purposes and is not intended for use in a production design.
#
*****
****

# *****
# *           Setup           *
# *****
# First check whether a project is already open.
# If one is open, close it before opening the new
# project
if {[get_design] != ""} {
close_design
}

new_design DUT
set_attr source_files [list src/DUT.cpp] [get_design]
set_attr compile_flags " -w" [get_design]
set_attr top_module_path "DUT" [get_design]

set_attr tech_lib_names [list tutorial.lib] [get_design]

define_clock -name CLK -period 5000 -rise 0 -fall 2500

define_sim_config -testbench_files "tb/main.cpp tb/tb.cpp" [get_design]
define_sim_config -testbench_kind "self_checking" [get_design]
define_sim_config -makefile_name "Makefile.sim" [get_design]
define_sim_config -simulator_args "-top sc_main -Isrc -Itb" [get_design]
define_sim_config -success_msg "PASS: Simulation completed." [get_design]
```

C.1.3 Building the Design (Flat)

Because you want to flatten the design hierarchy, you will set the **build_flat** design attribute in the **Design Properties** dialog, as follows:

```
Edit -> Design Properties
```

In the **Input** tab, select **Build Flat**, and then click the **Build** button.

C.2 Considering Loop Implementation

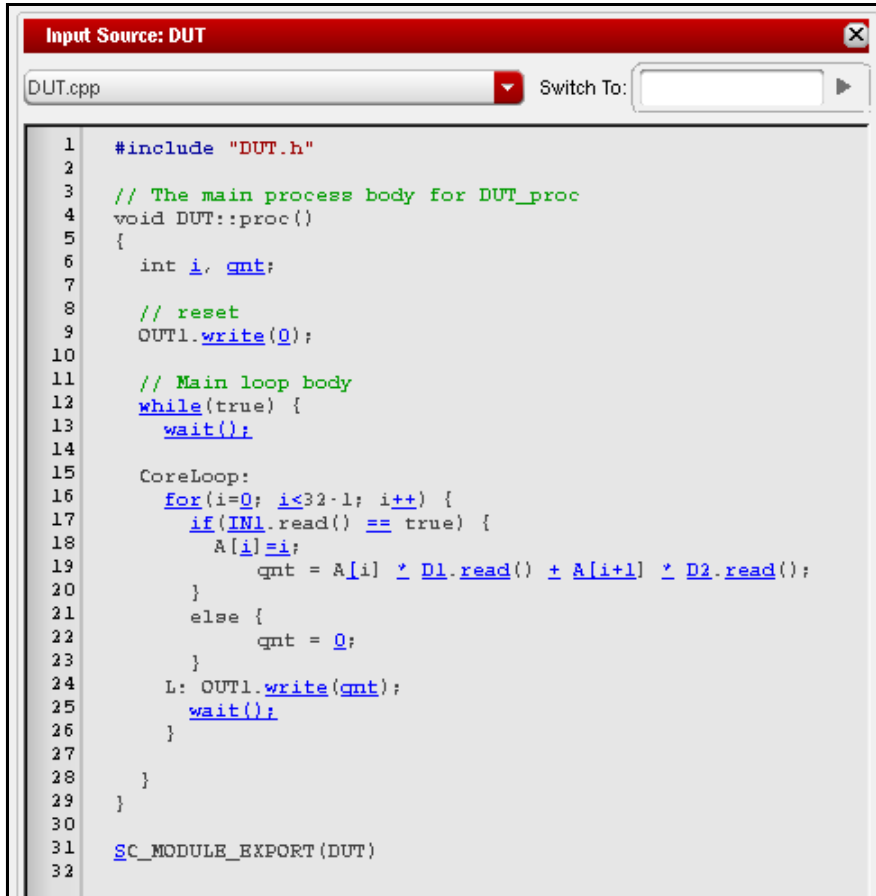
Now, you will consider the options for implementing a loop in this design.

The input SystemC source code should be displayed. If it is not, select from the Menu Bar:

```
View -> Input
```

You will now see the **Input Source** viewer, which displays the set of source files of the design, **DUT.cpp** and **DUT.h**, as shown in [Figure C-3 on page C-6](#) and [Figure C-2 on page C-5](#), respectively.

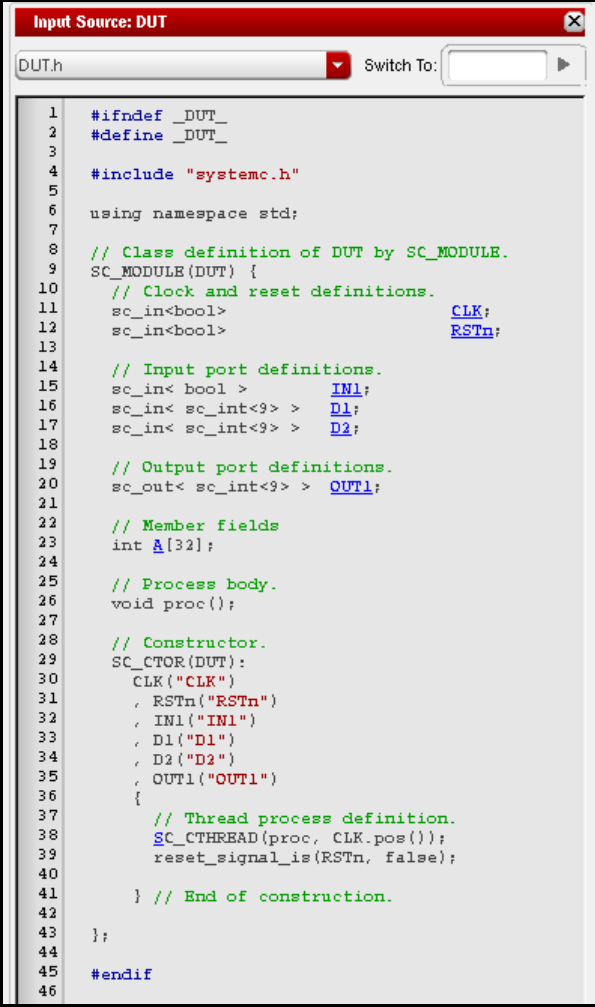
Figure C-2 Input Source Viewer Showing DUT.cpp



The screenshot shows a window titled "Input Source: DUT" with a red header bar. Below the header, there is a text field containing "DUT.cpp" and a "Switch To:" button with a dropdown arrow. The main area of the window displays C++ code for "DUT.cpp". The code is as follows:

```
1  #include "DUT.h"
2
3  // The main process body for DUT_proc
4  void DUT::proc()
5  {
6      int i, qnt;
7
8      // reset
9      OUT1.write(0);
10
11     // Main loop body
12     while(true) {
13         wait();
14
15         CoreLoop:
16         for(i=0; i<32-1; i++) {
17             if(IN1.read() == true) {
18                 A[i] = i;
19                 qnt = A[i] * D1.read() + A[i+1] * D2.read();
20             }
21             else {
22                 qnt = 0;
23             }
24             L: OUT1.write(qnt);
25             wait();
26         }
27     }
28 }
29
30
31 SC_MODULE_EXPORT(DUT)
32
```

Figure C-3 Input Source Viewer Showing DUT.h



```
1  #ifndef _DUT_
2  #define _DUT_
3
4  #include "systemc.h"
5
6  using namespace std;
7
8  // Class definition of DUT by SC_MODULE.
9  SC_MODULE(DUT) {
10     // Clock and reset definitions.
11     sc_in<bool> CLK;
12     sc_in<bool> RSTn;
13
14     // Input port definitions.
15     sc_in< bool > IN1;
16     sc_in< sc_int<9> > D1;
17     sc_in< sc_int<9> > D2;
18
19     // Output port definitions.
20     sc_out< sc_int<9> > OUT1;
21
22     // Member fields
23     int A[32];
24
25     // Process body.
26     void proc();
27
28     // Constructor.
29     SC_CTOR(DUT):
30         CLK("CLK")
31         , RSTn("RSTn")
32         , IN1("IN1")
33         , D1("D1")
34         , D2("D2")
35         , OUT1("OUT1")
36     {
37         // Thread process definition.
38         SC_CTHREAD(proc, CLK.pos());
39         reset_signal_is(RSTn, false);
40
41     } // End of construction.
42
43 };
44
45 #endif
46
```

In the **DUT.cpp** tab, note the **for** loop starting at line 16 (it is highlighted and underlined in blue):

```
for(i=0; i<32-1; i++) {
    if(IN1.read() == true) {
        A[i]=i;
        qnt = A[i] * D1.read() + A[i+1] * D2.read();
    }
    else {
        qnt = 0;
    }
    L: OUT1.write(qnt);
    wait();
}
```

You will now consider possible implementations of this **for** loop, keeping in mind that:

- Design requirements specify that the throughput of this loop be no more than two clock cycles, that is, data must be produced to the output port **OUT1** either every clock cycle or every other clock cycle.
- To complete the two operations to read elements of an array in this loop in a single clock cycle, you must either:
 - use a built-in RAM with two read ports, or
 - flatten the array to implement it with registers

Using a built-in RAM to implement this array provides the following options to implement the loop:

- “Option 1: RAM with 2 Read Ports, No Pipeline, Latency 1” on page C-7
- “Option 2: RAM with 2 Read Ports, No Pipeline, Latency 2” on page C-8
- “Option 3: RAM with 2 Read Ports, Pipeline” on page C-8
- “Option 4: RAM with 1 Read Port, No Pipeline, Latency 2” on page C-8
- “Option 5: RAM with 1 Read Port, Pipeline” on page C-8

These options will be briefly reviewed below, and then evaluated in more detail in “[Examining Implementation Options](#)” on page C-9.

C.2.1 Option 1: RAM with 2 Read Ports, No Pipeline, Latency 1

With option 1, you would use a RAM with two read ports and would not pipeline the loop, nor set any latency constraints. Therefore, all operations in a single iteration of the loop would be implemented in a single clock cycle. This would provide the throughput of 1, meeting the design requirements.

The potential problem with this option is whether the timing of the clock period can be met when implementing all operations of a single iteration in a single clock cycle.

C.2.2 Option 2: RAM with 2 Read Ports, No Pipeline, Latency 2

Option 2 is similar to option 1, except you would set a latency constraint for the loop, so it can take two clock cycles to implement the operations of a single iteration of the loop. This would result in a throughput of 2, which still meets the design requirements.

Since two clock cycles would be used, you could still use a single multiplier to implement the two multiplication operations in the loop, and by doing so, reduce the area of the resulting hardware, compared to using a dedicated multiplier for each multiplication.

C.2.3 Option 3: RAM with 2 Read Ports, Pipeline

With option 3, you would still use a RAM with two read ports, but you would pipeline the loop. Since there would be two ports to read two elements of the array in a single clock cycle, the initiation interval of the pipeline would be only one clock cycle, which would provide a throughput of 1.

Since the initiation interval would be 1, no resource instances would be shared to implement multiple operations.

C.2.4 Option 4: RAM with 1 Read Port, No Pipeline, Latency 2

Option 4 is similar to option 2, except you would use a RAM with only one read port. This means you could read only one element of the array in a single clock cycle and would therefore need to set a latency constraint so two clock cycles would be used to implement the operations in a single iteration of the loop. This option would provide a throughput of 2.

As with option 2, this option would provide a possibility of sharing a single multiplier to implement the two multiplications in the loop. Also, the size of the RAM might be smaller than the RAM with two read ports.

C.2.5 Option 5: RAM with 1 Read Port, Pipeline

With option 5, you would pipeline the loop, but would use a RAM with a single read port. This means you would need at least two clock cycles to read two elements of the array, and the initiation interval of the pipeline would therefore need to be set to two clock cycles. The throughput of the resulting implementation would be 2.

C.3 Examining Implementation Options

Now, you will examine the implementation options for the loop defined in the previous section, [“Considering Loop Implementation” on page C-4](#).

Quickly assessing options 1 and 2 precludes them from examination at this point, for these reasons:

- You should examine option 4 before examining option 2, because option 4 uses a RAM with a single read port, which would likely result in a smaller area than option 2.
- You should examine option 4 before examining option 1, because option 1 would be a viable option only if option 4 could meet the timing requirements of the clock period since option 1 would implement more operations than option 4 in a single clock cycle.

Now, you will explore the remaining options:

- [“Implementing Option 3: RAM with 2 Read Ports, Pipeline” on page C-9](#)
- [“Implementing Option 4: RAM with 1 Read Port, No Pipeline, Latency 2” on page C-18](#)
- [“Implementing Option 5: RAM with 1 Read Port, Pipeline” on page C-21](#)

C.3.1 Implementing Option 3: RAM with 2 Read Ports, Pipeline

As previously mentioned, to implement the loop in option 3, you would use a RAM with two read ports and apply pipelining to the loop.

Since there would be two ports to read two elements of the array in a single clock cycle, the initiation interval of the pipeline could be one clock cycle, which would provide a throughput of 1. And since the initiation interval would be 1, no resource instances would be shared to implement multiple operations.

Note See [“Pipelining Loops” on page 8-26](#) for more information.

To synthesize the design for option 3, you will follow these steps:

- [“Specifying the Pipelining Constraint for the Loop \(Option 3\)” on page C-10](#)
- [“Binding Arrays \(Option 3\)” on page C-13](#)
- [“Scheduling and Allocating Registers \(Option 3\)” on page C-14](#)
- [“Analyzing the Design \(Option 3\)” on page C-14](#)

C.3.1.1 Specifying the Pipelining Constraint for the Loop (Option 3)

Under the **DUT.cpp** tab of the **Input Source** viewer, hover on **for** at line 16 (highlighted in blue).

When the corresponding list is displayed, right-click on **CoreLoop_for_begin LOOP** and select **Pipeline** from the context menu.

You will see the **Pipeline Loop** dialog, as shown in Figure C-4 on page C-10.

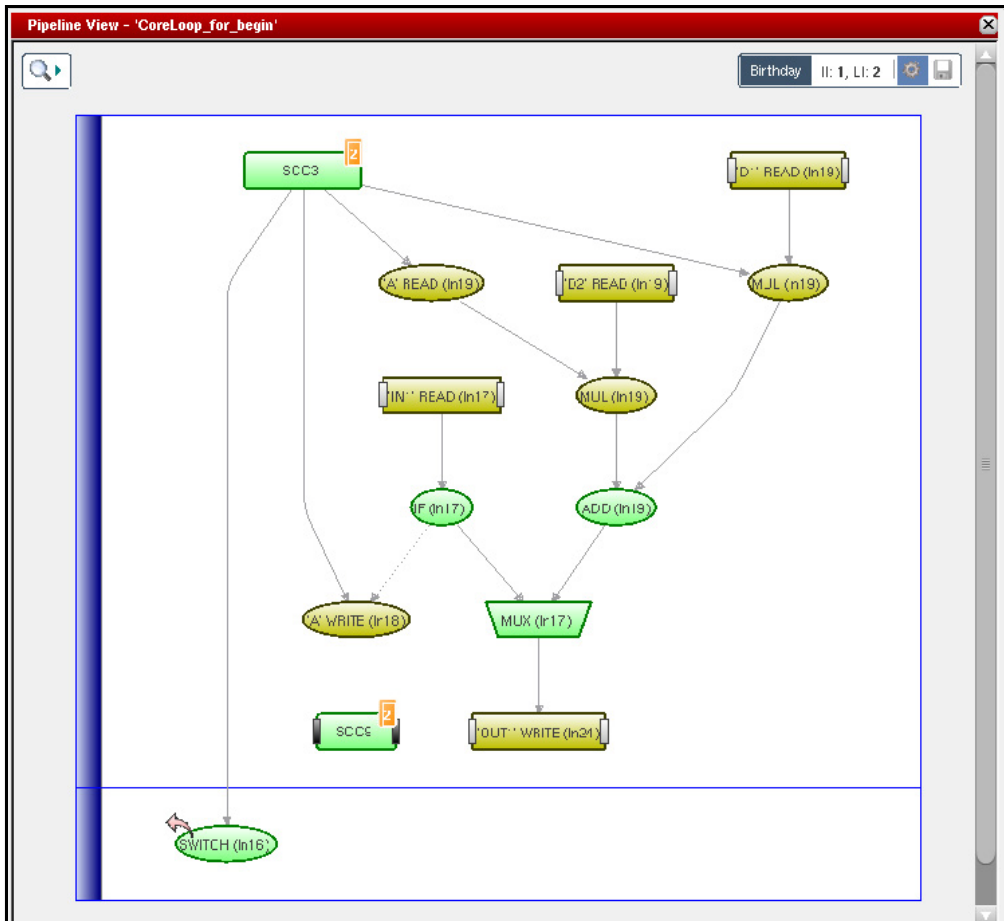
Note See also “[Pipelining Loops](#)” on page 8-26.

Figure C-4 Pipeline Loop Dialog



For this example (option 3), just leave the defaults, and click **OK**. You will then see the **Pipeline View** displayed, as shown in Figure C-5 on page C-11.

Figure C-5 Pipeline View - CoreLoop_for_begin



In the **Pipeline View**, you have the following helpful guides:

- The control palette at the top right provides:
 - the scheduling state, which is updated as the behavior goes through different stages: **Birthday**, **Scheduled**, or **Failed Schedule** (partial).
 - the initiation interval (**II**) and latency interval (**LI**)

- The main part of the viewer shows the pipeline diagram, in which rounded rectangles correspond to SCCs (*strongly connected components*; see “SCC” on page N-14), ovals correspond to individual ops, nodes with dark bars on either side are fixed and cannot be moved, and nodes with light bars are I/Os that can be moved by a designer, but will not be moved by the CtoS scheduler. (Note that, in this example, the operations that involve the incrementation of the loop index **i** form an SCC set.)

You can drag ops to move them to different clock cycles of the pipeline. This means that you are setting a constraint for those ops to be scheduled in those clock cycles in the pipeline. When you are dragging any of these rectangles, you will see *arcs* between the rectangle and other rectangles.

The arcs show the dependency among the operations. An arc will be highlighted in red if its order is reversed, that is, if between the two operations connected by the arc, the one to be executed earlier is positioned in a later clock cycle. This means that such a schedule is infeasible and will fail.

- The **Stage** and **Phase** are displayed in a box near the bottom of the main area that is displayed when you change either of these characteristics.
- The  icon configures the filtering of the pipeline graph, bringing up the **Pipelining and Filtering** dialog, as shown in [Figure 8-13 on page 8-36](#). This icon blinks when the pipeline graph is filtered. Hovering over it, you see all the filters in effect.
- Any dragging of rectangles is not final until you select the  icon, which becomes enabled when there are unsaved pipeline changes.

For this example (option 3), drag the rectangle for the output operation **write_DUT_OUT1_In24** from stage 1 to stage 2 (move it down past the horizontal line, and look to see that **Stage 1, Phase 1** goes to **Stage 2, Phase 1**).

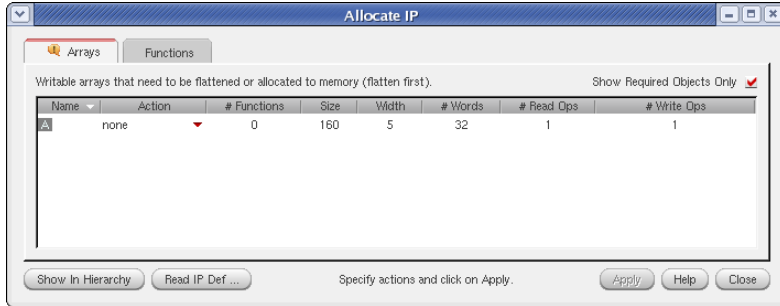
Important Be sure to select the  icon – moving the boxes does not save the changes.

You have thus issued a pipeline constraint that this loop is to be pipelined with initiation and latency intervals of 1 and 2, respectively, and the operation to write to **write_DUT_OUT1_In24** is to be scheduled in the second stage of the pipeline.

C.3.1.2 Binding Arrays (Option 3)

In the **Task Window**, double-click the red **Arrays** to get to the **Allocate IP** dialog.

Figure C-6 Allocate IP Dialog

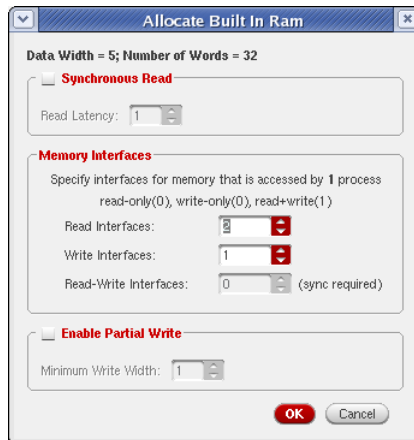


Right-click on **A**, and select **Allocate Built In RAM**. You will see the **Allocate Built In Ram** dialog, as shown in [Figure C-7 on page C-13](#), where you can choose the number of read interfaces.

Note See also “[Allocating Built-In RAM](#)” on [page 9-3](#).

For this example (option 3), select **2** for **Read Interfaces**. Leave all the other defaults (especially make sure **Synchronous Read** is unchecked, as you want to use asynchronous read ports for this RAM, and leave **Enable Partial Write** unchecked), and click **OK**.

Figure C-7 Allocate Built In Ram Dialog



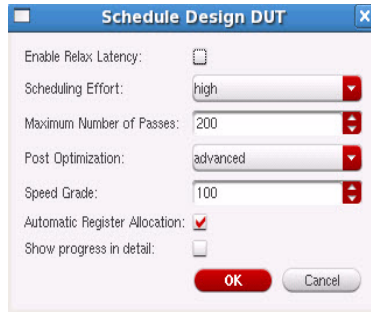
C.3.1.3 Scheduling and Allocating Registers (Option 3)

Select from the Menu Bar:

```
Edit -> Schedule
```

You will see the **Schedule** dialog, as shown in [Figure C-8 on page C-14](#).

Figure C-8 Schedule Dialog



For this example (option 3), just leave the defaults, and click **OK**. This will start scheduling, and when the CtoS scheduler has completed, you will get a message that behavior **DUT_proc** has been successfully scheduled, and registers will have been allocated.

C.3.1.4 Analyzing the Design (Option 3)

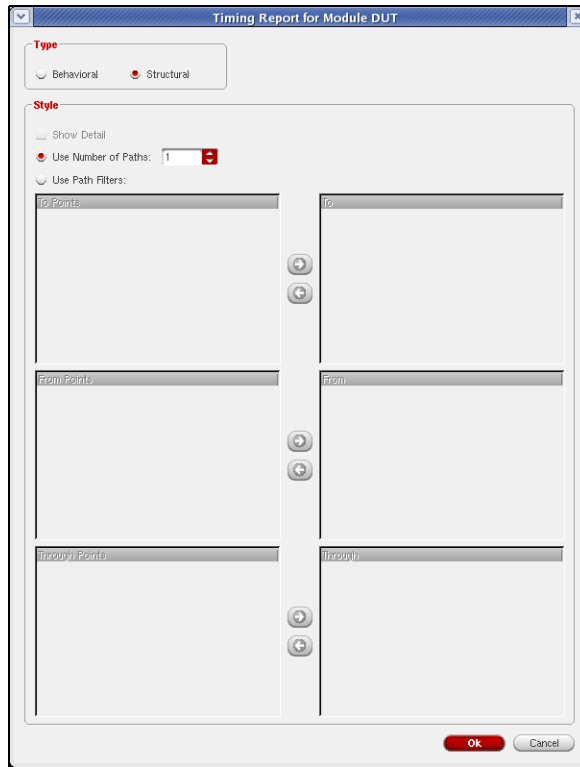
First, check that the resulting design meets the timing set by the clock period.

Select from the Menu Bar:

```
Report -> Timing -> Report
```

You will see the **Timing Report** dialog, as shown in [Figure C-9 on page C-15](#), which lets you select various options for the timing report.

Figure C-9 Timing Report Dialog



For this example (option 3), just leave the defaults, and click **OK** to start the timing analysis.

You will see a new tab, **Structural Timing Report for Module DUT**, as shown in [Figure C-10 on page C-16](#), in which the most critical path is displayed in terms of a sequence of resource instances and their input and output pins involved in the path.

The last column shows the arrival time at each pin involved in the path, and the arrival time at the last row indicates the time of the path, that is, the arrival time of the output pin of the last resource instance of the path.

Figure C-10 Structural Timing Report

Structural Timing Report for Module DUT					
Timing report for most critical path in module DUT					
	Instance Terminal	Master	Fanout	Delay (ps)	Arrival (ps)
1	read_DUT_D2_In19_Z_0/Q[0]	flipflop_9	1	148	148
2	umul_9x9x5/A[0]	umul_9x9x5		0	148
3	umul_9x9x5/Z[0]	umul_9x9x5	1	1692	1840
4	add_9_0/B[0]	add_9		0	1840
5	add_9_0/Z[0]	add_9	1	1024	2864
6	mux_qnt_In17/A_00[0]	mux_2_9		0	2864
7	mux_qnt_In17/Z[0]	mux_2_9	1	245	3109
8	OUT1_mux/A_00[0]	mux_2_9		0	3109
9	OUT1_mux/Z[0]	mux_2_9	1	245	3354
10	OUT1_reg/D[0]	flipflop_9_r_0		0	3354
11	OUT1_reg/CLK	flipflop_9_r_0		313	3667

Next, you will check the total area and also the area of the RAM you allocated.

To see the total area, select the **Summary Report** icon (bottom-most icon on the left border of CtoS GUI window, which looks like a notepad), or select from the Menu Bar:

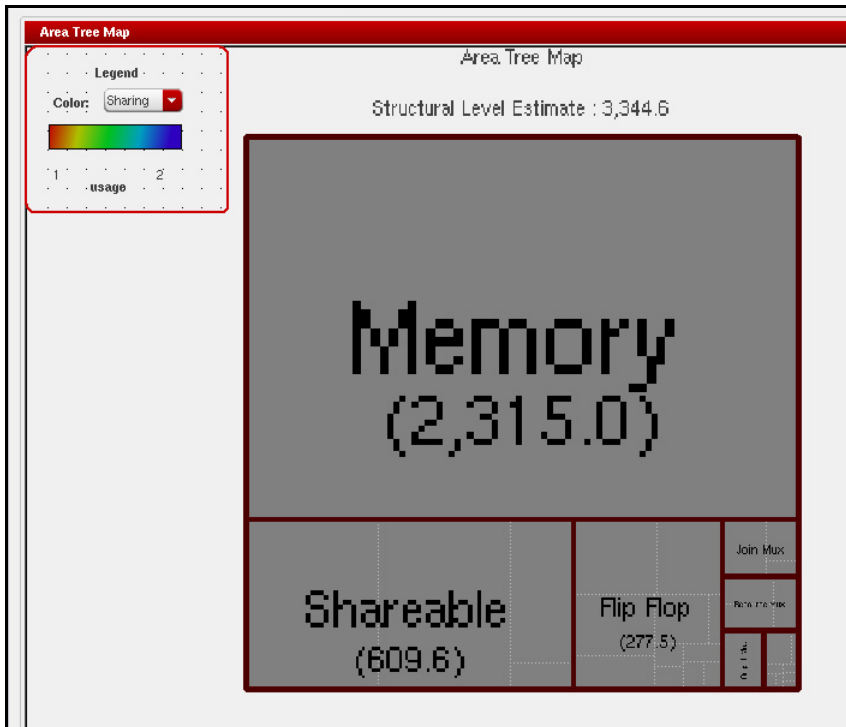
Window -> Summary Report

You will see a new window, **Summary Report**, as shown in [Figure C-11 on page C-16](#), and you can see the total area of the design in the entry, **Area**, under **Overview**.

Figure C-11 Summary Report Window

Summary Report	
Name	Count
Overview	
Name	DUT
'CLK' Clock Period (ps)	4,000
Minimum Slack (ps)	333
Area	3,344.6
Power	N/A
Behavior	
Modules	1
Processes	1
Functions	0
Arrays	1
Loops	1
States	1
Edges	1
Total Ops	27
Shareable Ops	6
Values	113
Tags	11
Structure	
Memories (bits)	160
Flip Flops (bits)	43
Muxes (bits)	40
Shareable Resources	4
Non Shareable Resources	14
Input Terminals (bits)	21
Output Terminals (bits)	9
Nets (bits)	165
Instance Terminals (bits)	398

Figure C-12 Area Tree Map - Expanded



To check the area of the RAM, and also any other resource instance, you can use the **Area Tree Map**.

Select from the Menu Bar:

Report -> Area

You will see the **Area Tree Map**, which shows the area of the **DUT_proc**.

To see the area of the RAM, right-click and select **Expand**. You will see the area of the RAM (**Memory**) and other components, as shown in [Figure C-12 on page C-17](#).

You can see that the area of the RAM is by far the dominant area in this design.

Note See “[Reporting Power and Area Using the Tree Map](#)” on page 13-29 for a more detailed explanation of the CtoS GUI tree maps.

C.3.2 Implementing Option 4: RAM with 1 Read Port, No Pipeline, Latency 2

As previously mentioned, to implement the loop in option 4, you would use a RAM with only one read port and set a latency constraint (and not apply pipelining to the loop).

With this option only one element of the array could be read in a single clock cycle and a latency constraint would need to be set so two clock cycles would be used to implement the operations in a single iteration of the loop.

This option provides a throughput of 2 and provides a possibility of sharing a single multiplier to implement the two multiplications in the loop. In addition, the size of the RAM might be smaller than the RAM with two read ports.

Important Before starting with option 4, if you have not done so already, close the design you used for option 3 by selecting:

```
File -> Close Design
```

Then, to again set up the design, in the input area (blinking `>`) of the *Command Window*, type:

```
source ctos_setup.tcl
```

Again, because you want to flatten the design hierarchy, you will set the **build_flat** design attribute in the **Design Properties** dialog, as follows:

```
Edit -> Design Properties
```

In the **Input** tab, select **Build Flat**, and then click the **Build** button.

To synthesize the design for option 4, you will follow these steps:

- [“Setting a Latency Constraint \(Option 4\)” on page C-19](#)
- [“Binding Arrays, Scheduling, Allocating Registers \(Option 4\)” on page C-20](#)
- [“Analyzing the Design \(Option 4\)” on page C-21](#)

C.3.2.1 Setting a Latency Constraint (Option 4)

For option 4, you must set a latency constraint for the loop.

Note See “[Constraining Latency](#)” on page 11-12 more about setting latency constraints, in general.

When you set this latency constraint, operations in a single iteration of the loop will be executed with a maximum of two clock cycles, that is, CtoS may insert one more **wait** statement somewhere inside the loop. More specifically, the part of the loop in which an additional **wait** statement may be inserted is between the top of the loop and just before the operation of writing to the output port **OUT1**.

Important Even if the additional **wait** statement is inserted after the **write** operation to **OUT1**, none of the operations in the loop body is scheduled in the second clock cycle. For example, suppose that the additional **wait** statement is put at the end of the loop, as shown in the following code sample:

```
for(i=0; i<32-1; i++) {
    if(IN1.read() == true) {
        qnt = A[i] * D1.read() + A[i+1] * D2.read();
    }
    else {
        qnt = 0;
    }
L:  OUT1.write(qnt);
    wait();           // this is the original wait statement.
    wait();           // this is the additional wait statement.
}
```

In this case, CtoS would not schedule any of the operations in the loop body between the first and second **wait** statements. This is due to the fact that CtoS will schedule operations to output ports in the original location shown in the code, that is, **OUT1.write()** will be in the first clock cycle. Since the value of **qnt** must be computed before **OUT1.write()**, all operations in the loop body must also be scheduled in the first clock cycle.

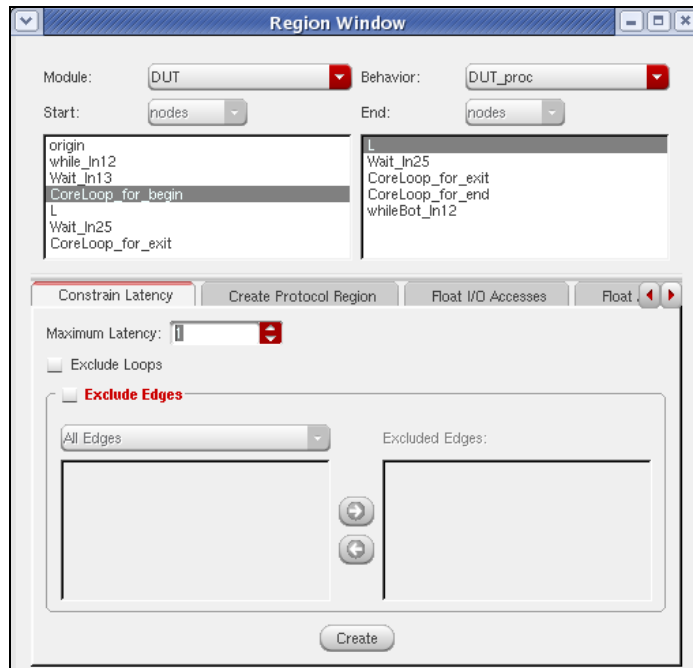
Therefore, in setting a latency constraint, it is important to eliminate the possibility that CtoS will insert the additional **wait** statement as shown in the previous code sample. This can be done by setting the constraint between the top of the loop and the label **L** and instructing CtoS to insert at most one **wait** statement in this region.

To set a latency constraint, open the **Input Source**, by selecting from the Menu Bar:

View -> Input

Hover over **for** at line 16 (which is highlighted in blue), right-click on **CoreLoop_for_begin LOOP**, then **Constrain Latency -> Set Region Start Node**. This will set the top of the loop as the starting point of the region.

Bring up the **Region Window**, as shown in [Figure C-13 on page C-20](#) by selecting from the *Tool Bar* the **Region Editor** icon . (If it is not displayed, select *Window -> Toolbars -> Region*.)

Figure C-13 Region Window

In the **Region Window**, select **L** as the **End** node.

In **Maximum Latency**, scroll to integer **1** (or just type it in) to indicate that CtoS may insert at most one **wait** statement in this region. Click **Create** and then **Close** to finalize this constraint.

Important Be sure to click **Create** before clicking **Close** to finalize this constraint. Just clicking **Close** will not set the constraint.

C.3.2.2 Binding Arrays, Scheduling, Allocating Registers (Option 4)

As with option 3, you will now specify a binding for the array **A** to a built-in RAM. Follow the instructions in [“Binding Arrays \(Option 3\)” on page C-13](#) to set this constraint. Remember that unlike option 3, for option 4, you need to leave the default of **1** for **Read Interfaces** (so you could just select **builtin** under **Action** – this provides the default behavior).

Tip This was previously shown in [“Allocate Built In Ram Dialog” on page C-13](#).

Next, perform scheduling and register allocation as in [“Scheduling and Allocating Registers \(Option 3\)” on page C-14](#).

Tip Remember to just leave the defaults in the **Schedule** dialog.

C.3.2.3 Analyzing the Design (Option 4)

As with option 3, first check to see if the timing requirement is met with option 4.

Open the **Timing Report** by selecting from the Menu Bar:

```
Report -> Timing -> Report
```

Then, **Ok** the **Timing Report** dialog.

Check the delay of the most critical paths of the resulting implementation.

Open the **Summary report** by selecting from the Menu Bar:

```
Window -> Summary Report (or just select the icon at the left)
```

Open the **Area Tree Map** by selecting from the Menu Bar:

```
Report -> Area
```

Right-click and select **Expand** to see the area of the RAM (**Memory**).

C.3.3 Implementing Option 5: RAM with 1 Read Port, Pipeline

As previously mentioned, to implement the loop in option 5, you would use a RAM with one read port and apply pipelining to the loop.

With this option, you would need at least two clock cycles to read two elements of the array, and the initiation interval of the pipeline therefore must be set to two clock cycles. The throughput of the resulting implementation would be 2.

Important Again, before starting with option 5, if you have not done so already, close the design you used for option 4 by selecting:

```
File -> Close Design
```

Then, to again set up the design, in the input area (blinking **>**) of the *Command Window*, type:

```
source ctos_setup.tcl
```

Again, set the **build_flat** design attribute in the **Design Properties** dialog, as follows:

```
Edit -> Design Properties
```

In the **Input** tab, select **Build Flat**, and then click the **Build** button.

To synthesize the design for option 5, you will follow these steps:

- [“Specifying the Pipelining Constraint for the Loop \(Option 5\)” on page C-22](#)
- [“Binding Arrays, Scheduling, Allocating Registers \(Option 5\)” on page C-23](#)

- “Analyzing the Design (Option 5)” on page C-23

C.3.3.1 Specifying the Pipelining Constraint for the Loop (Option 5)

Open the **Pipeline Loop** dialog, as described in “[Specifying the Pipelining Constraint for the Loop \(Option 3\)](#)” on page C-10 (quick reminder: **View** -> **Input**, hover over **for** loop at line 16 in blue, right-click on **CoreLoop_for_begin LOOP**, select **Pipeline**).

Unlike option 3, since only a single read port in the RAM is used, and there are two operations in the loop to read elements of the array **A**, at least two clock cycles are required to execute these operations using the RAM.

Therefore, you will set the initiation interval of the pipeline to two.

Scroll to or just type an integer **2** in the field for **Initiation Interval** in the **Pipeline Loop** dialog.

You will see that the minimum latency interval is automatically expanded to **3**, and you can leave it at that. Click **OK** on the dialog, and the **Pipeline View** will be displayed.

Now, again locate the rectangle **write_DUT_OUT1_ln24**. Move this rectangle to the third clock cycle, that is, down two levels.

This is necessary to specify the constraint that this operation must be scheduled at the last cycle of the pipeline; without it, CtoS will try to schedule it in the original stage, which is the first clock cycle of the pipeline.

Similarly, also find an oval **memread_DUT_A_ln19**. Move this rectangle to the second clock cycle, that is, the second phase of the first stage of the pipeline. You can see how this should look in “[Pipeline View - with Constraints](#)” on page C-23

Important Be sure to select the  icon – moving the boxes does not save the changes.

D CtoS Object Reference

In the CtoS Tcl environment, the CtoS database is exposed through object string identifiers (*object_ids*) and *object attributes*. This appendix describes these all of these objects and their attributes, including how to identify them for CtoS commands.

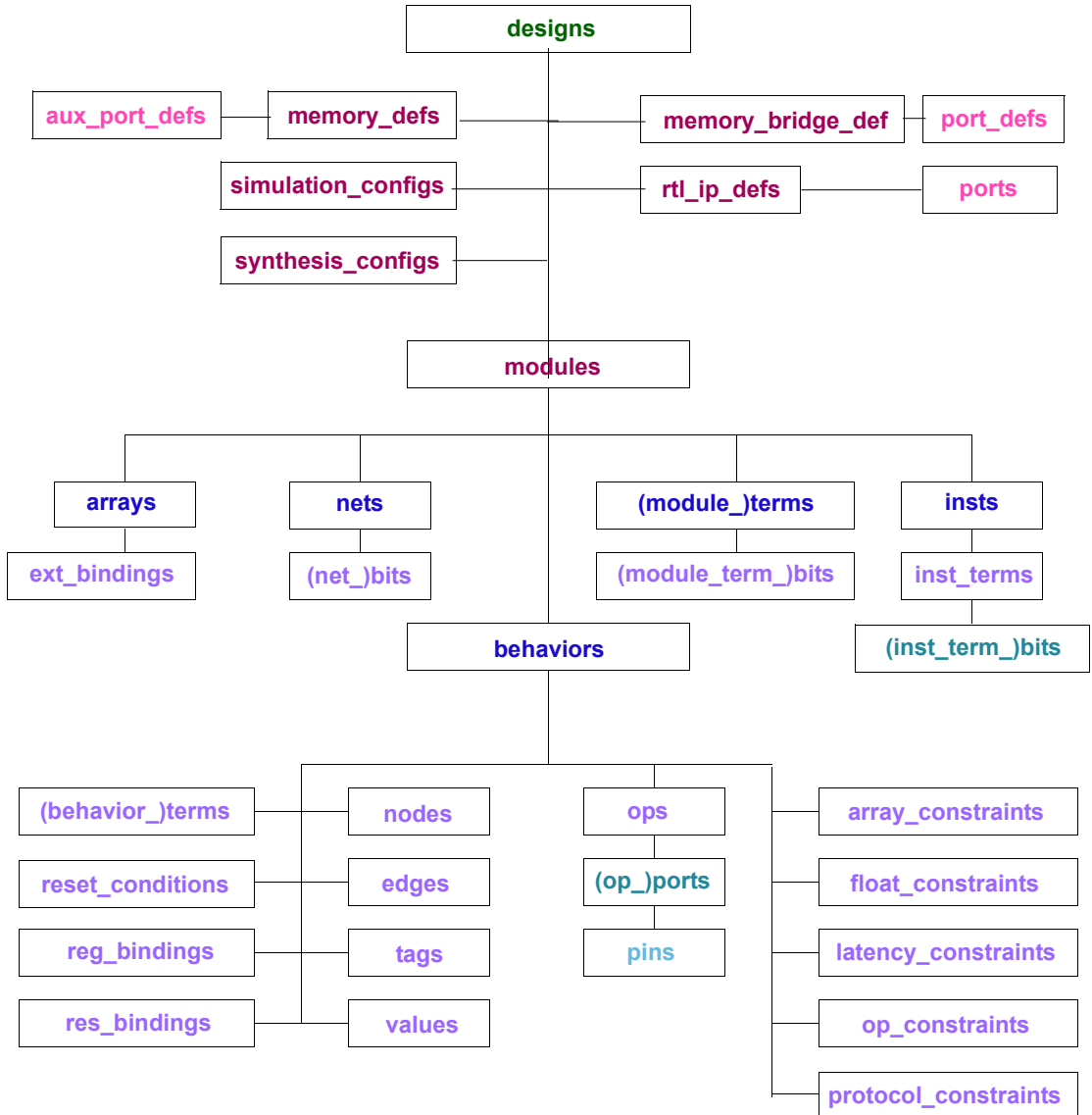
- “Design Hierarchy (Graphical Depiction)” on page D-3
- “Design Hierarchy (Tabular Depiction)” on page D-4
- “Object Identifiers (object_id’s)” on page D-5
- “Object Attributes” on page D-5
- “Identifying Objects for Commands” on page D-6
- “Design Object Attributes (Designs)” on page D-9
- “Memory Bridge Definition Object Attributes (memory_bridge_defs)” on page D-29
- “Memory Bridge Port Def Object Attributes (memory_bridge_port_defs)” on page D-29
- “Memory Definition Object Attributes (memory_defs)” on page D-30
- “Memory Definition Auxiliary Port Object Attributes (aux_port_defs)” on page D-33
- “Module Object Attributes (Modules)” on page D-34
- “Array Object Attributes (Arrays)” on page D-36
- “External Array Binding Object Attributes (external_array_binding)” on page D-39
- “Behavior Object Attributes (Behaviors)” on page D-41
- “Array Constraint Object Attributes (array_constraints)” on page D-46
- “Edge Object Attributes (Edges)” on page D-46
- “Floating Constraint Object Attributes (float_constraints)” on page D-48

- “Latency Constraint Object Attributes (latency_constraints)” on page D-49
- “Node Object Attributes (Nodes)” on page D-50
- “Operation Constraint Object Attributes (op_constraints)” on page D-56
- “Operation Object Attributes (Ops)” on page D-57
- “Operation Ports Object Attributes [(op_)ports]” on page D-60
- “Directive Object Attributes” on page D-61
- “Pin Object Attributes (Pins)” on page D-62
- “Protocol Constraint Object Attributes (protocol_constraints)” on page D-63
- “Register Binding Object Attributes (reg_bindings)” on page D-63
- “Reset Condition Object Attributes (reset_conditions)” on page D-65
- “Tag Object Attributes (Tags)” on page D-66
- “Behavior Terminal Object Attributes [(behavior_)terms]” on page D-67
- “Value Object Attributes (Values)” on page D-68
- “Instance Object Attributes (Insts)” on page D-69
- “Instance Terminal Object Attributes (inst_terms)” on page D-73
- “Instance Terminal Bits Object Attributes [(inst_term_)bits]” on page D-74
- “Net Object Attributes (Nets)” on page D-76
- “Net Bit Object Attributes [(net_)bits]” on page D-78
- “Module Terminal Object Attributes [(module_)terms]” on page D-79
- “Module Terminal Bits Object Attributes [(module_term_)bits]” on page D-81
- “RTL IP Definition Object Attributes (rtl_ip_defs)” on page D-83
- “RTL IP Definition Port Object Attributes [(rtl_ip_def_)ports]” on page D-84
- “Default Simulation Configuration Object Attributes (simulation_configs)” on page D-85
- “Default Synthesis Configuration Object Attributes (synthesis_configs)” on page D-86

D.1 Design Hierarchy (Graphical Depiction)

Here is a graphical depiction of the CtoS design hierarchy:

Figure D-1 Graphical Depiction of the CtoS Design Hierarchy



D.2 Design Hierarchy (Tabular Depiction)

Here is a tabular depiction of the CtoS design hierarchy:

designs					
	memory_bridge_defs				
		memory_bridge_port_defs			
	memory_defs				
		aux_port_defs			
	modules				
		arrays			
			external_array_bindings		
		behaviors			
			array_constraints		
			edges		
			float_constraints		
			latency_constraints		
			nodes		
			op_constraints		
			ops		
				(op_)ports	
					pins
			protocol_constraints		
			reg_bindings		
			res_bindings		
			reset_conditions		
			tags		
			(behavior_)s		
			values		
		insts			
			inst_s		
				(inst_term_)bits	
		nets			
			(net_)bits		
		(module_)terms			
			(module_term_)bits		
	rtl_ip_defs				
		(rtl_ip_def_)ports			
	simulation_configs				
	synthesis_configs				

D.3 Object Identifiers (object_id's)

An *object identifier* (*object_id*) is a hierarchical name composed of object scope names and object identifier names, separated by / (forward slashes).

The format of an *object_id* starts with a leading / to denote the root for the identification path, followed by alternating scope names and object names, as follows:

/root_scope_name/object_name/child_scope_name/object_name/child_scope_name/object_name ...

Any *object_id* starting with / is considered an *absolute* path, and any *object_id* not starting with / is considered a *relative* path.

The root object scope name in CtoS is */designs*.

All CtoS commands accept relative and absolute path *object_id*'s as arguments.

When a relative path *object_id* is used, CtoS will combine it with the current object path before converting it to a CtoS database object.

The special string *..* can be used in the *object_id* to refer to the previous scope in the object path.

The CtoS commands **cd**, **ls**, and **pwd** can be used to set and list child scopes and query the current object path.

Note See “**cd**” on page E-31, “**ls**” on page E-85, and “**pwd**” on page E-96.

D.4 Object Attributes

An object attribute is identified by its name and contains either a single value or a set of values.

Types include *integer* (which can be *unsigned*, *positive*, etc.), *double*, *boolean*, *string*, *list of strings*, etc.

Attributes are defined by the *database object*, which you cannot create or delete.

A *database object* may define an attribute as read-only (that is, its value *cannot* be changed) or read/write (that is, its value *can* be changed).

The Tcl commands **get_attr**, **list_attr**, and **set_attr** can be used to retrieve, list, and set values for attributes, respectively.

Note See “**get_attr**” on page E-73, “**list_attr**” on page E-82, and “**set_attr**” on page E-140.

D.5 Identifying Objects for Commands

Many CtoS Tcl commands take one or more *object_id*'s as arguments.

The simplest way to get the *object_id* for an object in your design is in the CtoS GUI. In most views, you can simply right-click the object and select **Print Object Id** (see “Getting Object IDs” on page 6-47).

In addition, [Table D-1 on page D-7](#) provides paths to all CtoS objects.

In the left column are all of the types of objects in CtoS. In the right column is each one's uniquely defined location, which corresponds to a virtual directory.

For example, to specify the *object_id* of an operation *op1* as the argument of a command **cmd**, you can specify the entire path in the virtual directory from the root to the object ID *op1*, as follows:

```
cmd -op /designs/my_design/modules/my_module/behaviors/my_process/ops/op1
```

You can use the **set** command in Tcl to set this path to a variable, and refer to this variable in CtoS Tcl commands. For example:

```
set op1 /designs/my_design/modules/my_module/behaviors/my_process/ops/op1
cmd -op $op1
```

Or you can browse to a parent directory of the object, and use a relative path, supplying only the necessary subset of the path to reference the object:

```
cd /designs/my_design/modules/my_module
cmd -op behaviors/my_process/ops/op1
```

Here are some additional examples of how to identify an object for use with a CtoS command:

- Here is an *object_id* for identifying a module named **topModule** in the design **DUT**:
- Here is an *object_id* for identifying a behavior named **DUT_proc** in the module **topModule**:

```
/designs/DUT/modules/topModule
```

```
/designs/DUT/modules/topModule/behaviors/DUT_proc
```

- Here is an *object_id* for identifying an edge named **edge_ln12** in the behavior **DUT_proc**:

```
/designs/DUT/modules/topModule/behaviors/DUT_proc/edges/edge_ln12
```

- Here is an *object_id* for identifying all **edges** in the behavior **DUT_proc**:

```
/designs/DUT/modules/topModule/behaviors/DUT_proc/edges/*
```

Table D-1 Identifying Objects in CtoS Commands

Object	Path of Object ID
Design	<code>"/designs/<i>des_name</i>"</code>
Memory Bridge Def	<code>"/designs/<i>des_name</i>/memory_bridge_defs/<i>memory_bridge_def_name</i>"</code>
Memory Bridge Port Def	<code>"/designs/<i>des_name</i>/memory_bridge_defs/<i>memory_bridge_def_name</i> /memory_bridge_port_defs/<i>memory_bridge_port_def_name</i>"</code>
Memory Def	<code>"/designs/<i>des_name</i>/memory_defs/<i>memory_def_name</i>"</code>
Memory Def Aux Port	<code>"/designs/<i>des_name</i>/memory_defs/<i>memory_def_name</i> /aux_port_defs/<i>aux_port_def_name</i>"</code>
Module	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>"</code>
Array	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/arrays/<i>array_name</i>"</code>
External Array Bindings	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/arrays/<i>array_name</i> /external_array_bindings/<i>external_array_binding_name</i>"</code>
Behavior	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/behaviors/<i>beh_name</i>"</code>
Array Constraint	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/behaviors/<i>beh_name</i> /array_constraints/<i>array_constraint_name</i>"</code>
Edge	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/behaviors/<i>beh_name</i> /edges/<i>edge_name</i>"</code>
Float Constraint	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/behaviors/<i>beh_name</i> /float_constraints/<i>float_constraint_name</i>"</code>
Latency Constraint	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/behaviors/<i>beh_name</i> /latency_constraints/<i>latency_constraint_name</i>"</code>
Node	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/behaviors/<i>beh_name</i> /nodes/<i>node_name</i>"</code>
Operation Constraint	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/behaviors/<i>beh_name</i> /operation_constraints/<i>operation_constraint_name</i>"</code>
Operation	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/behaviors/<i>beh_name</i> /ops/<i>op_name</i>"</code>
Operation Port	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/behaviors/<i>beh_name</i> /ops/<i>op_name</i>/ports/<i>port_name</i>"</code>
Operation Port Pin	<code>"/designs/<i>des_name</i>/modules/<i>mod_name</i>/behaviors/<i>beh_name</i> /ops/<i>op_name</i>/ports/<i>port_name</i>/pins/<i>pin_name</i>"</code>

Table D-1 Identifying Objects in CtoS Commands

Object	Path of Object ID
Protocol Constraint	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /behaviors/ <i>beh_name</i> /protocol_constraints/ <i>protocol_constraint_name</i> "
Register Binding	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /behaviors/ <i>beh_name</i> /reg_bindings/ <i>reg_binding_name</i> "
Reset Condition	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /behaviors/ <i>beh_name</i> /reset_conditions/ <i>reset_condition_name</i> "
Resource Binding	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /behaviors/ <i>beh_name</i> /res_bindings/ <i>res_binding_name</i> "
Behavior Tag	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /behaviors/ <i>beh_name</i> /tags/ <i>tag_name</i> "
Behavior Terminal	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /behaviors/ <i>beh_name</i> /terms/ <i>beh_term_name</i> "
Behavior Value	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /behaviors/ <i>beh_name</i> /values/ <i>value_name</i> "
Instance	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /insts/ <i>inst_name</i> "
Instance Terminal	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /insts/ <i>inst_name</i> /inst_terms/ <i>inst_term_name</i> "
Instance Term Bit	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /insts/ <i>inst_name</i> /inst_terms/ <i>inst_term_name</i> /bits/ <i>inst_term_bit_name</i> "
Net	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /nets/ <i>net_name</i> "
Net Bit	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /nets/ <i>net_name</i> /bits/ <i>net_bit_name</i> "
Module Term	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /terms/ <i>mod_term_name</i> "
Module Term Bit	" /designs/ <i>des_name</i> /modules/ <i>mod_name</i> /terms/ <i>mod_term_name</i> /bits/ <i>mod_term_bit_name</i> "
RTL IP Def	" /designs/ <i>des_name</i> /rtl_ip_defs/ <i>rtl_ip_def_name</i> "
RTL IP Def Ports	" /designs/ <i>des_name</i> /rtl_ip_defs/ <i>rtl_ip_def_name</i> /ports / <i>rtl_ip_def_port_name</i> "
CGIC IP Def	" /designs/x/cgic_ip_defs/"
CGIC IP Def Ports	" /designs/x/cgic_ip_defs/ports"
Sim Configs	" /designs/ <i>des_name</i> /simulation_configs/ <i>simulation_config</i> "

D.6 Design Object Attributes (Designs)

A *design* corresponds to a container used to encapsulate modules in the source file(s). There are two tables for design attributes:

- “[Design Object Attributes - When They Can Be Modified](#)” on page D-9, which identifies the *latest* place in the CtoS Design Flow at which each attribute may be modified.
- “[Design Object Attributes - Defined](#)” on page D-11, which gives detailed descriptions of each attribute.

Table D-2 Design Object Attributes - When They Can Be Modified

Attribute Name	Latest Place It Can Be Modified
name	During the start of the design
auto_save_dir	During the set up of the design
baseline_dir	During the set up of the design
build_flat	During the set up of the design
build_monolithic_structs	During the set up of the design
compile_flags	During the set up of the design
default_scheduling_effort	During the set up of the design
design_dir	During the set up of the design
eco_sharing_policy	During the set up of the design
enable_multiple_pipeline_stalls	During the set up of the design
enable_slec_verification	During the set up of the design
header_files	During the set up of the design
implementation_target	During the set up of the design
keep_all_signals	During the set up of the design
max_auto_flatten_array_size	During the set up of the design
name_max_length	During the set up of the design
name_module_prefix	During the set up of the design
name_module_prefix_policy	During the set up of the design
name_use_*	During the set up of the design
optimize_enable_if_loops_transform	During the set up of the design
required_match_percentage	During the set up of the design
reset_registers	During the set up of the design
sim_wrapper_filename	During the set up of the design

Table D-2 Design Object Attributes - When They Can Be Modified

Attribute Name	Latest Place It Can Be Modified
source_files	During the set up of the design
systemc_version	During the set up of the design
top_module_path	During the set up of the design
default_export_memories	Before allocating IP
prototype_memory_setup_delay	Before allocating IP
prototype_memory_launch_delay	Before allocating IP
default_clock	Before analyzing micro-architecture
optimize_enable_merge_join_states	Before analyzing micro-architecture
ram_def_files	Before analyzing micro-architecture
export_memories	Before managing states
fpga_dsp_cost	Before managing states
fpga_install_path	Before managing states
fpga_target	Before managing states
low_power_clock_gating	Before managing states
rc_startup_script	Before managing states
tech_lib_names	Before managing states
allow_op_delays_exceeding_clock_period	Before scheduling
schedule_slack_margin	Before scheduling
verilog_rtl_model_suffix	Before analyzing and implementing
verilog_top_wrapper_suffix	Before analyzing and implementing
auto_write_models	Any time – not restricted
default_speed_grade	Any time – not restricted
default_toggle_probablity	Any time – not restricted
default_value_1_probability	Any time – not restricted
fpga_work_dir	Any time – not restricted
max_output_line_length	Any time – not restricted
rc_work_dir	Any time – not restricted
systemc_out_header_ext	Any time – not restricted
systemc_out_source_ext	Any time – not restricted
verilog_out_file_ext	Any time – not restricted

Table D-2 Design Object Attributes - When They Can Be Modified

Attribute Name	Latest Place It Can Be Modified
verilog_pragma_keyword	Any time – not restricted
verilog_use_*	Any time – not restricted

Table D-3 Design Object Attributes - Defined

allow_op_delays_exceeding_clock_period	Specifies that CtoS should proceed with scheduling, even if there are ops that exceed the clock period. Note See also “ Multi-Phase Scheduler ” on page -11. Warning Setting the allow_op_delays_exceeding_clock_period design attribute to <i>true</i> has no effect on scheduling in <i>low power mode</i> .	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	before register allocation
auto_save_dir	The name of the directory in which CtoS saves the <i>baseline design</i> at two places in the Incremental Synthesis flow: after build (in a subdirectory elaborated) and after registers have been allocated for every behavior in the design (in a subdirectory rtl_ready). Note See also “ Incremental Synthesis ” on page 17-1.	
	Type	string
	Access	read/write
	When set?	must be set before build
auto_write_models	Enables the writing of simulation models after a successful build (post_build) and a successful allocate registers (final). If enabled, the models are written to the model_dir specified in the simulation configuration object. Also writes the verification wrapper and TLM wrapper (if a TLM design) after a successful build. Note See also “ Graphical Depiction of all CtoS Models ” on page 7-7.	
	Type	boolean [true (1) or false (0)] [default is true (1)]
	Access	read/write
	When set?	anytime, preferably before build

Table D-3 Design Object Attributes - Defined

baseline_dir	The name of the directory in which a design is to be saved during the <i>baseline</i> run in the Incremental Synthesis flow. Note See also “ Incremental Synthesis ” on page 17-1.	
	Type	string
	Access	read/write
	When set?	must be set before build
build_flat	Activates the flattening of the design hierarchy during the build process. By default, CtoS preserves the SC_MODULE hierarchy. Note See also “ Module Hierarchy ” on page 16-1.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	must be set before build
build_monolithic_structs	Specifies whether a design will be built according to the <i>monolithic object model</i> or the <i>field-fragmented object model</i> . If <i>true</i> , the design is built according to the <i>monolithic object model</i> , that is, arrays nested in non- sc_module structs/classes are flattened during elaboration. If <i>false</i> (the default), the design is built according to the <i>field-fragmented object model</i> , that is, arrays nested in structs/classes are elaborated as database arrays, which can be flattened later using the flatten_array command. Note See also “ Object Models ” on page 14-45 and “ flex_channels_nets ” on page E-69.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	must be set before build
clocks	The clocks for the design Note See also “ Create New Design Wizard (Page 3) ” on page 6-13	
	Type	list of clock definitions { clock_period time_of_rising_edge time_of_falling_edge }
	Access	read only
	When set?	not applicable

Table D-3 Design Object Attributes - Defined

compile_flags	The C/C++ compiler flags used for parsing the source file(s), which must be enclosed in double quotes(" ") since spaces are interpreted as argument separators in Tcl. The first character of a compiler flag cannot be a dash (–) because the CtoS argument parser interprets it as an option flag; therefore, use a space as the first character preceding a dash (–). Notes - Setting this attribute resets the design. - For a list of supported compiler flags, see “Create New Design Wizard (Page 2)” on page 6-11	
	Type	string
	Access	read/write
	When set?	must be set before build
default_clock	The default clock for the design. Note See also “Create New Design Wizard (Page 3)” on page 6-13	
	Type	list of three integers { clock_period rise_period fall_period }
	Access	read/write
	When set?	must be set before analyzing micro-architecture
default_export_memories	Sets the default for the <i>exported memory feature</i>, that is, top module ports are created to connect memories instead of memory instances in modules. This separates memories, RAMs or ROMs, from the actual design. The value of this attribute is the default for the allocate_memory and allocate_prototype_memory commands. Notes - Built-in RAMs are <i>not</i> exported. - See also “allocate_memory” on page E-9, “allocate_prototype_memory” on page E-14, “is_exported” on page D-37.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	must be set before allocating IP

Table D-3 Design Object Attributes - Defined

default_scheduling_effort	Controls which phases of the scheduler should be performed. Note See also “ Scheduling ” on page -2.	
	Type	string (high , medium or low) [default is high]
	Access	read/write
	When set?	must be set before scheduling
default_speed_grade	The default speed grade for resources. - A speed grade of 100 forces CtoS to utilize the fastest possible, but area-expensive, resources to implement the given operation. - A speed grade of 0 instructs CtoS to choose the slowest implementation, with probably the smallest area requirement. Note See also “ Relaxed Latency Scheduling Mode ” on page -4	
	Type	unsigned integer [0 (slowest) to 100 (fastest)] (default is 100)
	Access	read/write
	When set?	must be set before state creation (create_required_states , create_initial_resources , schedule)
default_toggle_probability	The <i>default toggling probability</i> for nets and values: the probability of a net or value to change its value between one clock cycle and the next (or within a clock cycle, if the TCF were generated from a back-annotated gate-level simulation including glitching effects). Notes ncsim generates two transitions even for a glitch within a delta cycle. - See also “Setting Default Toggle Probability” on page 18-5	
	Type	double (0.0 to 1.0 ; however, it can be more than 1.0 if glitches are considered) (default is 0.02)
	Access	read/write
	When set?	anytime

Table D-3 Design Object Attributes - Defined

default_value_1_probability	The <i>default value 1 probability</i> for nets and values: the probability of a net or value to stabilize to logic value 1 at the end of a clock cycle. Note See also “ Setting Default Toggle Probability ” on page 18-5	
	Type	double (0.0 to 1.0) (default is 0.5)
	Access	read/write
	When set?	anytime
design_dir	The directory in which to save a design.	
	Type	string (default is <i>design_name</i> in the current working directory)
	Access	read/write
	When set?	must be set before build
eco_sharing_policy	Control resource sharing between the baseline and ECO resources	
	Type	string (reuse_existing_resources, create_new_resources) [default is reuse_existing_resources]
	Access	read/write
	When set?	must be set before create initial resources
enable_multiple_pipeline_stalls Note This is a preliminary feature (see “ Stalling/Flushing a Pipelined Loop ” on page 8-49).	Enables <i>the use of multiple pipeline stalls</i> .	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	must be set before build
enable_slec_verification	Enables SLEC verification by skipping some optimizations that are not supported by SLEC. Note Setting this attribute may cause QOR degradation.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	must be set before build

Table D-3 Design Object Attributes - Defined

Note For the following four attributes, note that FPGA support is a preliminary feature (see “ Create New Design Wizard - FPGA (Page 4) ” on page 6-17).		
fpga_dsp_cost	The DSP cost, in terms of LUTs, for FPGA designs	
	Type	integer (default is 100)
	Access	read/write
	When set?	must be set before state creation (create_required_states , create_initial_resources , schedule)
fpga_install_path	The installation path to the executable for FPGA designs Note The executable for Xilinx must be the path to xst in the Xilinx installation, and for Altera, it must be the path to quartus_sta .	
	Type	string
	Access	read/write
	When set?	must be set before state creation (create_required_states , create_initial_resources , schedule)
fpga_target	The target vendor, device family, and part for FPGA designs	
	Type	list of three values {<vendor> <family> <part>}
	Access	read/write
	When set?	must be set before state creation (create_required_states , create_initial_resources , schedule)
fpga_work_dir	The FPGA working directory.	
	Type	string (default is fpga_work)
	Access	read/write
	When set?	anytime
header_files	The header files to include in a SystemC simulation model. You must assure the correct order of this ordered list (nested child types before parent type) of file names.	
	Type	list of strings
	Access	read/write
	When set?	must be set before build

Table D-3 Design Object Attributes - Defined

implementation_target	The type of implementation desired.	
	Type	string (ASIC or FPGA) (default is ASIC)
	Access	read/write
	When set?	must be set before build
is_scheduled	Specifies whether scheduling has been performed on this design	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read only
	When set?	not applicable
keep_all_signals	Specifies whether nets inferred from sc_signals during build will be preserved throughout synthesis flow, even if no process reads from these nets.	
	Notes - The build, schedule and write_rtl commands will give you a warning that QoR will be sub-optimal if this attribute is set. - See also “Debugging Simulation Mismatches” on page I-1.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	must be set before build
low_power_clock_gating	Controls whether CtoS optimizes the design for clock gating.	
	Note If this attribute is set to <i>true</i> , when you write an RC script, the following line will be added to your script (see “Understanding RC Run Scripts” on page 13-51): <code>set_attr lp_insert_clock_gating true</code>	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	must be set before state creation (create_required_states , create_initial_resources , schedule)

Table D-3 Design Object Attributes - Defined

max_output_line_length	The maximum length of a line of an output simulation and RTL file, both for SystemC and Verilog. This should be set to a high value in incremental synthesis mode, to minimize differences due to line breaking. Note See also “Incremental Synthesis” on page 17-1 .	
	Type	integer (default is 80)
	Access	read/write
	When set?	anytime
max_auto_flatten_array_size	This is maximum number of bits for an array to be considered for flattening in automatically specifying micro-architecture flow. Note See also “Default Array Actions” on page 8-100 .	
	Type	integer (default is 256)
	Access	read/write
	When set?	anytime
name	The design name	
	Type	string
	Access	read only
	When set?	not applicable
name_max_length	The maximum length of any name used in a design, after which truncation occurs, but before uniquification, which means actual name length may be 2-5 characters longer.	
	Type	integer (minimum is 16 , default is 80)
	Access	read/write
	When set?	must be set before build
name_module_prefix	The prefix for all module names, used to avoid name conflicts.	
	Type	string (the default is empty string) Note Can be any valid Verilog string; last character is usually an underscore (_) to recognize prefix, but this is not required.
	Access	read/write
	When set?	must be set before build

Table D-3 Design Object Attributes - Defined

name_module_prefix_policy	The policy for naming non-user-defined modules, used to avoid name conflicts.	
	Type	string (by_name_module_prefix_attr or by_sc_module) [default is by_name_module_prefix_attr]
	Access	read/write
	When set?	must be set before build
name_use_class_name	Controls whether the class name will be prepended to behavior names derived from class/field (including sc_module) members.	
	Type	boolean [true (1) or false (0)] [default is false (0)] Note Use true (1) for backward compatibility.
	Access	read/write
	When set?	must be set before build
name_use_file_name	Controls whether CDFG objects will include source filename(s).	
	Type	boolean [true (1) or false (0)] [default is false (0)] Note Use true (1) for backward compatibility.
	Access	read/write
	When set?	must be set before build
name_use_inline_location	Controls whether the string _inl , followed by the inlined call location (including the file and line number, as specified), will be added when inlining a function call.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	must be set before build
name_use_line_number	Controls whether the string _ln , followed by the line number, will be used for names of nodes and operations in the CDFG derived from statements and expressions, not from variables or fields.	
	Type	boolean [true (1) or false (0)] [default is true (1)]
	Access	read/write
	When set?	must be set before build

Table D-3 Design Object Attributes - Defined

name_use_node_index	Controls whether the string _i , followed by the CDFG node index, will be part of the name. (This is for debugging only.)	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	must be set before build
name_use_template_parameters	Controls whether class template parameters will be included in the name of database objects.	
	Type	boolean [true (1) or false (0)] [default is true (1)]
	Access	read/write
	When set?	must be set before build
name_use_unrolling_index	Controls whether the string _unr , followed by the current iteration index, will be added when unrolling a loop.	
	Type	boolean [true (1) or false (0)] [default is true (1)] Note Use false (0) for backward compatibility.
	Access	read/write
	When set?	must be set before build
name_use_var_line_number	Controls whether the string _ln , followed by the line number, will be used for names of CDFG nodes derived from local method variables. The filename is also used if name_use_file_name is also <i>true</i> (because variable names can also generate global objects, such as arrays, and hence knowing the source files is useful for multi-file designs). This option is useful if the SystemC coding style uses several variables with the same name (for example, for loop indices) within the same method.	
	Type	boolean [true (1) or false (0)] [default is false (0)] Note Use true (1) for backward compatibility.
	Access	read/write
	When set?	must be set before build

Table D-3 Design Object Attributes - Defined

optimize_enable_if_loops_transform	Specifies whether to apply the if_loop transform.	
	Type	boolean [true (1) or false (0)] [default is true (1)]
	Access	read/write
	When set?	must be set before build
optimize_enable_merge_join_states	Specifies whether to apply the merge_join_states optimization as part of final optimization.	
	Type	boolean [true (1) or false (0)] [default is true (1)]
	Access	read/write
	When set?	must be set before any optimization is performed
Note For the following two attributes, note that <i>Prototype Memories</i> is a preliminary feature (see “ Allocating Prototype Memory ” on page 9-7).		
prototype_memory_launch_delay	The launch delay, expressed as a percentage of the clock period, for a prototype memory; used by the allocate_prototype_memory command.	
	Type	positive integer [0-100] [default is 80]
	Access	read/write
	When set?	must be set before allocating IP (allocate_built_in_ram , allocate_prototype_memory , allocate_memory)
prototype_memory_setup_delay	The setup delay, expressed as a percentage of the clock period, for a prototype memory; used by the allocate_prototype_memory command.	
	Type	positive integer [0-100] [default is 20]
	Access	read/write
	When set?	must be set before allocating IP (allocate_built_in_ram , allocate_prototype_memory , allocate_memory)

Table D-3 Design Object Attributes - Defined

rc_startup_script	The absolute path of the optional script file to be used as an initialization script for Encounter RTL Compiler (RC). This script is sourced by RC after it is started, before the tech libraries are set. Important In order to reference tech libraries, you must have set them in the RC startup script before they are referenced. CtoS does not close the tech libraries that are opened in the RC startup script if the tech libraries are not set in CtoS or if the tech libraries are identical. Note See also “ASIC (CNDW, Page 4):” on page 6-16 for more detail, including a reference to an example.	
	Type	string
	Access	read/write
	When set?	must be set before state creation (create_required_states , create_initial_resources , schedule)
rc_work_dir	The directory where intermediate files are kept when CtoS runs Encounter RTL Compiler (RC).	
	Type	string (default is rc_work)
	Access	read/write
	When set?	anytime
required_match_percentage	The minimum amount of matching between the baseline and a new design when in Incremental Synthesis mode. A low value usually signals an error, because an incremental design must be similar to the baseline. If the new design does not meet the required match percentage, as compared with the baseline, CtoS will issue an error. Note See also “Incremental Synthesis” on page 17-1.	
	Type	unsigned integer (0 to 100) (default is 90 , that is, 90%)
	Access	read/write
	When set?	must be set before build

Table D-3 Design Object Attributes - Defined

reset_registers	Controls whether registers will be reset in a design. If only_required , then reset is added only if it is functionally required. If internal , then reset is added to all internal registers and sc_signal registers that are reset in the SystemC (and a warning is given for sc_signal registers that are not reset in the SystemC). If internal_and_outputs , then all internal and sc_signal registers are reset (and an info message is given for all sc_signals that are not reset in the SystemC so the user can review these to see if resetting these registers will cause a simulation mismatch). Note See also “Resetting Registers” on page -19 and “Reset All Registers” on page 6-15	
	Type	enum [only_required , internal , internal_and_outputs] [default is only_required]
	Access	read/write
	When set?	must be set before build
run_command_scripts	<i>reserved for future use</i>	
schedule_slack_margin	Sets <i>slack margin</i> as a percentage of the clock cycle, from 0 to 100 . This attribute applies only to paths with negative slack. Note See also “Scheduling” on page -2.	
	Type	unsigned integer (0 to 100). The default value is 0 - for timing critical behaviors or pipeline functions. Otherwise the value is (mux+flop delay) /clock_cycle * 100%.
	Access	read/write
	When set?	must be set before scheduling
sim_wrapper_filename	The name (including path) of the verification wrapper file the generated design expects to find when simulating the RTL. Note See also “Generating a SystemC Verification Wrapper” on page 7-11, “inline” on page E-78, and “write_wrapper” on page E-176	
	Type	string (default is empty string)
	Access	read/write
	When set?	must be set before build

Table D-3 Design Object Attributes - Defined

source_files	The synthesis source filename(s) in which the design's top module is instantiated. Note Setting this attribute resets the design. This means that the results of all operations are cleared. Attribute settings are not cleared.	
	Type	list of strings
	Access	read/write
	When set?	must be set before build
systemc_out_header_ext	The filename extension for CtoS-output SystemC header files.	
	Type	string (default is h)
	Access	read/write
	When set?	anytime
systemc_out_source_ext	The filename extension for CtoS-output SystemC source file(s).	
	Type	string (default is cpp)
	Access	read/write
	When set?	anytime
systemc_version	The version of SystemC with which to process design sources. The only current value is 2.2 , that is, CtoS will process design sources with header files derived from SystemC 2.2, the basis for <i>IEEE Standard 1666-2005 for SystemC</i> and also the basis for INCISIV headers, as of version 8.2.	
	Type	string (only current possible value is 2.2)
	Access	read/write
	When set?	must be set before build
tech_lib_names	The names of the technology libraries. Note that: - You <i>must</i> include a technology library when using RC; otherwise, you will get an error when CtoS tries to call RC. - See also “Specifying Vendor RAM Library File/Simulation Model” on page H-19 for details on the technology libraries used for Vendor RAMs.	
	Type	list of strings
	Access	read/write
	When set?	must be set before state creation (create_required_states , create_initial_resources , schedule)

Table D-3 Design Object Attributes - Defined

top_module	The name of the top module	
	Type	object_id
	Access	read only
	When set?	not applicable
top_module_path	The hierarchical path to the module instance to be synthesized. If using the SC_MODULE_EXPORT macro in the module's .cpp file, it is just the module name; otherwise, it is the period (.) separated hierarchical path to the instance. Notes - Setting this attribute resets the design. - See also “Instantiation of the Top Module” on page 14-4.	
	Type	string
	Access	read/write
	When set?	must be set before build
type	The type of this object.	
	Type	string (design)
	Access	read only
	When set?	not applicable

Table D-3 Design Object Attributes - Defined

verilog_mux_style	Specifies the coding style of muxes used in CtoS-generated RTL models. The legal values are: • reverse_one_hot (default). Produces <i>case</i> with constant control expression and variable labels. • casez. Produces <i>casez</i> with variable control expression and constant labels. • one_hot. Produces <i>case</i> with variable control expression and constant labels. Note See also “Controlling Select Expression, Case Label Generation” on page J-2.	
	Type	enum [reverse_one_hot , one_hot , or casez] [default is reverse_one_hot]
	Access	read/write
	When set?	anytime
verilog_out_file_ext	The filename extension for output Verilog files. Note See also “Generating Verilog Behavioral Models” on page 7-9.	
	Type	string (default is v)
	Access	read/write
	When set?	anytime
verilog_pragma_keyword	Sets the name for all pragmas generated in RTL Note See “Controlling pragma Keyword for Generated RTL” on page J-4.	
	Type	any string of non-zero length (default is pragma)
	Access	read/write
	When set?	can be set and/or unset any time before/after generating RTL (write_rtl)
verilog_rtl_model_suffix	The filename extension for output Verilog RTL model files.	
	Type	string (default is _rtl)
	Access	read/write
	When set?	cannot be changed after generating RTL (write_rtl , to ensure that write_rtl and write_rc_script are consistent)

Table D-3 Design Object Attributes - Defined

verilog_top_wrapper_suffix	The filename extension for output Verilog top wrapper files. This extension <i>cannot</i> be an empty string because this can result in a name conflict of the top module and the top wrapper module of the design. Note See “Generating a Top Wrapper for Exported Memories (Only after Scheduling)” on page 7-15; “Exporting Memories” on page 9-20; “write_top_wrapper” on page E-173	
	Type	string (default is _top_wrapper) (<i>cannot</i> be empty string)
	Access	read/write
	When set?	must be set before allocating registers (allocate_registers)
verilog_use_aligned_widths	Ensures that inputs to certain arithmetic and all comparison/equality operations are padded to the same matching bit width in the generated RTL. Also, results of shift operations will be performed with matching bit widths. Note See also “Controlling Mismatched Operand Bit Widths” on page J-3	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	anytime
verilog_use_case_default_x	Controls how the default case will be generated in CtoS-generated output simulation models or RTL. When <i>true</i> (the default), the default case will be assigned a value of X ; when <i>false</i> , the default case will be assigned a value of 0 . Note See also “Controlling Default Case” on page J-2.	
	Type	boolean [true (1) or false (0)] [default is true (1)]
	Access	read/write
	When set?	anytime
verilog_use_indexed_part_select	Controls whether the <i>indexed part select</i> feature of Verilog-2001 will be used in CtoS-generated output simulation models or RTL. The default is <i>true</i> ; use <i>false</i> to specify that <i>case</i> statements be used, instead. Note See also “Controlling Use of Indexed Part Selects” on page J-5.	
	Type	boolean [true (1) or false (0)] [default is true (1)]
	Access	read/write
	When set?	anytime

Table D-3 Design Object Attributes - Defined

verilog_use_non_blocking_delay_control	Controls whether <i>delay control</i> (a #1) will be used for every non- blocking assignment in CtoS-generated output simulation models or RTL. The default is <i>false</i>, but when <i>true</i>, CtoS will use a #1 intra- assignment delay for clocked always blocks, which can make debugging easier by having clk and data changes separated in waveforms. Warning Contrary to some designers’ opinions, this coding style will <i>not</i> help to avoid Verilog race conditions. Also, since logic synthesis tools, such as Encounter RTL Compiler (RC), will ignore these added intra-assignment delays, the use of this design attribute is not recommended for final RTL. Note See “Controlling Use of Delay Control with Non-Blocking Assignments”	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	anytime
verilog_use_state_names	Controls the naming of FSM states in the generated RTL. For more information, see “Renaming States in Support of FSM Coverage” on page 7-25. Note You can set this attribute only when verilog_mux_style is set to one_hot.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	anytime
verilog_use_wire_array	Controls whether a <i>wire array</i> will be used to model lookup resources, a feature of Verilog-2001, in CtoS-generated output simulation or RTL models. The default is <i>false</i>, which means that a function call with a case statement embedded inside the called function will be used, instead. Note See also “Controlling Use of Wire Arrays to Model Lookup Resources” on page J-4.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	When set?	anytime

Child Scopes

memory_defs	The vendor memory definitions in this design
memory_generators	<i>reserved for future use</i>
modules	The modules in this design
simulation_configs	The simulation configurations for this design
rtl_ip_defs	The RTL IP definitions in this design

D.7 Memory Bridge Definition Object Attributes (*memory_bridge_defs*)

This table shows the *memory bridge definition* object attributes. *Memory bridges are a preliminary feature.*

interface_type	The type of memory interface to which this bridge will connect.	
	Type	list of strings (async_read or sync_read or sync_write or sync_read_write)
	Access	read-only
name	The user-defined name for this memory bridge definition, which must be unique; it will be checked against names of defined SC_MODULE s.	
	Type	string
	Access	read-only

Child Scope

memory_bridge_port_defs	The port definitions for the memory bridge.

D.8 Memory Bridge Port Def Object Attributes (*memory_bridge_port_defs*)

This table shows the *memory bridge port definition* object attributes. *Memory bridges are a preliminary feature.*

connected_to	The Verilog expression that will be assigned to the input port. This form can be used when, for whatever reason, bit widths are necessary.	
	Type	string (for multi-bit, must use a format in Table H-1 on page H-14)
	Access	read-only

is_mem_port	Indicates whether this port is a CtoS standard port name. If this value is false, the input port is an input to the user's memory.	
	Type	boolean [true (1) or false (0)]
	Access	read-only
name	The name of the port.	
	Type	string
	Access	read-only

D.9 Memory Definition Object Attributes (*memory_defs*)

This table shows the *memory definition* object attributes.

Note See “Allocating Memory” on page 9-2 & “IP Definition Files for Memories and RTL IP” on page H-1

bridges	The bridges for each previously defined port protocol, in order of declaration.	
	Type	array of strings
	Access	read-only
clock_per_port	Indicates whether each port has its own clock.	
	Type	boolean (true or false)
	Access	read-only
clock_port	The name of the clock port on the user's memory cell. Important If bridges is defined and clock_per_port is false , this must be specified.	
	Type	string
	Access	read-only
clock_posedge	Indicates whether clock sensitivity is positive edge (true) or negative edge (false).	
	Type	boolean (true or false)
	Access	read-only
format	The programming format, only for a ROM.	
	Type	string [valid values are { hex or bin or arm }]
	Access	read-only

has_reset	Indicates whether the memory has a reset mode, which must reset all memory bits to 0 to be compatible with CtoS. This works with reset_high and reset_async .	
	Type	boolean (true or false)
	Access	read-only
has_stall	For Vendor RAMs only, indicates whether each memory interface has a user-implemented STALLx pin. Note <i>Stalling a memory is a preliminary feature (see “RAMdef XML Elements” on page H-3).</i>	
	Type	boolean (true or false)
	Access	read-only
interface_types	The memory interface type for each memory interface.	
	Type	list of strings [valid values are { async_read or sync_read or sync_write or sync_read_write }]
	Access	read-only
liberty_filename	The Liberty library filename, used only during synthesis – not in simulation.	
	Type	string
	Access	read-only
min_write_width	The minimum number of bits the memory is capable of writing, which is used to enable the partial memory write optimization in CtoS. A port should not be generated unless this is present.	
	Type	positive integer
	Access	read-only
name	The memory’s name, contingent on the use of the wrapper_filename attribute: • If wrapper_filename <i>is</i> specified, this name must match the module name in the Verilog file specified by the wrapper_filename. • If wrapper_filename is <i>not</i> specified, this is the name of the module created by CtoS to represent the memory master.	
	Type	string
	Access	read-only
num_words	The number of words in this memory.	
	Type	integer
	Access	read-only

read_latency	The read latency for a Vendor memory.	
	Type	unsigned integer (valid values are 1-3) (default is 1) (vendor ROMs are always 1) Note Specifying read latency is a preliminary feature (see “Allocating Memory” on page 9-2).
	Access	read-only
reset_async	Indicates whether reset is asynchronous (true) or synchronous (false). Important If has_reset is false , this is ignored.	
	Type	boolean (true or false)
	Access	read-only
reset_high	Indicates whether reset is active high (true) or active low (false). Important If has_reset is false , this is ignored.	
	Type	boolean (true or false)
	Access	read-only
reset_port	The name of the reset terminal on the user’s memory cell. Important If bridges is defined and has_reset is true , this must be specified.	
	Type	string
	Access	read-only
tech_cell_name	The name of the technology cell used to implement the memory. CtoS creates a module with this name to represent the technology cell. Instances of this module are connected to memory bridge instances.	
	Type	string
	Access	read-only
verilog_filename	The Verilog simulation model filename; specify this when you do not want to use CtoS automatically generated RTL. Notes and Limitations - This is <i>required</i> for FPGA designs – and is <i>optional</i> for ASIC designs. - This verilog_filename does not work in side-by-side simulation; you will get an error on an undefined module.	
	Type	string
	Access	read-only

width	The number of bits in each word of this memory.	
	Type	integer
	Access	read-only
wrapper_filename	The Verilog wrapper file, contingent on the use of the bridges attribute: • If bridges is <i>not</i> specified, this file contains a module wrapping the actual memory instance whose purpose is to map ports from the vendor memory to the CtoS expected ports. • If bridges is specified, this is used for other purposes, e.g., composite memories.	
	Type	string
	Access	read-only
xilinx_search_dir	The Xilinx search directory, which contains the built-in memory model. Limitation This is used only for FPGA designs – and not for ASIC designs.	
	Type	string
	Access	read only

Child Scope

aux_port_def	The optional definition of any auxiliary ports and their bit widths.

D.10 Memory Definition Auxiliary Port Object Attributes (*aux_port_defs*)

This table shows the *memory definition auxiliary port* object attributes for Vendor RAMs.

Note See “[Specifying Auxiliary Ports in Vendor RAMs](#)” on page H-19.

is_input	Indicates whether this is an input port	
	Type	boolean [true (1) or false (0)]
	Access	read-only
name	The name of this port.	
	Type	string
	Access	read-only

width	The number of bits in each word of this port.	
	Type	integer
	Access	read-only

D.11 Module Object Attributes (Modules)

A *module* corresponds to a Verilog module or an **SC_MODULE** class in SystemC.

area	An area of the module	
	Type	double
	Access	read only
design	The design containing this module	
	Type	object_id
	Access	read only
is_optimized	Specifies whether this module has been optimized	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read only
is_user_defined	Identifies user defined modules.	
	Type	boolean
	Access	read only
max_delay	The maximum delay of the module	
	Type	integer
	Access	read only
mod_type	The type of this module	
	Type	string
	Access	read only
name	The name of this module	
	Type	string
	Access	read only

power	The estimated power for the module, which is set during power analysis (using the analyze command with the -power option). [Note that <i>Power Estimation</i> is a <i>preliminary feature</i> (see “ Low Power Estimation Using CtoS ” on page 18-1).]	
	Type	double (default is 0.0)
	Access	read only
schedule_slack	The slack of this module after successfully scheduled. It is the minimum slack of all paths in this module.	
	Type	integer or “ <i>not calculated</i> ”
	Access	read only
slack	The slack for this module after netlisting. The value is assigned on the first call to report_timing or analyze -timing. It is the minimum slack of all paths in this module.	
	Type	integer or “ <i>not calculated</i> ”
	Access	read only
type	The type of this object	
	Type	string (module)
	Access	read only

Child Scopes

arrays	The arrays contained in this module
behaviors	The behaviors contained in this module
insts	The instances contained in this module
nets	The nets contained in this module
terms	The terminals contained in this module

D.12 Array Object Attributes (Arrays)

In CtoS, an *array* is defined as a database object (as opposed to a *memory*, which is defined as a resource).

addr_width	The address width of this array	
	Type	positive integer
	Access	read only
data_width	The data width of this array	
	Type	positive integer
	Access	read only
data_words	The number of words in this array	
	Type	positive integer
	Access	read only
design	The design containing this array	
	Type	object_id
	Access	read only
dont_touch	When set to <i>true</i> by ctos dont_touch pragma, specifies that this array should not be flattened or trimmed during optimization (see “Allocating Vendor ROM” on page 9-14).	
	Important If calls to dont_touch functions evaluate to either constant zero or one, those functions <i>will</i> be optimized out.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
external_array_bindings	Specifies the list of external bindings in which this array is the master.	
	Type	list of id
	Access	read only

is_allocated	Specifies whether this array is fully allocated. If interfaces are automatically allocated, this attribute is set by the allocate_builtin_ram (“ allocate_builtin_ram ” on page E-7), allocate_prototype_memory (“ allocate_prototype_memory ” on page E-14), or allocate_memory (“ allocate_memory ” on page E-9) commands; otherwise, set by the allocate_memory_interfaces command (“ allocate_memory_interfaces ” on page E-11).	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read only
is_exported	Specifies whether this array is to be exported (“ Exporting Memories ” on page 9-20; “ default_export_memories ” on page D-13) during netlisting. This attribute is set by the allocate_memory command (“ allocate_memory ” on page E-9) or the allocate_prototype_memory command (“ allocate_prototype_memory ” on page E-14).	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read only
is_external	Specifies whether this array is external. This attribute is set by the build command (“ build ” on page E-30).	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read only
is_shared	Specifies whether this array is accessed by multiple processes (including external processes).	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read only
inst	The memory instance that is bound to the array by the allocate_memory command (“ allocate_memory ” on page E-9).	
	Type	object [empty string or object]
	Access	read only
min_write_width	Smallest number of bits that can be modified in writing this array. If value is zero, this value has not been specified, and the entire data word must be written each time.	
	Type	unsigned integer
	Access	read/write
module	The module containing this array	
	Type	object_id
	Access	read only

name	The name of this array	
	Type	string
	Access	read only
read_only	Specifies whether this array is read-only	
	Type	boolean [true (1) or false (0)]
	Access	read only
read_ops	The array read operations performed on the array	
	Type	list of object_ids
	Access	read only
src_links	The SystemC source file locations for this array	
	Type	list of strings of the form: file:line:char
	Access	read only
type	The type of this object	
	Type	string (mem)
	Access	read only
uarch_action	The micro-architecture action to be applied to this array.	
	Type	string (flatten , builtin , prototype , vendor , rom , or no_action) (default is no_action)
	Access	read/write
	When set?	must be set before calling apply_uarch_action
write_ops	A list of array write operations performed on the array	
	Type	list of object_ids
	Access	read only

Child Scopes (of arrays)

external_array_bindings	The external array bindings in this array
directives	Contains the directives of this array

D.13 External Array Binding Object Attributes (external_array_binding)

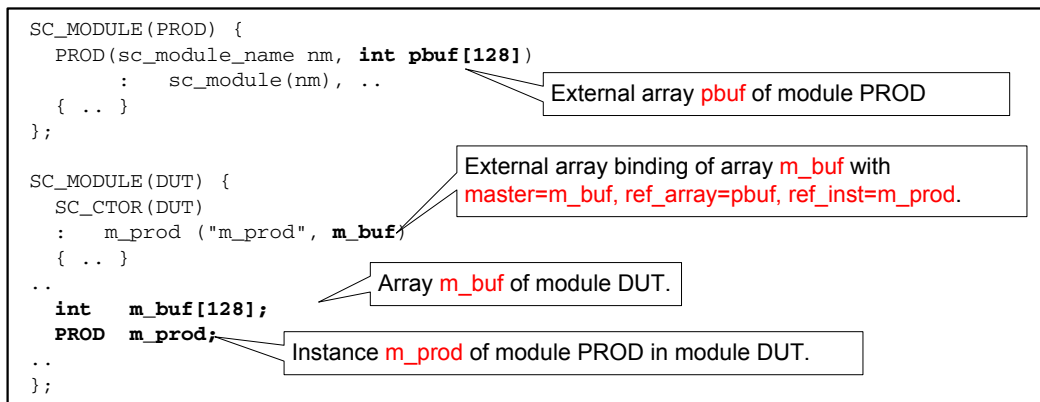
An *external array binding* is a child scope of *arrays* and is named as follows:

[name of master]_[name of ref inst]_[name of ref array]

See also [Figure D-2 on page D-39](#) for how this naming relates to the source code names.

master	The name of the array to which ref_array refers	
	Type	array_id
	Access	read only
name	The name of the external array binding, used for constructing an id	
	Type	string (must not be empty)
	Access	read only
ref_array	The name of the external array in the submodule instance	
	Type	array_id
	Access	read only
ref_inst	The name of the submodule instance	
	Type	inst_id
	Access	read only

Figure D-2 External Array Binding Object Attributes Related to Source Code Names



D.14 Pipeline Function Object Attributes (pipeline_functions)

The pipeline function object is available under the **pipeline_functions** scope under the module of the behavior. The name of the pipeline function object is the same as the name of the behavior. The pipeline function object has the following attributes:

spec_behavior	The pipeline specification behavior.	
	Type	behavior
	Access	read only
impl_behavior	The pipeline implementation behavior. This is a valid object after the optimize step is completed.	
	Type	behavior
	Access	read only
min_latency	The minimum latency specified by the user.	
	Type	unsigned int. The value must be greater than or equal to 1 .
	Access	read only
max_latency	The maximum latency specified by the user.	
	Type	unsigned int. The value must be greater than or equal to the min_latency .
	Access	read only
latency	The actual latency of the pipeline function after schedule. This is equal to the min latency before schedule.	
	Type	unsigned int. The value must be greater than or equal to the min_latency .
	Access	read only

D.15 Behavior Object Attributes (Behaviors)

A *behavior* is the internal representation of a member function (or function).

dont_touch	When set to <i>true</i> by the ctos dont_touch pragma, specifies that this behavior's interfaces (input/output ports) should not be trimmed or optimized out during optimization (see "Importing RTL IP into SystemC Designs" on page 9-41).	
	Important If calls to dont_touch functions evaluate to either constant zero or one, those behaviors <i>will</i> be optimized out.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read only
enable_addsubs	Controls whether the create_initial_resources and schedule commands will use <i>addsub</i> resources. You can manually create <i>addsubs</i> with the create_resource command; however, no ops in the design will be mapped onto any existing <i>addsub</i> resource. If you have already created an op constraint (either soft or hard) to an <i>addsub</i> resource, before resource allocation, CtoS will not let you disable this attribute.	
	For FPGA based implementations, <i>addsubs</i> are not used by default. To enable the use the <i>addsub</i> resources on FPGA implementations, the enable_addsubs attribute can be applied at all or corresponding behavior.	
	Type	boolean [true (1) or false (0)] [default is true (1)]
	Access	read/write
enable_const_resources	Allows you to disable the use of const-resources. In ECO mode, you must explicitly set the attribute to equal the baseline value (which is true for any other behavior attribute today). This attribute exists in on BST::Behavior and is visible from TCL.	
	Note : - This attribute must be set before the manage_resources step in the design flow. - You cannot constrain a const input value op to a generic resource by calling constrain_op manually.	
	Type	boolean [true (1) or false (0)] [default is true (1)]
	Access	read/write

error_recovery_terminal	The terminal for signaling recovery from an illegal control state	
	Type	object_id
	Access	read only
external_input_delay	The input delay of a behavior, which is set using the external_delay command.	
	Type	integer or “ <i>not set</i> ”
	Access	read only
external_output_delay	The output delay of a behavior, which is set using the external_delay command.	
	Type	integer or “ <i>not set</i> ”
	Access	read only
is_combinational	Specifies whether this behavior is purely combinational (has no state nodes)	
	Type	boolean [true (1) or false (0)]
	Access	read only
is_process	Specifies whether this behavior is a process	
	Type	boolean [true (1) or false (0)]
	Access	read only
module	The module containing this behavior	
	Type	object_id
	Access	read only
name	The name of this behavior	
	Type	string
	Access	read only
number_of_schedule_passes	The number of passes that CtoS needed to schedule this behavior	
	Type	integer
	Access	read only
one_value	The value representing the constant 1	
	Type	object_id
	Access	read only

optimize_enable_arithmetic_trees	Enables the arithmetic-tree optimization	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
origin	The origin node in the CDFG.	
	Type	object_id
	Access	read only
pipeline_function	Specifies the pipeline function object from a behavior when the function is set to have a pipeline function implementation.	
	Type	object_id
	Access	read/write
power	The estimated power for the behavior, which is set during power analysis (using the analyze command with the -power option). Note <i>Power Estimation is a preliminary feature (see “Low Power Estimation Using CtoS” on page 18-1).</i>	
	Type	double
	Access	read only
relax_latency	Indicates whether the CtoS scheduler and related commands, for example, create_required_states , should run in <i>relaxed latency scheduling mode</i> . Note <i>Relaxed latency scheduling mode is a preliminary feature (see “Relaxed Latency Scheduling Mode” on page -4).</i>	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
reset_value	The value representing the reset value in this behavior	
	Type	object_id
	Access	read only
schedule_failure_edge	Indicates the edge on which scheduling failed.	
	Type	object_id (default is empty string)
	Access	read only

schedule_slack	The slack of this behavior after successfully scheduled. It is the minimum slack of all instances in this behavior.	
	Type	integer or “ <i>not calculated</i> ”
	Access	read only
scheduling_effort	Indicates the number of phases the scheduler should perform.	
	Type	string (high , medium or low) [default is high]
	Access	read/write
slack	The slack for this behavior after netlisting. The value is assigned on the first call to report_timing or analyze_timing. It is the minimum slack of all instances in this behavior.	
	Type	integer or “ <i>not calculated</i> ”
	Access	read only
src_links	The SystemC source file locations for this behavior	
	Type	list of strings of the form: file:line:char
	Access	read only
synthesis_mode	The synthesis mode of the behavior.	
	Type	string (manual , simple , cycle_accurate , relaxed_latency or pipelined) (default is manual)
	Access	read/write
	When set?	must be set before Allocating IP
timing_criticality	The timing criticality of the behavior.	
	Type	string (high , auto or medium) (default is auto)
	Access	read/write
	When set?	must be set before scheduling
type	The type of this object	
	Type	string (behavior)
	Access	read only

uarch_action	The micro-architecture action to be applied to this function.	
	Type	string (inline , or no_action) (default is no_action)
	Access	read/write
	When set?	must be set before calling apply_uarch_action
zero_value	The value representing the constant 0	
	Type	object_id
	Access	read only

Child Scopes

array_constraints	The regions of the behavior where it is legal to add states
directives	Contains the directives of this behavior
edges	The edges in this behavior's CDFG
latency_constraints	The regions of the behavior where it is legal to add states
nodes	The nodes in this behavior's CDFG
op_constraints	The op constraints of this behavior
ops	The operations of this behavior
reg_bindings	The register bindings of this behavior
reg_bindings	The register bindings of this behavior
reset_conditions	The reset conditions of this behavior
tags	The tags of this behavior
terms	The terminals of this behavior
values	The values of this behavior

D.16 Array Constraint Object Attributes (*array_constraints*)

Here are the *array constraint* object attributes.

end_node	The ending node for the region to which this constraint applies	
	Type	object_id
	Access	read only
name	The name of this node	
	Type	string
	Access	read only
start_node	The starting node for the region to which this constraint applies	
	Type	object_id
	Access	read only
type	The type of this object	
	Type	string (array_constraint)
	Access	read only

D.17 Edge Object Attributes (Edges)

An *edge* is part of the CDFG, and edges connect nodes.

behavior	The behavior containing this node	
	Type	object_id
	Access	read only
born_ops	The operations on this edge in the original source specification	
	Type	list of object_ids
	Access	read only

control_value	The values representing the activation of this edge	
	Type	object_id
	Access	read only
is_backward	Specifies whether this edge is a backward edge implementing a loop	
	Type	boolean [true (1) or false (0)]
	Access	read only
name	The name of this node	
	Type	string
	Access	read only
op_constraints	The op constraints on this edge	
	Type	list of op constraints
	Access	read only
res_bindings	The resources bound to ops scheduled on this edge	
	Type	list of resource bindings
	Access	read only
scheduled_ops	The operations on this edge after scheduling	
	Type	list of ops
	Access	read only
sink	The node to which this edge transitions	
	Type	object_id
	Access	read only
source	The node from which this edge transitions	
	Type	object_id
	Access	read only
type	The type of this object	
	Type	string (edge)
	Access	read only

Child Scopes

directives	Contains the directives of this edge

D.18 Floating Constraint Object Attributes (*float_constraints*)

Floating constraints are created by the **float_io_accesses** command and describe a region of the CDFG for which I/O nets may be *floated*.

Note See “[float_io_accesses](#)” on page E-71. The *float_io_accesses* command is a preliminary feature.

end_node	The ending node for the region to which this constraint applies.	
	Type	object_id
	Access	read only
name	The name of the floating constraint	
	Type	string
	Access	read only
start_node	The starting node for the region to which this constraint applies.	
	Type	object_id
	Access	read only
type	The type of this object	
	Type	string (float_constraint)
	Access	read only
volatile	Specifies whether this floating constraint is volatile.	
	Type	boolean [true (1) or false (0)]
	Access	read only

D.19 Latency Constraint Object Attributes (latency_constraints)

Latency constraints are created by the **constrain_latency** command and describe a region of the CDFG where states may be added.

Note See also “[constrain_latency](#)” on page E-36.

behavior	The behavior to which this latency constraint applies	
	Type	object_id
	Access	read only
end_node	The ending node for the region of the CDFG to which this constraint applies.	
	Type	object_id
	Access	read only
max_latency	The maximum number of cycles for any path in this region	
	Type	unsigned integer
	Access	read only
start_node	The starting node for the region of the CDFG to which this constraint applies.	
	Type	object_id
	Access	read only
type	The type of this object	
	Type	string (latency_constraint)
	Access	read only

D.20 Node Object Attributes (Nodes)

A *node* is part of the CDFG, and nodes are connected by edges.

annotation_paths	The annotation path of a state node. This attribute can return zero, one, or more annotation_paths for a state node. State nodes can end up with multiple annotation_paths due to merging of states defined in the source code or due to zipping. After build, the annotation_path of state node is set to the label (if any) on the corresponding wait statement in the SystemC source code. During inlining, the annotation_paths attribute of all states in the function are prepended with the value of the annotation pragma associated with the function call (if any). A slash '/' is the separation character. During unrolling, the annotation_paths attribute of all states in the loop are prepended with the 'unrX' where X represents the iteration number and with slash '/' as the separation character. For more information, see "Annotating Function Calls" on page 14-76.	
	Type	string
	Access	read only
	Node type	state nodes only
behavior	The behavior containing this node	
	Type	object_id
	Access	read only
	Node type	all node types
in_edges	The edges that have this node as a sink	
	Type	list of object_ids
	Access	read only
	Node type	all node types

init_interval	The number of states used as an II for the pipeline, which determines the throughput. This value will be the same as the value for the -init_interval option to the pipeline_loop command, or if that value was not given, this will be the default value of 1 , or if the loop is not pipelined, it will return 0 .	
	Type	unsigned integer
	Access	read/write
	Node type	join nodes of pipelined loops
is_function_call_loop	Specifies whether this node is the join node of a sequential function call loop.	
	Type	boolean [true (1) or false (0)]
	Access	read only
	Node type	join nodes of a sequential function call loops
is_loop	Specifies whether this node is the join node of a loop.	
	Type	boolean [true (1) or false (0)]
	Access	read only
	Node type	loop join nodes
is_pipeline_loop	Specifies whether this is the join node of a pipelined loop.	
	Type	boolean [true (1) or false (0)]
	Access	read only
	Node type	join nodes of pipelined loops
is_stall_loop	Specifies whether this is the join node of a stall loop.	
	Type	boolean [true (1) or false (0)]
	Access	read only
	Node type	loop join nodes
is_state	Specifies whether this is a state node	
	Type	boolean [true (1) or false (0)]
	Access	read only
	Node type	state nodes

lat_interval	Starts out as the same value as the min_lat_interval node attribute. The CtoS scheduler may increase it if it needs to add latency to reduce negative slack. If the loop is not pipelined, it will return 0.	
	Type	unsigned integer
	Access	read only
	Node type	join nodes of pipelined loops
loop_level	The level of the innermost loop that contains this node (0 means this node is not contained in a loop).	
	Type	integer (≥ 0)
	Access	read only
	Node type	all node types
max_lat_interval	The maximum number of states to be used for a latency interval for the pipeline, which determines the number of stages in the pipeline. This value will be the same as the value supplied with the -max_lat_interval option to the pipeline_loop command; or if that value was not given, this will be the default value of (-min_lat_interval + 1) or 100, whichever is greater; or if the loop is not pipelined, it will return 0.	
	Type	unsigned integer
	Access	read/write
	Node type	join nodes of pipelined loops
merged_join_states	Specifies whether this is a loop join node on which the state before the loop and the state at the bottom of the loop have been merged and moved to the top of the loop.	
	Type	boolean [true (1) or false (0)]
	Access	read only
	Node type	loop join nodes

min_lat_interval	The minimum number of states to be used for a latency interval for the pipeline, which determines the number of stages in the pipeline. This value will be the same as the value supplied with the -min_lat_interval option to the pipeline_loop command; or if that value was not given, this will be the default value of (-init_interval + 1) or the original loop latency, whichever is greater; or if the loop is not pipelined, it will return 0.	
	Type	unsigned integer
	Access	read/write
	Node type	join nodes of pipelined loops
name	The name of this node. While renaming the state node, the new name that you provide must adhere to the following guidelines: • It must start with an alphabetic letter underscore followed by alphanumeric characters or underscores • It cannot be a Verilog keyword • It must not conflict with names of nodes in the same behavior • It must not conflict with names of state nodes, instances or nets in the module of the node's behavior	
	Type	string
	Access	read/write for state nodes, and read only for all other type of nodes
	Node type	all node types
out_edges	The edges that have this node as a source	
	Type	list of object_ids
	Access	read only
	Node type	all node types
pipeline_stage_mux	Returns the join mux op that codes the active stages of the pipelined loop. If the join node is not a pipelined loop, or the design has not been scheduled, the attribute will be " ". The width of this mux is the number of pipeline stages. Bit i is true if and only if stage i+1 is active.	
	Type	op (" " or an op)
	Access	read only
	Node type	join nodes of pipelined loops

preserve	Specifies whether this node was elaborated for a labeled non-control statement (not an if/switch/for/do/while statement) and should therefore be preserved during transforms.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write
	Node type	all node types
reg_bindings	The register bindings containing values in this state	
	Type	list of register bindings
	Access	read only
	Node type	all node types
src_links	The SystemC source file locations for this node	
	Type	list of strings of the form: file:line:char
	Access	read only
	Node type	all node types
transformed_do_while	Specifies whether a do/while transform has been applied to this loop join node.	
	Type	boolean [true (1) or false (0)]
	Access	read only
	Node type	loop join nodes
transformed_if_loops	Specifies whether the loop transform has been applied to this join node.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read only
	Node type	join nodes
type	The type of this object	
	Type	string (either origin , join , fork or simple)
	Access	read only
	Node type	all node types

uarch_action	The micro-architecture action to be applied to this loop join node.	
	Type	string (unroll , break , pipeline , or no_action) (default is no_action)
	Access	read/write
	Node type	loop join nodes

Child Scopes

directives	Contains the directives of this node

D.21 Operation Constraint Object Attributes (op_constraints)

Operation constraints are created by the **schedule** command, prior to being set by the **constrain_op** command.

Note See also “[constrain_op](#)” on page E-38 and “[schedule](#)” on page E-138.

behavior	The behavior containing this constraint	
	Type	Behavior
	Access	read only
constraint_type	The type of the constraint	
	Type	string (soft , hard , restrict_sharing)
	Access	read only
created_by	Describes where the constraint originated	
	Type	string (user , incremental)
	Access	read only
edge	The edge to which the op is constrained	
	Type	Edge
	Access	read only
inst	The resource on which the operation is scheduled	
	Type	Inst
	Access	read only
is_respected	Indicates whether CtoS scheduler was able to bind op to specified edge or resource	
	Type	boolean [true (1) or false (0)] [default is true (1) before schedule]
	Access	read only
name	The name of this constraint	
	Type	string
	Access	read only
op	The operation for this constraint	
	Type	Op
	Access	read only
type	The type of this object	
	Type	string (op_constraint)
	Access	read only

D.22 Operation Object Attributes (Ops)

An *operation* corresponds to an operation in the CDFG.

behavior	The behavior containing this operation	
	Type	object_id
	Access	read only
birth_edge	The edge on which this operation was specified in the source file description	
	Type	object_id
	Access	read only
in_ports	The input ports on this operation	
	Type	list of ports
	Access	read only
min_speed_grade	Set during CDFG timing analysis – a suggestion to the CtoS scheduler of the minimum speed grade for this operation that will result in non-negative slack. Note If you run the constrain_op command on a resource with a speed grade smaller than min_speed_grade , you will get a warning.	
	Type	unsigned integer [0 (slowest) to 100 (fastest)] [default is design's default_speed_grade] (see “ default_speed_grade ” on page D-14.)
	Access	read only
name	The name of this operation	
	Type	string
	Access	read only
op_constraint	The op constraint associated with this op	
	Type	op_constraint
	Access	read only
op_type	The type of this operation	
	Type	string (add , sub , custom , etc.)
	Access	read only

out_ports	The output ports on this operation	
	Type	list of ops
	Access	read only
power	Set during CDFG timing analysis – the power estimation for this operation. Note <i>Power Estimation is a preliminary feature (see “Low Power Estimation Using CtoS” on page 18-1).</i>	
	Type	double [default is 0 (not set)]
	Access	read only
res_binding	If the operation is scheduled or partially scheduled, this identifies the binding, which captures the information about which edge and resource implement this operation.	
	Type	object_id or empty string
	Access	read only
scheduled_edge	The edge on which this operation is scheduled.	
	Type	object_id
	Access	read only
scheduled_phase	The phase in which this operation is scheduled. This will return 0 if the op is not scheduled at all, or is scheduled but not in a pipeline.	
	Type	unsigned integer
	Access	read only
scheduled_stage	The stage in which this operation is scheduled. This will return 0 if the op is not scheduled at all, or is scheduled but not in a pipeline.	
	Type	unsigned integer
	Access	read only
seq_slack	Set during CDFG timing analysis – the sequential slack for this operation	
	Type	integer [default is INT_MAX (not set)]
	Access	read only

slack	Set during CDFG timing analysis – the slack of the bound instance for this operation	
	Type	integer [default is INT_MAX (op not bound)]
	Access	read only
src_links	The SystemC source file locations for this operation	
	Type	list of strings of the form: file:line:char
	Access	read only
tag	The tag that controls whether or not this operation is executed	
	Type	object_id
	Access	read only
type	The type of this object	
	Type	string (op)
	Access	read only
uarch_action	The micro-architecture action to be applied to this custom op.	
	Type	string (inline , or no_action) (default is no_action)
	Access	read/write
	Op Type	custom ops
use_dedicated_resource	Directs the scheduler to use a dedicated resource for this op. Note This attribute must be set after final optimize (transitioning to Analyze Micro-architecture Step) and before Managing States Step.	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read/write

Child Scopes

directives	Contains the directives of this operation
ports	The ports on this op

D.23 Operation Ports Object Attributes [(op_)ports]

A *port* corresponds to a port of a behavior. This table shows the object attributes for ports of an op.

behavior	The behavior containing this port	
	Type	object_id
	Access	read only
direction	The direction of this port	
	Type	string (in or out)
	Access	read only
op	The op to which this port belongs	
	Type	object_id
	Access	read only
signed	Specifies whether this port is signed	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read only
src_links	The SystemC source file locations for this port	
	Type	list of strings of the form: file:line:char
	Access	read only
type	The type of this object	
	Type	string (port)
	Access	read only
width	The width in bits of this port. Each bit is represented by a pin.	
	Type	positive integer
	Access	read only

Child Scope

pins	The pins on this operation port

D.24 Directive Object Attributes

A *directive* corresponds to is an instruction for transforming the design. It is specified using a pragma embedded in the SystemC source code of the design. It is represented in the database as a directive object. A directive object consists of:

- a **text**, which is a string consisting of the tokens following **#pragma ctos**.
- an **object**, which is a database object inferred from the position of the pragma in the source code.
- a **secondary object**, which is a second database object inferred from the position of the pragma in the source code. The secondary object is optional. Most directives do not have a secondary object.

The concatenation of the **text** of the **directive** with the string id of the object, and the string id of the secondary object, if there is one, must form a valid CtoS tcl command.

The secondary object is used for directives associated with a block statement (**{ .. }**), where the object is the node inferred from the **{** and the secondary object is the node inferred from the **}**.

name	The name of the directive	
	Type	string
	Access	read only
text	The text following #pragma ctos	
	Type	string
	Access	read only
command	The first token following #pragma ctos	
	Type	string
	Access	read only
object	The database object implied from the position of the pragma	
	Type	string
	Access	read only
secondary_object	The second database object implied from the position of the pragma (optional)	
	Type	string
	Access	read only

Note The name of a directive will be generated as follows:

- Start with the name of the command (**token1**), unless the command is **set_attr** in which case we start with **<token1>_<token2>**.
- If this is unique (within the scope of the directive) return, otherwise append **_<i>**, where **i** = 0, 1, .. until a unique name is found.

D.25 Pin Object Attributes (Pins)

A *pin* corresponds to a pin in a port of a behavior.

behavior	The behavior containing this pin	
	Type	object_id
	Access	read only
direction	The direction of this pin	
	Type	string (in or out)
	Access	read only
index	The index of this pin in the owning port	
	Type	unsigned integer
	Access	read only
op	The op to which this pin belongs	
	Type	object_id
	Access	read only
port	The port to which this pin belongs	
	Type	object_id
	Access	read only
type	The type of this object	
	Type	string (pin)
	Access	read only
value	The value to which this pin is connected, if connected.	
	Type	object_id or empty string for unconnected outputs
	Access	read only

D.26 Protocol Constraint Object Attributes (protocol_constraints)

Protocol constraints describe a region of the CDFG in which the relative timing of I/O ops will be retained (see “[Creating Protocol Regions](#)” on page 11-14).

end_node	The ending id (of the node or I/O op) for the region to which this constraint applies.	
	Type	object_id
	Access	read only
name	The name of the protocol constraint	
	Type	string
	Access	read only
start_node	The starting id (of the node, I/O op, or function) for the region to which this constraint applies.	
	Type	object_id
	Access	read only
type	The type of this object	
	Type	string (protocol_constraint)
	Access	read only

D.27 Register Binding Object Attributes (reg_bindings)

A *register binding* corresponds to a register binding in a behavior.

behavior	The behavior containing this register binding	
	Type	object_id
	Access	read only
is_inverted	The bit of the register used to store this value	
	Type	boolean [true (1) or false (0)]
	Access	read only

CtoS Object Reference

Register Binding Object Attributes (reg_bindings)

is_primary	The bit of the register used to store this value	
	Type	boolean [true (1) or false (0)]
	Access	read only
index	The bit of the register used to store this value	
	Type	unsigned integer
	Access	read only
name	The name of this register binding	
	Type	string
	Access	read only
node	The state node for this register binding	
	Type	object_id
	Access	read only
primary_binding	The primary binding of a secondary binding (or the binding, itself, if the attribute is queried on a primary binding)	
	Type	register binding
	Access	read only
reg	The register for this register binding	
	Type	object_id or empty string if the register has not yet been assigned for this value in this state.
	Access	read only
secondary_binding	The collection of secondary bindings of a primary binding (or an empty list if the attribute is queried on a secondary binding)	
	Type	Tcl list of register bindings
	Access	read only
tag	The tag for this register binding	
	Type	Tag
	Access	read only
type	The type of this object	
	Type	string (reg_binding)
	Access	read only

value	The value for this register binding	
	Type	object_id
	Access	read only

D.28 Reset Condition Object Attributes (reset_conditions)

A reset_condition corresponds to a reset condition in a behavior.

net	The path to the scalar net of the reset condition	
	Type	object_id
	Access	read only
is_active_high	Indicates the reset condition is active high.	
	Type	boolean (true or false)
	Access	read only
is_async	Indicates the reset condition is asynchronous	
	Type	boolean (true or false)
	Access	read only

D.29 Resource Binding Object Attributes (res_bindings)

A *resource binding* corresponds to a resource binding in a behavior.

behavior	The behavior containing this resource binding	
	Type	object_id
	Access	read only
edge	The edge for this resource binding	
	Type	object_id
	Access	read only

inst	The instance for this resource binding	
	Type	object_id
	Access	read only
is_failure	Indicates that this is a failed resource binding	
	Type	boolean [true (1) or false (0)] [default is false (0)]
	Access	read only
name	The name of this resource binding	
	Type	string
	Access	read only
op	The operation for this resource binding	
	Type	object_id
	Access	read only
tag	The tag for this resource binding	
	Type	object_id
	Access	read only
type	The type of this object	
	Type	string (res_binding)
	Access	read only

D.30 Tag Object Attributes (Tags)

The *tag* in a behavior describes a predicate or condition under which an operation should be performed.

behavior	The behavior containing this tag	
	Type	object_id
	Access	read only
children	The set of children tags of this tag	
	Type	list of object_ids
	Access	read only

control_value	The control value of this tag	
	Type	object_id
	Access	read only
is_root	Specifies whether this tag is a root tag	
	Type	boolean [true (1) or false (0)]
	Access	read only
level	The level of this tag	
	Type	positive integer
	Access	read only
name	The name of this tag	
	Type	string
	Access	read only
parent	The parent of this tag	
	Type	object_id
	Access	read only
reg_bindings	The register bindings associated with this tag	
	Type	object_id
	Access	read only
type	The type of this object	
	Type	string (tag)
	Access	read only

D.31 Behavior Terminal Object Attributes [(behavior_)terms]

A *behavior terminal* corresponds to a terminal in a behavior.

behavior	The behavior containing this terminal	
	Type	object_id
	Access	read only

direction	The direction of this behavior terminal	
	Type	string (in or out)
	Access	read only
name	The name of this behavior terminal	
	Type	string
	Access	read only
net	The net of the connections for this behavior terminal	
	Type	object_id
	Access	read only
signed	Specifies whether this behavior terminal is signed	
	Type	boolean [true (1) or false (0)]
	Access	read only
type	The type of this object	
	Type	string (term)
	Access	read only
width	The bit width of this behavior terminal	
	Type	positive integer (width > 0)
	Access	read only

D.32 Value Object Attributes (Values)

A *value* corresponds to a value in the CDFG.

behavior	The behavior containing this value	
	Type	object_id
	Access	read only
controlled_edges	The set of edges for which this is the control value	
	Type	list of object_ids
	Access	read only

controlled_tags	The set of tags for which this is the control value	
	Type	list of object_ids
	Access	read only
name	The name of this value	
	Type	string
	Access	read only
reg_bindings	The register bindings for this value in different states	
	Type	list of object_ids
	Access	read only
sinks	The input pins using this value	
	Type	object_ids
	Access	read only
source	The output pin sourcing this value if this value is variable	
	Type	object_id or empty string
	Access	read only
type	The type of this object	
	Type	string (value)
	Access	read only
value_type	The type of this value	
	Type	string (either 0 , 1 , x , z , u , or v)
	Access	read only

D.33 Instance Object Attributes (Insts)

An *instance* corresponds to a reference of another module that is instantiated in this module.

area	The area of the object	
	Type	double
	Access	read only
behavior	The behavior that owns this instance	
	Type	object_id or empty string
	Access	read only

CtoS Object Reference

Instance Object Attributes (Insts)

control_delay	The control delay for this instance	
	Type	integer
	Access	read only
data_delay	The data delay for this instance	
	Type	integer
	Access	read only
is_user_created	Specifies whether this instance was created by a designer, rather than by CtoS	
	Type	boolean [true (1) or false (0)]
	Access	read only
launch_delay	The launch delay for this instance	
	Type	integer
	Access	read only
master	The module being referenced by this instance	
	Type	object_id
	Access	read only
max_delay	The maximum delay of this instance. This attribute can be used only after you have run the schedule command.	
	Type	integer [default is INT_MAX (not set) if the timing analyzer has not run on the corresponding implementation]
	Access	read only
module	The module containing this instance	
	Type	object_id
	Access	read only

mux_purpose	If this instance is a multiplexer (mux), the purpose of the mux, as follows:	
	"" (empty string): This instance is not a mux (default). join: This instance implements a join mux op (a join mux selects a value at a join node in the CDFG, based on the in-edge that is active). value: This instance implements a non-join mux op in the data flow (that is, c ? a : b). resource: This instance implements a mux that selects the proper input for a shared resource. register: This instance implements a mux that selects the proper input for a shared register. output: This instance implements a mux that selects the value to be written to an output register when there are multiple writes to the output.	
	Type	string ("", join , value , resource , register , or output) (default is "")
	Access	read only
name	The name of this instance	
	Type	string
	Access	read only
op_constraints	The op constraints associated with this instance	
	Type	Op Constraint
	Access	read only
power	The estimated power consumption of this instance Note <i>Power Estimation is a preliminary feature (see “Low Power Estimation Using CtoS” on page 18-1).</i>	
	Type	double
	Access	read only
reg_bindings	The register bindings associated with this instance	
	Type	list of object_ids
	Access	read only

CtoS Object Reference

Instance Object Attributes (Insts)

res_bindings	The resource bindings associated with this instance	
	Type	list of object_ids
	Access	read only
setup_delay	The setup delay for this instance	
	Type	integer
	Access	read only
schedule_slack	The slack for this instance after successful schedule.	
	Type	integer or “not calculated”
	Access	read only
slack	The slack for this instance after netlisting. The value is assigned on the first call to report_timing or analyze -timing.	
	Type	integer or “not calculated”
	Access	read only
speed_grade	The speed grade of this instance	
	Type	unsigned integer [0 (slowest) to 100 (fastest)] (default is 100)
	Access	read only
src_links	The SystemC source file locations for this instance	
	Type	list of strings of the form: file:line:char
	Access	read only
type	The type of this object	
	Type	string (inst)
	Access	read only

Child Scope

inst_terms	The terminals in this instance

D.34 Instance Terminal Object Attributes (inst_terms)

An *instance terminal* corresponds to a reference of a master terminal in an instance.

direction	The direction of this instance terminal	
	Type	string (in or out)
	Access	read only
input_delay	The input delay of this instance terminal. The unset value is INT_MAX .	
	Type	unsigned integer
	Access	read only
inst	The instance containing this terminal	
	Type	object_id
	Access	read only
is_clock	Specifies whether this instance terminal is a clock	
	Type	boolean [true (1) or false (0)]
	Access	read only
net	The bit net connected to this instance terminal (only for bus_bit_inst_term and scalar_inst_term).	
	Type	object_id
	Access	read only
output_delay	The output delay of this instance terminal. The unset value is INT_MAX .	
	Type	unsigned integer
	Access	read only
signed	Specifies whether this instance terminal is signed	
	Type	boolean [true (1) or false (0)]
	Access	read only
slack	Specifies whether the slack of all paths goes through this instance terminal	
	Type	boolean [true (1) or false (0)]
	Access	read only

src_links	The SystemC source file locations for this instance terminal	
	Type	list of strings of the form: file:line:char
	Access	read only
term	The master terminal that this instance terminal is referencing	
	Type	object_id
	Access	read only
type	The type of this instance terminal	
	Type	string (bus_inst_term , bus_bit_inst_term or scalar_inst_term)
	Access	read only
width	The bit width of this instance terminal	
	Type	positive integer
	Access	read only

Child Scope

bits	The bits of this instance terminal, which have the same properties as the parent (instance terminal).

D.35 Instance Terminal Bits Object Attributes [(inst_term_)bits]

Here are the *instance terminal bits* object attributes.

direction	The direction of this instance terminal bit	
	Type	string (in or out)
	Access	read only
input_delay	The input delay of this instance terminal bit. The unset value is INT_MAX .	
	Type	unsigned integer
	Access	read only
inst	The instance containing this terminal bit	
	Type	object_id
	Access	read only

is_clock	Specifies whether this instance terminal bit is a clock	
	Type	boolean [true (1) or false (0)]
	Access	read only
net	The bit net connected to this instance terminal bit (only for bus_bit_inst_term and scalar_inst_term).	
	Type	object_id
	Access	read only
output_delay	The output delay of this instance terminal bit. The unset value is INT_MAX .	
	Type	unsigned integer
	Access	read only
signed	Specifies whether this instance terminal bit is signed	
	Type	boolean [true (1) or false (0)]
	Access	read only
slack	Specifies whether the slack of all paths goes through this instance terminal bit	
	Type	boolean [true (1) or false (0)]
	Access	read only
src_links	The SystemC source file locations for this instance terminal bit	
	Type	list of strings of the form: file:line:char
	Access	read only
term	The master terminal that this instance terminal bit is referencing	
	Type	object_id
	Access	read only
type	The type of this instance terminal bit	
	Type	string (bus_inst_term , bus_bit_inst_term or scalar_inst_term)
	Access	read only
width	The bit width of this instance terminal bit	
	Type	positive integer
	Access	read only

D.36 Net Object Attributes (Nets)

A *net* corresponds to a net in a module.

driver	The output bit inst terminal connected to the bit net	
	Type	Instance Terminal Bits
	Access	read only
external_input_delay	The input delay of this net, which is set using the external_delay command.	
	Type	integer or “ <i>not set</i> ”
	Access	read only
external_output_delay	The output delay of this net, which is set using the external_delay command.	
	Type	integer or “ <i>not set</i> ”
	Access	read only
full_name	The full name of this net	
	Type	string
	Access	read only
io_ops	The I/O ops associated with this net	
	Type	list of I/O Ops
	Access	read only
keep_signal	Specifies whether the net has been inferred from an sc_signal , and the declaration of the signal has either the ctos keep_signal pragma or the keep_all_signals design attribute is set. For nets that have not been inferred from sc_signals , the value of the attribute is not applicable.	
	Type	boolean [true (1) , false (0) , or not applicable]
	Access	read only
loads	The input bit inst terms connected to the bit net	
	Type	list of Instance Terminal Bits
	Access	read only

module	The module containing this net	
	Type	object_id
	Access	read only
name	The name of this net	
	Type	string
	Access	read only
signed	Specifies whether this net is signed	
	Type	boolean [true (1) or false (0)]
	Access	read only
src_links	The SystemC source file locations for this net	
	Type	list of strings of the form: file:line:char
	Access	read only
type	The type of this net	
	Type	string (bus_net , bus_bit_net or scalar_net)
	Access	read only
width	The bit width of this net	
	Type	positive integer
	Access	read only

Child Scope

bits	The bits of this net, which have the same properties as the parent (net).

D.37 Net Bit Object Attributes [(net_)bits]

Here are the *net bit* object attributes.

driver	The output bit inst terminal connected to this net bit	
	Type	string
	Access	read only
full_name	The full name of this net bit	
	Type	string
	Access	read only
io_ops	The I/O ops associated with this net bit	
	Type	list of I/O Ops
	Access	read only
loads	The input bit inst terms connected to this net bit	
	Type	object_id
	Access	read only
module	The module containing this net bit	
	Type	object_id
	Access	read only
name	The name of this net bit	
	Type	string
	Access	read only
signed	Specifies whether this net bit is signed	
	Type	boolean [true (1) or false (0)]
	Access	read only
type	The type of this net bit	
	Type	string (bus_net , bus_bit_net or scalar_net)
	Access	read only

width	The bit width of this net bit	
	Type	positive integer
	Access	read only

D.38 Module Terminal Object Attributes [(module_)terms]

A *terminal* corresponds to a terminal in a module.

direction	The direction of this module terminal	
	Type	string (in or out)
	Access	read only
full_name	The full name of this module terminal	
	Type	string
	Access	read only
input_delay	The input delay of this module terminal	
	Type	integer
	Access	read only
is_clock	Specifies whether this module terminal is a clock	
	Type	boolean [true (1) or false (0)]
	Access	read only
module	The module containing this module terminal	
	Type	object_id
	Access	read only
name	The name of this module terminal	
	Type	string
	Access	read only
output_delay	The output delay of this module terminal	
	Type	integer
	Access	read only

CtoS Object Reference

Module Terminal Object Attributes [(module_)terms]

signed	Specifies whether this module terminal is signed	
	Type	boolean [true (1) or false (0)]
	Access	read only
slack	The slack of this module terminal	
	Type	integer
	Access	read only
src_links	The SystemC source file locations for this module terminal	
	Type	list of strings of the form: file:line:char
	Access	read only
type	The type of this object	
	Type	string (bus_term , scalar_term , etc.)
	Access	read only
width	The bit width of this module terminal	
	Type	positive integer
	Access	read only

Child Scope

bits	The bits of this terminal, which refers to BitInstTerms, and has the same properties as the parent (terminal).

D.39 Module Terminal Bits Object Attributes [(module_term_)bits]

Here are the *module terminal bits* object attributes.

direction	The direction of this module terminal bit	
	Type	string (in or out)
	Access	read only
full_name	The full name of this module terminal bit	
	Type	string
	Access	read only
input_delay	The input delay of this module terminal bit	
	Type	integer
	Access	read only
is_clock	Specifies whether this module terminal bit is a clock	
	Type	boolean [true (1) or false (0)]
	Access	read only
module	The module containing this module terminal bit	
	Type	object_id
	Access	read only
name	The name of this module terminal bit	
	Type	string
	Access	read only
net	The bit net connected to this module terminal bit	
	Type	Net Bit
	Access	read only
output_delay	The output delay of this module terminal bit	
	Type	integer
	Access	read only

CtoS Object Reference

Module Terminal Bits Object Attributes [(module_term_bits)]

signed	Specifies whether this module terminal bit is signed	
	Type	boolean [true (1) or false (0)]
	Access	read only
slack	The slack of this module terminal bit	
	Type	integer
	Access	read only
type	The type of this object	
	Type	string (bus_bit_term)
	Access	read only
width	The bit width of this module terminal bit	
	Type	positive integer
	Access	read only

D.40 RTL IP Definition Object Attributes (*rtl_ip_defs*)

Here are the *RTL IP definition* object attributes.

name	The name of RTL IP, which must match the name of the behavior of the custom op	
	Type	string
	Access	read-only
pipeline_depth	The total number of cycles after which output is valid, after input is available for pipelined RTL IP	
	Type	unsigned integer (see Table H-6 on page H-28 and Table H-7 on page H-28 for restrictions)
	Access	read-only
rtl_filename	The filename for the RTL IP source, for example, my_add.v	
	Type	string
	Access	read-only
rtl_language	The language for the design specification	
	Type	string (verilog or liberty) Note The rtl_filename must be omitted or empty when the rtl_language is liberty.
	Access	read-only
rtl_type	The type of the RTL design	
	Type	string (combinational or pipelined)
	Access	read-only
type	The type of this object	
	Type	string (rtl_ip_def)
	Access	read only

Child Scope

ports	The optional definition of any auxiliary ports and their bit widths.

D.41 RTL IP Definition Port Object Attributes [(rtl_ip_def_)ports]

Here are the *RTL IP definition port* object attributes.

active_edge	The active edge of the clock (only for port_type of clock)	
	Type	string (fall or rise)
	Access	read-only
active_level	The active level of the reset (only for port_type of reset , stall , valid_in , valid_out)	
	Type	string (low or high)
	Access	read-only
direction	The direction of the port (only for port_type of input or output)	
	Type	string (in or out or io)
	Access	read-only
name	The name of this port	
	Type	string
	Access	read-only
port_type	The type of the this port	
	Type	string (clock or input or output or reset or stall or valid_in or valid_out)
	Access	read-only
reset_type	The type of the reset (only for port_type of reset)	
	Type	string (asynch or synch)
	Access	read-only
type	The type of this object	
	Type	string (rtl_ip_port)
	Access	read only
width	The width of this port (only for port_type of input or output)	
	Type	unsigned integer
	Access	read only

D.42 Default Simulation Configuration Object Attributes (simulation_configs)

Here are the *default simulation configuration* object attributes.

Note See also “[Defining a Simulation Configuration](#)” on page 7-18 and “[define_sim_config](#)” on page E-55.

design	The name of the design for which to define the simulation configuration	
	Type	string (default is name of design)
	Access	read only
makefile_name	The name of the makefile	
	Type	string (default is makefile)
	Access	read only
model_dir	The name of the model directory	
	Type	string (default is model)
	Access	read only
name	The name of the simulation configuration	
	Type	string (default is default_sim_config)
	Access	read only
simulator_args	The set of simulator arguments	
	Type	string
	Access	read only
success_msg	The message generated by the simulation run, indicating success.	
	Type	string
	Access	read only
testbench_files	The list of testbench files	
	Type	string
	Access	read only
testbench_kind	The type of testbench	
	Type	string (simple_diff self_checking) (default is simple_diff)
	Access	read only
type	The type of this object	
	Type	string (simulation_config)
	Access	read only

D.43 Default Synthesis Configuration Object Attributes (*synthesis_configs*)

Here are the *Default Synthesis Configuration* object attributes.

Note See also “Generating Gates in CtoS” on page 13-42 and “define_synth_config” on page E-56.

config_file_name	The name of the <i>User Configuration File</i>	
	Type	file name (default is empty string)
	Access	read only
run_dir	The name of the run directory	
	Type	run directory (default is run_synth_gates)
	Access	read only
script	The name of the script	
	Type	string (default is default_synthesis_flow)
	Access	read only
script_type	The type of the script	
	Type	string (standard_flow , custom_flow , direct_rc_script) (default is standard_flow)
	Access	read only

E CtoS Command Reference

This appendix provides the syntax for all CtoS commands and executables, in alphabetical order.

It also includes a table that relates the CtoS commands to the CtoS Design Flow:

- [“Command Usage Related to CtoS Design Flow” on page E-2.](#)

For additional help with entering CtoS commands, you might find the following sections useful:

- [“Conventions and Syntax in this User Guide” on page 3-3](#)
- [“Identifying Objects for Commands” on page D-6](#)
- [“CtoS Design Flow \(Chart Format\)” on page 4-2](#)

E.1 Command Usage Related to CtoS Design Flow

Many CtoS Tcl commands can be used only at certain times in the CtoS Design Flow [see “[CtoS Design Flow \(Text Format with References\)](#)” on page 4-3]. Here is a detailed table, showing this command usage.

Table E-1 Command Usage Related to CtoS Design Flow

Command Name	Start	Set up Design	Specify Micro-arch	Allocate IP	Analyze Micro-arch	Manage Array Dep	Manage States	Manage Resources	Scheduling	Schedule Complete	Manage Registers	Registers Allocated	Analyze, Implement	End
new_design	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
open_design	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
define_clock		✓	✓	✓										
remove_clock		✓												
external_delay			✓	✓	✓	✓	✓	✓						
write_setup		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
build		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
inline			✓											
inline_calls			✓											
pipeline_function			✓											
convert_to_lookup			✓											
break_combinational_loop			✓											
find_combinational_loops			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
unroll_loop			✓											
pipeline_loop			✓											
flatten_array			✓											
merge_arrays			✓											
split_array			✓											
restructure_array			✓											
split_op			✓											
set_synthesis_mode			✓											
set_uarch_action			✓	✓										
apply_uarch_action			✓	✓										
report_summary			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
report_area			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
report_sccs			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
write_wrapper			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
write_verilog_wrapper			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
write_sim			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
define_control_error_terminal			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
undefine_control_error_terminal			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
read_ip_defs			✓	✓										
use_ip			✓	✓										
use_dsp			✓	✓										
allocate_builtin_ram				✓										
allocate_memory				✓										
allocate_prototype_memory				✓										
allocate_memory_interfaces				✓										
create_rom_program			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
optimize					✓									
analyze_timing_pre_schedule					✓	✓	✓	✓	✓	✓	✓	✓		

Command Name	Start	Set up Design	Specify Micro-arch	Allocate IP	Analyze Micro-arch	Manage Array Dep	Manage States	Manage Resources	Scheduling	Schedule Complete	Manage Registers	Registers Allocated	Analyze, Implement	End
report_timing -type pre_schedule					✓	✓	✓	✓	✓	✓	✓	✓		
analyze -power -pre_schedule					✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
report_pipeline_errors					✓	✓	✓	✓	✓					
create_array_dependencies					✓									
break_array_dependency						✓								
break_array_inter_iteration_dependencies						✓								
float_array_accesses			✓	✓	✓	✓								
report_array_dependencies			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
float_io_accesses			✓	✓	✓	✓								
create_protocol_region			✓	✓	✓	✓								
set_stall_value			✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
create_required_states							✓							
create_state			✓	✓	✓	✓	✓	✓	✓					
constrain_latency			✓	✓	✓	✓	✓	✓	✓					
report_latency - birthday			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
report_opspan							✓	✓	✓	✓				
create_initial_resources								✓						
create_resource			✓	✓	✓	✓	✓	✓	✓					
constrain_op			✓	✓	✓	✓	✓	✓	✓					
report_resources			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
schedule									✓					
report_schedule										✓	✓	✓	✓	✓
report_latency -scheduled										✓	✓	✓	✓	✓
unschedule										✓	✓	✓	✓	✓
create_register											✓			
bind_value											✓			
report_registers												✓	✓	✓
allocate_registers											✓			
find_by_annotation_path			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
report_states			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ename_state													✓	✓
write_rename_states_script													✓	✓
write_rtl													✓	✓
connect										✓	✓	✓	✓	✓
write_top_wrapper													✓	✓
analyze -timing (structural)													✓	✓
report_timing (structural)													✓	✓
report_paths													✓	✓
report_slack													✓	✓
analyze -area (structural)													✓	✓
report_area (structural)													✓	✓
read_tcf			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
analyze -power (structural)													✓	✓
report_power (structural)													✓	✓
save_design			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
close_design		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
check_design			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
define_tlm_transactor_pair		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
undefine_tlm_transactor_pair		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
write_tlm_wrapper			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
report_tlm_transactor_pairs		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
set_baseline			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
report_incremental			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

CtoS Command Reference

Command Usage Related to CtoS Design Flow

Command Name	Start	Set up Design	Specify Micro-arch	Allocate IP	Analyze Micro-arch	Manage Array Dep	Manage States	Manage Resources	Scheduling	Schedule Complete	Manage Registers	Registers Allocated	Analyze, Implement	End
define_sim_config		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
write_sim_makefile			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
launch_sim			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
define_synth_config		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
write_synth_makefile			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
write_rc_run_script													✓	✓
write_rc_script													✓	✓
write_gates													✓	✓

E.1.1 **add_command**

Syntax

```
add_command [-h] -tcl_name name -help string [-first_step step_type]  
            [-last_step step_type] cmd_name
```

Precondition

The Tcl procedure must exist.

Postcondition

The command has been added.

Action

Adds the specified Tcl procedure as a CtoS command.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-tcl_name *name*

Specifies the name of the Tcl procedure to execute.

-help *string*

Specifies the help description for this command.

[-first_step *step_type***]**

Specifies the first valid command step (the default is **start**).

[-last_step *step_type***]**

Specifies the last valid command step (the default is **end**).

cmd_name

Specifies the name of the command to be added.

E.1.2 **add_message**

Syntax

```
add_message [-h] -owner name -msg formatted_text [-extended_msg text]  
            [-severity info|warning|error] msg_id
```

Precondition

None

Postcondition

The message has been added.

Action

Adds the specified message to the CtoS message system to be subsequently printed with the **print_message** command ([“print_message” on page E-95](#)).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-owner *name*

Specifies the scope of the message id.

-msg *formatted_text*

Provides a short description of the message that supports formatted arguments.

[-extended_msg *text*]

Provides a longer description to be printed by the help command for this message id.

[-severity *info|warning|error*]

Identifies the severity level of the message (the default is **error**).

msg_id

Specifies the identifier of the message.

E.1.3 allocate_builtin_ram

Syntax

```
allocate_builtin_ram [-h] [-no_auto_interface_allocation]
                    [-read_interfaces num] [-write_interfaces num]
                    [-read_write_interfaces num] [-min_write_width width]
                    [-read_latency num_cycles] [-sync_read] [-clock clock_net] [array_id]
```

Note The *-no_auto_interface_allocation*, *-min_write_width*, *-read_latency*, and all the *_interfaces* options are preliminary features.

Preconditions

The module containing the array has completed the Specifying Micro-architecture Step, and you have done nothing in the Analyze Micro-architecture Step or beyond.

The specified array must *not* have already been bound to a memory and must *not* have external access.

Postcondition

You may continue allocating IP, or if done, you may proceed to the Analyze Micro-architecture Step.

Action

Creates a synthesizable memory instance and binds it to the specified array. See also “[Allocating Built-In RAM](#)” on page 9-3.

Known Limitation

Built-in RAMs cannot be allocated to external arrays – you must use prototype memories instead.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-no_auto_interface_allocation]

Indicates that you do *not* want CtoS to automatically assign memory interfaces to processes accessing this array. The default is *false* (that is, the option is not specified). If you do specify it, you must explicitly call the **allocate_memory_interfaces** command (“[allocate_memory_interfaces](#)” on page E-11).

[-read_interfaces num]

Specifies the number of read interfaces. The default is the number of processes reading the memory.

[*-write_interfaces num*]

Specifies the number of write interfaces. The default is the number of processes writing to the memory.

[*-read_write_interfaces num*]

Specifies the number of read-write interfaces. The default is **0**.

[*-min_write_width width*]

Specifies the smallest write unit to be supported by the memory, which must be a non-zero positive integer less than the word size. If it is not an integral divisor of the word size, the optimizer will automatically resize the array (and notify you). If you use this option, you must also specify an *array_id*, that is, it cannot default to *all* arrays.

[*-read_latency num_cycles*]

Specifies the latency of memory reads in clock cycles. The default is **1**, and the maximum value is **3**.

[*-sync_read*]

Specifies that all memory reads are synchronous.

[*-clock clock_net*]

Specifies the clock net to be connected to the clock of the memory. If the **-clock** option is not specified, the clock of the processes that access the array is used.

Note the following restrictions for the **-clock** option:

- *clock_net* must be hierarchically connected to a scalar in-port on the top module, which is defined as a clock (using the **define_clock** command).
- The **array_id** argument is mandatory while using this **-clock** option.

[*array_id*]

Identifies an array. The default is all arrays in the current design except with **-min_write_width**.

E.1.4 allocate_memory

Syntax

```
allocate_memory [-h] [-export] [-no_export] [-register_input] [-register_output]
               [-multiplex {counts}] [-no_auto_interface_allocation] [-clock clock_net]
               memory_def array_id
```

Note The *-export* *-no_export*, *-register_input*, *-register_output*, *-multiplex {counts}*, and *-no_auto_interface_allocation* options are preliminary features.

Preconditions

The module containing the array has completed the Specifying Micro-architecture Step, and you have done nothing in the Analyze Micro-architecture Step or beyond.

The specified array must *not* have already been bound to a memory.

Postcondition

You may continue allocating IP, or if done, you may proceed to the Analyze Micro-architecture Step.

Action

Creates a Vendor memory (RAM or ROM) instance and binds it to the specified array. The Vendor memory width and number of words should be at least as large as the array width and number of words.

The Vendor memory should have sufficient read and write ports to provide one port per behavior reading or writing the array, respectively. If there are insufficient ports, you can specify *-multiplex count* for one of the interfaces. When sharing interfaces, auto interface allocation is not supported and you must specify the option **-no_auto_interface_allocation** and then explicitly assign interface using the **allocate_memory_interfaces** command.

If the memory should be registered then specify the **register_input** and **register_output** options. The total latency is increased for each register above the read latency specified in the memory definition.

See also [“Allocating Vendor RAM” on page 9-10](#) and [“Allocating Vendor ROM” on page 9-14](#)

Known Limitations

Vendor memories that support asynchronous read cannot be allocated to arrays that have external access or shared interfaces.

For more limitations, see also [“Limitation when Using Vendor Memories” on page 9-14](#).

In this release, only one memory interface can be shared and this array access can not be an external access.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-export] [-no_export]`

Specifies whether you want to export this memory. If neither option is specified, the value of the **default_export_memories** design attribute (“[default_export_memories](#)” on page D-13) is used. After allocation, the **is_exported** array attribute (“[is_exported](#)” on page D-37) on the memory instance reflects whether this memory will be exported. These options are invalid for external arrays.

`[-register_input]`

Creates registers between RTL and memory inputs. The default value is **false**.

`[-register_output]`

Creates registers between memory outputs and RTL. The default value is **false**.

`[-multiplex {counts}]`

Specifies the multiplex count for each interface. This list is ordered by the memory interfaces in the memory definition.

`[-no_auto_interface_allocation]`

Indicates that you do *not* want CtoS to automatically assign memory interfaces to processes accessing this array. This must be specified if the array has external access or shared interfaces. The default is *false* (that is, the option is not specified). If you do specify it, you must explicitly call the **allocate_memory_interfaces** command (“[allocate_memory_interfaces](#)” on page E-11).

`[-clock clock_net]`

Specifies the clock net to be connected to the memory clock. If the **-clock** option is not specified, the clock of the processes that access the array is used.

Note the following restrictions for the **-clock** option:

- *clock_net* must be hierarchically connected to a scalar in-port on the top module, which is defined as a clock (using the **define_clock** command).
- It is an error to specify the **-clock** option when **clock_per_port** is specified in Vendor RAM Def.

memory_def

Identifies the memory definition of the Vendor memory.

array_id

Identifies an array to be allocated.

E.1.5 **allocate_memory_interfaces**

The allocate_memory_interfaces command is a preliminary feature.

Syntax

```
allocate_memory_interfaces [-h] -interfaces indices  
    [-process behavior_id | -inst inst_id] array_id
```

Preconditions

The module containing the specified array has completed the Specifying Micro-architecture Step, and you have done nothing in the Analyze Micro-architecture Step or beyond.

Also note the following about the command arguments:

- A vendor memory, prototype memory, or builtin-RAM must have already been allocated to the specified *array*.
- A previous **allocate_memory_interfaces** command must not have been run on the specified *array* and *process* or *instance* pair. In other words, you should include all of the indices for a particular pair at the same time and not issue individual commands for each index.
- Each *index*:
 - must be in the range of **0** to the number of interfaces on the allocated memory definition minus **1**
 - must not have been allocated already for the specified array
- If an *instance* is specified, the instance must belong to the same module as the module containing the array.
- If a *process* is specified, the process must belong to the same module as the module containing the array.

Postconditions

This command will check to see if the Allocate IP step is complete for the module containing the specified array.

If the following conditions are met, the module will be considered to be in the next step, Analyze Micro-architecture:

- All processes accessing the array have been assigned interfaces, and
- All instances referred to in the external array bindings of the array have been assigned interfaces (applies only to arrays with external access).

Action

Allocates the specified interfaces of the memory (allocated to the specified array) to the module of the specified process or instance, as follows:

- If a *process* is specified, this command instantiates bridges within the module containing the process. If the array is external, it adds memory-interface terminals to the module containing the array. If the module containing the array (for example, **M1**) and that containing the process (for example, **M2**) are *not* the same, it adds memory-interface terminals to all modules between **M1** and **M2** in the module hierarchy (including **M2**).
- If an *instance* is specified, the command adds memory interface terminals to the master module of the instance.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`-interfaces indices`

Provides a list of the indices of the interfaces (unsigned integers with value ≥ 0), for example `{0 1 2}`

`-process behavior_id`

Identifies the process to which memory interfaces are to be allocated.

`-inst inst_id`

Identifies the module instance containing external arrays to which memory interfaces are to be assigned.

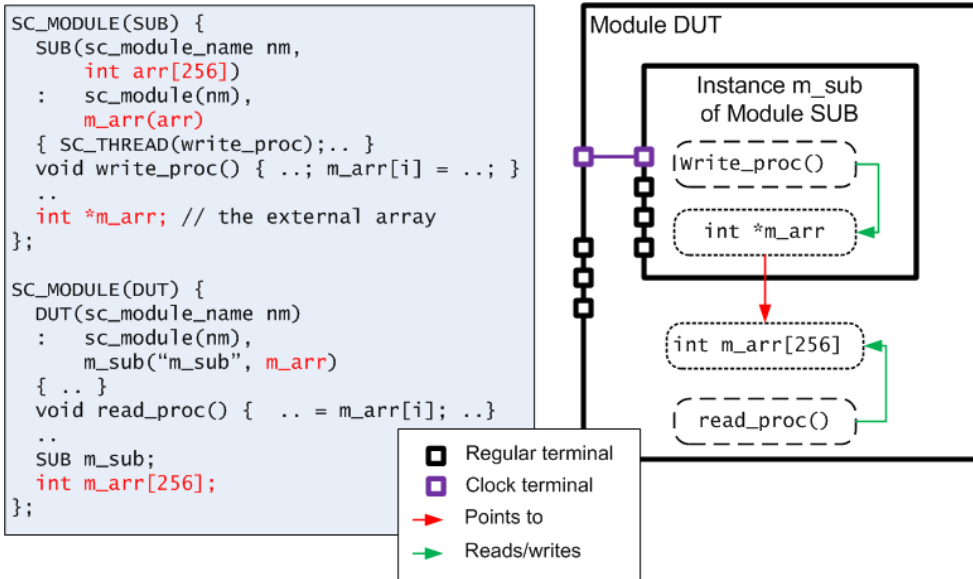
`-array array_id`

Identifies the array whose memory's interfaces are being allocated.

Example

An example of allocating memory interfaces for a 2-ported RAM is shown on the following page.

Figure E-1 Allocating Memory Interfaces Example, 2-port RAM



Synthesis of SUB:

```

cd $SUB
allocate_ram $ramdef arrays/arr -no_auto_interface_allocation
allocate_memory_interfaces arrays/arr behaviors/write_proc 0
# allocation for arrays/arr is now complete

```

Synthesis of DUT:

```

cd $DUT
allocate_ram $ramdef arrays/m_arr -no_auto_interface_allocation
allocate_memory_interfaces arrays/m_arr behaviors/read_proc 1
allocate_memory_interfaces arrays/m_arr insts/m_sub 0
# allocation for arrays/m_arr is now complete

```

Synthesis of entire design (note that the order is unimportant, except the **allocate_memory_interfaces \$arr** commands must come after the **allocate_memory \$arr** commands):

```

cd $DUT
allocate_ram $ramdef arrays/m_arr -no_auto_interface_allocation
allocate_memory_interfaces arrays/m_arr behaviors/read_proc 1
allocate_memory_interfaces arrays/m_arr $SUB/behaviors/write_proc 0
# allocation for arrays/m_arr is now complete
cd $SUB
allocate_ram $ramdef arrays/arr -no_auto_interface_allocation
allocate_memory_interfaces arrays/arr behaviors/write_proc 0
# allocation for arrays/arr is now complete

```

E.1.6 allocate_prototype_memory

The allocate_prototype_memory command is a preliminary feature.

Syntax

allocate_prototype_memory [-h]

Memory with shared interfaces:

```
allocate_prototype_memory -interface_types {types} -multiplex {counts}
    -no_auto_interface_allocation
    [-read_latency num_cycles] [-register_input] [-register_output]
    [-min_write_width width]
    [-export] [-no_export]
    [-chip_enable_low] [-write_enable_low]
    [-clock clock_net]
    [array_id]
```

Memory without shared interfaces:

```
allocate_prototype_memory [-h]
    [-read_interfaces num] [-write_interfaces num] [-read_write_interfaces num]
    [-read_latency num_cycles] [-register_input] [-register_output]
    [-min_write_width width]
    [-export] [-no_export]
    [-chip_enable_low] [-write_enable_low]
    [-clock clock_net]
    [array_id]
```

Preconditions

The module containing the array has completed the Specifying Micro-architecture Step, and you have done nothing in the Analyze Micro-architecture Step or beyond.

The specified array must *not* have already been bound to a memory.

Postcondition

You may continue allocating IP, or if done, you may proceed to the Analyze Micro-architecture Step.

Action

Creates a prototype RAM for an array, similar to the **allocate_builtin_ram** command ([“allocate_builtin_ram” on page E-7](#)).

Setup and launch delays are calculated from the clock period and the **prototype_memory_setup_delay** and **prototype_memory_launch_delay** design attributes ([“prototype_memory_setup_delay” on page D-21](#) and [“prototype_memory_launch_delay” on page D-21](#)).

If the memory being prototyped has sufficient read and write ports to provide one port per behavior reading or writing the array, respectively then you may declare the number using the **-read_interfaces**, **-write_interfaces** and **-read_write_interfaces** options. CtoS can perform automatic interface allocation with the following rules:

- **read_interfaces** have the lowest index numbers.
- **read_write_interfaces** have the highest index numbers.
- **write_interfaces** have index numbers between those of read interfaces and those of write interfaces.

For example:

```
allocate_prototype_memory -write_interfaces 1 -read_write_interfaces 1
                          -read_interfaces 1
type of interface 0: read
type of interface 1: write
type of interface 2: read_write
```

If there are insufficient ports, you must provide an ordered list of interfaces (**-interfaces** option) and multiplex count for of the interfaces (**-multiplex** option). When sharing interfaces, auto interface allocation is not supported and you must also specify the option **-no_auto_interface_allocation** and then explicitly assign interface using the **allocate_memory_interfaces** command.

If the memory should be registered then specify the **register_input** and **register_output** options. The total latency is increased for each register above the specified read latency.

Area is not estimated and is set to zero.

Prototype memories should not be used in the final implementation of a design.

See also [“Allocating Prototype Memory” on page 9-7](#).

Known Limitations

See [“Limitations when Using Prototype Memories” on page 9-10](#).

In this release, only one memory interface can be shared and this array access can not be an external access.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-interface_types {types}]

Specifies the ordered list of the interface types. Interface type can be ‘rw’, ‘r’, or ‘w’.

[-multiplex {counts}]

Specifies the multiplex count for each interface in the same order as interface types declared above.

`[-no_auto_interface_allocation]`

Indicates that you do *not* want CtoS to automatically assign memory interfaces to processes accessing this array. This must be specified if the array has external access or interface is shared. The default is *false* (that is, the option is not specified). If you do specify it, you must explicitly call the **allocate_memory_interfaces** command ([“allocate_memory_interfaces” on page E-11](#)).

`[-read_interfaces num]`

Specifies the number of read interfaces. The default is **0**.

`[-write_interfaces num]`

Specifies the number of write interfaces. The default is **0**.

`[-read_write_interfaces num]`

Specifies the number of read-write interfaces. The default is the number of processes accessing the memory.

`[-min_write_width width]`

Specifies the smallest write unit to be supported by the memory, which must be a non-zero positive integer less than the word size. If it is not an integral divisor of the word size, the optimizer will automatically resize the array (and notify you). If you use this option, you must also specify an *array_id*, that is, it cannot default to *all* arrays.

`[-read_latency num_cycles]`

Specifies the latency of memory reads in clock cycles. The default is **1**, and the maximum value is **3**. If value 3 is specified, then the `register_input` and `register_output` options must not be set.

`[register_input]`

Creates registers between RTL and memory inputs. The default value is **false**.

`[register_output]`

Created registers between memory outputs and RTL. The default value is **false**.

`[-export] [-no_export]`

Specifies whether you want to export this memory. If neither option is specified, the value of the **default_export_memories** design attribute ([“default_export_memories” on page D-13](#)) is used. After allocation, the **is_exported** array attribute ([“is_exported” on page D-37](#)) on the memory instance reflects whether this memory will be exported. These options are invalid for external arrays.

`[-chip_enable_low]`

Specifies the chip enable to be active low.

`[-write_enable_low]`

Specifies the write enable to be active low.

`[-clock clock_net]`

Specifies the clock net to be connected to the memory clock. If the **-clock** option is not specified, the clock of the processes that access the array is used.

Note the following restrictions for the **-clock** option:

- *clock_net* must be hierarchically connected to a scalar in-port on the top module, which is defined as a clock (using the **define_clock** command).
- The **array_id** argument is mandatory while using this **-clock** option.

`[array_id]`

Identifies an array. The default is all arrays in the current design, except if you are using the **-min_write_width** option, in which case, you *must* specify an array with *array_id*.

E.1.7 allocate_registers

Syntax

```
allocate_registers [-h] [-min regs | muxes | regs_and_muxes | share] [object_id]
```

Preconditions

The design, module or behavior has completed the Scheduling Step – the **schedule** command (“[schedule](#)” on page E-138) has been run – and you have not previously run the **allocate_registers** command (“[allocate_registers](#)” on page E-18).

If you want this command to (1) add reset logic to registers with a *valid reset path* (as specified by the SystemC code), (2) reset all other non-**sc_signals** to zero, and (3) produce warnings about **sc_signals** that are not reset, set the **reset_registers** design attribute (“[reset_registers](#)” on page D-23) to *true* before running this command.

Postcondition

You have completed the Manage Registers Step and may proceed to the Analyze and Implement Step.

Action

Allocates registers for all processes and non-inlined sequential functions; generates a netlist automatically.

By default, **allocate_registers** adds reset to registers where it is functionally required. The **reset_registers** attribute (“[reset_registers](#)” on page D-23) can be used to reset other registers if this is a test requirement.

See also “[Allocating Registers](#)” on page -20.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-min regs | muxes | regs_and_muxes | share]

Specifies what should be *minimized* when registers are allocated:

- **regs** allocates the *absolute minimum* number of registers and is recommended when a register is *expensive* with respect to a mux. This is the default for ASIC targets.
- **muxes** allocates one register for each output bit of a *resource* whose result is produced in one clock cycle and used in another clock cycle. This is the default for FPGA targets
- **regs_and_muxes** strikes a balance between the two previous options, sometimes achieving better results than just **regs**. *Note that the -min regs_and_muxes option is a preliminary feature.*
- **share** allocates one register for each output bit of a *source code operation* whose result is produced in one clock cycle and used in another clock cycle. As compared with minimizing muxes, minimizing sharing is much less efficient, because if a resource is shared among three ops, with **-min muxes** you get *one* register, but with **-min share** you get *three* registers. However, this may be useful to minimize interconnect cost on FPGAs by reducing input muxes, albeit at the expense of registers.

[*object_id*]

Identifies a design, module, or behavior for which to allocate registers. The default is the current design.

E.1.8 analyze

The *analyze* command is part of Power Estimation, a preliminary feature.

Syntax

```
analyze [-h] [-timing] [-area] [-power] [-propagate] [-pre_schedule] [object_id]
```

Preconditions

For pre-scheduling analysis, you are in the Analyze Micro-architecture Step.

For RTL analysis, you are in the Analyze and Implement Step.

For pre-scheduling or RTL power analysis, the INCISIV simulator has been run to generate the TCF (toggle count format) file.

Notes on Pre-scheduling vs. Post-scheduling Analysis

Generally, you run analysis *after* you perform scheduling. However, you may analyze timing and power at the behavioral level *before* scheduling.

The precision of pre-scheduling analysis and reporting may be much lower than post-scheduling, because sharing decisions can heavily affect timing (due to multiplexers and false paths) and power (due to additional toggling induced by sharing).

Instead of using pre-scheduling power analysis, it is strongly recommended that you use a *fast-scheduling algorithm*, by setting the **scheduling_effort** design attribute (“[default_scheduling_effort](#)” on [page D-14](#)) to *false*, followed by RTL power analysis.

Postconditions

If **-timing** or **-power** is specified, the database is annotated with timing or power information, respectively (see also “[Low Power Estimation Using CtoS](#)” on [page 18-1](#)).

Also, you may use the following report commands to see the results of the analysis:

- **report_area** (“[report_area](#)” on [page E-103](#))
- **report_timing** (“[report_timing](#)” on [page E-130](#))
- **report_power** (“[report_power](#)” on [page E-116](#))

Action

Analyzes the specified module for timing, area, and/or power; or if you do not specify a particular analysis, *all* analyses are run. This analysis results in:

- changes to the speed grades of instances (see also “[Relaxed Latency Scheduling Mode](#)” on page -4)

Note This does not affect the final RTL.

- more accurate results from the execution of subsequent report commands, as a result of the use of speed grades. In the case of the **report_power** command (“[report_power](#)” on page E-116), the use of **analyze-power** is mandatory in order to get any results at all.

Important In order to analyze area, the **analyze** command needs to analyze timing first, and in order to analyze power, it needs to analyze timing and area first.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-timing]

Performs timing analysis, that is, finds the critical path and the slack of every resource.

[-area]

Performs timing analysis and area recovery, that is, analyzes the area of the resources, setting the best speed grades on instances that do not make the slack (more) negative.

[-power]

Performs timing analysis, area recovery, and power analysis, that is, estimates power consumed by all resources. If you use the **-power** option with the **-propagate** option, probabilities are propagated from resource outputs to driven resource inputs [if the TCF (toggle count format) file does not cover all resources].

[-propagate]

Propagates toggle and value probabilities across instances for power analysis.

[-pre_schedule]

Indicates that the analysis of timing should be done at the CDFG level (that is, considering operations, as opposed to RTL resources) without considering binding information, even if it is present (that is, even if scheduling has already been performed). The default is pre-scheduling analysis before you have run the **allocate_registers** command (“[allocate_registers](#)” on page E-18) and RTL analysis afterward. Note that **pre_schedule** option is not supported for power analysis.

[object_id]

Identifies the design whose top module is to be analyzed. The default is the current design.

E.1.9 apply_directive

Syntax

```
apply_directive -h
apply_directive directive_id ...
apply_directive [-root object_id] [-step micro_architecture|allocate_ip]
```

Note *This command is a preliminary feature.*

Preconditions

The preconditions of the commands embedded in the specified directives.

Postcondition

The postconditions of the commands embedded in the specified directives.

Action

For each directive specified:

- The command prints a message stating that the directive is being applied.
- The command sets the current virtual directory (**cd** command) to the behavior closest to the directive. For directives whose object is an array it sets the virtual directory to the module to which the array belongs.
- The command prints a message stating that the command associated with the directive is being executed.
- The command then executes the command associated with the specified directive.
- The command then restores the current virtual directory.
- If the command associated with the directive succeeds, the directive is removed, and the steps above are repeated for the next directive specified. If there are no more directives to be processed the **apply_directive** command succeeds.
- If the command associated with the directive fails, the **apply_directive** command fails and no further directives are pursued. The **apply_directive** command prints a message stating that the given directive has failed.

Command Usages

- **Directives are specified explicitly.** In this case no (root or step) options must be specified. This is the basic usage of the **apply_directive** command. It allows the user to define his own strategy (order) for directives. This is not intended for the average CtoS user.
- **No directives are specified explicitly.** Instead, all directives in the design are applied. The **root** option allows you to restrict this to only directives that affect given design, module or behavior. The **step** option allows you to restrict the directives to only directives that will not advance the step beyond given value.

Command Arguments

`-h`

Provides a brief description of the command syntax and arguments.

`[-root object_id]`

Considers only the directives that affect the given design, module, or behavior.

`[-step micro_architecture|allocate_ip]`

Considers only the directives that do not advance the step beyond the specified step.

`directive_id ...`

Specifies the directive(s) to be applied.

Known Limitations

The **apply_directive** command does not take any **uarch_action** (specified using the **set_uarch_action** or **set_synthesis_mode** commands) into account. Therefore, the **apply_directive** command issues a warning message if the design has any behavior, loop, or array whose **uarch_action** attribute is different from **no_action**.

Similarly, the **set_synthesis_mode** and **apply_uarch_action** commands do not take directives into account. So, these commands issue a warning if the design has any directives.

If you are planning to use directives and synthesis modes on the same design CtoS recommends the following:

- Do not specify any **allocate_built_in_ram**, **allocate_memory**, **allocate_prototype_memory** directives.
- Apply all directives before specifying **synthesis_mode**, and before setting any **uarch_action**.

If you are using directives in your design, it is quite possible that the **apply_directive** command will fail. The log will show which directive failed and the command associated with that directive. Resolving failures depends on the reason for the failure. For instances:

- If there is a typo in the pragma from which the directive has been inferred, you will eventually need to fix this typo in the source text. However, you can continue synthesizing the design by first removing the directive using the **remove_directive** command (see [“remove_directive” on page E-101](#)), and then typing the corrected command in the command window.
- If a pre-condition for the command associated with the directive is not met. You can invoke a transform to address the pre-condition, and then you can re-try the **apply_directive** command. It may also be appropriate to add a pragma to the source code for the "missing" transform.

E.1.10 **apply_uarch_action**

Syntax

`apply_uarch_action [-h] [object_id]`

Note *This command is part of Specifying Micro-architecture with Synthesis Modes, a preliminary feature.*

Precondition

The design has completed the Set up Design Step – the **build** command has been run. This command will have no effect if synthesis mode or micro-architecture actions are not previously set with the **set_synthesis_mode** or **set_uarch_action** commands.

Postcondition

If no arrays are allocated to memories, then you can continue in the Specifying Micro-architecture Step. If at least one but not all arrays are allocated to memories, then you can continue in the Allocate IP step. If all arrays are allocated to memories, then you can continued with Analyzing Micro-Architecture Step or any subsequent step.

Action

Apply micro_architecture actions previously set on loops, functions, custom_ops and arrays contained in the specified design, module or behavior.

See also [“Resolving Functions” on page 8-3](#)

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[object_id]`

Specifies the design, module or behavior that you want to apply micro-architecture actions.

E.1.11 **bind_value**

Syntax

`bind_value [-h] [-reg register_id] [-index index] [-state state_id] value_id`

Preconditions

The behavior containing the value has completed the Scheduling Step – the **schedule** command (“[schedule](#)” on page E-138) *has* been run – and the **allocate_registers** command (“[allocate_registers](#)” on page E-18) *has not* been run.

If you want to bind a value to a *particular* register, you must first use the **create_register** command (“[create_register](#)” on page E-45) to create that register.

If specified, the register, index and/or state must be from the same behavior. If a state is specified, a register binding must exist for the value in that state. If a register is specified, it must be available for each use.

Postcondition

You may continue to bind other values to registers, or use the **allocate_registers** command to complete the Manage Registers Step.

Action

Binds the specified value to an existing register, or CtoS will automatically create one.

The **find -reg_binding** command (“[find](#)” on page E-60) will help you to find all register bindings. The value and the node attributes of a register binding are its value and state node.

See also “[Binding a Value](#)” on page -18

Return Value

The register name (*register_id* if specified: CtoS-created if not)

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-reg register_id]`

Identifies a register resource. The default is that a new register resource is created for this value.

`[-index index]`

Identifies the register index of the binding. The default is **0**.

`[-state state_id]`

Identifies a state node. The default is every state in which a register is needed for that value.

`value_id`

Identifies the value to be bound.

E.1.12 break_array_dependency

Syntax

```
break_array_dependency [-h] array_access_1 array_access_2
```

Precondition

The design is in the Manage Array Dependencies Step, which implies that array dependencies are known for the behavior containing the array, as determined by the **create_array_dependencies** command (“[create_array_dependencies](#)” on page E-41), and you have *not* run the **create_required_states** command (“[create_required_states](#)” on page E-46).

Postcondition

You may break additional array dependencies or proceed to the Manage States Step.

Action

Breaks an array dependency, that is, the dependency from *array_access_1* to *array_access_2* is replaced by a dependency from *array_access_1*'s dependencies to *array_access_2*'s, and in similar fashion, for *array_access_2* to *array_access_1*.

A classic case in which you might want to break a *created* dependency would be the following:

```
for (i=1; i<100000; i++) {  
    A[i] = A[i+200] + 37;  
}
```

Because CtoS does limited analysis to determine if there is overlap between array indexes, an assumption is made that reads from **A** and writes to **A** must be done in order. You could use this command to prevent this false order dependency from being included.

See also “[Breaking Array Dependencies](#)” on page 11-3.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

array_access_1

Identifies the leader array operation, such as read/write array access or a sequential function call, which contains array accesses.

array_access_2

Identifies the trailer array operation, such as read/write array access or a sequential function call, which contains array accesses.

E.1.13 **break_array_inter_iteration_dependencies**

Syntax

```
break_array_inter_iteration_dependencies [-h] [-num_iterations integer]  
loop_join_id array_id
```

Precondition

The design is in the Manage Array Dependencies Step, which implies that array dependencies are known for the behavior containing the array, as determined by the **create_array_dependencies** command (“[create_array_dependencies](#)” on page E-41), and you have *not* run the **create_required_states** command (“[create_required_states](#)” on page E-46).

Postcondition

You may break additional array dependencies or proceed to the Manage States Step.

Action

Breaks the inter-iteration array dependencies for a given array and loop.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-num_iterations integer]

Identifies the number of independent loop iterations (the default is '1' for regular loops or, for pipelineloops, 'maximum latency -1' when maximum latency is given or 'number of stages - 1' otherwise).

loop_join_id

Identifies the loop to break inter-iteration array dependencies.

array_id

Identifies the array to break inter-iteration dependencies.

E.1.14 break_combinational_loop

Syntax

```
break_combinational_loop [-h] [-exclude_loops] loop_join_id
```

Precondition

The design has completed the Set up Design Step – the **build** command (“[build](#)” on page E-30) has been run.

If you want to specify the *exact* place to break the loop, it is recommended that you use the **create_state** command (“[create_state](#)” on page E-50), instead of the **break_combinational_loop** command.

Postcondition

You may continue with the Specify Micro-architecture Step, or proceed to the Allocate IP Step.

Action

Breaks the combinational loop represented by a join node.

The loop is broken by inserting states on the combinational paths in the loop.

If any path in the loop intersects inner loops, this command is invalid unless you have specified the **-exclude_loops** option.

See also “[Breaking Combinational Loops](#)” on page 8-22.

Known Limitations

See “[Limitations When Creating States in Combinational Functions](#)” on page 11-7.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-exclude_loops]

Specifies that the set of paths should contain only *simple* paths, that is, paths that do not cross a backward edge; therefore, no loops will be included.

loop_join_id

Identifies a loop join node (that is, the *head* of the loop). See also “[loop_join_id](#)” on page N-8.

E.1.15 build

Syntax

```
build [-h] [-verbose]
```

Preconditions

You must have created the design and set the **source_files**, **compile_flags**, and **top_module_path** design attributes, as well as any other optional attributes that you want to set (see [“Design Object Attributes - When They Can Be Modified”](#) on page D-9).

To return to the Specify Micro-architecture Step, you can run the **build** command at any time; however, you will lose all of your micro-architectural and scheduling decisions.

By default, CtoS *preserves* the **SC_MODULE** hierarchy, but if you want to *flatten* the hierarchy into a single database module, you must have set the **build_flat** design attribute ([“build_flat”](#) on page D-12) to *true*.

It is recommended that you set the *model directory* using the **define_sim_config** command ([“define_sim_config”](#) on page E-55) if you do not want automatically generated models [which can be created after a successful **build** if the **auto_write_models** design attribute ([“auto_write_models”](#) on page D-11) is set to *true*] to use the default of *./model*.

Interrupt

To stop this process, select the **Interrupt** button (see [“Interrupt Button”](#) on page 6-32). The command stops fairly quickly and resets the design back to before **build**. You can rerun the **build** again (from the beginning) after interrupting it.

Postcondition

You have completed the Set up Design Step and may proceed to the Specify Micro-architecture Step.

Action

Reads input files, elaborates them, and creates data structures representing the many modules and behaviors that make up the design in the CtoS database. See also [“Building a Design”](#) on page 6-28.

Return Value

The object ID of the newly created design

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-verbose]

Reports elaboration progress in more detail (see [“Using the -verbose Option of Build for Enhanced Source Context”](#) on page 6-34).

E.1.16 **cd**

Syntax

`cd [-h] object_path`

Precondition

None

Postcondition

No change

Action

Sets the current scope for identifying objects by a relative object ID.

See also [“Navigating the Virtual Design Directory” on page 6-47](#).

Return Value

The current scope

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

object_path

Identifies the object path to be used as the current scope.

E.1.17 check_design

Syntax

```
check_design [-h] [object_id]
```

Precondition

The design has completed the Set up Design Step – the **build** command (“[build](#)” on [page E-30](#)) has been run.

Note The **check_design** command works well after the Set up Design Step, but you may get a more accurate report after completing the Allocate IP Step.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Reviews a design for inconsistency, producing warnings or errors.

Return Value

A Boolean value indicating whether any such condition was found, that is, it returns *false* if it finds any errors.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[object_id]`

Identifies a design, module, or behavior to be checked. The default is the current design.

E.1.18 **close_design**

Syntax

```
close_design [-h] [-force] [design_id]
```

Precondition

The design has completed the Start Step – the **new_design** command ([“new_design” on page E-88](#)) has been run.

It is highly recommended that you use the **save_design** command ([“save_design” on page E-137](#)) before closing the design; CtoS does not automatically save the design before closing it.

Postcondition

There is no current design.

Action

Closes the specified design.

If no design is specified, CtoS closes and unsets the current design, if there is one.

See also [“Closing a Design” on page 6-53](#).

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-force]`

Indicates that the design is to be closed regardless of state, and you are not prompted to save it (by the CtoS GUI).

`[design_id]`

Identifies the design to be closed. The default is the current design.

E.1.19 connect

Syntax

```
connect [-h] object_id_1 object_id_2
```

Precondition

You have completed the Manage Registers Step – and the **allocate_registers** command ([“allocate_registers” on page E-18](#)) has been run.

Note You will not get an error if you use this command just after scheduling, but before allocating registers. However, it may not perform as desired.

Postcondition

You may connect more objects; when done, you may write out your RTL, using the **write_rtl** command ([“write_rtl” on page E-161](#)).

Action

Connects power or test management terminals of memories and coarse-grained clock gating cells. Such input instance terminals are initially tied off to ground; such output instance terminals are initially left unconnected.

The **connect** command allows to connect input instance terminals for power or test management of memories and coarse-grained clock gating cells to primary inputs of the module in which the memories or clock gating cells are instantiated. It also allows you to connect output terminals of memories and coarse grained clock gating cells to primary outputs of the user-defined module in which the memory is instantiated.

See also [“Connecting Terminals to Add Power or Test” on page -20](#).

Known Limitations

Connections made using the **connect** command are not incorporated into the scheduled simulation models. They are reflected in the RTL model.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

object_id_1, object_id_2

Identifies the two objects to connect; one should be a driver (either an input terminal or an output instance terminal), and the other should be a load (either an output terminal or an input instance terminal).

Both the driver and the load should be in the same module.

E.1.20 **constrain_latency**

Syntax

```
constrain_latency [-h] [-name name] [-max num_states] [-exclude_loops]  
                  [-exclude list_of_edges] start_node_id [end_node_id]
```

Preconditions

The design has completed the Set up Design Step – the **build** command ([“build” on page E-30](#)) *has* been run – and you have *not* run the **schedule** command ([“schedule” on page E-138](#)).

If you want to insert states in a *specific* location, use the **create_state** command ([“create_state” on page E-50](#)), instead of using the **constrain_latency** command.

You cannot set a latency constraint on a region that contains any edge that is *non-expandable* – edges may not be expandable for several reasons, including the presence of op constraints or a protocol region. You will get a warning if you try to do this.

Postconditions

You may continue with the Manage States Step or proceed to the Manage Resources Step.

If this command is successful, latency constraint object attributes have been set [see [“Latency Constraint Object Attributes \(latency_constraints\)” on page D-49](#)].

Action

Sets a constraint for the maximum latency allowed on a set of paths between two control-flow nodes.

See also [“Constraining Latency” on page 11-12](#).

Return Value

The object ID of the newly created latency constraint.

Known Limitations

See [“Limitations When Creating States in Combinational Functions” on page 11-7](#).

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-name name]`

Specifies the name of the latency constraint

`[-max num_states]`

Specifies the maximum number of cycles allowed between the start and end nodes. It cannot be lower than the existing maximum number of cycles that exist when the constraint is set.

`[-exclude_loops]`

Specifies that the set of paths should contain only *simple* paths, that is, paths that do not cross a backward edge; therefore, no loops will be included.

`[-exclude list_of_edges]`

Specifies a list of edges that are not allowed on the paths for which the latency is to be constrained.

start_node_id

Specifies the starting node of the path.

If you do not specify an end node, the start node must be the start of a loop, and the end node will automatically be set to the end of the loop.

If you do not specify an end node, and the start node is not the start of a loop, you will get an error.

You will also get an error if you use the origin node of a sequential function; you must select a different node and add a label to the source code of the function, if necessary.

`[end_node_id]`

Specifies the ending node of the path.

If you do not specify an end node, the start node must be the start of a loop, and the end node will automatically be set to the end of the loop.

If you do not specify an end node, and the start node is not the start of a loop, you will get an error.

E.1.21 constrain_op

Syntax

```
constrain_op [-h] [-type soft | hard | restrict_sharing]
              [-edge edge_name / (-stage stage_number -phase phase_number)]
              [-inst instance_id [-overwrite]]
              op_id
```

Note You must specify at least one edge or instance constraint, that is, you must include at least one of these options: **-edge**, **-stage** or **-inst**.

Preconditions

The behavior in which the op is to be constrained has completed the Specifying Micro-architecture Step, and you have *not* run the **schedule** command ([“schedule” on page E-138](#)).

If the op specified has already been constrained, you will get a warning (unless you have specified the **-overwrite** option, which is not recommended), and the previous constraint will be removed. This includes previous constraints to an instance – if the last call to **constrain_op** does not specify an instance constraint, previous instance constraints will be ignored.

If you are planning to inline a function with *custom ops*, you must do this *before* constraining the op. CtoS does not preserve *custom op* constraints during inlining (see [“inline” on page E-78](#) and [“inline_calls” on page E-79](#)).

If you are planning to unroll a loop, you must do this *before* constraining the op. CtoS does not preserve op constraints during unrolling (see [“unroll_loop” on page E-151](#)).

The edge (specified with the **-edge** option) must meet the qualifications in [“Constraining ops to Edges” on page 11-17](#).

The resource instance (specified with the **-inst** option) must meet the qualifications in [“Constraining ops to Resources” on page 11-23](#).

If you are using **-stage** and **-phase** (with or without **-inst**), you must have run the **pipeline_loop** command ([“pipeline_loop” on page E-92](#)).

Important If you use the **pipeline_loop** command multiple times, every time you use it, the existing constraints specified with the **constrain_op** command will be checked for consistency, as follows:

- If the new **pipeline_loop** command has an initiation interval *different from* the previous **pipeline_loop** command, the stage and phase constraints of all operations from this pipeline loop are .
- If the new **pipeline_loop** command has the *same* initiation interval as the previous **pipeline** command, but *different* latency intervals, the constraints are reset if they constrain a stage inconsistent with a new latency interval (that is, it exceeds the maximum stage number).

Postconditions

You may continue with the Specify Micro-architecture Step or any subsequent step, up to running the **schedule** command (“[schedule](#)” on page E-138).

If the constraint is successful, the scheduler creates an *op constraint* (see “[Operation Constraint Object Attributes \(op_constraints\)](#)” on page D-56).

Action

Creates, prior to scheduling, an *op constraint* for the specified op on an edge [specified by either the edge name (no pipelining) or the stage/phase numbers (with pipelining)] and/or on a resource instance.

See also “[Constraining ops to Resources](#)” on page 11-23, “[Constraining ops to Edges](#)” on page 11-17, and “[Constraining ops to Expandable Edges](#)” on page 11-19.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-type soft | hard | restrict_sharing]`

Specifies the type of constraint:

- **soft** constraints *can* be removed by the CtoS scheduler during scheduling.
Note Incremental synthesis uses soft constraints during the incremental **schedule** step.
- **hard** (default) constraints *cannot* be removed during scheduling.
- **restrict_sharing** specifies to the CtoS scheduler that no other op may share the specified resource. Only ops that specify the particular resource using **restrict_sharing** constraints will share it.

`[-edge edge_name]`

Specifies the edge on which the operation is to be constrained.

`[-stage stage_number]`

Specifies the stage number inside a pipelined loop. This number must satisfy the following:

$$1 \leq [\text{stage\_number}] \leq \text{ceil}(\text{LI}/\text{II})$$

When using this option, the specified operation must not be part of the FSM control logic or a join mux (the scheduling for such ops is fixed).

`[-phase phase_number]`

Specifies the phase number for the stage; this number must satisfy the following:

$$1 \leq [\text{phase\_number}] \leq \text{II}$$

`[-inst instance_id [-overwrite]]`

Specifies the resource instance on which the operation is to be constrained. The **-overwrite** option specifies that a warning not be issued with you overwrite a constraint.

Warning Using the **-overwrite** option is *not* recommended – when you overwrite a constraint, it is usually *unintentional* and the warning is helpful. (This option was created mostly for the CtoS GUI.)

op_id

Identifies the operation to be constrained.

E.1.22 **convert_to_lookup**

Syntax

```
convert_to_lookup [-h] behavior_id
```

Preconditions

The design has completed the Set up Design Step – the **build** command (“[build](#)” on [page E-30](#)) has been run – and you have done nothing in the Allocate IP Step or beyond.

The behavior (***behavior_id***) must meet the requirements in “[Converting a Function to a Table Lookup](#)” on [page 8-15](#)

Postcondition

You may continue with the Specify Micro-architecture Step, especially to use the **inline** or **inline_calls** commands (see “[inline](#)” on [page E-78](#) and “[inline_calls](#)” on [page E-79](#)).

Action

Converts the implementation of a combinational function into a *table lookup*.

The function’s inputs drive the address of the array, and its outputs are driven by the values read out of the array.

See also “[Converting a Function to a Table Lookup](#)” on [page 8-15](#).

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

behavior_id

Identifies the behavior to be converted.

E.1.23 **create_array_dependencies**

Syntax

```
create_array_dependencies [-h] [object_id]
```

Preconditions

The design has completed the Allocate IP Step and optionally completed the Analyze Micro-architecture Step, and you have *not* run the **create_required_states** command ([“create_required_states” on page E-46](#)).

Additionally, you cannot run this command more than once.

Note This command (and the corresponding Manage Array Dependencies Step) is optional. If skipped, it will automatically be run by commands later in the design flow.

Postcondition

You are now in the Manage Array Dependencies Step and can use the **report_array_dependencies** command ([“report_array_dependencies” on page E-105](#)) to determine if you want to break dependencies using the **break_array_dependency** command ([“break_array_dependency” on page E-27](#)).

Action

Analyzes the use of the memories in the object and builds a set of data structures that capture the read and write dependencies of each array in the specified object.

This command begins the Manage Array Dependencies Step in the CtoS Design Flow.

See also [“Creating Array Dependencies” on page 11-2](#).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[object_id]

Identifies a design, module, or behavior for which to create array dependencies. The default is the current design.

E.1.24 create_initial_resources

Syntax

create_initial_resources [-h] [object_id]

Preconditions

The design has completed the Manage States Step and is ready for the Manage Resources Step. You must run this command before the **schedule** command ([“schedule” on page E-138](#)), and you cannot run it more than once.

If you do not want to use *addsub* resources, set the **enable_addsubs** behavior attribute ([“enable_addsubs” on page D-41](#)) to *false* before running this command.

Note This command (and the corresponding Manage Resources Step) is optional. If skipped, it will automatically be run by commands later in the design flow.

Interrupt

To stop this process (and it will stop *after processing the current behavior*), select the **Interrupt** button (see [“Interrupt Button” on page 6-32](#)). This will leave the design in the state of having some behaviors with initial resources and others without.

Postcondition

This command automatically creates required states, if you have not already created them in the CtoS GUI or with the **create_required_states** command ([“create_required_states” on page E-46](#)).

You are now in the Manage Resources Step, and you can use the **create_resource** command ([“create_resource” on page E-47](#)) and/or **constrain_op** command ([“constrain_op” on page E-38](#)) to guide the CtoS scheduler to map specific ops to resources.

Action

Creates the minimum number of shareable resources for use by the CtoS scheduler. Depending on the setting of the **default_scheduling_effort** design attribute (or the **scheduling_effort** behavior attribute), this command will either allow complex ops to share the same resource, if they are mutually exclusive (scheduling effort is set to **medium** or **high**) or create a unique resource for every shareable op (scheduling effort is set to **low**). See also [“Creating Initial Resources” on page 11-20](#).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[object_id]

Identifies a design, module, or behavior for which to create initial resources. The default is the current design.

E.1.25 **create_protocol_region**

Syntax

```
create_protocol_region [-h] [-name name] [-stallable] start_id [end_id]
```

Preconditions

The design has completed the Set up Design Step – the **build** command (“[build](#)” on page E-30) has been run.

The region to be defined must *not* contain any pipelined loops.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

If the command is successful, *protocol constraints* are created [see “[Protocol Constraint Object Attributes \(protocol_constraints\)](#)” on page D-63].

Action

Creates a region in which relative timing of I/O ops and shared memory accesses will be retained. Protocol regions are thus defined on the behavior and can be accessed by name.

See also “[Creating Protocol Regions](#)” on page 11-14

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-name *name*]

Specifies the name of the protocol region.

[-stallable]

Allows the protocol region to be in a stalling pipeline loop and extend over multiple states.

Notes:

- This option is invalid if the region is not within a pipeline.
- If the region is in a pipeline, then the protocol region cannot contain the pipeline state expansion point.
- If this option is not present, then:
 - the region cannot contain states if the pipelined loop contains stall loops.
 - the region cannot contain a loop fork or join node.

start_id

Specifies the starting id (of the node, I/O op, or function) for the protocol region.

[*end_id*]

Specifies the ending id (of the node or I/O op) for the protocol region.

Note the following restrictions on **start_id** and **end_id**:

- **start_id** and **end_id** must be in the same behavior and must be reachable to each other.
- **end_id** is not valid if **start_id** is a function.
- **end_id** is optional if **start_id** is the starting node of a loop (implied **end_id** is last node of loop).
- **end_id** is optional if **start_id** is an I/O op (implies that I/O op is fixed on its birth edge and is the only element of the protocol).

E.1.26 create_register

Syntax

```
create_register [-h] -name register_name [-width width] behavior_id
```

Precondition

The design has completed the Scheduling Step – the **schedule** command (“[schedule](#)” on page E-138) *has* been run – and the **allocate_registers** command (“[allocate_registers](#)” on page E-18) has *not* been run.

Postcondition

You can run the **bind_value** command (“[bind_value](#)” on page E-26) to bind values to this register, or if you have completed the Manage Registers Step, you may run the **allocate_registers** command and proceed to the Analyze and Implement Step.

Action

Creates a register with the specified name and width.

See also “[Creating Specific Registers](#)” on page -18.

Return Value

The object ID of the newly created register

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-name register_name

Specifies the name for the register to be created.

[-width width]

Specifies the width of the D and Q ports of the register. The default is **1**.

behavior_id

Identifies the behavior binding of the register.

E.1.27 create_required_states

Syntax

```
create_required_states [-h] [-verbose] [object_id]
```

Note The *-verbose* option is a preliminary feature.

Preconditions

The design has completed the Allocate IP Step and optionally completed the Analyze Micro-architecture and Manage Array Dependencies Steps.

You must have defined a latency constraint on the affected path(s), using the **constrain_latency** command ([“constrain_latency” on page E-36](#)), or set the **relax_latency** attribute ([“relax_latency” on page D-43](#)) on the behavior.

You must run this command before running the **create_initial_resources** command ([“create_initial_resources” on page E-42](#)), and you cannot run it more than once.

Note This command (and the corresponding Manage States Step) is optional. If skipped, it will automatically be run by commands later in the design flow.

Interrupt

To stop this process, select the **Interrupt** button (see [“Interrupt Button” on page 6-32](#)). The command stops fairly quickly, leaving the states it has already created. You can run the **create_required_states** command again, basically continuing where it left off.

Postcondition

You are now in the Manage States Step, and you can continue to create resources (see [“Creating Individual Resources” on page 11-21](#)) or to create individual states (see [“Creating Individual States” on page 11-9](#)), or you may proceed to the Manage Resources Step.

Action

Inserts states, as long as constraints are not violated, to increase the latency between ops, which is required to avoid *sequential conflicts* or conflicts due to *resource contention* between array ops. For more detail about these conflicts, see [“Creating Required States” on page 11-8](#).

Known Limitations

See [“Limitations When Creating States in Combinational Functions” on page 11-7](#).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-verbose]

Reports the progress in more detail.

[object_id]

Identifies a design, module, or behavior for which states are to be created. The default is the current design.

E.1.28 create_resource

Syntax

```
create_resource [-h] [-name resource_name] [-speed_grade speed_grade]  
               (-type resource_type -widths {resource_widths}) | -op op_id  
               behavior_id
```

Note You *must* type in the **{ }** (curly braces) when you specify the resource widths, for example:

```
create_resource -widths {8 32 32} ...
```

Preconditions

The design has completed the Set up Design Step – the **build** command ([“build” on page E-30](#)) has been run.

You know the array dependencies for the object specified.

You can run this command before or after you have run the **create_initial_resources** command ([“create_initial_resources” on page E-42](#)).

Postcondition

You can now constrain an op to this resource, using the **constrain_op** command ([“constrain_op” on page E-38](#)).

Action

Creates a resource from a type and width specification, or an op specification.

See also [“Creating Individual Resources” on page 11-21](#).

Return Value

The object ID of the newly created resource

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-name resource_name]`

Specifies a name for the resource to be created.

`[-speed_grade speed_grade]`

Specifies the speed grade for the resource, from **0** (slowest) to **100** (fastest and the default).

`-type resource_type`

Specifies the type of resource to create. [Table 11-1 on page 11-22](#) shows supported types and widths.

`-widths {resource_widths}`

Specifies a variable-sized list of required resource bit widths (the default is **1** width).

For some resource types, a single number expresses the width (and is used for the output width and all input widths).

Signed and unsigned multiplier resources require three widths: output width, width of the wider input, and width of the narrower input.

[Table 11-1 on page 11-22](#) shows supported types and required widths. The **{Z, A}** format means the first width is for both **Z** and **A**. The second width is for **B**, and it must be smaller than **A**. Note that you *must* type in the **{ }** (curly braces) when you specify resource widths.

`-op op_id`

Identifies the op for which to create a new resource, which must be *inside* the behavior, specified with **behavior_id**, that calls the function or references the array. Do not use the **-type** and **-widths** options with the **-op** option.

Limitation The **op_id** must identify an op that is a call to a combinational function or a read from a read-only array. Also, the module containing the op must have completed final optimization. You can manually force this by running **optimize module** before calling **create_resource -op**.

`behavior_id`

Identifies the behavior in which the resource is used.

E.1.29 **create_rom_program**

Syntax

```
create_rom_program [-h] -format format -o output read_only_array
```

Precondition

The design has completed the Set up Design Step – the **build** command ([“build” on page E-30](#)) has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Creates a ROM programming file.

See also [“Allocating Vendor ROM” on page 9-14](#).

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`-format format`

Identifies the output format of the programming file, which can be **bin**, **hex**, or **arm**.

`-o output`

Specifies the name of the output programming file to generate.

`read_only_array`

Specifies a read-only array in the design.

E.1.30 create_state

Syntax

```
create_state [-h] [-before] edge_id [new_state_id]
```

Precondition

The design has completed the Set up Design Step – the **build** command (“[build](#)” on page E-30) has been run – and you have not run the **schedule** command (“[schedule](#)” on page E-138).

The edge must meet the qualifications in “[Creating Individual States](#)” on page 11-9.

The behavior must meet the qualifications in “[Limitations When Creating States in Combinational Functions](#)” on page 11-7.

Postcondition

You can now constrain an op explicitly to an edge before or after the newly created state (see “[Constraining ops to Resources](#)” on page 11-23).

Action

Inserts a new state after (or before, if using **-before**) the specified edge.

See “[Creating Individual States](#)” on page 11-9 for more detail, including limitations.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-before]

Indicates that the edge should be inserted *before* the specified edge.

edge_id

Specifies the edge after which the state is to be inserted.

[new_state_id]

Specifies the new state name. The default is **expand_inline_number_of_next_node**.

E.1.31 **ctos (executable)**

Syntax

```
ctos [-h] [-v[ersion]] [-V] [-log log_file] [-replay replay_filename] [-n] [-32]  
      [-64] [src_script]
```

Action

Brings up *CtoS* at the command line.

Command Arguments

`[-h]`

Provides a brief description of the supported command-line arguments.

`[-v[ersion]]`

Shows version information.

`[-V]`

Shows additional version information.

`[-log log_file]`

Logs the complete session in the ***log_file***. The default is **ctos.log**.

`[-replay replay_filename]`

Runs the ***replay_filename***, which should be a CtoS log file that will be scanned for commands.

`[-n]`

Skips the loading of user initialization files.

`[-32]`

Brings up the 32-bit CtoS executable.

`[-64]`

Brings up the 64-bit CtoS executable.

`[src_script]`

Provides a start-up script to the Tcl engine.

E.1.32 ctosgui (executable)

Syntax

```
ctosgui [-h] [-v[ersion]] [-V] [-d working_dir] [-log log_file] [-n]  
        [-geometry WxH+X+Y] [-32] [-64] [src_script]
```

Action

Brings up the *CtoS GUI*.

Command Arguments

[-h]

Provides a brief description of the supported command-line arguments.

[-v[ersion]]

Shows version information.

[-V]

Shows additional version information.

[-d *working_dir*]

Specifies a working directory.

[-log *log_file*]

Logs the complete session in the *log_file*. The default is **ctosgui.log**.

[-n]

Skips the loading of user initialization files.

[-32]

Brings up the 32-bit CtoS GUI executable.

[-64]

Brings up the 64-bit CtoS GUI executable.

[-geometry *WxH+X+Y*]

Lets you specify the initial size and position of the main window, where *W* is the width, *H* is the height, *X* is the x-position, and *Y* is the y-position, for example, **-geometry 400x600+100+200**.

[*src_script*]

Provides a start-up script to the CtoS GUI.

E.1.33 define_clock

Syntax

```
define_clock [-h] -name clock_name -period clock_period [-rise rise_period]
             [-fall fall_period]
```

Precondition

The design has completed the Start Step – the **new_design** command (“[new_design](#)” on page E-88) has been run – and you have done nothing in the Analyze Micro-architecture Step or beyond.

Postcondition

You can now refer to this clock when setting external delays using the **external_delay** command (“[external_delay](#)” on page E-58).

Action

Adds a clock definition to the current design. See also “[Create New Design Wizard \(Page 3\)](#)” on page 6-13.

Important CtoS provides limited support for designs with multiple clocks. If a design has more than one clock, one of them must be the *base*, and its frequency must be a multiple of all other frequencies. In addition, CtoS will not create circuitry to ensure clock domains are properly interfaced; defining such circuitry is solely your responsibility. See “[Limited Support for Multiple Clocks](#)” on page 1-8.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-name clock_name

Identifies the clock, which must be an input port of the top module (specified in **top_module_path**). It cannot have any embedded spaces.

-period clock_period

Specifies the clock period, which must be a minimum of 0, can step by 100ps, up to a maximum of 2^{24} , which is defined by the Encounter RTL Compiler (RC).

[-rise rise_period]

Specifies the time in picoseconds when, in the clock period, the rising edge occurs. It must be a non-negative integer less than the **clock_period** and equal to **fall_period** only if zero, which is the default.

[-fall fall_period]

Specifies the time in picoseconds when, in the clock period, the falling edge occurs. It must be a non-negative integer less than the **clock_period** and equal to **fall_period** only if zero. The default is 50% of the **clock_period**.

E.1.34 **define_control_error_terminal**

Syntax

```
define_control_error_terminal [-h] terminal_id behavior_id
```

Precondition

The design has completed the Set up Design Step – the **build** command (“[build](#)” on [page E-30](#)) has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Uses the *object identifier* of the specified terminal to set the **error_recovery_terminal** behavior attribute (“[error_recovery_terminal](#)” on [page D-42](#)) for the specified behavior, in order to indicate the activation of error recovery for the given behavior.

This attribute can be cleared using the **undefine_control_error_terminal** command (“[undefine_control_error_terminal](#)” on [page E-149](#)).

See also “[Generating an RTL Description after Scheduling](#)” on [page 13-36](#).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

terminal_id

Specifies the scalar output terminal to be used for signaling recovery from an illegal control state of the specified behavior, that is, as the error recovery terminal.

behavior_id

Specifies the behavior (which must be an **SC_CTHREAD**, **SC_SYNC_RESET_THREAD**, or **SC_ASYNC_RESET_THREAD** process) that is to use the error terminal.

E.1.35 define_sim_config

Syntax

```
define_sim_config [-h] [-makefile_name name] [-model_dir directory]  
                  [-simulator_args list_of_args] [-testbench_files filenames]  
                  [-testbench_kind simple_diff|self_checking] [-success_msg string] [design_id]
```

Preconditions

The design has completed the Start Step – the **new_design** command (“[new_design](#)” on page E-88) has been run.

Postcondition

You may continue with the **write_sim_makefile** command (“[write_sim_makefile](#)” on page E-168).

If the command is successful, the simulation configuration object attributes are set or modified [see “[Default Simulation Configuration Object Attributes \(simulation_configs\)](#)” on page D-85].

Action

Defines a design’s simulation configuration; see also “[Defining a Simulation Configuration](#)” on page 7-18.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-makefile_name name]

Specifies the name of the **makefile**. The *name* may be just a name or a full file path, for example:

```
-makefile_name ./sim/Makefile.ies
```

Important The **work** and **run** directories used by the IES simulator are created in the same directory as the **makefile**. Thus, if you specify a path, the **work** and **run** directories will also be in this path, for example, in this case, you would have **./sim/work** and **./sim/run**.

[-model_dir directory]

Specifies the name of the model directory.

[-simulator_args list_of_args]

Specifies the set of simulator arguments.

[-testbench_files filenames]

Specifies the list of testbench files.

[-testbench_kind simple_diff|self_checking]

Specifies the type of testbench. The default is **simple_diff**.

[-success_msg string]

Specifies the message, generated by the simulation run, that indicates success.

[design_id]

Specifies the design to which this command applies. The default is the current design.

E.1.36 **define_synth_config**

Syntax

```
define_synth_config [-h]  
define_synth_config -standard_flow name [-config_file_name filename]  
                    [-run_dir directory] [design_id]  
define_synth_config -direct_rc_script filename [-run_dir directory] [design_id]
```

Preconditions

The design has completed the Set up Design Step – the **build** command (“[build](#)” on page E-30) has been run.

Postcondition

Subsequent calls to the **write_rc_run_script** command (“[write_rc_run_script](#)” on page E-158), **write_gates** command (“[write_gates](#)” on page E-157), and **write_synth_makefile** command (“[write_synth_makefile](#)” on page E-169) will use the new values specified in the *Synthesis Configuration*.

If the command is successful, the *Synthesis Configuration* object attributes are set or modified [see “[Default Synthesis Configuration Object Attributes \(synthesis_configs\)](#)” on page D-86].

Action

Defines a design’s *Synthesis Configuration*; see also “[Configuring Logic Synthesis Runs](#)” on page 13-46.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-standard_flow name

Specifies the name of the standard *Foundation Flow* to run logic synthesis.

[-config_file_name filename]

Specifies the file to configure the specified *Foundation Flow*.

Important This option is not compatible with the **-direct_rc_script** option.

-direct_rc_script filename

Specifies the file path of the RC script to run logic synthesis.

[-run_dir directory]

Specifies the run directory for logic synthesis. The default is **run_synth_gates**.

[design_id]

Specifies the design to which this command applies. The default is the current design.

E.1.37 define_tlm_transactor_pair

Syntax

```
define_tlm_transactor_pair [-h] -transactor name1 -mirror_transactor name2  
                             [design_id]
```

Precondition

The design has completed the Start Step – the **new_design** command (“[new_design](#)” on page E-88) has been run.

Postcondition

You may continue with the Set up Design step or any subsequent step.

Action

Defines a pair of dual TLM transactors for the design. If a transactor pair with ***name1*** as **-transactor** has already been defined, the existing pair is replaced by the new pair.

See also “[TLM Library](#)” on page 15-65.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-transactor *name1*

Specifies the name of the transactor module. This is the name of the **SC_MODULE** that constitutes the transactor.

-mirror_transactor *name2*

Specifies the name of the dual (mirror) transactor. This is the name of an **SC_MODULE** that is a dual of the **SC_MODULE** name specified with the **-transactor** option.

Note If the transactor **SC_MODULE** or the mirror transactor **SC_MODULE** is defined in a namespace, ***name1*** or ***name2***, respectively, must be qualified with that namespace (for example, **xbus::xbus_slave_transactor**).

[*design_id*]

Specifies the design to which this command applies. The default is the current design.

E.1.38 external_delay

Syntax

```
external_delay [-h] [-input delay] [-output delay]  
               [-clock clock] [-edge rise|fall] object_id
```

Precondition

- The design has completed the Set up Design Step.
- The **build** command ([“build” on page E-30](#)) has been run.
- Before all modules in the user defined module tree of the behavior are scheduled.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Sets the external delay of a user-created net, module port, or behavior terminal. For a module port or behavior terminal, you must use the **-input** or **-output** options consistently with the direction of the port; else, you will get an error. With a net, you can use either the **-input** or the **-output** option, or both (in two separate calls). See also [“Specifying External Delay for Process I/O Nets” on page 11-10](#).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-input delay]

Specifies an input delay. That is the time between the reference edge of the launching clock and the time when the input signal at the specified ports or pins becomes stable and the reference edge of the capturing clock. The delay is specified in picoseconds and should be between **0** and the clock period.

[-output delay]

Specifies an external output delay. That is the delay between the time when the output signal at the specified ports or pins becomes stable and the reference edge of the capturing clock. The delay is specified in picoseconds and should be between **0** and the clock period.

Note The output of a thread process or a clocked method is driven directly by a flip-flop, so you need to specify an **external_delay -output** command *only* for outputs of combinational methods.

`[-clock clock]`

Specifies the clock that was created with the **define_clock** command, on which this delay is based. If you do not specify this *clock*, the default is the first defined clock in the current design. If this clock was not specified, or there is more than one clock in your design, you will get an error.

`[-edge rise|fall]`

Specifies the edge of the clock to which this external delay is relative.

object_id

Specifies the net or port on a module or the behavior terminal for which external delay should be set.

E.1.39 find

Syntax

```
find [-h] [-ignore_case] [-root path] [-object_type] name_pattern
```

Precondition

The design has completed the Set up Design step – the **build** command (“[build](#)” on page E-30) has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Finds objects in the virtual directory that match the given arguments. See also “[Finding Objects in the CtoS Virtual Directory](#)” on page 6-48.

Return Value

A list of object IDs. If no objects are found that match, an empty list is returned.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-ignore_case]`

Ignores case when matching root path or name pattern.

`[-root path]`

Specifies the starting directory for the search (the default is the current working directory or the current design). You may use the `*` and `?` wildcards; however, the `/` separator is not matched with wildcards.

`[-object_type]`

Specifies the type of object for which you want to search: `arg`; `array`; `array_constraint`; `behavior`; `behavior_term`; `design`; `directive`; `edge`; `external_array_binding`; `float_constraint`; `inst`; `inst_term`; `inst_term_bit`; `latency_constraint`; `memory_bridge_def`; `memory_bridge_port_def`; `memory_def`; `memory_generator`; `module`; `module_term`; `module_term_bit`; `net`; `net_bit`; `node`; `op`; `op_constraint`; `pin`; `port`; `protocol_constraint`; `reg_binding`; `res_binding`; `reset_condition`; `rtl_ip_def`; `rtl_ip_port`; `simulation_config`; `synthesis_config`; `tag`; `transactor`; `value`; `*` (all objects)

name_pattern

Specifies the glob-style name pattern for matching object names. You may use the * and ? wildcards; however, the / separator is not matched with wildcards. CtoS tries to match ***name_pattern*** to objects assuming scopes are specified in the pattern. If no objects are found, CtoS tries to match objects assuming scopes are *not* specified in the pattern.

E.1.40 **find_by_annotation_path**

Syntax

`find_by_annotation_path [-h] -behavior behavior_id annotation_path`

Precondition

The design has completed the Set up Design step – the **build** command (“[build](#)” on page E-30) has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Returns the list of states that match the **annotation_path** in the specified behavior. For more information, see “[Renaming States in Support of FSM Coverage](#)” on page 7-25.

Return Value

A list of states that match the **annotation_path** in the specified behavior.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`-behavior behavior_id`

Specifies the behavior to be searched.

`annotation_path`

Specifies the annotation path to be matched.

E.1.41 **find_combinational_loops**

Syntax

```
find_combinational_loops [-h] [object_id]
```

Precondition

The design has completed the Set up Design step – the **build** command (“[build](#)” on page E-30) has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Identifies the *join nodes*, which are the nodes that CtoS uses as the *signature* of a combinational loop.

You can then unroll or break the combinational cycles.

See also “[Resolving Loops](#)” on page 8-16.

Return Value

A list of join nodes representing combinational loops.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[object_id]

Identifies a design, a module, or behavior for which to find join nodes. The default is the current design.

E.1.42 **find_from_rtl**

Syntax

```
find_from_rtl [-h] [-line line_number] filename [module_id]
```

Precondition

You must be in the Analyze and Implement Step, which implies you have run the **write_rtl** command ([“write_rtl” on page E-161](#)).

Postcondition

You may continue with the Analyze and Implement Step.

Action

Lets you find database objects in your RTL code by providing a filename (and optionally a line number), which should be associated with an *assign* statement in which the operand in question is performed.

See also [“Generating an RTL Description after Scheduling” on page 13-36](#).

Known Limitation

This command currently supports finding operands only.

Return Value

A list of object IDs associated with operands performed at the specified file location.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-line *line_number*]

Specifies a line number, relative to the top of the RTL file.

filename

Specifies the filename of the RTL file.

[*module_id*]

Identifies the module in which to find the RTL code. The default is the top-level module of the current design.

E.1.43 **find_in_rtl**

Syntax

```
find_in_rtl [-h] object_id [module_id]
```

Precondition

You must be in the Analyze and Implement Step, which implies you have run the **write_rtl** command ([“write_rtl” on page E-161](#)).

Postcondition

You may continue with the Analyze and Implement Step.

Action

Lets you locate a database object representation in your RTL code.

The location identified by the returned filename and line number represents the beginning of the *assign* statement responsible for the operand.

See also [“Generating an RTL Description after Scheduling” on page 13-36](#).

Known Limitation

This command currently supports finding operands only.

Return Value

The filename and line number associated with the given database object.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

object_id

Specifies the database object that you want to look up in the RTL code.

[module_id]

Identifies the module in which to find the object. The default is the top-level module of the current design.

E.1.44 find_source

Syntax

```
find_source [-h]
            [-label label_name] symbol | [-line line_number] filename
```

Precondition

The design has completed the Set up Design step – the **build** command (“build” on page E-30) has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Lets you look up database object ID(s) from a given source reference. The source reference can be either:

1. a symbol name, representing either a class method declaration **<class::method>** or a function declaration **<function>**, and an optional label somewhere in the body

or

2. a filename and a line number

The function returns the object ID(s) associated with the provided source location, which depends on the set of arguments you used:

1. In the first case (symbol, label), the command returns database objects for all matching code locations. When several candidates match the symbol (for example, several overloaded methods or functions), the command considers all of them when looking for objects. If a label is not specified, all of these candidates are returned; if a label is specified and found in more than one candidate, database objects for all of these locations are returned.
2. In the second case (filename, line number), the command returns all objects associated with a given source line specified by the filename-line number pair.

Known Limitation

Namespaces, although supported by the rest of CtoS, are not currently supported in the *symbol* argument.

Return Value

A list of object ID(s) of all object(s) corresponding to matching code locations

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-label label_name]`

Specifies any legal C/C++ label.

symbol

Specifies a C/C++ class method or a global function reference (<**class::method**> or <**function**>)

`[-line line_number]`

Specifies a line number relative to the top of the file.

filename

Specifies a filename to which the line number is relative.

E.1.45 **flatten_array**

Syntax

```
flatten_array [-h] array_id ...
```

Precondition

The module containing the array(s) has completed the Set up Design Step – the **build** command (“[build](#)” on page E-30) has been run.

The array (*array_id*) must meet the requirements in “[Flattening Arrays](#)” on page 8-61.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Removes the array(s) and replaces all reads and writes of the array(s) with the equivalent reading and writing variables representing each word in the array(s).

Important As indicated in the syntax, the **flatten_array** command accepts one or more arrays. It is faster to flatten multiple arrays with one **flatten_array** command than to use multiple **flatten_array** commands for each array.

See also “[Flattening Arrays](#)” on page 8-61.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

array_id

Identifies the array(s) to be flattened.

E.1.46 **flex_channels_nets**

Syntax

`flex_channels_nets [-h] initiator_name`

Precondition

The design has completed the Set up Design Step – the **build** command (“[build](#)” on page E-30) has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Returns a list of nets in the specified flex channel initiator.

See also “[Flex Channels Library](#)” on page 16-1.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

initiator_name

Identifies the name of the initiator in the design.

E.1.47 **float_array_accesses**

Syntax

```
float_array_accesses [-h] array_id start_node_id [end_node_id]
```

Preconditions

The design has completed the Set up Design Step – the **build** command (“[build](#)” on page E-30) has been run, and the **create_required_states** command (“[create_required_states](#)” on page E-46) has *not* been run.

You must also meet the requirements in “[Requirements for Floating Array Accesses](#)” on page 8-94.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step, up to the Manage States Step.

Action

Allows you to *float* all array accesses to a specified array (located in the set of edges E) to be scheduled at any edge of E, even if the array is accessed by multiple processes. The set of edges E is defined as a set of edges *forward reachable* from a specified starting node and *backward reachable* from an optionally specified ending node. The set of edges E does not include the edges of the functions called from the region. Array dependencies will be respected; CtoS will not reorder a write and a read or write unless it can determine the addresses are always unique.

See also “[Floating Array Accesses](#)” on page 8-94.

Command Arguments

[-h]

Provides a brief description of the syntax and arguments.

array_id

Specifies a particular array (which could be the result of the **merge_arrays** command).

start_node_id

Specifies the start node of the CDFG region (set of edges E).

[end_node_id]

Specifies the end node of the CDFG region (set of edges E).

E.1.48 float_io_accesses

The *float_io_accesses* command is a preliminary feature.

Syntax

```
float_io_accesses [-h]
                  [-name name][-volatile][-nets {net_ids}] start_node_id end_node_id
                  [-name name][-volatile][-nets {net_ids}] loop_join_id
                  [-name name][-volatile][-nets {net_ids}] origin_node_id
```

Preconditions

The design has completed the Set up Design Step – the **build** command (“[build](#)” on page E-30) has been run, and the **create_required_states** command (“[create_required_states](#)” on page E-46) has *not* been run.

The *floating region* must meet the requirements in “[Requirements for Floating I/O Regions](#)” on page 8-92.

Postconditions

After an op has been marked for floating, you cannot constrain that op to a specific edge using the **constrain_op** command (“[constrain_op](#)” on page E-38).

If this command is successful, floating constraint object attributes will have been set for this object [see “[Floating Constraint Object Attributes \(float_constraints\)](#)” on page D-48].

Action

Allows you to *float* all I/Os, or specified I/Os, in a region; that is, to specify the region of *stability* or *don't care* for this set of I/Os.

The I/O accesses include reads and writes from/to **sc_ins**, **sc_outs**, and **sc_signals**. See also “[Floating I/O Accesses](#)” on page 8-91.

By default, a floating constraint specifies input nets that are stable (i.e. should not change value) in the region, and the specified output nets can be safely ignored until the region is exited. An assertion checker is added to the generated simulation models to ensure the inputs do not change in this region.

Floating constraints can also be applied if the design doesn't care if the value of an input changes for many cycles. The **volatile** option gives the flexibility to float I/O accesses in this region and thus gain QOR improvement via register reduction. When the **volatile** option is specified, the assertion-checker task is *not* added to the simulation model, because it is valid for the value of this input to change in this region.

Important Specifying **volatile** option may lead to simulation mismatches if the testbench expects certain values in that region. This may result in individual bits being set at different cycles in that region.

Note The **volatile** option is not supported for region that starts with a fork node.

Command Arguments

[*-h*]

Provides a brief description of the syntax and arguments.

[*-name name*]

Specifies the name of the new floating constraint. The default is **FLOAT_REGION_***identifier*.

[*-volatile*]

Specifies inputs are volatile in restricted region.

[*-nets {net_ids}*]

Specifies one or a list of nets that can be floated in the region. If not specified, all nets can be floated.

start_node_id end_node_id

Specifies the start and end nodes of the region.

loop_join_id

Identifies a loop join node (the *head* of the loop). In this case, read values must be constant for the entire loop (all iterations of the loop), and each net write will be freely moved within the basic block where it belongs (see “[loop_join_id](#)” on page N-8).

origin_node_id

Specifies the *origin node* of the behavior. In this case, read values must be constant for the entire *behavior*; and specifying a floating write will result in an error.

E.1.49 get_attr

Syntax

```
get_attr [-h] attribute_name [object_id]
```

Precondition

The design has completed the Start Step – the **new_design** command has been run.

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Returns the value of a *single* attribute for a *single* object.

Return Value

The value of the attribute

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

attribute_name

Specifies the name of the attribute whose value you want to retrieve.

[object_id]

Identifies the object for which the value of the specified attribute is to be retrieved. A wildcard **object_id** can be used, with the restriction that the wildcard character (*) is the last character and is not used to match an object type. When a wildcard is specified, **get_attr** will return the specified attribute for each of the strings matched by the wildcard.

Example

For a design, **Mydesign**, to get the value of the built-in attribute **compile_flags**, you would use:

```
get_attr compile_flags /designs/Mydesign
```

To store this in a Tcl variable named **\$cf**, you would use:

```
set cf [get_attr compile_flags /designs/Mydesign]
```

Note You must type in the square brackets, as this is a Tcl command, and the square brackets indicate that the command is to return the value of the command in the brackets.

E.1.50 get_design

Syntax

`get_design [-h]`

Precondition

The design has completed the Start Step – the **new_design** command has been run.

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Returns the object ID of the current design.

Return Value

The object ID of the current design

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

E.1.51 get_install_path

Syntax

`get_install_path [-h]`

Precondition

None.

Postcondition

No change.

Action

Returns the filepath to the installation directory of the running CtoS executable.

Return Value

The filepath of the current design

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

E.1.52 get_version

Syntax

`get_version [-h]`

Precondition

None

Postcondition

No change

Action

Returns a list of version numbers in the following order: major release, minor release, supplemental release.

Return Value

A list of version numbers

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

E.1.53 help

Note In addition to the **help** command, you can add the **-h** option to any CtoS command to print out help information for that command.

Syntax

```
help [-h] [[-doc] [command_name]] | [error_message_id] [-quick_start]
```

Precondition

None

Postcondition

No change

Action

Provides brief help for CtoS commands, gives more detail about error messages, or (if no arguments are specified) lists all available CtoS commands.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[[-doc] [command_name]]

Brings up the entire *CtoS User Guide*. If you specify a particular command (for example, **help -doc set_attr**), the page for that command is displayed. If you do not specify a command (**help -doc**), the beginning of the *CtoS Commands* chapter is displayed.

If you simply specify a command name (without the **-doc** option), you will get a brief help listing for that command, and the *CtoS User Guide* is not displayed.

[error_message_id]

Provides more detail about the specified error message.

[-quick_start]

Brings up the *CtoS Quick Start Guide*.

E.1.54 inline

Syntax

```
inline [-h] object_id ...
```

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Depending on the type of object(s):

- If the argument is a *function*, all calls to that function are inlined.
- If the argument is an *operation*, the operation must be for calling a function, and in that case the function is inlined only for that operation.

Important The **inline** command accepts multiple arguments and will inline all of the specified functions or ops. It is faster to inline multiple functions or ops with one **inline** command than to use multiple **inline** commands.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

object_id

Identifies the operation(s) or function(s) to be inlined.

E.1.55 inline_calls

Syntax

inline_calls [-h] [-all] [object_id]

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Inlines all calls to sequential functions *inside* the specified behavior, module, or design, as well as all combinational functions that must be inlined before scheduling (the combinational functions that are *not* synthesizable). However, it does not inline functions marked **pragma ctos dont_touch**.

If you use the **-all** option, *all* function calls, including *all* combinational function calls, are inlined. Again, however, even with the **-all** option, functions marked **pragma ctos dont_touch** are *not* inlined.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-all]

Specifies that *all* function calls, including *all* combinational function calls, be inlined. However, functions marked **pragma ctos dont_touch** are *not* inlined.

[object_id]

Identifies a behavior, module or design. The default is the current design.

E.1.56 launch_sim

Syntax

```
launch_sim [-h] [-target name] [-user_args args] [design_id]
```

Preconditions

The design has completed the Set up Design Step – the **build** command has been run.

The **makefile** specified in the simulation configuration object exists. You may use the **write_sim_makefile** command ([“write_sim_makefile” on page E-168](#)) to generate a **makefile** or write a custom one.

The files required by the target exist. CtoS automatically writes some models for you if you have enabled the **auto_write_models** design attribute. You may manually generate the models with the **write_sim**, **write_rtl**, **write_wrapper**, **write_tlm_wrapper**, and **write_top_wrapper** commands.

Postcondition

None

Action

Launches simulation by calling the **makefile** specified by the **define_sim_config** command.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-target name]

Specifies the name of the target in the **makefile**.

[-user_args args]

Specifies additional arguments for the simulator.

[design_id]

Specifies the design to which this command applies. The default is the current design.

E.1.57 **lcd**

Syntax

`lcd [-h] [dir_path]`

Precondition

None

Postcondition

No change

Action

Sets the current working directory for your Linux file system (similar to the Linux command **cd**).

Return Value

The current directory

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

dir_path

Identifies the directory to set for the current working directory.

E.1.58 **list_attr**

Syntax

```
list_attr [-h] [object_id]
```

Precondition

The design containing the object has completed the Start Step – the **new_design** command has been run.

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Returns the list of attribute names of the specified object.

Return Value

A list of attribute names

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[object_id]

Identifies the object for which to retrieve the attribute names. The default is the current scope.

E.1.59 **lls**

Syntax

```
lls [-h] [option] ... [file] ...
```

Precondition

None

Postcondition

No change

Action

Lists information about the file(s) specified or the file(s) in the current directory (the default), from within the CtoS environment (similar to the Linux command **ls**).

Return Value

The list of files requested.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[option]

See the CtoS GUI Help for a list of options (they are similar to the Linux command **ls**).

[file]

Identifies the file for which to display information.

E.1.60 lpwd

Syntax

`lpwd [-h]`

Precondition

None

Postcondition

No change

Action

Returns the current working directory of your Linux file system, from within your CtoS environment (similar to the Linux command **pwd**).

Return Value

The current working directory.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

E.1.61 **ls**

Syntax

ls [-h] [object_path]

Precondition

None

Postcondition

No change

Action

Lists the subscopes of the current or specified scope

Return Value

The subscopes of the current or specified scope

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[object_path]

Identifies the scope whose subscopes are to be listed. The default is the current scope.

E.1.62 merge_arrays

Syntax

```
merge_arrays [-h] [-data | -addr] [-min_write_width integer]
             [-name new_array_name] array_ids
```

Precondition

The design containing the arrays has completed the Set up Design Step – the **build** command has been run – and none of the arrays has gone through the Allocate IP Step.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Merges specified arrays into a single array; beneficial as it is smaller than multiple individual arrays.

Note This command cannot be applied to arrays, which are accessed by multiple process.

See also “[Merging Arrays](#)” on page 8-63.

Command Arguments

[-h]

Provides a brief description of the syntax and arguments.

[-data | -addr]

For **-data**, the arrays are mapped to *different bits of the word*. In the resulting array, the number of words is the maximum of the words in the arrays being merged, and the data width is at least the sum of the data widths of the arrays being merged.

For **-addr**, the arrays are mapped to *different addresses* in the resulting array, so accesses to the arrays must share the same read and write port, which may create contention. In the resulting array, the number of words is at least the sum of the words in the arrays being merged, and the data width is the maximum between the *data's* of the arrays being merged.

[-name new_array_name]

Specifies the name of the resulting merged array. The default is the name of the first array specified.

[-min_write_width integer]

Specifies the minimum write width of the resulting array.

array_ids

Specifies the names of the arrays to be merged. They should be read-write, not read-only. At least two must be specified, but there is no limit.

E.1.63 new_design

Syntax

new_design [-h] [design_name]

Precondition

None

Postcondition

You have completed the Start Step and may proceed to the Set up Design Step.

Action

Creates a new design with the specified name, or defaults to a uniquely created name, and sets the current design to this new design.

Return Value

The object ID of the newly created design

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[design_name]

Specifies the name of the new design. If you specify the same name as another design open in this session, you will get an error. If you do not specify a *design_name*, the default is **Design_0** (which is automatically incremented by 1, that is **Design_1**, **Design_2**, etc., each time you create a new design).

Note Design names cannot contain the slash (/) character.

E.1.64 open_design

Syntax

```
open_design [-h] [-not_current] [-rename new_design_name] design_directory
```

Preconditions

A valid design database exists in *design_directory*.

A design with the same name must not exist in the current session (unless you will be using the **-rename** option).

Postcondition

You may proceed to the Set up Design Step.

Action

Opens a design that was saved in the specified directory.

Important When a design is saved, CtoS *does not* save the source code, but *does* save the parse tree from which you could derive the source code. It *does not* save the **.lib** files, but *does* save the *path* to the **.lib** files (however, this is not very useful without the **.lib** files). It *does not* save the RC cache, but *does* save timing data.

Additionally, CtoS *does not* save the Tcl interpreter (and thus *does not* save Tcl variables, which are in the interpreter) – it saves only the CtoS design database.

Return Value

The object ID of the newly opened design

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-not_current]

Opens the design, but does not set it as the current design.

[-rename design_name]

Specifies a new name for the design. The default is the name of the design, as saved.

design_directory

Specifies the directory, relative to the current directory, containing the design database.

E.1.65 **optimize**

Syntax

`optimize [-h] [object_id]`

Precondition

The design has completed the Allocate IP Step, and you have done nothing in the Manage Array Dependencies Step or beyond.

Note This command is automatically run by the **schedule** command, if you have not run it beforehand.

Postcondition

You are now in the Analyze Micro-architecture Step.

Action

Performs a final set of optimizations after all micro-architectural transforms have been applied.

When a *module* is specified, in addition to optimizing the behaviors in the module, inter-behavior optimization, such as shared memory or variable trimming, is also performed.

When a *behavior* is specified, the final optimization is applied on the module to which the behavior belongs; consequently, all behaviors in the module are optimized.

CtoS provides a number of other transforms (such as the **unroll_loop** command) that you must apply on a case-by-case basis, as they can improve or degrade the performance of a design, depending on many factors.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[object_id]`

Identifies a design, module, or behavior to be optimized. The default is the current design.

E.1.66 pipeline_function

Syntax

```
pipeline_function [-h] [-latency {min [max]}] [-extra_timing_effort]  
behavior_id
```

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

The function meets the requirements listed in [“Requirements for Pipelining a Function” on page 8-12](#).

Postcondition

You may continue with the Specifying Micro-architecture Step or proceed to the Allocate IP Step.

Action

Sets up the specified function for pipeline scheduling. The number of pipeline stages will be the minimum number of initiation intervals needed to stay within the latency interval (see also [“Pipelining Functions” on page 8-12](#)).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-latency {min [max]}]

Specifies the minimum and maximum number of states to be used for the latency interval for the pipeline, which determines the number of stages in the pipeline. The default minimum latency is **1**, and the default maximum latency is **100**. The **min** value must be less than the **max** value.

Note If the **min** is specified without specifying **max**, then the pipeline function is implemented with fixed latency (that is, **min** = **max**).

[-extra_timing_effort]

Enables you to increase run-time effort to find the minimal latency within a specified range that closes timing. Default is **false**.

behavior_id

Specifies the combinational function to be pipelined.

E.1.67 pipeline_loop

Syntax

```
pipeline_loop [-h] [-init_interval num_states] [-extra_timing_effort]
               [-min_lat_interval num_states] [-max_lat_interval num_states]
               [-allow_io_reordering] [-expand_before {net_ids}]
               [-expand_after {net_ids}] [-expand_at {node_id}] loop_join_id
```

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

The loop meets the requirements in [“Requirements for Loops to be Pipelined” on page 8-29](#).

Postcondition

You may continue with the Specifying Micro-architecture Step or proceed to the Allocate IP Step.

The following node object attributes may be set, depending on your option choices: **init_interval**, **min_lat_interval**, and **max_lat_interval** (see [“Node Object Attributes \(Nodes\)” on page D-50](#)).

Action

Sets up the specified loop for pipeline scheduling. The number of pipeline stages will be the minimum number of initiation intervals needed to equal or exceed the latency interval (see also [“Pipelining Loops” on page 8-26](#)).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-init_interval num_states]

Specifies the number of states used as an initiation interval for the pipeline, which determines the throughput. The default is **1**.

[-extra_timing_effort]

Enables you to increase run-time effort to find the minimal latency within a specified range that closes timing. Default is **false**.

Note For the following two options (**-min_lat_interval** and **-max_lat_interval**), the legal range is:
 $\max(\text{init\_interval} + 1, \text{original loop latency}) \leq \text{min\_lat\_interval} \leq \text{max\_lat\_interval}$

`[-min_lat_interval num_states]`

Specifies the minimum number of states to be used for the latency interval for the pipeline, which determines the number of stages in the pipeline. The default is `(-init_interval + 1)` or the original loop latency, whichever is greater.

The **-min_lat_interval** must be greater than the largest latency of a sequential op that is scheduled in the pipeline. Since the largest op latency currently allowed in CtoS is **2**, the only time this can cause a problem is when the **-min_lat_interval** is set (or defaults) to **2**, and there is an op with latency **2** that might be scheduled in the pipeline. This can result in either of the following:

- The sequential op gets a null op span, or
- The latency of the pipeline is increased at scheduler initialization time to accommodate the sequential op. CtoS will issue an information message if this happens.

`[-max_lat_interval num_states]`

Specifies the maximum number of states to be used for the latency interval for the pipeline, which determines the number of stages in the pipeline. The default is `(-min_lat_interval + 1)` or **100**, whichever is greater.

`[-allow_io_reordering]`

Requests that CtoS ignore the check that all read and write operations to a given signal fit within an initiation interval. See also [“Allowing I/O Reordering during Pipelining” on page 8-59](#).

Note For the following three options (**-expand_at**, **-expand_before**, and **-expand_after**), the following rules apply:

- If the **-expand_at** option is specified, the expansion point is set between the last DFG op before the label and the first DFG op after the label.
- If either the **-expand_before** option or the **-expand_after** option is specified, the insertion point will be before/after the first/last read or write operation to any of the nets. And, if there are no read or write operations to the net in the loop, you will get an error.
- If more than one of the three options are specified, you will get an error.
- If neither of the three options are specified, the following mechanisms are used:
 - If one or more labels starting with the text "HLS_EXPAND_HERE" or "HLS_DEFAULT_EXPAND_HERE" is found in the loop body, then the last one (that is, the one closest to the loop end) is selected, as if it was specified with the **-expand_at** option. If the label is in a protocol region, the initial label of the protocol region is selected. The label must not be in a stall loop or a conditional.
 - Otherwise, the end of the loop is selected for expansion

[*-expand_before* {*net_ids*}]

Specifies a list of net names. The first access to any of these nets is the expansion point at which CtoS will insert additional states in the loop to increase latency. If the access is inside the stall loop, the insertion point will be before the stall loop.

[*-expand_after* {*net_ids*}]

Specifies a list of net names. The last access to any of these nets is the expansion point at which CtoS will insert additional states in the loop to increase latency. If the access is inside the stall loop, the insertion point will be after the stall loop.

[*-expand_at* {*node_id*}]

Specifies the label node at which CtoS will insert an additional state. The expansion point is set between the last DFG op before the label and the first DFG op after the label. The label node must not be inside a stall loop or inside a conditional statement.

loop_join_id

Identifies a loop join node (that is, the *head* of the loop). See also “[loop_join_id](#)” on page N-8.

E.1.68 **print_message**

Syntax

`print_message [-h] -owner name msg_id [args ...]`

Precondition

The message must exist.

Postcondition

No change

Action

Prints the specified message to the output window, replacing format parameters with specified arguments.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`-owner name`

Specifies the scope of the message id.

`msg_id`

Specifies the identifier of the message.

`[args ...]`

Provides the formatting name/value pairs.

E.1.69 pwd

Syntax

pwd [-h]

Precondition

None

Postcondition

No change

Action

Shows the current scope.

Known Limitation

For objects with multiple path-RegBinding and ResBinding?, this command returns the native object path.

Return Value

The current scope

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

E.1.70 read_ip_defs

Syntax

```
read_ip_defs [-h] -ip_def filename [-overwrite] [design_id]
```

Precondition

The design has completed the Set up Design Step – the **build** command has been run – and you have done nothing in the Scheduling Step.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Reads in an IP definition XML file and creates IP definition objects.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`-ip_def filename`

Identifies the filename of the XML file. This could be for a **memory_def** or an **rtl_ip_def**.

`[-overwrite]`

Allows existing IP definitions to be replaced.

`[design_id]`

Identifies a design to read IP definitions. The default is the current design.

E.1.71 **read_tcf**

The read_tcf command is part of Power Estimation, a preliminary feature.

Syntax

```
read_tcf [-h] [-accumulate] [-weight double] [-scale double] file_name module_id
```

Preconditions

For pre-scheduling analysis, you are in the Analyze Micro-architecture Step.

For RTL analysis, you are in the Analyze and Implement Step.

For either pre-scheduling or RTL analysis, the INCISIV simulator has been run to generate the TCF (toggle count format) file.

Postcondition

You can now use the **analyze** command with the **-power** option.

Action

Reads in switching activity described in TCF (toggle count format). TCF is the Cadence standard format to describe switching activity information in a design.

Transition and level (also called toggling and value) probabilities are changed for all the nets or values specified in the TCF file.

Command Arguments

[*-h*]

Provides a brief description of the command syntax and arguments.

[*-accumulate*]

Accumulates probabilities, as opposed to overwriting them, if multiple probabilities are specified for a given signal in the TCF (toggle count format) file(s). If this option is specified, probabilities are accumulated with a weight incremented every time a new probability is read for a net (past and current values are weighed evenly).

[*-weight double*]

Uses the specified weight for the probabilities, which affects both toggling and level probabilities. This is useful to specify the different probabilities of different scenarios specified by different TCF files, and it is only useful in connection with the **-accumulate** option. The total sum of the weight values provided to *n* files must be *n*. The default value is 1.0.

[*-scale double*]

Multiplies the specified scaling factor, which must be a positive floating point number, by the clock frequency of the module without re-running the simulation. This only affects toggling probabilities. The default value is 1.0.

file_name

Specifies the name of the TCF file, which must be an existing file.

module_id

Specifies the name of the module for which to annotate switching activity. The specified module identifier must be the same as the top instance in the TCF (as specified by the scope in the **dumpptcf** command of **ncsim**), since no warning can be issued when an entry in a TCF file does not match the current design. However, this problem can be discovered by noticing a low percentage of asserted nets in the output message. This message can also be used to identify both a low coverage of the design by the testbench or the **dumpptcf** probe, if only a few nets have non-zero or non-default values.

E.1.72 remove_clock

Syntax

```
remove_clock [-h] clock_name
```

Precondition

The design has completed the Start Step – the **new_design** command has been run – and you have not run the **build** command.

This “pre-build” precondition is a temporary restriction, which will be removed in a future release.

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Removes a clock definition from the current design.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

clock_name

Identifies the clock to be removed.

E.1.73 remove_directive

Syntax

```
remove_directive directive_id ...
```

Note *This command is a preliminary feature.*

Action

Removes one or more directives from the database and performs a non-final optimization at the end of the command. This command enables users to ignore some of the directives.

Command Arguments

```
directive_id ...
```

Specifies the directive(s) to be removed.

E.1.74 rename_state

Syntax

```
rename_state [-h] -behavior behavior_id -annotation_paths {path ...} name
```

Precondition

The design has completed the Manage Registers Step - the **allocate_registers** command has been run.

Postcondition

You may rename more states; when done, you may write out your RTL, using the **write_rtl** command ([“write_rtl” on page E-161](#)).

Action

Renames the state that matches all specified annotation paths in the specified behavior. See also [“Renaming States in Support of FSM Coverage” on page 7-25](#).

Warning CtoS issues an error if multiple states have the specified annotation path. The user should add labels on state or pragmas on function calls to avoid this conflict.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

behavior behavior_id

Identifies the behavior in which annotation paths are searched.

-annotation_paths

Specifies all annotation path(s) of the state to be renamed.

name

Specifies the new name for the state.

E.1.75 report_area

Syntax

```
report_area [-h] [-detail] [object_id]
```

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Technology libraries have been specified.

Note When you run this command early in the design flow, the results will be only as accurate as the state of the design. During the Scheduling Step, CtoS will optimize logic across resource boundaries and use smaller, but slower implementations where possible. Therefore, you will get better results if you run this command after the Manage Registers Step, and after you have run the **analyze** command with the **-area** option.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Generates a report of the area of all of the resources associated with a behavior. At the module level, it reports the area for each behavior, shared array or shared variable, or user instance. At the design level, it reports the area of the top-level module.

The units are the units in the Encounter RTL Compiler (RC) technology library. This is typically gates or square microns/microns squared.

Warning This is an overly conservative number; to get a more accurate report, use the area reports from your logic synthesis tool.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-detail]

Displays area details.

[object_id]

Identifies a design, module, or behavior for which to report the area. The default is the current design.

Example

Here is a sample report from the **report_area** command:

```

Area for the behavior ctos_system1_fourier_inst_main of module ctos_system1
  Total Area   Instances   Master Name
  -----
    31.0464         11   if_then_else
    76.2048          1   mux_2_4
   114.307          2   mux_2_3
   190.512          2   mux_2_5
   211.68          12   unary_or_2
   248.371         22   unary_and_2
    529.2           2   addsub_4
   609.638          1   mux_2_32
  1181.17          2   mux_2_31
  1909.35          3   gtle_32
  2353.88          1   addsub_32
  2371.52          1   addsub_31
   5783.8           1   smul_10_22_22
 35881.2           2   ram_16x32_1ar_1w
 51491.9           Total Area for the behavior
                        ctos_system1_fourier_inst_main of module ctos_system1

```

E.1.76 **report_array_dependencies**

Syntax

`report_array_dependencies [-h] [object_id]`

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Reports the array dependencies in the context of the specified design, module, behavior or array. These dependencies are reported as pairs of memory ops or memory ops and sequential function calls.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[object_id]`

Identifies a design, module, behavior, or array for which to report array dependencies. The default is the current design.

E.1.77 **report_behavioral_power**

Syntax

`report_behavioral_power [-h]`

Precondition

The design has completed the Allocate IP Step. You must run this command before the **create_initial_resources** and the **schedule** command.

Postcondition

You can run the **create_initial_resources** or any subsequent step.

Action

Reports the behavioral power estimation.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

E.1.78 **report_incremental**

Syntax

```
report_incremental [-h]  
                  [-kind all | states | resources | op_bindings | registers | value_bindings]
```

Precondition

The design has completed the Set up Design Step – the **build** command has been run – and you have run the **set_baseline** command.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Reports on how much information was copied successfully from the baseline design. If you use the **-kind** option, it prints out in Tcl form (which can be redirected to a file if needed):

- as a Tcl *command* those resources, states, registers and bindings that could be copied to the new design, and
- as a Tcl *comment* a detailed reason for those that could not be copied.

Known Limitation

The information that is printed by the **report_incremental** command (both the detailed and the non-detailed form) is not saved when you save a design. Therefore, this command can only be executed in the same CtoS session in which the **schedule** and **allocate_registers** commands (which produce that information) are executed.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-kind all | states | resources | op_bindings | registers | value_bindings]

Specifies the kind of information on which to report:

- the number of resources, registers, states, op bindings and/or value bindings that have been copied from the baseline to the new design and
- the number of those that could not be copied at all because the new design is too different.

E.1.79 **report_latency**

Syntax

```
report_latency [-h] [-birthday] object_id_1 object_id_2
```

Preconditions

With the **-birthday** option, the behavior containing the objects has completed the Set up Design Step – the **build** command has been run.

Without the **-birthday** option, the behavior has completed the Scheduling Step – the **schedule** command has been run.

With or without the **-birthday** option, both objects must be in the same behavior, must be part of a scheduled thread process or a sequential function, and must not contain any combinational loops.

Postcondition

With the **-birthday** option, you may continue with the Specify Micro-architecture Step.

Without the **-birthday** option, you may continue with the Analyze and Implement Step.

Action

Reports the latency of the paths between ops in the data flow or between nodes in the control flow, giving the minimum and maximum number of states (ignoring loops) found on paths from start to end.

If non-inlined function calls are found along the paths traversed, the states encountered in those functions will be counted in computing minimum and maximum latency, but the sample path will list only the states in the behavior of start and end.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-birthday]

Indicates that the original source code binding of each op should be used, instead of a scheduled binding.

object_id_1

Specifies a node or an op where the paths start, which must be in the same behavior as *object_id_2*.

object_id_2

Specifies a node or an op where the paths end, which must be in the same behavior as *object_id_1*.

E.1.80 report_opspan

Syntax

```
report_opspan [-h] [-detail] op_id
```

Precondition

The design has completed the Manage Array Dependencies Step, and you have not successfully scheduled the design.

Postcondition

You can continue in the Guide Scheduler step, or proceed to scheduling, if you issued this command before the **schedule** command ([“schedule” on page E-138](#)).

Action

Reports diagnostics for an op in the following three tables:

- **Start Span Boundary Table:** contains up to two critical ops that cause the span to *start* on a particular edge. The edge is the *earliest* edge of the **op_id**, or the edge on which the op span restriction was applied, for example:

```
Critical Reasons Why Span For <op1> Starts At <e1>
```

Edge	Reason	Object
L1_do_begin	array write limited by fork or join	

This shows the earliest edge of **op1** is limited by the fact that **op1** is a array write op and cannot be scheduled outside the loop that starts at **L1_do_begin**.

See also [“Reasons that limit op span” on page E-110](#).

- **End Span Boundary Table:** contains up to two critical ops that cause the span to *end* on a particular edge. The edge is the *latest* edge of the **op_id**, or the edge on which the op span restriction was applied, for example:

```
Critical Reasons Why Span For <op1> Ends At <e4>
```

Edge	Reason	Object
e8	io fixed on edge	write_ln93

This shows the latest edge of **op1** is limited by **write_ln93**, which has latest edge **E8** due to the write being fixed on edge.

See also [“Reasons that limit op span” on page E-110](#).

Table E-2 Reasons that limit op span

reason	description
io fixed on edge	I/Os are fixed on edge. You can override this with the float_array_accesses command (" float_array_accesses " on page E-70).
mux or ctrl fixed on edge	Control ops and join mux ops are fixed on edge.
float io access	I/Os are allowed to float to region boundary defined by the float_array_accesses command (" float_array_accesses " on page E-70).
shared array fixed on edge	Shared array accesses are fixed on edge. You can override this with the float_array_accesses command (" float_array_accesses " on page E-70).
float array access	Array op is allowed to float to region boundary defined by the float_array_accesses command (" float_array_accesses " on page E-70).
hard op constraint	Op is fixed on edge due to user-specified constrain_op command (" constrain_op " on page E-38).
array write limited by fork or join	Array write is limited to the basic block relative to the birth edge to avoid simulation mismatches.
no predecessor	Op is limited to the basic block relative to the birth edge. Run the check_design command (" check_design " on page E-32) for warnings.
no successor	Op is limited to the basic block relative to the birth edge due to speculation.

- **Op Span Table:** in which each row represents an edge in the op span, sorting topologically.

The edge status can be one of the following:

- **valid**
- **not valid**
- **assigned** (which indicates that the CtoS scheduler has selected this edge)

The schedule status of the op being analyzed can be one of the following:

- **Not Yet Scheduled:** applies to a behavior before scheduling, as well as a behavior that failed scheduling, and this particular op has not been scheduled.
- **Scheduled:** has one entry with status *assigned* to indicate the scheduled edge.
- **Failed Schedule:** has all edges with status *not valid*.

Before scheduling, here is an **Op Span Table** for **op1**:

Op Span for op1 (Not Yet Scheduled)

Edge	Status	Reason	Object
edge1	valid		
edge2	valid		
edge3	not valid	sequential dependency	op2
edge4	not valid	violates timing	op3
edge5	not valid	on fork stem	node4

These results tell you that:

- **edge3** was removed by the sequential refinement triggered by **op2**, which must be a sequential predecessor of **op1**.
- **edge4** was removed by the timing refinement so **op3** must be a timed predecessor of **op_id**.
- **edge5** was removed by the fork optimization, and **node4** must be a fork node.

After a failed schedule, but with **op1** being successfully scheduled, it would look like this:

Op Span for op1 (Failed Schedule)

Edge	Status	Reason	Object
edge1	assigned		
edge2	valid		
edge3	not valid	sequential dependency	op2
edge4	not valid	fails timing	op3
edge5	not valid	on fork stem	node4

See also [“Reasons for refinement of an op” on page E-112](#).

Table E-3 Reasons for refinement of an op

refinement reason	description	agent	agent description
sequential predecessor	op not allowed on edge due to sequential predecessor op	op	offending predecessor op
sequential successor	sequential op not allowed on edge due to successor op	op	offending successor op
violates predecessor timing	op not allowed on edge that causes negative slack due to predecessor op	op	timed predecessor
violates successor timing	op not allowed on edge that causes negative slack due to successor op	op	timed successor
resource contention	op not allowed on edge that results in resource contention	op	competing op
inter-iteration memory contention	memory op not allowed on edge that results in memory contention	op	competing op
on fork stem	op not allowed on fork stem to promote sharing	node	fork node
SCC out of loop	op belongs to an SCC that cannot be moved after the loop fork	node	loop fork node
not on pipeline last edge	op cannot be scheduled on the last edge of a pipeline	node	loop join node
not in first stage	op must be scheduled in first stage if used in computation of loop exit condition	op	exit op
op not with SCC	all ops belonging to an SCC that must be scheduled in the same stage	node	loop join node
in preserve region	op not allowed in preserved region	edge	preserved edge
out of preserved region	op not allowed out of preserved region	edge	preserved edge
not same as soft constraint	user-specified soft constraint	edge	user-specified edge

Table E-3 Reasons for refinement of an op

refinement reason	description	agent	agent description
in call busy loop	op not allowed on edge in call-busy loop	op	function call
in reset path	op not allowed on edge in reset path		
not loop invariant	array op cannot moved before loop due to loop array dependency	op	other op in same loop
near fork join	sequential op not allowed on edge adjacent to fork or join node	node	adjacent non-simple node
not in stall loop stage	op must remain in same stage as stall loop	node	stall loop join
not out of pipeline	sequential op with no stall pin is not allowed in pipeline with stall loop	node	stall loop join
array op in nested loop	array op not allowed in nested loop to reduce number of accesses	node	loop join
SCC violates timing	op belongs to an SCC which would cause negative slack on this edge		
memory cluster	op belongs to memory cluster that must be scheduled in the same stage	node	loop join

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-detail]

If you specify this option, the report will contain a longer **description** (in [Table E-3 on page E-112](#)).

If not specified, it will contain only the shorter **refinement reason**.

op_id

Specifies an op for which to report op span.

E.1.81 **report_paths**

Syntax

```
report_paths [-h] [-detail] [module_id]
```

Precondition

The design has completed the Manage Registers Step – the **allocate_registers** command has been run.

Postcondition

You may continue with the Analyze and Implement Step.

Action

Reports the number of paths between inputs, outputs, and registers with different clock or reset signals.

This command is useful for reviewing the design and identifying problems where signals in one clock or reset domain are unexpectedly used in another clock or reset domain.

This command reports the following:

- If applicable, the clock edge (**r** for rising or **f** for falling), reset assertion level (**hi** if a **1** resets the register or **lo** if a **0** resets the register), and the reset synchronicity (**sync** for synchronous reset or **async** for asynchronous reset).
- Information about whether there is logic in the path between the two registers (paths with logic between two asynchronous clocks should be carefully examined).

It does *not* report the following:

- Paths between two registers with the same clock edge and the same reset signal, level, and synchronicity.
- Paths between a register with reset and a register without reset if the two registers use the same clock and clock edge.

If **report_paths** is run before **write_rtl** is run on the specified module, internal names are used for the registers in the detailed report. After the **write_rtl** command has been run, the register names in the RTL are used.

The paths terminating at a port section of the detailed report will list the same name as the source and destination for registered outputs: the first name refers to the register, and the second refers to the output.

The paths reported are pre-synthesis and do not account for paths that may be removed by optimization during logic synthesis, or paths that might be introduced by logic sharing. Furthermore, if re-timing is selected during logic synthesis, the number of paths will probably change.

Command Arguments

[`-h`]

Provides a brief description of the command syntax and arguments.

[`-detail`]

Indicates that every path is to be enumerated showing specific register names (thus, output can be very long when using this option).

[`module_id`]

Specifies a module for which to report paths. The default is that all paths in the top module are reported.

Example

Here is a sample report from the **report_paths** command:

Type	Source	Dest.	Logic?	#Paths
----	-----	-----	-----	-----
Port to Reg				
	IN1a	r CLK1/sync lo RST_N	yes	32
	IN1b	r CLK1	no	32
	IN2a	r CLK1/sync lo RST_N	yes	32
	IN2b	r CLK1	no	32
	ENn_IN1	r CLK1/sync lo RST_N	yes	4
	ENn_IN1	r CLK1	yes	2
Port to Port				
	IN1a	UNION	yes	32
	IN2a	UNION	yes	32
Reg to Reg				
	r CLK1/sync lo RST_N	r CLK3/sync lo RST_N	yes	64
Reg to Port				
	r CLK1	RDYn_IN1	no	1
	r CLK1	RDYn_IN2	no	1
	r CLK3/sync lo RST_N	OUT	no	32

E.1.82 report_power

The report_power command is part of Power Estimation, a preliminary feature.

Syntax

```
report_power [-h] [-detail] [object_id]
```

Preconditions

You are in the Analyze and Implement Step.

- The INCISIV simulator has been executed to generate the TCF (toggle count format) file.
- Technology libraries have been specified.

Postcondition

You may continue with the Analyze and Implement Step.

Action

Reports the estimated power consumption of all resources in the specified design, module, or behavior – automatically calling power analysis (the **analyze** command with the **-power** option), if needed.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-detail]`

Displays power details.

`[object_id]`

Identifies the design, module, or behavior for which to report power. The default is the top-level module.

E.1.83 report_registers

Syntax

```
report_registers [-h] [-detail] [object_id]
```

Precondition

The design has completed the Scheduling Step – the **schedule** command has been run.

Postcondition

You may continue with the Manage Registers Step or any subsequent step.

Action

Displays the register bits used per state for a design, module, or behavior, as well as how many registers belong specifically to a pipeline stage (that is, whose value is computed in the pipeline itself).

- If **report_registers** is called on a module when you are using the CtoS GUI, and you are using the Register Viewer, registers for each behavior in that module are reported in a separate tab. This is also true if no *object_id* is provided since the default is the top-level module; in that case, it will report the registers for each behavior in the top-level module in a separate tab.
- If **report_registers** is called on a module when you are using the CtoS command line, multiple listings are also reported.

Command Arguments

[-h]

Provides a brief description of the syntax and arguments.

[-detail]

Displays verbose usage details, including both primary and a nested list of *secondary register binding*. The report includes a **Kind** heading indicating **primary**, **secondary inverted**, or **secondary non-inverted**. Four register binding attributes support this feature: **is_primary**, **is_inverted**, **primary_binding**, and **secondary_bindings**.

[object_id]

Identifies a design, module or behavior for which to report. The default is the current design.

Examples

Here are two examples of this report, with and without the **-detail** option.

- Without the **-detail** option, below the line for each state that is part of a pipeline, one line per pipeline stage is added, reporting how many registers belong to that stage. A * denotes registers computed outside the pipeline, but used inside the pipeline. For example:

Number of registers for behavior main

State	Stage	#Registers
-----	-----	-----
xformState_ln38		1
state_ln44		61
state_ln50		61
state_ln74		315
	1	103
	2	91
	3	79
	4	30
	*	2
state_ln75		295
	1	20
state_ln127		1

- With the **-detail** option, the stage to which each register bit belongs is reported after the state (that is, the phase), for example:

Register bindings for behavior main

Value	State	Stage	Tag	Register
-----	-----	-----	---	-----
...				
read_in_j_ln42_Z_7	state_ln74	*	-	read_in_j_ln42_Z_0[7]
read_in_j_ln42_Z_31	state_ln74	*	-	read_in_j_ln42_Z_0[8]
muxD_i_whileT_ln62_Z_0	state_ln74	1	-	read_in_i_ln41_Z_0[0]
muxD_i_whileT_ln62_Z_1	state_ln74	1	-	read_in_i_ln41_Z_0[1]

A register belongs to a stage of the pipeline if the value it memorizes is computed in the pipeline (this excludes registers whose value is computed before the pipelined loop and whose stage is shown as a * in the report).

E.1.84 report_resources

Syntax

```
report_resources [-h] [-detail] [object_id] [all]
```

Preconditions

Minimally, the design has completed the Set up Design Step – the **build** command has been run.

However, to get maximum delay and area information, the design has completed the Scheduling Step – the **schedule** command has been run – and you must use the **-detail** option.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Generates a report of all resources, in a design, module, or behavior, capable of implementing more than one operation (thus each is a *shareable* resource). These resources may be specified with “[constrain_op](#)” on page E-38.

You may get a resource report after using the **create_initial_resources** command (“[create_initial_resources](#)” on page E-42), which analyzes the design and determines the minimum resources (and their size) required to synthesize this design. Later, you can manually add, or use the **schedule** command (“[schedule](#)” on page E-138) to add, resources to resolve timing and causality issues encountered during scheduling.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-detail]

Additionally displays the maximum delay and area, if the design has completed the Scheduling Step. (This option can be used any time, but it will not provide this information if the design has not completed the Scheduling Step.)

[object_id]

Identifies a design, module, or behavior for which to generate a resource report. The default is the current design.

[all]

Additionally, lists the resources with resource bindings as well.

Example

Here is a sample report from the **report_resources** command:

```
Resources allocated to module example, behavior example_thread1:
Count  ModuleType  Master
-----  -
      2   add      add_10
      2   add      add_9

Resources allocated to module example, behavior example_thread2:
Count  ModuleType  Master
-----  -
      1   ram      ram_64x8_1ar_1w
      2  addsub      addsub_14
      1  addsub      addsub_9
```

Here is an explanation of the columns in this report:

- **ModuleType:** This column can be one of the resource types defined in [“Resource Types and Required Widths for Creating Resources” on page 11-22](#), as well as *ram* for memories, or *custom* for resources representing functions (not inlined).
- **Master:** For resources manually created with the **create_resource** command, the **Master** is the name specified as the name of the resource. If you did not specify a name, or if CtoS creates the resource, the name is generated with *moduleType* appended, with sizes for the *significant terminals* of the module (also described in [“Resource Types and Required Widths for Creating Resources” on page 11-22](#)).

The reason that some names have 1, 2, or 3 widths appended has to do with required resource bit widths (the default is 1 width). For some resource types, a single number expresses the width (and is used for the output width and all input widths). Signed and unsigned multiplier resources require three widths: output width, width of the wider input, and width of the narrower input. Again, refer to [“Resource Types and Required Widths for Creating Resources” on page 11-22](#) for supported types and required widths.

For *ram* resources, the **Master** is the name of the memory definition specified with the **allocate_memory** command.

For *custom* resources, the **Master** is the name of the function.

E.1.85 **report_sccs**

Syntax

```
report_sccs [-h] [-detail] [object_id]
```

Precondition

The design has completed the Set up Design Step – the **build** command has been run. However, the content of the report may change due to changes to the design through the flow.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Generates a report of all the SCCs in the context of the behavior, join node, or the op. For a behavior or join node, all the SCCs in its scope are reported.

Depending on the **-detail** option, a summary or detailed report is shown. For an op, the detailed report of the SCC is shown.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-detail]

Displays a list of ops in the SCC. The default is false.

[*object_id*]

Identifies a behavior, loop join node, or op for which to generate a report of SCCs.

When you specify an op as an *object_id* argument, a detailed report is produced, even if you do not specify the **-detail** option.

Example

The SystemC code for the loop **LP_for_begin** in an *sc_module* **simple** is shown below. It is followed by the summary and the detailed SCC report for the loop **LP_for_begin**.

```
01 SC_MODULE(simple) {
02   public:
03     sc_in< bool > clk;
04     ...
22     sc_out< bool > out1_valid;
```

```

...
47  int  w;
48  }
...
73  w = 0;
74  wait();
75  while(1) {
76    LP: for (unsigned int i = 0; i < 8; i++) {
77      wait();
78      k = in1.read();
79      wait();
80      if (w < k) {
81        w = w + k;
82      } else if (w > k) {
83        w = w - k;
84      }
85      out1.write(w);
86      OUT: out1_valid = !out1_valid;
87    }
88  }
89  }

```

The SCC **mux_w_ln76** is formed because the value of the variable **w** is being modified in each iteration of the loop and needs to be stored for the next iteration. The list of ops involved in the modifications of the value of variable are listed in the detailed report.

The SCC **mux_i_ln76** is formed for the same reason, the value of **i** at the end of each iteration is being incremented. These kind of SCCs are common on variables used as a counter for a loop.

The SCC **write_simple_out1_valid_ln88** is formed because the value to be written to the port/signal depends on its value from previous iteration. These kind of SCCs are common on module ports or signals that are used to implement a handshake protocol.

There are other kinds of SCCs formed due to array accesses (not in the example). These are formed because of loop carried dependencies amongst the array accesses.

Summary Report

Without the **-detail** option, the report consists of a table with each row containing information about each SCC under the behavior or join node. If an op is provided as the argument, the detail report for the SCC it belongs to is produced even without the **-detail** option.

The loop LP_for_begin has the following SCCs:

SCC Name	#Ops	Stage	Variable
-----	----	-----	-----
mux_w_ln76	7	2	w

```

mux_i_ln76          3      1  i
write_simple_out1_valid_ln86  3      6  out1_valid

```

In addition to the summary report, you can create a more detailed report, as described in [“Detailed Report”](#).

Detailed Report

With the **-detail** option, each SCC in the scope of join node has information about the SCC followed by a table of the SCC ops, SCC variable and declaration location of the variable. For an op, the same is reported for the SCC it belongs to, if any. The ops table contains all the ops that are involved in modification of the variable through the loop (listed in topological order starting from the join node/read op).

SCC ops for op mux_w_ln76

```

Behavior  simple_main
Loop      LP_for_begin
Stage     2

Ops      Variable  Location
---      -
mux_w_ln76  w      simple.cpp:47:15
lt_ln80
gt_ln82
sub_ln83
add_ln81
mux_w_ln80
case_mux_w_ln80

```

SCC ops for op mux_i_ln76

```

Behavior  simple_main
Loop      LP_for_begin
Stage     1

Ops      Variable  Location
---      -
mux_i_ln76  i      simple.cpp:76:31
add_ln76
if_mux_i_ln76_0

```

SCC ops for op write_simple_out1_valid_ln86

Behavior simple_main
Loop LP_for_begin
Stage 6

Ops	Variable	Location
---	-----	-----
read_simple_out1_valid_ln86	out1_valid	simple.cpp:22:22
if_read_simple_out1_valid_ln86		
write_simple_out1_valid_ln86	out1_valid	simple.cpp:22:22

E.1.86 report_schedule

Syntax

```
report_schedule [-h] [-by_state] [-display ops|res|states] [-op op_id]  
               [-res res_name] [-state state_id] [object_id]
```

Precondition

The design has completed the Scheduling Step – the **schedule** command has been run.

Postcondition

You may continue with the Manage Registers Step or any subsequent step.

Action

Generates a report describing any edge, op, and resource bound as the result of scheduling the specified object.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-by_state]

Sorts the output by state. Note that in this format, many entries will be listed under multiple states.

[-display ops|res|states]

Indicates the type of object to display. If used, this option replaces the multicolumn output format with a single column output showing only the type of object selected, without duplications. The **-display states** option shows only those states, which have at least one op scheduled on any of their combinational out path.

[-op op_id]

Identifies an operation.

[-res res_name]

Identifies a resource instance.

[-state state_id]

Identifies a state node.

[object_id]

Identifies a design, module or behavior for which to generate a report. The default is the current design.

E.1.87 report_slack

Syntax

```
report_slack [-h] [object_id]
```

Preconditions

The design has completed the Manage Registers Step – the **allocate_registers** command has been run. Technology libraries have been specified.

Postcondition

You may continue with the Analyze and Implement Step.

Action

For a *module* or *behavior*, this command reports *slack per state*. The slack for a state is the difference between the clock period and the maximum delay of the paths that are sensitizable at the state.

For a *loop join node*, this command shows the slack for stage and phase pairs in a pipelined loop.

The critical slack reported in any state *can* be more than the worst slack reported for the module, either because the critical path is in another behavior, or because the critical path is a false path involving paths from two different states that share a resource. You can use the **constrain_op** command ([“constrain_op” on page E-38](#)) to remove a false path to prevent the CtoS scheduler from sharing the offending resource.

Command Arguments

[-h]

Provides a brief description of the syntax and arguments.

[object_id]

Identifies the module or behavior for which to report slack per state, or the loop join node of a pipelined loop for which to report the slack for stage and phase pairs in the pipelined loop.

Example

Here is a sample report from the **report_slack** command:

```
Slack per state in behavior example1_main
State          Slack (ps)
-----
Wait_ln69      15319
Wait_ln25      15859
Wait_ln57      17330
Wait_ln97      17370
Wait_ln44      17560
Wait_ln18      17960
```


E.1.88 **report_states**

Syntax

`report_states [-h] [object_id]`

Preconditions

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Reports states and their annotation paths in the specified object. See also [“Renaming States in Support of FSM Coverage” on page 7-25](#).

Command Arguments

`[-h]`

Provides a brief description of the syntax and arguments.

`[object_id]`

Identifies the design, module, or behavior for which states and their annotation paths are reported.

E.1.89 **report_summary**

Syntax

`report_summary [-h] [object_id]`

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or any subsequent step.

Action

Summarizes the behavioral information of the design. If the design has completed the Scheduling Step – the **schedule** command has been run – it also summarizes the structural information.

Slack reporting for **report_summary** and CtoS GUI Summary Report

Please note the following about the reporting of slack for both the **report_summary** command and the CtoS GUI **Summary Report**:

- In either the report generated by the **report_summary** command or the CtoS GUI **Summary Report**, the slack value, after scheduling, is the minimum slack found by the CtoS scheduler.
- The slack value is not shown after the allocation of registers (**allocate_registers**) and before a Timing Report (**report_timing**).
- In either the report generated by the **report_summary** command or the CtoS GUI **Summary Report**, the slack value, after a timing report (using the **report_timing** command), of the top module is the minimum slack of that timing report.
- The CtoS GUI **Summary Report** is automatically updated.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[object_id]`

Identifies a design, module, or behavior for which to generate a summary report. The default is the current design.

Example

Here is a sample report from the **report_summary** command:

Summary report for complete Design /designs/xbus_hw_idct

Name	Count
----	-----
Overview	
Name	xbus_hw_idct
'clk' Clock Period (ps)	28,000
Minimum Slack after Schedule (ps)	not calculated
Minimum Slack (ps)	not calculated
Area	N/A
Power	N/A
Behavior	
Modules	1
Processes	1
Functions	3
Arrays	3
Loops	5
States	8
Edges	37
Total Ops	224
Shareable Ops	150
Values	2,974
Tags	11
Structure	
Memories (bits)	0
Flip Flops (bits)	0
Muxes (bits)	0
Shareable Resources	0
Non Shareable Resources	0
Input Terminals (bits)	69
Output Terminals (bits)	32
Nets (bits)	103
Instance Terminals (bits)	0

E.1.90 report_timing

Syntax

report_timing -h

Structural Timing Report:

```
report_timing
  [-type structure] [-num_paths integer]
  [module_id | behavior_id | state_id]
```

Structural Timing Report with to, from and through points:

```
report_timing
  [-type structure] [-num_paths integer]
  [-to string_id [-from string_id [-through string_id]]]
  [module_id]
```

Pipeline Timing Report:

```
report_timing
  -stage stage_number -phase phase_number [-num_paths integer]
  loop_join_id
```

Behavioral Timing Report:

```
report_timing
  -type behavior [-detail] [-num_paths integer]
  [module_id | state_id]
```

Preschedule Timing Report:

```
report_timing
  -type pre_schedule [-num_paths integer]
  [module_id]
```

Note The *-type pre_schedule* option is part of Power Estimation, a preliminary feature.

Preconditions

With the **-type pre_schedule** option, the design has completed the Allocate IP Step.

With the **-type structure** or **-type behavior** options, the design has completed the Manage Registers Step – the **allocate_registers** command has been run.

With any option, technology libraries have been specified.

Postconditions

With the **-type pre_schedule** option, you may continue with the Analyze Micro-architecture Step or any subsequent step.

With the **-type structure** or **-type behavior** options, you may continue with the Analyze and Implement Step.

Action

Reports the timing of a module, behavior, state, or pipelined loop.

CtoS automatically calls Encounter RTL Compiler (RC) internally during the execution of this command.

CtoS will also automatically call the **analyze** command with the **-timing** option, if necessary.

The report title distinguishes each timing report, by using string and object identifiers. For example:

```
Timing report for the most critical path from string_id1 to string_id3
passing through string_id2 in module object_id.
```

Important An empty name in a report indicates a false path.

Known Limitations and Caveats

The **report_timing** command has the following known limitations and caveats:

- Timing analysis does not work for hierarchical designs; it can be used on leaf modules only.
- The precision of pre-scheduling analysis and reporting (**pre_schedule** type) can be much lower than post-scheduling.
The reason is that sharing decisions can heavily affect timing (due to multiplexers and false paths) and power (due to additional toggling induced by sharing).
- The **pre_schedule** type can be used only for a state, so you must identify a *module_id* in this case.
- The **pre_schedule** type can be used only *before* the **allocate_registers** command.
- The **behavior** and **structure** types can be used only *after* the **allocate_registers** command.

Command Arguments

[-h]

Provides a brief description of the syntax and arguments.

[-type structure | behavior | pre_schedule]

Selects the type of names to be displayed. If type is **structure**, resource names are displayed. If type is **behavior**, op names are displayed. If type is **pre_schedule**, critical sequences of ops (delimited by I/O or other fixed operations) across multiple states are displayed.

`[-detail]`

Includes structural details for the behavior report type.

`[-num_paths integer]`

Indicates the number of critical paths to display. The default is **1**.

`[-to string_id]`

Specifies a list of the ending point(s) of the path, which *must* be in a custom module. These points could be output bit ports, output ports, input bit instance ports, input instance ports, insts, or a list of any of these. If you specify an instance port, it *must* be a sequential input instance port, such as the D port of a flip-flop. If you specify an instance, it *must* have a sequential port.

`[-from string_id]`

Specifies a list of the starting point(s) of the path, which *must* be in a custom module. These points could be input bit ports, output bit instance ports, output instance ports, insts, or a list of any of these. If you specify an instance port, it *must* be a sequential output instance port, such as the Q output of a flip-flop. If you specify an instance, it *must* have a sequential port.

`[-through string_id]`

Specifies a list of the point(s) through which a path must traverse. These points could be input or output bit instance ports, input or output instance ports, insts, or a list of any of these. If you specify an instance port, it must *not* be sequential.

`[-stage stage_number]`

For a *Pipeline Timing Report*, specifies the stage number for which the critical paths will be displayed. This number must be less than or equal to the number of stages in the pipelined loop.

Note Control ops in pipelining are not associated with a particular stage, because in pipelining several pipeline stages are executed simultaneously.

To distinguish which op takes place in which stage, an additional control mechanism is implemented using the *pipeline stage vector* [[“PSV \(Pipeline Stage Vector\)” on page 8-60](#)]. The PSV takes values from control ops as input and combines them with a control value showing which stage is active (multiple stages could be active simultaneously).

The output values of the PSV are used to tag ops in pipeline stages. Therefore, ops from different stages consume values from control ops (in the form of PSV-produced tags), and control ops cannot be assigned to a particular stage. They will thus show up in timing reports for different stages.

See also [“Note about Control ops in the Pipeline View” on page 8-38](#)

`[-phase phase_number]`

For a *Pipeline Timing Report*, specifies the phase number for which the critical paths will be displayed. This number must be less than or equal to the initiation interval of the pipelined loop.

`module_id | behavior_id | state_id | loop_join_id`

Identifies a module, behavior, state, or loop join node in a pipelined loop for which to generate a timing report.

Examples

Here are two examples of timing reports:

- [“Example: Using -stage option” on page E-133](#)
- [“Example: Pre-scheduling timing report” on page E-134](#)

Example: Using -stage option

Here is a sample report of the **report_timing** command, using the **-stage** option:

Timing report of most critical path for stage 1 and phase 1 of pipelined loop
loop_while_begin

Instance Terminal	Master	Fanout	Delay (ps)	Arrival (ps)
-----	-----	-----	-----	-----
ctrlOr_ln53_Z_0_Wait_ln57/Q	flipflop_1_r1_0	10	120	120
sub_ln75_en/A[0]	unary_and_3		400	520
sub_ln75_en/Z	unary_and_3	1	400	920
I_mux_sub_12x1_A/C[1]	mux_2_12		0	920
I_mux_sub_12x1_A/Z[0]	mux_2_12	1	280	1200
sub_12x1/A[0]	sub_12x1		0	1200
sub_12x1/Z[1]	sub_12x1	3	400	1600
mux_j_ln72/A_00[1]	mux_2_12		190	1790
mux_j_ln72/Z[0]	mux_2_12	1	400	2190
merged_I_mux_mux_j_ln72_Z_0/A_01[0]	mux_2_11		0	2190
merged_I_mux_mux_j_ln72_Z_0/Z[0]	mux_2_11	1	400	2590
mux_j_ln72_Z_0/D[0]	flipflop_12		0	2590
mux_j_ln72_Z_0/CLK	flipflop_12		80	2670

Note The last part of this critical path, **mux_j_ln72_Z_0/CLK**, is the setup time for the memory address. The setup time is the time by which the signal must arrive before the clock edge.

Example: Pre-scheduling timing report

In the following pre-scheduling timing report, the total available time is determined by the two fixed I/O ops `read_DUT_MODE_ln42_unr0` and `write_DUT_DATA_OUT_7_ln81_unr7`, which in the original source code, or from “[Creating Individual States](#)” on [page 11-9](#), are separated exactly by one state [`wait()` statement].

This means that (in the absence of external delay directives), the total available time is two clock cycles.

The original clock cycle, of 8000 ps, is reduced to take into account the delay of registers and of their sharing muxes, to 6846 ps; hence the total available time is twice that amount, that is, 13692 ps.

The critical path is 44647 ps; hence the slack, which must be distributed across two clock cycles, is -30955.

Neglecting the estimation errors inherent in any pre-scheduling timing, power, or area report, you could expect the post-scheduling negative slack to be half of -30955, again since it must be distributed across two clock cycles.

```
Performing behavioral timing analysis for behavior 'DUT_process' with the effective
clock period 6846.
```

```
It is reduced from the original clock period 8000 by wire delay margin 10% and delay 393
to account for registers and their sharing MUXes.
```

```
Timing report for most critical path in process DUT_process with slack -30955 and total
available time 13692
```

Op Name	Op Type	Master	Delay (ps)	Arrival (ps)	Slack (ps)
-----	-----	-----	-----	-----	-----
read_DUT_MODE_ln42_unr0	read	-	3000	0	-30955
lt_ln44_unr4	gt	gtle_5x4	1943	3000	-30955
if_ln44_unr4	if	if_then_else	0	4943	-30955
...					
add_ln70_unr10	add	add_18	4635	38656	-30955
mux_exit_mux__datareg_tmp_ln33	mux	mux_8_70	621	43291	-30955
mux__datareg_tmp_ln33_1	mux	mux_10_80	735	43912	-30955
write_DUT_DATA_OUT_7_ln81_unr7	write	-	0	44647	-30955

E.1.91 **report_tlm_transactor_pairs**

Syntax

```
report_tlm_transactor_pairs [-h] [design_id]
```

Precondition

The design has completed the Start Step – the **new_design** command has been run.

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Displays the TLM transactor pairs defined for the specified design.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[design_id]`

Identifies the design for which transactor pairs are to be displayed. The default is the current design.

E.1.92 restructure_array

Syntax

```
restructure_array [-h] addr_width_delta array_id
```

Precondition

The module containing *array_id* has completed the Set up Design Step – the **build** command (“[build](#)” on [page E-30](#)) has been run.

The array (*array_id*) must meet the requirements in “[Requirements for Original and Resulting Array in Restructuring](#)” on [page 8-78](#).

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Changes the number of words in the specified array. The array is scaled up (an *increased* number of words) if *addr_width_delta* is a *positive* integer and scaled down (a *decreased* number of words) if *addr_width_delta* is a *negative* integer, in multiples of powers of 2.

See also “[Restructuring Arrays](#)” on [page 8-78](#).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

addr_width_delta

Indicates the amount to restructure: a positive integer indicates scaling up, and a negative integer indicates scaling down (this number cannot be 0).

array_id

Identifies the array to be restructured.

E.1.93 save_design

Syntax

save_design [-h] [-dir *design_directory*] [*design_object_id*]

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

You may proceed to the Specify Micro-architecture Step or any subsequent step.

Action

Saves the design. You can optionally specify a directory into which to save it. If a design database already exists in the directory to be used, it will be overwritten. You can also optionally specify the object id of a design other than the current design.

Important When a design is saved, CtoS *does not* save the source code, but *does* save the parse tree from which you could derive the source code. It *does not* save the **.lib** files, but *does* save the *path* to the **.lib** files (however, this is not very useful without the **.lib** files). It *does not* save the RC cache, but *does* save timing data.

Additionally, CtoS *does not* save the Tcl interpreter (and thus *does not* save Tcl variables, which are in the interpreter) – it saves only the CtoS design database.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-dir *design_directory*]

Specifies the directory in which to save the design. The default is the value of the **design_dir** design attribute (“[design_dir](#)” on page D-9), but if this attribute is not specifically set, the default is the value of the **name** design attribute (“[name](#)” on page D-18).

[*design_object_id*]

Identifies the design to be saved. The default is the current design.

E.1.94 schedule

Syntax

```
schedule [-h] [-verbose] [-max_negative_slack] [-post_optimize none|simple|  
advanced] [-use_birthday] [-passes num_passes] [-low_power] [object_id]
```

Note The *-low_power* option is part of Power Estimation, a preliminary feature.

Preconditions

The design has completed the Manage Resources Step.

If you do *not* want to use addsub resources, you must have set the **enable_addsubs** behavior attribute ([“enable_addsubs” on page D-41](#)) to *false* before running this command.

Interrupt

To halt scheduling, select the **Interrupt** button ([“Interrupt Button” on page 6-32](#)); the CtoS scheduler will stop at the end of the current pass. The design state is *Failed Schedule*; use the CDFG viewer to see failed ops.

Postcondition

You have completed the Scheduling Step and may proceed to the Manage Registers Step.

Action

First checks to see if you have already run the following commands, and if applicable to your design, runs them automatically, in the following order:

- **create_array_dependencies**
- **create_required_states**
- **create_initial_resources**

Then, it performs scheduling for the specified object. It also analyzes the op spans, if successful.

The behavior’s slack attribute is updated with the difference between the clock rate and the critical path as analyzed by the CtoS scheduler.

See [“Scheduling” on page -2](#) and [“Results of Scheduling” on page -17](#) for complete information on the CtoS scheduling process.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-verbose]

Controls the type of messages issued. If this option is included, the CtoS scheduler will display each op scheduled with negative slack, and each op that cannot be scheduled, with an explanation.

`[-max_negative_slack]`

Specifies the tolerance for starting the sharing phase as percentage of clock cycle. The default value is 100 (that is, a clock cycle).

This option is applicable only if the scheduling effort is medium or high.

- If negative slack < max_negative_slack, a warning is displayed and the scheduling stops with a valid RTL without sharing.
- If the negative slack is between 0 and max_negative slack, a warning is displayed but the scheduling continues to the sharing phase.

`[-post_optimize none | simple | advanced]`

Specifies the post-schedule optimization level. **none** skips all optimizations; **simple** skips the *Unsharing* optimization; **advanced** (the default) runs all optimizations.

`[-use_birthday]`

Specifies that the original source code schedule should be used.

Note In the CtoS 14.1 release, the command-line option ‘useBirthday’ has been renamed to ‘use_birthday’. Support of ‘useBirthday’ is still available in the CtoS 14.1 release, but it will be completely removed from future releases. Instead, use ‘use_birthday’.

`[-passes num_passes]`

Specifies the number of passes allowed. The default is **200**.

`[-low_power]`

Performs pre-scheduling timing analysis, evaluating the criticality of all operations; then chooses the best set of resources, in terms of delay/area/power trade-offs to implement the given set of operations.

`[object_id]`

Identifies a design, module or behavior to schedule. The default is the current design.

E.1.95 set_attr

Syntax

```
set_attr [-h] attribute_name attribute_value [object_id]
```

Precondition

The design has completed the Start Step – the **new_design** command has been run.

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Sets the value of an object's attribute. If no object is specified, the current scope is used.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

attribute_name

Specifies the name of the attribute to be set; see [“CtoS Object Reference” on page D-1](#) for a complete list.

attribute_value

Specifies the new value of the attribute.

[object_id]

Identifies the object whose attribute is to be set. The default is the current scope.

Example

For a design, **Mydesign**, to set the value of the built-in attribute **compile_flags**, you would use:

```
set_attr compile_flags " -w -DJUST_TESTING " /designs/Mydesign
```

Here is how to append an extra define of a compiler flag **FIXED_WAIT** to the value for **compile_flags** that you fetched for the current value of this attribute (in **\$cf**, described in [“get_attr” on page E-73](#)):

```
set ef [join [join $cf] " -DFIXED_WAIT "]  
set_attr compile_flags " $ef " /designs/Mydesign
```

Note You must put a blank after the first " and before the \$, and type in the square brackets, as this is a Tcl command, and the value of the command in the brackets is to be returned.

E.1.96 set_baseline

Syntax

```
set_baseline [-h] [-verbose level] baseline_id
```

Precondition

The design has completed the Set up Design Step– the **build** command has been run.

Note This command is intended for use immediately after the initial **build**, to enable the reuse of scripts for micro-architectural selection.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Sets the baseline design for Incremental Synthesis and renames the current design to be as similar as possible, after first checking to see if the current design already has a baseline.

It does this in the following three steps:

1. It compares the *baseline design* and *current design* and tries to identify portions that were not changed. Again, if there are no changes, it does not perform any matching or renaming.

Note This process is heuristic in nature and is similar to the UNIX *diff* command; thus it has no guarantee of identifying the same similarities as a human designer. However, if *diff* finds few differences, this matching is very likely to be highly successful.

2. For all nodes and ops of the *current design* for which it could find a correspondence to the *baseline design*, it renames those nodes and ops to be *identical* to the corresponding names in the *baseline design*. Conversely, the names of any nodes and ops of the *current design* that could not be matched are appended with **_v** and the **eco_version** (*ECO version number*) to clearly identify them.

Note The **eco_version** is an integer attribute of a design, which is automatically initialized to zero when a design is created, and is set to one higher than the corresponding attribute of the most recently used baseline in a **set_baseline** command¹. In this manner, nodes and ops of the *current design* used in scripts and in Tcl scripts remain identical. This is helpful in reusing scripts that were initially defined from the *baseline design*.

1. Manipulation of this attribute via **set_attr**, although possible, is not recommended. The ECO version number of zero is not appended, so **_v** and **eco_version** are appended only after matching and renaming occurs.

3. It flags, in the baseline design (which must remain open in its scheduled version throughout the scheduling, register allocation and RTL generation steps of the new design), the following:

- set of resources
- created states
- operation bindings
- value bindings
- RTL instance order

so that the **create_initial_resources**, **schedule**, **allocate_registers** and **write_rtl** commands can reuse them as much as possible later.

Note This command sets the baseline information for the current design, so if you close the current design and open a new one, the baseline information for the new design must be set again. Also, if the baseline design is closed, all of the following commands (**schedule**, **allocate_registers** and **write_rtl**) behave non-incrementally. The information about which design was used as the baseline for the current design is not saved when the new design is saved. However, the effects of the **set_baseline** commands on the current design are saved, so that the saved design can be used as the baseline for a new Incremental Synthesis step, and so on.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-verbose level]`

Indicates that you want to print out information about matched and unmatched node names.

baseline_id

Specifies the identifier for the baseline design.

E.1.97 **set_design**

Syntax

`set_design [-h] design_object_id`

Precondition

You have opened at least one design using **open_design -not_current** ([“open_design” on page E-89](#)).

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Lets you set the *current* design to any of your open designs.

If you have a current design when you use this command, that design will remain present in the session, but will no longer be the current design.

See also [“Setting the Current Design” on page 6-48](#).

Return Value

The object ID of the new current design

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

design_object_id

Specifies the design that you want to set as the current design.

E.1.98 set_stall_value

Syntax

```
set_stall_value [-h] -value bit_value net_id
```

Precondition

- You must use this command before the Netlist Step.
- The design has completed the Set up Design Step— the **build** command has been run.
- The pipelined loop must have initiation interval (II) equal to 1

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Lets you to set a stall value to be applied to the specified net.

Return Value

The object ID of the new current design

Known Limitations

The **set_stall_value** command has the following limitations:

- The **report_timing** command on a behavior and the **report_slack** command do not account for the combinational path created by the **set_stall_value** command
- The simulation model after scheduling does not model the combinational path

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-value bit_value

Specifies the single bit constant stall value to be asserted during stall. Specified net must be written by only a single behavior.

-net_id

Specifies the net to be used during the stall. The net width must be 1.

E.1.99 set_synthesis_mode

Syntax

```
set_synthesis_mode [-h]
                    -mode manual|simple|cycle_accurate|relax_latency|pipeline
                    [object_id]
```

Note *This command is part of Specifying Micro-architecture with Synthesis Modes, a preliminary feature.*

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

You may override some actions on loops, functions or arrays with **set_uarch_action** command or apply the actions with **apply_uarch_action** command.

Action

Lets you set the synthesis mode on the specified design, module or behavior. Depending on the mode, micro-architecture actions will be set on loops, functions and arrays contained in the specified object which are hints to make it synthesizable. A subsequent call to **apply_uarch_action** command will result in a design that is synthesizable.

See also [“Resolving Functions” on page 8-3](#)

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-mode manual|simple|cycle_accurate|relax_latency|pipeline

Specifies mode to apply.

[object_id]

Specifies the design, module or behavior that you want to set the synthesis mode.

E.1.100 set_uarch_action

Syntax

set_uarch_action [-h]

Set Loop Action

set_uarch_action [-h] -action unroll|break|pipeline|no_action join_node_id

Set Function Action

set_uarch_action [-h] -action inline|no_action behavior_id|custom_op

Set Array Action

set_uarch_action [-h] -action flatten|builtin|prototype|vendor|rom|no_action
array_id

Note *This command is part of Specifying Micro-architecture with Synthesis Modes, a preliminary feature.*

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

You may continue setting additional actions and apply actions with **apply_uarch_action** command. You can also continue with other commands allowed in the Specifying Micro-architecture Step or Allocate IP Step.

Action

Lets you set the micro-architecture action on the loop, function or array.

See also [“Resolving Functions” on page 8-3](#).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-action <action>

Specifies action to apply.

object_id

Applies action to this object.

E.1.101 split_array

Syntax

```
split_array [-h]
split_array -addr bit_index array_id
split_array -data bit_index array_id
```

Precondition

The design containing the array has completed the Set up Design Step – the **build** command has been run – and the array has not gone through the Allocate IP Step.

The array (*array_id*) and the bit index (*bit_index*) must meet the requirements in “Requirements for Original Array, Bit Index, Resulting Array” on page 8-65.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

In rare situations, it is possible that the resultant array(s) *array_id_0* or *array_id_1* could be optimized out by this (**split_array**) command.

Action

Splits an array, based on either the address or data bits of the array elements, into two smaller arrays, named *array_id_0* and *array_id_1*. See also “Splitting Arrays” on page 8-65.

Note If the names *array_id_0* and/or *array_id_1* are already in use, the first available identifier (*array_id_x*) is used.

Command Arguments

[-h]

Provides a brief description of the syntax and arguments.

-addr

All elements in the specified array whose addresses contain the bit *bit_index* set to **0** are mapped to *array_id_0*, and all elements whose addresses contain the bit *bit_index* set to **1** are mapped to *array_id_1*. Accesses to the same array will use different read and write ports, which may not create contention. The data width of the two new arrays will be the same as that of the original array.

-data

Bits **0** to (*bit_index* - **1**) in each input array element are placed in *array_id_0*, and the rest of the bits become part of *array_id_1*. The data width of *array_id_0* will be *bit_index* bits, and the data width of *array_id_1* will be (**M** - *bit_index*) bits, where **M** is the data width of the original array. The total number of words in the two new arrays will be the same as the number of words in the original array.

bit_index

Specifies the bit index to be used for splitting the array.

array_id

Specifies the name of the array to be split.

E.1.102 **split_op**

The split_op command is a preliminary feature.

Syntax

```
split_op [-h] [-max_out_width integer] op_id
```

Preconditions

The design has completed the Set up Design Step – the **build** command has been run.

The op to be split should not have been previously bound.

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Converts the arithmetic operations of addition, subtraction and multiplication from one **M** bits operation into multiple **N** bits operations.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-max_out_width integer]

Specifies the maximum output width. The range is from **4** to the width of the op. The default is **16**.

op_id

Specifies the add, subtract, or multiplier op to be split.

E.1.103 **undefine_control_error_terminal**

Syntax

`undefine_control_error_terminal [-h] behavior_id`

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Clears the **error_recovery_terminal** behavior attribute (“[error_recovery_terminal](#)” on page D-42) of the specified behavior, which was set using the **define_control_error_terminal** command (“[define_control_error_terminal](#)” on page E-54).

See also “[Generating an RTL Description after Scheduling](#)” on page 13-36.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

behavior_id

Specifies the behavior for which the **error_recovery_terminal** will be undefined.

E.1.104 **undefine_tlm_transactor_pair**

Syntax

```
undefine_tlm_transactor_pair [-h] -transactor name [design_id]
```

Precondition

The design has completed the Start Step – the **new_design** command has been run.

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Removes the definition of a pair of dual TLM transactors for the design from the set of transactor pairs in the design. If no such transactor pair exists, no action is taken.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-transactor *name*

Specifies the name of the transactor module. This is the name of the **SC_MODULE** that constitutes the transactor.

Note If the transactor **SC_MODULE** is defined in a namespace, *name* must be qualified with that namespace (for example, **xbus::xbus_slave_transactor**).

[*design_id*]

Specifies the design to which this command applies. The default is the current design.

E.1.105 unroll_loop

Syntax

```
unroll_loop [-h] [-num_pre_body integer] [-num_in_body integer]  
            [-num_tries integer] loop_join_id ...
```

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

Loop unrolling is automatically followed by an internal optimization step.

You may continue with the Specify Micro-architecture Step or proceed to the Allocate IP Step.

Action

Unrolls the specified loop(s), either partially or completely, depending on other arguments used. If neither **-num_pre_body** nor **-num_in_body** is specified, the loop(s) is completely unrolled (there must be a fixed bound on the number of iterations). If either **-num_pre_body** and/or **-num_in_body** is specified, the loop(s) is partially unrolled.

-num_pre_body specifies the number of *iterations* of a loop that should unrolled before the loop body, while **-num_in_body** specifies the number of *copies* (including the original) that are present in the loop body.

Specifying **-num_pre_body** is useful when the first iteration of a loop is substantially different from the other iterations, or you would like to overlap the execution of the first iterations with the computation that precedes the loop. Unrolling the loop body can be used as an alternative to (or in conjunction with) the **pipeline_loop** command to increase the throughput of a loop by overlapping the computations of several iterations of the loop body.

Note that the following would leave a loop unchanged, so this set of options is not supported and generates an error:

```
unroll_loop -num_pre_body 0 -num_in_body 1
```

Important Note both of these considerations when working with *multiple loops*:

- When multiple loops are specified, the innermost loops will be unrolled first (so that new loops are not created when unrolling both an inner and outer loop).
- It is faster to unroll multiple loops by using one **unroll_loop** command than by using multiple **unroll_loop** commands.

Command Arguments

[*-h*]

Provides a brief description of the command syntax and arguments.

[*-num_pre_body integer*]

Specifies the number of *iterations* of the loop(s) to be unrolled prior to the loop body (the default is **0**).

[*-num_in_body integer*]

Specifies the number of original loop bodies to be in the loop(s) after unrolling (the default is **1**).

Note Both *num_pre_body* and *num_in_body* may be specified, but their sum must not exceed the total number of times the loop body will be invoked. If a loop is declared data-dependent, either *num_pre_body* or *num_in_body* must be specified.

[*-num_tries integer*]

Indicates the limit of the number of iterations of a loop to be tested to determine if the loop terminates. Symbolic simulation is used to test whether loop termination is data independent. To prevent analysis time from being excessive, only this number of iterations are tested. If the loop has still not terminated, the loop will be treated as an infinite or unbounded loop and will not be unrolled.

If you *do not* use this option, the command defaults to a limit of 256 iterations; if you *do* use this option, you may specify any limit.

loop_join_id ...

Identifies a loop join node(s) (that is, the *head* of the loop). See also “[loop_join_id](#)” on page N-8.

Example

Suppose you have the following code:

```
for (UINT i = 0; i < 16; i++) {
    out[i] = a[i] + b[i];
}
```

Unrolling this loop using:

```
unroll_loop -num_pre_body 2 -num_in_body 2 object_id
```

will produce code that looks like this:

```
out[0] = a[0] + b[0];
out[1] = a[1] + b[1];
for (UINT i = 2; i < 16; i += 2) {
    out[i] = a[i] + b[i];
    out[i+1] = a[i+1] + b[i+1];
}
```

E.1.106 **unschedule**

Syntax

`unschedule [-h] [object_id]`

Precondition

The design has completed the Manage Registers Step, but you have not run the **allocate_registers** command.

Postcondition

You may go back to the Manage Resources Step.

Action

Removes all binding information from the previous scheduling attempt.

However, it does not remove any resources or states created by the CtoS scheduler, by the user, or by the **create_initial_resources** command.

Known Limitation

The **unschedule** command will fail if there is a pipelined loop in the design.

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[object_id]`

Identifies a design, module, or behavior for which to undo scheduling. The default is the current design.

E.1.107 use_dsp

Syntax

```
use_dsp [-h] [-latency num_cycles] (op_id|[-min_width min_width] [behavior_design_id])
```

Preconditions

The design has completed the Set up Design Step – the **build** command has been run.

You must have set FPGA as the Implementation Target.

Postcondition

You may continue with the Specifying Micro-architecture Step or any subsequent step, up to the Manage Array Dependencies Step.

Action

Binds multiply ops to FPGA Digital Signal Processing (DSP) resources. You can choose to map a specific multiplier, or all multipliers (above a minimum width), in a behavior or in the entire design. For each of the multiplier ops specified, CtoS:

- replaces the op with a custom op in which the combinational behavior contains a single multiply op.
- binds the combinational behavior to a RTL IP provided by CtoS.

Using reset_all_registers design attribute with this command

DSP RTL IP resources for Xilinx have a *synchronous* reset with *high* active level, while those for Altera have an *asynchronous* reset with *low* active level. These resets are connected to **0** or **1** if the **reset_registers** design attribute is **only_required** (the default). Conversely, these resets are connected to the reset of the behavior of the **use_dsp** command arguments if **reset_registers** is **internal** or **internal_and_outputs**.

In the latter case (**reset_registers** is **internal** or **internal_and_outputs**), if a behavior has multiple resets, or if the reset type of the behavior is different from the reset type of the DSP resource, you will receive ERROR (CTOS-15089).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-latency num_cycles]

Identifies the latency of the pipelined DSP resource. The range is **0** to **3**, and the default is **3**.

op_id

Identifies the multiply op that is mapped. Only input port widths less than or equal to 18 are supported.

[-min_width min_width]

Specifies the minimum width for ops, that is, ops with smaller output widths are not mapped. This value must be greater than or equal to **0** and less than or equal to **36**.

[*behavior_design_id*]

Identifies the behavior or design of which all multiply ops are mapped.

E.1.108 use_ip

Syntax

```
use_ip [-h] rtl_ip_def function_behavior | module
```

Preconditions

The design has completed the Set up Design Step – the **build** command has been run, and you have *not* run the **schedule** command.

Postcondition

You may continue with the Specifying Micro-architecture Step or any subsequent step, up to the Manage Array Dependencies Step.

Action

Specifies that a combinational function behavior must be implemented using given **rtl_ip_def** object or specifies that the given module must be implemented using the **cgic_ip_def** object.

The function will be synthesized using the user-provided RTL IP, instead of the SystemC function body.

This command can be used to substitute hand-written RTL for a SystemC function.

Important This command cannot be used to substitute hand-written RTL for an **sc_module**, or a memory – for a memory, you should use either the **allocate_prototype_memory** or **allocate_memory** commands.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

rtl_ip_def

Identifies the RTL IP definition object to use for binding.

function_behavior

Identifies a behavior for the combinational function to bind.

module

Identifies a module to bind

E.1.109 write_gates

Syntax

write_gates [-h] [*design_id*]

Precondition

The design has completed the Manage Registers Step – the **allocate_registers** command has been run – and you have also run the **write_rtl** command on all modules in the design.

Postcondition

You may continue with the Analyze and Implement Step.

Action

Generates gates by calling RC with an *RC Run Script* (**rc_run.tcl**). The *RC Run Script* is automatically generated [it calls the **write_rc_run_script** command (“[write_rc_run_script](#)” on page E-158] in the run directory specified in the *Synthesis Configuration* [see “[Default Synthesis Configuration Object Attributes \(synthesis_configs\)](#)” on page D-86].

For a *Foundation Flow*, a gates file is generated in the run directory, again as specified in the *Synthesis Configuration*, with the name <topMod><VerilogRTLSuffix>.vg, where <VerilogRTLSuffix> is the design object attribute, **verilog_rtl_model_suffix** (“[verilog_rtl_model_suffix](#)” on page D-26).

Other reports are also generated in the run directory under **reports** (see “[Viewing Reports Generated by Default Synthesis Flow](#)” on page 13-52).

Note All other processes will be blocked by this command until RC exits; however, if you are generating gates using the CtoS GUI, other processes are not blocked.

See also “[Generating Gates in CtoS](#)” on page 13-42.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[*design_id*]

Specifies the design to which this command applies. The default is the current design.

E.1.110 write_rc_run_script

Syntax

```
write_rc_run_script [-h] [-rtl_file filename] [-report_power] [module_id]
```

Precondition

The design has completed the Manage Registers Step – the **allocate_registers** command has been run – and you have also run the **write_rtl** command on all modules in the design.

If you are using prototype memories, you will get a warning.

FPGAs are not supported.

Postcondition

You may continue with the Analyze and Implement Step.

Action

Generates the following file(s) in the run directory specified in the *Synthesis Configuration*:

- the *RC Run Script*, named **rc_run.tcl**, which is a Tcl script runnable within RC.

If the *Synthesis Configuration* specifies the use of a *Foundation Flow*, it also generates:

- the *RTL Design Configuration*, named **topModule_rtl_config.tcl**, which describes the RTL to synthesize (see also “[Understanding RTL Design Configurations](#)” on page 13-47).
- the *SDC File*, named **topModule_rc.sdc**, which represents the clocks and external delays.

See also “[Understanding RC Run Scripts](#)” on page 13-51.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-rtl_file filename]

Specifies the CtoS-generated RTL filename. If not specified, the RTL filename from a previously created [by the **write_rtl** command (“[write_rtl](#)” on page E-161)] RTL output parse tree is used.

[-report_power]

Generates power reporting commands if a TCF (toggle count format) file has been read.

[module_id]

Specifies the module being synthesized. The default is the top-level module.

E.1.111 write_rc_script

Syntax

```
write_rc_script [-h] [-o filename] [-rtl_file filename] [-report_power] [module_id]
```

Precondition

The design has completed the Manage Registers Step – the **allocate_registers** command has been run.

Note This command does not support RTL IP.

Postcondition

You may continue with the Analyze and Implement Step.

Action

Writes an Encounter RTL Compiler (RC) script file for use as input to RC for the specified module in the design. An SDC file is also generated for the module. See also “[Generating RC Scripts](#)” on page 13-53.

Note If the **low_power_clock_gating** design attribute (“[low_power_clock_gating](#)” on page D-17) is set to *true*, the following line will be added to your script:

```
set_attr lp_insert_clock_gating true
```

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-o filename]

Specifies the output script filename. The base name of this filename (with the **.sdc** extension) is also used for the SDC filename. The default output script filename is *top_level_module_rc.tcl*, and it is written to the current directory; the default filename for the SDC file is *top_level_module.sdc*.

[-rtl_file filename]

Specifies the CtoS-generated RTL filename. If not specified, the RTL filename from a previously created (by the **write_rtl** command) RTL output parse tree is used.

Warning Do not specify the **-rtl_file** option if you are performing hierarchical synthesis. The list of RTL files generated during hierarchical synthesis will be automatically looked up by CtoS (if you have run the **write_rtl** command with the **-recursive** option before running the **write_rc_script** command).

[-report_power]

Generates power reporting commands if a TCF (toggle count format) file has been read.

[module_id]

Specifies the module being synthesized. The default is the top-level module.

E.1.112 **write_rename_states_script**

Syntax

```
write_rename_states_script [-h] -file <filename> [object_id]
```

Precondition

The design has completed the Manage Registers Step - the **allocate_registers** command has been run.

Postcondition

You may continue with the Analyze and Implement Step.

Action

Writes script that can be used in subsequent session to rename states to the current names.

Warning If multiple states in a behavior have same set of annotation paths, they are skipped in the generated script. CtoS issues warning messages for such states. The user should add labels on the state and/or pragmas on function calls to avoid this conflict.

See also [“Renaming States in Support of FSM Coverage” on page 7-25](#).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-file <filename>

Specifies the filename of the script.

[object_id]

Identifies the design, module, or behavior for which to write the rename script.

E.1.113 write_rtl

Syntax

```
write_rtl [-h] -file filename|-dir directory [-non_recursive]
          [-verilog_module per_sc_module|per_process|per_function]
          [-file_policy per_verilog_module|per_sc_module] [-tcf][[-psl psl_filename]
          [-slec slec_filename] [-control_error_recovery] [module_id]
```

Preconditions

The design has completed the Manage Registers Step – the **allocate_registers** command has been run.

To set filename extensions, you can modify the design attributes **verilog_rtl_model_suffix** and **verilog_top_wrapper_suffix** before issuing this command. These extensions should not be the same because this can result in a name conflict of the top module and the top wrapper module of the design. See also “[verilog_rtl_model_suffix](#)” on page D-26, and “[verilog_top_wrapper_suffix](#)” on page D-27.

For CtoS to generate non-blocking assignments with a #1 delay control, you must have set the **verilog_use_non_blocking_delay_control** design attribute to *true* before issuing this command. See also “[verilog_use_non_blocking_delay_control](#)” on page D-28.

Warning Contrary to some designers’ opinions, this coding style will *not* help avoid Verilog race conditions. Furthermore, since logic synthesis tools, such as Encounter RTL Compiler (RC), will ignore these added intra-assignment delays, the use of this design attribute is not recommended for final RTL.

Postcondition

You may continue with the Analyze and Implement Step.

Action

Generates a synthesizable RTL description for the specified module or all modules in the design.

Known Limitation

SLEC and PSL output are not supported for default **recursive** behavior of **write_rtl**; that is, if the **-non_recursive** option of the **write_rtl** command is not specified, the **-slec** and **-psl** options are not supported.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-file filename|-dir directory

Generates a single file for all the modules if the **-file** option is specified.

When the **-dir** option is specified, separate files are generated within specified directory. The filenames of each module are defined by CtoS based on verilog module name and verilog file extension design attribute. Even non user-defined modules will be written to separate files.

Note These are exclusive options and only one must be specified.

`[-non_recursive]`

Indicates that a RTL model is generated for the specified module instead of the hierarchical design.

By default, it is set to **recursive**; so, if you do not specify the **-non_recursive** option a model is generated for the given database module, and every module transitively instantiated from that module. Every database module that corresponds to an **SC_MODULE** in the input source is written to a separate file or single file is derived by **-dir|-file** option, respectively.

`[-verilog_module per_sc_module|per_process|per_function]`

Specifies whether to write a verilog module for each SystemC module, process, or function constructs. The default value is **per_sc_module**, and the **write_rtl** command generates one Verilog module for every user defined SystemC module.

`[-file_policy per_verilog_module|per_sc_module]`

Specifies if files must be generated for each Verilog module or SystemC module. The default value is **per_verilog_module**.

Note This option is applicable only with the **-dir** option. There is no change in behavior when this option is used in conjunction with **-file**, because all the modules are placed in a single file.

`[-tcf]`

Generates RTL compatible with TCF (toggle count format) power analysis. TCF is the Cadence standard format to describe switching activity information in a design.

The switching activity information contained in the file is required for accurate power analysis or power optimization of a design. The **-tcf** option instructs model generation to promote all process-local registers up to the module scope.

Note When **-tcf** is used while generating the RTL, an unused register is created to generate a correct TCF file. To avoid warnings from the linting tools, this register is created under the lint pragma. If the users wish to run the generated RTL through a linting tool, they should enable the lint pragma processing. In **hal**, this is done by passing the **-pragma** and **-lintpragma** options to the **hal** command.

`[-psl psl_filename]`

Specifies the file in which properties of the synthesized design are generated as PSL assertions. An assertion that the control FSM has a 1-hot encoding is produced.

`[-slec slec_filename]`

Specifies the file to be used for output of a script and an XML file for verifying the generated RTL using SLEC. See also “[Generating a SLEC Script and XML File](#)” on page 13-38 for more detail.

Important If *all* of the memories in a design are exported, do not use the **-slec** option for **write_rtl**, but do use the **-slec** option with the **write_top_wrapper** command. If *only some* of the memories are exported, use the **-slec** option with both commands. See also [“Generating SLEC Files with Exported Memories” on page 13-39](#) for more detail.

`[-control_error_recovery]`

Indicates that if the state machine ever enters an illegal state, the state machine will be reset and an error will be signaled on an output terminal specified by the **define_control_error_terminal** command.

`[module_id]`

Identifies a module for which to generate the RTL description. The default is the top-level module.

E.1.114 write_setup

Syntax

```
write_setup [-h] [-o name] [object_id]
```

Precondition

The design has completed the Start Step – the **new_design** command has been run.

Postcondition

You may continue with the Set up Design Step or any subsequent step.

Action

Writes design attributes into a Tcl file, which can then be run for design setup.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-o name]

Identifies the name of the output Tcl file. The default is *design_name_setup.tcl*.

[object_id]

Identifies a design for which attributes have been set.

E.1.115 write_sim

Syntax

```
write_sim [-h] [-suffix suffix_string] [-birthday] [-single_file] [-recursive]  
          [-tcf] [-o name_string] [module_id]
```

Preconditions

With the **-birthday** option, the design has completed the Set up Design Step – the **build** command has been run.

Without the **-birthday** option, the design has completed the Scheduling Step – the **schedule** command has been run.

If you want CtoS to generate non-blocking assignments with a #1 delay control, you must have set the **verilog_use_non_blocking_delay_control** design attribute to *true* before running this command.

Warning Contrary to some designers' opinions, this coding style will *not* help avoid Verilog race conditions. Furthermore, since logic synthesis tools, such as Encounter RTL Compiler (RC), will ignore these added intra-assignment delays, the use of this design attribute is not recommended for final RTL.

Postcondition

You may continue with the Specifying Micro-architecture Step or any subsequent step.

Action

Generates a simulation model using the original source schedule (**-birthday**) or the schedule created by CtoS. If you use the **-birthday** option, all behaviors in the module must be fully scheduled, and actual read latencies are used. If you use the **-suffix** option, every module name is suffixed with the specified string (this is used to create models that can be simulated side-by-side with the original model).

The generated simulation model is compliant with the *IEEE Standard 1364-1995/2001 for Verilog HDL* standard.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-file *filename* | -dir *directory*

Generates a single file for all the modules if the **-file** option is specified.

When the **-dir** option is specified, separate files are generated within specified directory. The filenames of each module are defined by CtoS based on verilog module name and verilog file extension design attribute. Even non user-defined modules will be written to separate files. This option is independent of the **-non_recursive** option.

Note These are exclusive options and only one must be specified.

[`-non_recursive`]

Indicates that a RTL model is generated for the specified module instead of the hierarchical design.

By default, it is set to **recursive**; so, if you do not mention the **-non_recursive** option a model is generated for the given database module, and every module transitively instantiated from that module. Every database module that corresponds to an **SC_MODULE** in the input source is written to a separate file.

[`-suffix suffix_string`]

Specifies an optional string to be used as a suffix to every module name. The default is **_sim**.

[`-birthday`]

Specifies that the schedule from the original source code is to be used.

[`-single_file`]

Indicates that declarations and definitions should be combined into a single file for SystemC output. The default is that two separate header and source files are produced.

[`-recursive`]

Indicates that a model be generated for the given database module, and every module transitively instantiated from that module. Every database module that corresponds to an **SC_MODULE** in the input source is written to a separate file. If you use the **-o** option with the **-recursive** option, it specifies a directory in which to generate the output files. The names of the output files are the names of the database module concatenated with the specified suffix and file extension.

If you do not use the **-recursive** option, then database modules corresponding to **SC_MODULES** in the input source, transitively instantiated within the given module, are *not* included in the output. In this case, the **-o** option would specify the filename of output file, as usual.

[`-tcf`]

Generates a simulation model compatible with TCF (toggle count format) power analysis. TCF is the Cadence standard format to describe switching activity information in a design. The switching activity information contained in the file is required for accurate power analysis or power optimization of a design. The **-tcf** option instructs model generation to promote all process-local registers up to the module scope. In addition, it causes the creation of extra registers, which are assigned a value of 1 in every clock cycle for which an edge or tag of the CDFG is active.

[`-o name_string`]

The value for the *name_string* depends on whether you are also using the **-recursive** option:

- with the **-recursive** option, this is the directory name in which to generate output files (see the **-recursive** option, above).
- with no **-recursive** option, this is the output simulation model filename. The default is **top_level_module_sim** with the extension specified by the **verilog_out_file_ext** design attribute (whose default is **v**).

[*module_id*]

Identifies a module for which to generate the simulation model. The default is the top-level module.

E.1.116 **write_sim_makefile**

Syntax

`write_sim_makefile [-h] [-overwrite] [design_id]`

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

If you have exported memories, the design has completed the Allocate IP step.

If you do not want to use the defaults, you have defined the simulation configuration with the **define_sim_config** command ([“define_sim_config” on page E-55](#)).

Postcondition

None

Action

Writes a simulation makefile based on the simulation configuration of the specified design.

See also [“Defining a Simulation Configuration and Generating Simulation Makefiles” on page 7-17](#).

Note You can have an overall makefile that references **Makefile.sim** and **Makefile.synth** (see also [“write_synth_makefile” on page E-169](#)).

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-overwrite]`

Indicates that you want to overwrite an existing makefile.

`[design_id]`

Specifies the design to which this command applies. The default is the current design.

E.1.117 write_synth_makefile

Syntax

```
write_synth_makefile [-h] [-overwrite] -name file [design_id]
```

Precondition

The design has completed the Manage Registers Step – the **allocate_registers** command has been run – and you have also run the **write_rtl** command on all modules in the design.

If you do not want to use the defaults, you have defined the *Synthesis Configuration* with the **define_synth_config** command ([“define_synth_config” on page E-56](#)).

Postcondition

None

Action

Writes a logic synthesis makefile based on the *Synthesis Configuration* of the specified design.

See also [“Generating Gates from Logic Synthesis Makefiles” on page 13-53](#).

Note You can have an overall makefile that references **Makefile.sim** and **Makefile.synth** (see also [“write_sim_makefile” on page E-168](#)).

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-overwrite]

Indicates that you want to overwrite an existing makefile.

-name file

Specifies the name of the makefile to generate.

[design_id]

Specifies the design to which this command applies. The default is the current design.

E.1.118 write_tlm_wrapper

Syntax

```
write_tlm_wrapper [-h] [-o filename] [module_id]
```

Preconditions

The design has completed the Set up Design Step – the **build** command has been run.

Transactor pairs have been defined using the **define_tlm_transactor_pair** command for all transactors in the given module.

Postcondition

You may continue with the Specifying Micro-architecture Step or any subsequent step.

Terminology

For this command description, the following terminology is used:

- *top module* refers to the module for which the TLM wrapper is to be generated.
- *port* refers to any object of type **sc_in<T>**, **sc_out<T>**, **sc_clock**, **sc_port<IF>**, or **sc_export<IF>**.
- The *core instance* is an **SC_MODULE** instance for which no mirror transactor has been specified.
- *transactor instances* are module instances for which a module transactor pair for their master module has been defined in the design.

Action

Generates a TLM wrapper for the specified module in the specified file.

The command first determines whether the given module satisfies well-formedness requirements, and if so, an **SC_MODULE** definition for the TLM wrapper is generated in the output file.

- The *top module* must not contain any **sc_signal<T>** members.
- The *top module* must be purely structural, that is, it must not declare any processes.
- The *top module* must instantiate one or more transactors for which a mirror transactor has been defined.
- The *top module* must precisely instantiate the *core instance*.

- All **sc_in**<T>, **sc_out**<T>, and **sc_clock** members of both the *transactor instances* and the *core instance* must be bound to **sc_in**<T>, **sc_out**<T> or **sc_clock** members of the *top module*.
- Each **sc_port**<IF> member of a *transactor instance* must be bound to an **sc_export**<IF> member of the *core instance*.
- Each **sc_port**<IF> member of the *core instance* must be bound to one **sc_export**<IF> member of a *transactor instance*.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-o filename]

Specifies the name of the output file. The default is *top_module_name_ctos_tlm_wrapper.cpp*.

[module_id]

Identifies the *top module*, for which to generate the wrapper. The default is the top-level module.

Design Requirements

Before a TLM wrapper can be created for it, a design must meet the following requirements.

- **Each port of a transactor must have a dual port with the same name in the mirror transactor.**

The dual port is a port with the same name, but with a type as indicated in [Table E-4 on page E-171](#).

Table E-4 Acceptable types for dual port (TLM wrapper requirement)

Type of port in transactor	Type of dual port in mirror transactor	Comment
sc_in<bool>	sc_in<bool>	for a clock or reset port
sc_in<T>	sc_out<T>	for a port that is not used as clock or reset
sc_out<T>	sc_in<T>	
sc_port<IF>	sc_export<IF>	
sc_export<IF>	sc_port<IF>	

Limitation This requirement is currently not checked because CtoS is not processing the definition or declaration of the mirror transactor.

- **Only one module instance in the top module is allowed to lack a mirror transactor.**

That module instance is the *core instance* (all other instances being *transactor instances*).

Note You will get ERROR (CTOS-14035) if the *top module* contains more than one module instance whose master module lacks a transactor pair definition.

- **The top module must have a core instance.**

Note You will get ERROR (CTOS-14039) if this requirement is not met.

- **The top module must have at least one transactor instance.**

Note You will get ERROR (CTOS-14040) if this requirement is not met.

- **No `sc_signal<T>` objects are allowed in the top module.**

There are two reasons for this:

- The *top module* must not contain any processes, and
- The ports of the module instances in the *top module* must be connected to ports of the *top module*, or to ports of other module instances.

Note You will get ERROR (CTOS-14036) if this condition is not met.

- **The top module cannot contain `sc_port<T>` or `sc_export<T>` members.**

Note You will get ERROR (CTOS-14037) if this requirement is not met.

- **All ports of the module instances in the top module must be bound.**

Note You will get ERROR (CTOS-14038) if this requirement is not met

- **Each `sc_port<IF>` object of the core instance must be bound to an `sc_export<IF>` object of a transactor. Each `sc_port<IF>` object of a transactor instance must be bound to an `sc_export<IF>` object of the core instance.**

Note You will get ERROR (CTOS-14041) if this requirement is not met. Direct binding of an `sc_port<IF>` object of a module instance in the *top module* to another module instance is not supported.

- **Each `sc_in<T>`, `sc_out<T>` and `sc_clock` object of a module instance in the top module must be bound to a similar object of the top module.**

Note You will get ERROR (CTOS-14042) if this requirement is not met.

- **Any clock and reset port of a transactor instance must be bound to a port of the top module for which a port of the core instance is also bound to that port.**

Limitation This requirement is currently not checked.

E.1.119 write_top_wrapper

Syntax

```
write_top_wrapper [-h] -o filename [-slec filename] [design_id]
```

Note The *-slec* option is a preliminary feature.

Preconditions

The design has completed the Allocate IP Step - the **allocate_memory**, **allocate_prototype_memory**, **allocate_memory_interfaces** commands have been run.

Note To generate the correct top wrapper for designs with exported memories, the RTL has to be generated for all the modules transitively instantiated from the top module. For more information, see “Generating a Top Wrapper for Exported Memories” on page 13-40.

To generate the index file with the correct set of RTL files, you must have run the **write_rtl** command on all modules in the design.

To use the *-slec* option, the design must consist of a single user-defined module.

To set filename extensions, before issuing this command, you must have set the design attributes **verilog_rtl_model_suffix** (“verilog_rtl_model_suffix” on page D-26) and **verilog_top_wrapper_suffix** (“verilog_top_wrapper_suffix” on page D-27). These extensions *cannot* be empty strings because this can result in a name conflict of the top module and the top wrapper module of the design.

Postcondition

You may continue with the Analyze and Implement Step.

Action

Writes a simulation model of the top level wrapper module.

For designs with exported memories, generates the top wrapper module (see “Exporting Memories” on page 9-20 and “Generating SLEC Files with Exported Memories” on page 13-39).

This top wrapper module instantiates all exported memories of the synthesized design, as well as the rest of the RTL of the design. It is used for simulating the RTL.

In addition, an index file is generated which lists each of the RTL files of the design and is typically used as an **-F** argument for the simulator. The name of the generated index file is *<moduleName>.f* written to the *model_dir* specified in the *simulation configuration*

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

-o filename

Specifies a name for the generated wrapper file.

`-slec filename`

Specifies a name for the generated SLEC file, which must be a valid filename in a writable directory.

`[design_id]`

Specifies a design for which to generate the wrapper. The default is the current design.

E.1.120 write_verilog_wrapper

Syntax

```
write_verilog_wrapper [-h] [-dir directory] [module_id]
```

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Note The **header_files** attribute must be specified before build.

Postcondition

You may continue with the Specifying Micro-architecture Step or any subsequent step.

Action

Generates two files:

- Top-level Verilog verification wrapper (<mod_name>_ctos_wrapper.<verilog_file_ext>)
- Type wrapper, instantiated and used in the Verilog verification wrapper (<mod_name>_ctos_type_wrapper.<systemc_file_ext>)

The type wrapper provides the following functionality:

- Conversion of non-supported user-defined types (by Incisive) on port boundaries of SystemC DUT to sc-bv (SystemC bit vector) on port boundaries, and vice-versa.
- Connect meta-ports to their hierarchical counterparts in SystemC.

Command Arguments

[-h]

Provides a brief description of the command syntax and arguments.

[-dir directory]

Specifies an output directory. If the **-dir** option is not specified, CtoS automatically creates a *model* directory, and the output is written to the *model* directory.

[module_id]

Specifies a module for which to generate the wrapper. The default is the top-level module.

E.1.121 write_wrapper

Syntax

```
write_wrapper [-h] [-compare_signals] [-o filename] [module_id]
```

Precondition

The design has completed the Set up Design Step – the **build** command has been run.

Postcondition

You may continue with the Specifying Micro-architecture Step or any subsequent step.

Action

Generates a file, with the optionally specified filename, that contains the verification wrapper for the specified module (default is the top-level module).

See also [“Generating a SystemC Verification Wrapper” on page 7-11](#).

Command Arguments

`[-h]`

Provides a brief description of the command syntax and arguments.

`[-compare_signals]`

Enables the comparing of internal **sc_signals** in the module in the verification wrapper.

`[-o filename]`

Specifies the name for the generated wrapper file. The wrapper module constructor has parameters that are used to name the *model under test* and *reference model* instances. This is done indirectly by providing the suffixes of the module names via the constructor. If you do not use this option, CtoS uses the name in the **sim_wrapper_filename** design attribute; if that is not set, the default is **module_name_ctos_wrapper**. If you specify **-o filename**, and it is different from **sim_wrapper_filename**, you will get a warning.

`[module_id]`

Specifies a module for which to generate the wrapper. The default is the top-level module.

F CtoS Example Reference

CtoS has created many examples to help you understand how best to use the C-to-Silicon Compiler. The example hierarchy is intended to help locate examples more rapidly according to specific areas of application.

The following table provides a listing and a brief description of all of these examples, which can be found in the following directory:

```
install_directory/share/ctos/examples/
```

The high-level directories are as follows:

- `analysis`: Illustrates design analysis and reports.
- `coding_style`: Illustrates about the modeling styles for different application areas.
- `extensions`: Includes the AE ware providing libraries and TCL scripts extending SystemC and CtoS capabilities.
- `features`: Provides the CtoS basic feature and command examples.
- `flows`: Provides the full flow examples illustrating the integration between CtoS and other tools.
- `libraries`: Provides the examples illustrating productized library components
- `training`: Provides the documentation and examples used for CtoS training.

For additional help, links to the related sections of this user guide are provided.

Setup for Examples

Before running any example, make sure you have specified the location of your CtoS installation, in one of the following ways:

- Add **CtoS** to your **\$PATH**.
- Call **make**, specifying “**CTOS_ROOT=<path>**”
- Set **CTOS_ROOT** in your **.cshrc** file.

Running the Examples

You can simulate all of the examples by typing **ctos ctos.tcl** or **ctosgui and source ctos.tcl**, which then will create the simulation setup (**Makefile.sim**) for simulation.

Top-Level directory	Subdirectory1	Subdirectory2	Subdirectory3	Description
analysis	power			Shows how to use the CtoS <i>Low Power Design Flow</i> to design a 128- point, in-place FFT processor; uses simulation launch to automate analysis flow. See “ Scheduling for Low Power Design ” on page -16.
	scheduling			Shows how to work through scheduling messages resulting from tight constraints. See “ Scheduling ” on page -2.
coding_style	arbiter			Show how CtoS supports the modeling of systems that are configurable, control-dominated, have TLM communication, and synchronize processes using sc_signals .
	sinc_filter			This example is run through a realistic design refinement process with CtoS.

Top-Level directory	Subdirectory1	Subdirectory2	Subdirectory3	Description
extensions	examples	constrain_mem_io		Shows how to schedule master memory operations together with associated write operations.
		hier_timing_check		Shows how hierarchical timing budgeting can be performed.
		non_bus_ports		Shows how the RTL I/O interfaces for structures and arrays should be broken down into individual structure members or array elements.
	doc			The documentation is about how the features.
	include			The SystemC libraries of the extensions.
	scripts			The tcl scripts of the extensions.
features	arrays See “Resolving Arrays (Memories)” on page 8-61.	external	compose	Shows how to setup CtoS for synthesizing the whole system consisting of DUT, Producer, and Consumer.
			consumer	Shows how to model sharing of an array by two modules using the external array feature.
			full	Show how to setup CtoS for synthesizing the whole system consisting of DUT, Producer, and Consumer.

Top-Level directory	Subdirectory1	Subdirectory2	Subdirectory3	Description
			producer	Shows how to model sharing of an array by two modules using the external array feature.
		vendor_ram		Shows how to create a Vendor RAM instance and bind it to a given array. See “Allocating Vendor RAM” on page 9-10 .
		vendor_ram_bridge		Shows how to use memory bridges. See “Using Memory Bridges to Allocate Vendor RAMs” on page H-9 .
		prototype_ram		Shows how to use prototype memories. See “Allocating Prototype Memory” on page 9-7 .
		builtin_ram		Shows how to create a synthesizable RAM instance and bind it to a given array. See “Allocating Memory” on page 9-2 .
		export_memories		Shows how to use the <i>exported memory</i> feature, setting memory connectivity at the top level of a design, separating the memories from the actual design. See “Exporting Memories” on page 9-20 .

Top-Level directory	Subdirectory1	Subdirectory2	Subdirectory3	Description
		floating		Shows why floating array accesses may be needed and how to use them. See “Floating Array Accesses” on page 8-94.
		partial_write		Shows how to utilize memories capable of supporting partial write operations. A typical benefit of these memories is efficient data access for multi-field structures. See “Specifying Minimum Write Width in Vendor RAMs” on page H-18.
		merge_arrays		Shows how to merge arrays together to form a single array. A typical benefit of this operation is that a single array will be smaller than multiple individual arrays. See “Merging Arrays” on page 8-63.
		stalled_reg_mem		Shows how to use registered memories. See “Registered Memories” on page 9-21.
		memory_dep		Shows how to resolve dependencies between memory accesses to an array. See “Creating and Managing Array Dependencies” on page 11-2.

Top-Level directory	Subdirectory1	Subdirectory2	Subdirectory3	Description
		vendor_rom		Shows how CtoS supports Vendor ROMs. A simple cordic sin that utilizes a ROM for its lookup table is implemented. See “Allocating Vendor ROM” on page 9-14.
		flatten_array		Shows how to transform an array into variables representing each element of the array, that is, how to transform a memory into individual registers.
	coarse_grain_clock_gating			Shows how process level clock gating cells are modeled in SystemC and how they are replaced by RTL_IP during synthesis.
	eco			Shows how CtoS preserves design information during an ECO flow. See “Design Flow Bug Fixing (ECO)” on page 17-3.
	io	volatile_input		Shows how to declare inputs as floating and volatile.
		fixed_input		Shows how to use the floating I/O feature. See “Floating I/O Accesses” on page 8-91.

Top-Level directory	Subdirectory1	Subdirectory2	Subdirectory3	Description
		external_delay		Shows how to declare external delays on inputs and outputs. See “Specifying External Delay for Process I/O Nets” on page 11-10.
	functions See “Inlining Functions” on page 8-3.	1func_1call		Shows how to inline a function at one point in the code where it is called.
		1func_allcalls		Shows how to inline a function at all the points in the program at which it is called.
		allfuncs		Shows how to inline all functions called within a specified behavior or the entire design.
		mustfuncs		Shows how to inline functions required to be before scheduling. See “Functions that Must be Inlined before Scheduling” on page 8-4.
		pipelineing		Shows how the pipeline_function command can be used to transform large combinational functions to a multi-cycle operation.
	loops See “Resolving Loops” on page 8-16.	body_unroll		Shows how to unroll the body of a loop. See “Unrolling Loops” on page 8-17.

Top-Level directory	Subdirectory1	Subdirectory2	Subdirectory3	Description
		breaking		Shows how to break a combinational loop. See “Breaking Combinational Loops” on page 8-22.
		full_unroll		Shows how to fully unroll a loop. See “Unrolling Loops” on page 8-17.
		partial_unroll		Shows how to partially unroll a loop. See “Unrolling Loops” on page 8-17.
		pipelining		Shows the application of the pipeline command.
	rtl_ip			Shows the CtoS external IP binding feature, which lets you bind a SystemC function to an external RTL IP. See “Importing RTL IP into SystemC Designs” on page 9-41.
	schedule	speedgrade		Shows how changing the default speed grade can impact the CtoS scheduler. See “Relaxed Latency Scheduling Mode” on page -4.
		constrain_latency		Shows how changing the default speed grade can impact the CtoS scheduler. See “Relaxed Latency Scheduling Mode” on page -4.

Top-Level directory	Subdirectory1	Subdirectory2	Subdirectory3	Description
flows	fpga	altera		Described in “CtoS Tutorial for FPGA designs” on page B-1.
		xilinx		Described in “CtoS Tutorial for FPGA designs” on page B-1.
	eco	jpeg_idct	base	Described in “CtoS Tutorial for ASIC designs” on page A-1.
			version1	Described in “CtoS Tutorial for ASIC designs” on page A-1.
libraries	fixed_point	n_sqr_avg		Shows how to use the CtoS fixed-point library; see “CtoS Libraries” on page 15-1.
	flex_channels	blocking		Shows how to use Flex Channels at the input and output interface. Flex Channels can be simulated either at TLM level or signal level. The signal level can be synthesized by CtoS, which produces RTL and associated simulation models.
		hello_world		Shows the basic usage of the Flex Channel library.
		hierarchy		Shows how to use the put/get channels in a hierarchical design.

Top-Level directory	Subdirectory1	Subdirectory2	Subdirectory3	Description
		may_block		Shows the uses of Flex Channel at the input and output interface. It explains how to simulate and synthesize the example.
		multi_wait_in_cycle		Shows how to use the traits to configure the may-block initiators to call wait() when there is more than one transaction attempted in a clock cycle.
		non_blocking		Shows the uses of Flex Channel at the input and output interface. The communication semantics shown in the example is the non-blocking put and non-blocking get for a point-to-point communication.
		pipeline		Shows the design pattern for building multi-stall pipelines with put/get channels.
		traits_default		Shows how to globally define the default traits for initiators and channels.
		traits_param		Shows how to explicitly specify the traits for initiators and channels.
training	analysis			Shows a small toy example to illustrate CtoS Synthesis capabilities.

G Structure of the Verification Wrappers

This appendix provides additional detail about verification wrappers, as follows:

- [“Structure of the SystemC Verification Wrapper” on page G-1](#)
- [“Structure of the Verilog Verification Wrapper” on page G-9](#)

G.1 Structure of the SystemC Verification Wrapper

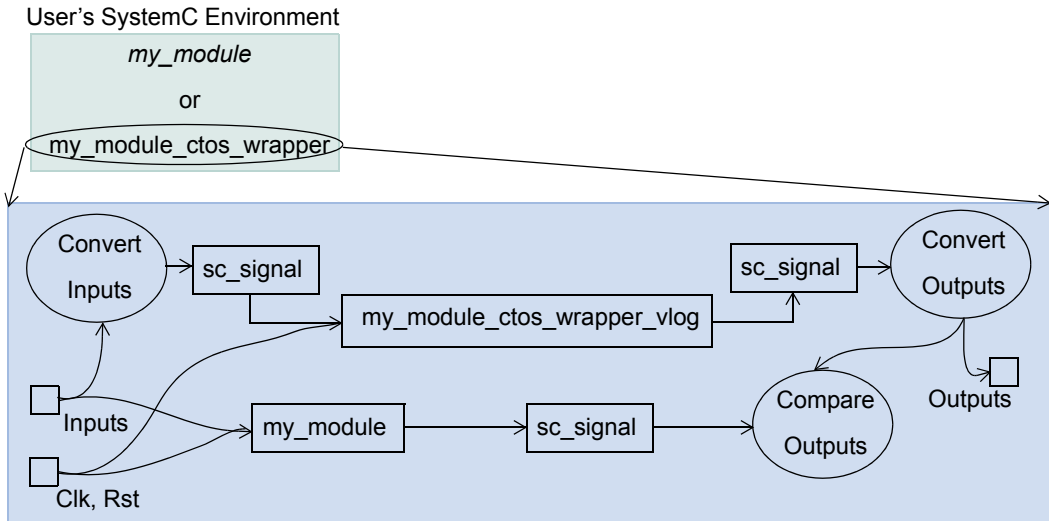
This section covers the following:

- [“Overview of SystemC Verification Wrappers” on page G-2](#)
- [“Typical Use Cases for SystemC Verification Wrappers” on page G-3](#)
- [“Wrapper Module Constructor Arguments” on page G-3](#)
- [“Other Customizable SystemC Wrapper Member Data” on page G-5](#)
- [“Converting Inputs and Outputs” on page G-6](#)
- [“Comparing Outputs” on page G-6](#)
- [“Comparing Internal Signals” on page G-6](#)
- [“Requirements for SC_MODULE Ports” on page G-7](#)
- [“Limitations When Writing Wrappers” on page G-7](#)

G.1.1 Overview of SystemC Verification Wrappers

The structure of a SystemC verification wrapper is shown in Figure G-1 on page G-2. CtoS chooses a module instance (**my_module**) in a design for synthesis; in addition to regular CtoS commands for generating the simulation and/or RTL model, a wrapper is generated for **my_module**.

Figure G-1 Structure of SystemC Verification Wrapper



The resulting wrapper contains two SystemC class definitions:

- The first is a foreign module declaration (and named **my_module_ctos_wrapper_vlog**) to instantiate the CtoS-generated Verilog simulation or RTL module (**my_module_sim** or **my_module_rtl**) and thus has the same ports as the CtoS-generated Verilog models.
- The second, referred to as the *wrapper module* (and named **my_module_ctos_wrapper**), has the same ports as the input SystemC module and therefore can replace the input SystemC module in the testbench.

The wrapper module contains one or two module instances. The first module instance is called the *model under test*, while the optional second module instance is called the *reference model* instance. The outputs of the *model under test* instance drive the outputs of the wrapper module. The outputs of the *reference model* instance do not drive the outputs of the wrapper module and are used only for comparison.

Note Details for specifying module instances are described in “[Wrapper Module Constructor Arguments](#)” on page G-3

G.1.2 Typical Use Cases for SystemC Verification Wrappers

In a typical verification wrapper use case, the *reference model* is the original SystemC module, and the *model under test* is either the simulation or the RTL model generated by CtoS, and you may be trying to answer one or both of the following questions:

- Given the same inputs, will the *model under test* and the *reference model* produce the same outputs?
- If the original SystemC module is replaced by the *model under test*, will the overall design still behave as expected?

You can get the answer to the first question by output comparison inside the wrapper.

You can get the answer to the second question by analyzing global design outputs.

The two questions are particularly distinct if CtoS output reflects transforms that change cycle-by-cycle behavior (for example breaking or pipelining a loop). In this case, the simple output comparison in the generated wrapper may not be meaningful, and you may want to manually modify it, perhaps by inserting appropriate buffers. However, the second question can still be answered if the output of CtoS is used as the *model under test*.

If you want to answer the first question, as well, you should use two configurations:

- The first configuration is the standard, in which CtoS output is the *model under test*, and the original module is the *reference model*.
- A second configuration is to use the original module as the *model under test*, and the CtoS output as the *reference model*. This exercises the design with the same I/O sequences observed in the original SystemC simulation. (Because of changes in timing, the first, or standard, configuration would exercise different I/O sequences.)

Another typical use is to compare two models generated by CtoS. For example, you may generate models before and after a particular set of transforms and then analyze the changes introduced by the transforms.

G.1.3 Wrapper Module Constructor Arguments

The wrapper module constructor takes arguments that let you specify the models to be instantiated within the wrapper and, when two models are instantiated, whether the outputs of those models are to be compared.

The second argument of the constructor specifies the suffix of the *model under test*.

This specifies indirectly the name of the Verilog or SystemC module of the *model under test*: it is the concatenation of the name of the input module, _ (an underscore), and the specified suffix.

Similarly, the third argument of the constructor specifies the suffix of the *reference model*.

The fourth argument specifies whether you want a comparison of the outputs of the *model under test* and the *reference model*.

```
testbench::testbench(const sc_module_name &name)
: sc_module(name),
  ...
  my_module_ctos_wrapper("moduleName_ctos_wrapper", /* wrapper module */
                        /* instance name */
                        "_sim", /* model under test suffix */
                        "", /* reference model suffix */
                        true) /* Compare outputs */
                        /* enable */

/* wrapper module constructor */
my_module_ctos_wrapper::my_module_ctos_wrapper(sc_module_name name,
                                                char const *ctosDutSuffix = "",
                                                char const *ctosRefSuffix = NULL,
                                                bool compare = false)
```

Table G-1 on page G-5 shows some of the most common choices for constructor arguments.

Table G-1 Common Constructor Arguments

Instantiation	Model under Test	Reference Model	Compare	Comment
<code>my_module_ctos_wrapper ("my_module_ctos_wrapper", "_sim", NULL, false)</code>	SIM	NONE	FALSE	Verilog SIM model by itself
<code>my_module_ctos_wrapper ("my_module_ctos_wrapper", "_rtl", NULL, false)</code>	RTL	NONE	FALSE	Verilog RTL model by itself
<code>my_module_ctos_wrapper ("my_module_ctos_wrapper", "_sim", "", true)</code>	SIM	ORIG	TRUE	Compare SIM model with original SystemC model
<code>my_module_ctos_wrapper ("my_module_ctos_wrapper", "_rtl", "", true)</code>	RTL	ORIG	TRUE	Compare RTL model with original SystemC model
<code>my_module_ctos_wrapper ("my_module_ctos_wrapper", "_sim", "_rtl", true)</code>	SIM	RTL	TRUE	Compare SIM model with RTL model

G.1.4 Other Customizable SystemC Wrapper Member Data

To enable additional customization, the wrapper contains a few public member data fields:

```
class sc_event m_ctosWrapperError;
bool m_enableCompare;
char const *m_origSuffix;
```

- **m_ctosWrapperError** is an **sc_event** type that flags errors that occur during output comparison.
- **m_enableCompare** is a Boolean field (which defaults to true) that lets you conditionally turn off the compare feature.
- **m_origSuffix** defines the suffix of the original SystemC model ("", that is, empty string). If you want to use a different suffix, you can change it with this field.

G.1.5 Converting Inputs and Outputs

To convert between Verilog and SystemC types, two combinational **SC_METHOD**s are defined inside the wrapper. They are **ctos_convert_inputs** and **ctos_convert_outputs**.

```
SC_METHOD(ctos_convert_inputs);
sensitive << m_pktIn;

SC_METHOD(ctos_convert_outputs);
sensitive << *m_pktOut_vlog_dut;
dont_initialize();
```

While converting Verilog wrapper outputs, the verification wrapper assigns the Verilog outputs to the verification wrapper outputs unconditionally. Any Xs in the Verilog outputs is converted based on the simulator's behavior. However, a warning is issued by the verification wrapper whenever Xs are present in the Verilog outputs.

G.1.6 Comparing Outputs

To compare the outputs of the *model under test* with the outputs of the *reference model* (if present), a clocked **SC_METHOD** is defined (only if comparison is required and a reference model is provided) and is named **ctos_compare_outputs**. In the following example, the active clock phase for the *model under test* and the *reference model* is the positive edge of the clock **m_clk** – hence the compare method is sensitive to the negative edge of the clock **m_clk**.

```
if ((m_ctosRefSuffix!=NULL)&& m_ctos_compare) {
    SC_METHOD(ctos_compare_outputs);
    sensitive << m_clk.neg();
    dont_initialize();
}
```

G.1.7 Comparing Internal Signals

A common technique to speed up diagnoses of simulation mismatches is to increase observability and to compare not only the primary outputs of both models, but also internal observation points.

Since **sc_signals** are used for modeling communication between processes, they are natural candidates for adding observation points. The **-compare_signals** option of the **write_wrapper** command supports the comparing of **sc_signals**.

Note See also “[write_wrapper](#)” on page E-176 and “[Debugging Simulation Mismatches](#)” on page I-1.

G.1.8 Requirements for SC_MODULE Ports

The types of the ports of **SC_MODULE** are subject to some requirements in order for CtoS to generate a wrapper that interfaces the Verilog models produced by CtoS to your SystemC environment.

Ports of type **sc_in<T>** or **sc_out<T>**, where **T** is a user-defined class or struct, must meet the following requirements:

- If **T** is a class, all of its data members must have public access.
- If **T** is a class, it must define a default constructor.
- **T** must define the following operators as members:

`==` (equal) and `=` (assignment)

- If the system is to be simulated with NC-SC, the following functions must be defined:

```
extern void sc_trace(sc_trace_file *tf, const T &p, std::string name);  
extern ostream& operator << (::std::ostream &os, const T &p);
```

G.1.9 Limitations When Writing Wrappers

There are a few limitations when writing wrappers:

- The wrapper is derived from **sc_module**; therefore, you cannot create a global instance of this object. You can instantiate the wrapper object only inside another **sc_module** or inside the **sc_main** function.

Note This is specified in the SystemC LRM, section 4.1.1: *Instances of class **sc_module** and class **sc_module** may only be created within a module or within function **sc_module**. Instances of class **sc_module** and class **sc_module** can only be created within a module. It shall be an error to instantiate a module or primitive channel other than within a module or within function **sc_module**, or to instantiate a port or export other than within a module.*

- All ports of the **SC_MODULE** must be members of the top-level module class.
- Ports that are members of a base class of the **SC_MODULE** class are not supported.
- Usage of an unnamed type in the port interface is not supported.
- Arrays of pointers to port are not supported.
- Ports that are fields of a struct that is a member variable of the **SC_MODULE** class are not supported,

– *continued on following page* –

Limitations when writing wrappers (continued from previous page):

- The clock and reset ports of the CtoS-generated Verilog module are connected directly to the corresponding ports of the wrapper. All ports of the CtoS-generated Verilog module are connected to **sc_signals** instantiated inside the wrapper.
 - For input ports of the Verilog module, a combinational **SC_METHOD** reads from the corresponding input ports of the wrapper, performs the type conversion, and writes the converted values to the corresponding signals.
 - For output ports of the Verilog module, a combinational **SC_METHOD** reads from the corresponding signals, performs the type conversion, and writes the converted values into the corresponding outputs ports of the wrapper.
- The need for type conversion causes the need for **sc_signals**, which results in a delay of one delta cycle on the data transmission between the ports of the wrapper and the ports of the CtoS-generated Verilog model. Clock and reset ports are connected directly so they do not suffer from this latency.
- Currently, only reset signals of type boolean are supported.
- If a cycle-by-cycle comparison between the outputs of the *model under test* and the outputs of the *reference model* is desired, the wrapper defines a unique clocked **SC_METHOD** per unique clock and reset combination. The active clock phase of this **SC_METHOD** is opposite that of the active clock phases of the *model under test* and the instances of the *reference model*.

Each of these compare methods compares the outputs of the *model under test* and the outputs of the *reference model*, performing any conversions necessary between Verilog and SystemC types, and reports mismatches if any were found. In addition, the compare methods also reports error messages if unknown values are encountered.

- Only a cycle-by-cycle comparison is supported, that is, if either the *model under test* or the *reference model* have differently timed input, the comparison fails. For example, the use of the **break_combinational_loop**, **pipeline_loop** or **constrain_latency** commands is likely to cause the *model under test* and the *reference model* to have different timing – leading to false comparisons.
- If outputs are driven by a combinational process, CtoS generates a single **SC_METHOD** sensitive to all clocks of the design. This is done to prevent the wrapper from reporting any transient mismatches. If there are no clocks in the design, a separate combinational **SC_METHOD** is defined sensitive to the output signal. The comparison is performed each time the value of the output signal changes. This can give rise to transient mismatches that you must resolve manually.

G.2 Structure of the Verilog Verification Wrapper

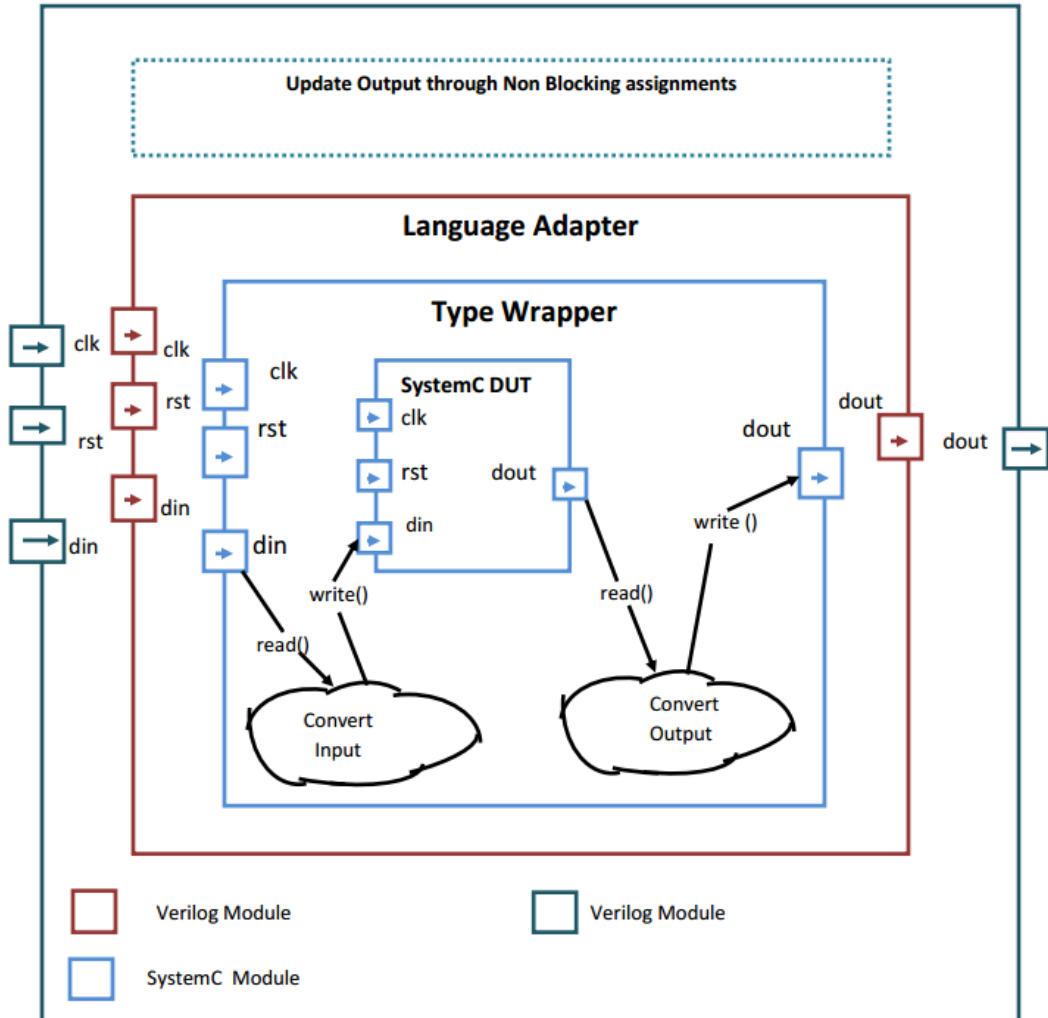
The structure of a verilog verification wrapper is as follows:

1. At the top, verilog verification wrapper is generated. See [“Top Level Verilog Verification Wrapper” on page G-11](#).
2. The verification wrapper instantiates the Verilog/SystemC Language Adaptor using Incisive "foreign" module feature to instantiate the SystemC DUT. See [“Language Adaptor” on page G-11](#).
3. The "foreign" module feature does not support user defined types at the port boundaries. If the Input SystemC module has user-defined data types, then the "foreign" module cannot instantiate the SystemC DUT directly. Instead, a SystemC Type Wrapper module wraps the SystemC DUT and is responsible for converting the user-defined data types to native SystemC types (sc_lv).
4. SystemC Type Wrapper contains all the required logic for supporting bit level connections for composite type like **sc_ufixed** and connecting the internal components of SystemC design hierarchy for flex channels. See [“Type Wrapper” on page G-11](#).

A graphical representation of the structure of a verilog verification wrapper is shown in [Figure G-2 on page G-10](#).

Figure G-2 Structure of Verilog Verification Wrapper

Verilog Verification Wrapper



G.2.1 Top Level Verilog Verification Wrapper

The verilog verification wrapper is a verilog module with I/O declarations that matches the verilog RTL model produced by CtoS. The output pins corresponding to **sc_out** and **sc_signal** of the SystemC model are updated through non-blocking assignments. It internally instantiates the SystemC top-level module (specified as the input to CtoS) using the Incisive feature of instantiating SystemC as a foreign module.

G.2.2 Language Adaptor

The Language Adaptor is a verilog module that uses the "foreign" attribute of Incisive to support instantiation of a SystemC module in a Verilog hierarchy.

In the simplest case, the Language Adaptor directly instantiates the SystemC DUT as a "foreign" module. When the ports on your SystemC DUT are not compatible with the data types supported by Incisive's "foreign" module, then the Language Adaptor instantiates a Type Wrapper (SystemC module) that supports this conversion (as described in next section).

Examples of incompatible data types include user-defined types or composite types like **sc_ufixed**.

Note The list for supported types is provided in the “*Data Types and Value Mapping*” section of the *SystemC® Simulation Reference Guide*.

G.2.3 Type Wrapper

This is a SystemC module that instantiates the SystemC DUT. It is responsible for converting user defined types on port boundaries of SystemC DUT to **sc_bv** (SystemC logic vector) on port boundary.

The connection of the following is done in Type Wrapper:

- Bit fields in Verilog Module to the corresponding types in SystemC.
- Individual pins of Verilog Module to the hierarchical ports (flex channels) of SystemC.

H IP Definition Files for Memories and RTL IP

As part of the CtoS memory and RTL IP allocation process, you must create an *IP definition file*, in XML format, for each memory or RTL IP you plan to allocate.

For the steps in the actual allocation process, see [“Allocating Memory and RTL IP” on page 9-1](#)

This appendix describes the IP definition file in detail. This file consists of:

- *for memories*: a mapping from the actual input/output ports on a memory technology cell to a standardized set of port names used by CtoS.
- *for RTL IP*: a set of specific elements describing the external RTL.

After this file has been read in, CtoS uses almost every element in the file to set a corresponding attribute in the CtoS database for the memory or RTL IP, as defined in [“Memory Definition Object Attributes \(memory_defs\)” on page D-30](#) and [“RTL IP Definition Object Attributes \(rtl_ip_defs\)” on page D-83](#). Note that, after this file has been read in, you cannot redefine the attributes for that object – they are read-only.

This appendix first describes the general IP definition file XML elements in:

- [“IP Definition \(XML\) File Description” on page H-2](#)

Then, it describes how to create these IP definition files (and other files, as needed) specifically for:

- [“Vendor RAMs” on page H-6](#)
- [“Vendor ROMs” on page H-25](#)
- [“RTL IP” on page H-28](#)

H.1 IP Definition (XML) File Description

This section lists all of the XML elements in the IP definition files for each of the different memory and IP types.

First – if you would like to see an example of an XML file, see the following figures:

- [“IP Definition File for Vendor RAM \(Wrapper Method\) Example” on page H-7](#)
- [“IP Definition File for Vendor RAM \(Memory Bridges\) Example” on page H-10](#)
- [“IP Definition File for Multiple Clock Ports \(Memory Bridges\) Example” on page H-17](#)
- [“IP definition File Showing Declaration of Auxiliary Ports” on page H-20](#)
- [“IP Definition File for Vendor ROM \(Wrapper Method\) Example” on page H-27](#)
- [“IP Definition File for RTL IP” on page H-29](#)

Secondly – in the following sections, each element in these files is presented, in hierarchical order, and linked to its corresponding object definition, when applicable:

- [“RAMdef XML Elements” on page H-3](#)
- [“ROMdef XML Elements” on page H-4](#)
- [“rtl_ip_def XML Elements” on page H-5](#)
- [“memory_bridge_def XML Elements” on page H-5](#)
- [“cgic_ip_def XML Elements” on page H-6](#)

H.1.1 RAMdef XML Elements

```

<ctos_ip_definitions>
  <RAMDef> [see Memory Definition Object Attributes (memory_defs)]
    <name>
    <wrapper_filename>
    <tech_cell_name>
    <liberty_filename>
    <verilog_filename>
    <num_words>
    <width>
    <min_write_width>
    <interface_types>
      <interface_type *
    <bridges>
      <bridge_name *
    <clock_per_port>
    <clock_port>
    <has_reset>
    <read_latency>
    <reset_async>
    <reset_high>
    <clock_posedge>
    <aux_ports> [see Memory Definition Auxiliary Port Object Attributes
(aux|port_defs)]
      <aux_port_def *
        <name>
        <is_input>
        <width>
        <value>
      <xilinx_search_dir>
      <reset_port>

```

Notes

The asterisk (*) beside *interface_type*, *bridge_name*, and `<aux_port_def>` means you can, and may actually need to, have more than one of these entries. Here is more detail about these three:

- ***interface_type***: You must have one line for each interface of the memory, *in the order that matches the Verilog module declaration in the wrapper file*. You can see the valid types in “[Access Protocols \(interface_types\) in Vendor RAMs](#)” on page H-21. For example, for a memory with two interfaces:

```

<interface_types>
  <elem>sync_read_write</elem>
  <elem>async_read</elem>
</interface_types>

```

- **bridge_name**: Similarly, if you are using memory bridges, you must have one line for each interface that specifies the name of a bridge definition to be used for that interface, in the same order – for example, for two memory bridges for the previous two interface types:

```
<bridges>
  <elem>bridge_srw</elem>
  <elem>bridge_ar</elem>
</bridges>
```

For more information, see [“Using Memory Bridges to Allocate Vendor RAMs”](#) on page H-9.

- **<aux_port_def>**: You simply specify one definition for each auxiliary port. Note that the order is not important, since you are also specifying the names – for example, in [Figure H-7 on page H-20](#), the **<aux_port_def>** entries could be in any order

For more information, see [“Specifying Auxiliary Ports in Vendor RAMs”](#) on page H-19.

H.1.2 ROMdef XML Elements

```
<ctos_ip_definitions>
  <ROMDef> [see Memory Definition Object Attributes \(memory\_defs\)]
    |— <name>
    |— <wrapper_filename>
    |— <tech_cell_name>
    |— <liberty_filename>
    |— <verilog_filename>
    |— <num_words>
    |— <width>
    |— <bridges>
    |   |— bridge_name *
    |— <clock_port>1
    |— <clock_posedge>
    |— <format>
    |— <aux_ports> [see Memory Definition Auxiliary Port Object Attributes \(aux\_port\_defs\)]
    |   |— <aux_port_def> *
    |       |— <name>
    |       |— <is_input>
    |       |— <width>
    |       |— <value>
```

Note The asterisk (*) beside *bridge_name* and **<aux_port_def>** means you can, and may actually need to, have more than one of these entries. For more detail, see the *Notes* in the previous section, [“RAMdef XML Elements”](#) on page H-3.

H.1.3 rtl_ip_def XML Elements

```
<ctos_ip_definitions>
  <rtl_ip_def> [see RTL IP Definition Object Attributes (rtl_ip_defs)]
    <name>
    <rtl_filename>
    <rtl_language>
    <rtl_type>
    <pipeline_depth>
    <ports> [see RTL IP Definition Port Object Attributes [(rtl_ip_def_)ports]]
      <clock_port>
        <name>
        <active_edge>
      <reset_port>
        <name>
        <type>
        <active_level>
      <valid_in_port>
        <name>
        <active_level>
      <valid_out_port>
        <name>
      <stall_port>
        <name>
        <active_level>
      <input_port>
        <name>
        <width>
      <output_port>
        <name>
        <width>
```

H.1.4 memory_bridge_def XML Elements

```
<ctos_ip_definitions>
  <memory_bridge_def> [see Memory Bridge Definition Object Attributes (memory_bridge_defs)]
    <name>
    <interface_type>
    <bridge_ports> * [see Memory Bridge Port Def Object Attributes (memory_bridge_port_defs)]
      <name>
      <is_mem_port>
      <connected_to>
```

Note The asterisk (*) beside *bridge_ports* means you can, and may actually need to, have more than one of these entries. For examples of this, see [Figure H-4 on page H-10](#) and [Figure H-6 on page H-17](#).

H.1.5 cgic_ip_def XML Elements

```
<ctos_ip_definitions>
  <cgic_ip_def>                                [see RTL IP Definition Object Attributes (rtl_ip_defs)]
    <name>
    <rtl_language>
    <rtl_filename>
    <ports>                                     [see RTL IP Definition Port Object Attributes [(rtl_ip_def_)ports]]
      <clock_port>
        <name>
        <active_edge>
      <enable_port>
        <name>
        <active_level>
      <gated_clock_port>
        <name>
      <aux_input_port>
        <name>
        <width>
```

Restrictions:

- The value of the **rtl_language** element must be either **verilog** or **liberty**.
- If the CGIC is a cell of the technology library, the name of the cell is given in the **name** element, the value of the **rtl_language** element must be **liberty**, and the **rtl_filename** must not be specified.
- If the CGIC is provided in a verilog file, the name of the module in the verilog file is given in the **name** element, the value of the **rtl_language** element must be **verilog**, and the name of the verilog file is given in the **rtl_filename** element.
- The ports **clock_port**, **enable_port**, and **gated_clock_port** are mandatory.

H.2 Vendor RAMs

In this section, the IP definition file for Vendor RAM is described in detail.

First – CtoS has two methods for allocating Vendor RAM, and each has different IP definition elements:

- “Using Wrapper Files to Allocate Vendor RAMs” on page H-7
- “Using Memory Bridges to Allocate Vendor RAMs” on page H-9

Secondly – the following sections describe more about the IP definition file and apply to either method:

- “Specifying Minimum Write Width in Vendor RAMs” on page H-18

- “Specifying Reset Behavior in Vendor RAMs” on page H-18
- “Specifying Negative Edge Clock in Vendor RAMs” on page H-18
- “Specifying Vendor RAM Library File/Simulation Model” on page H-19
- “Specifying Auxiliary Ports in Vendor RAMs” on page H-19
- “Using Vendor RAMs in FPGA Designs” on page H-20
- “Access Protocols (interface_types) in Vendor RAMs” on page H-21
- “Verilog Interface Per-Port Terminals for Vendor RAM” on page H-22
- “How CtoS Binds Processes to Vendor RAMs” on page H-23
- “Limitations Reported during Scheduling and Model Generation” on page H-24

H.2.1 Using Wrapper Files to Allocate Vendor RAMs

To use the *wrapper file* method for allocating Vendor RAM, you create two files for each Vendor RAM:

- the IP definition file, as shown in [Figure H-1 on page H-7](#)
- the RAM Verilog wrapper file, as shown in [Figure H-2 on page H-8](#)

The RAM Verilog wrapper file must meet the following requirements:

- It must be synthesizable Verilog, supported by the required implementation tool [that is, Encounter RTL Compiler (RC), etc.]
- It must conform to SystemVerilog P1800.
- It must contain a module wrapping the actual memory instance whose purpose is to map ports from the vendor memory to the CtoS expected ports.

To connect these two files, note the following correspondence:

- You must include the **<wrapper_filename>**, identifying the RAM Verilog wrapper file, in the IP definition file.
- The **module** name in this RAM Verilog wrapper file must match the **<name>** in the IP definition file.

Note The elements of the IP definition file are described in “[RAMdef XML Elements](#)” on page H-3 and are stored in the CtoS database as “[Memory Definition Object Attributes \(memory_defs\)](#)” on page D-30.

Figure H-1 IP Definition File for Vendor RAM (Wrapper Method) Example

```
<?xml version="1.0"?>
<ctos_ip_definitions>
  <RAMDef>
    <name>wrap_sram_2048x8_2rw</name>
    <wrapper_filename>wrap_sram_2048x8_2rw.v</wrapper_filename>
    <liberty_filename>sram_2048x8_2rw.lib</liberty_filename>
    <verilog_filename>sram_2048x8_2rw.v</verilog_filename>
    <num_words>2048</num_words>
```

```
<width>8</width>
<min_write_width>0</min_write_width>
<interface_types>
  <elem>sync_read_write</elem>
</interface_types>
<has_reset>>false</has_reset>
<reset_async>>false</reset_async>
<reset_high>>true</reset_high>
<clock_posedge>>true</clock_posedge>
</RAMDef>
</ctos_ip_definitions>
```

Figure H-2 RAM Verilog Wrapper File Example

// This is the CtoS wrapper for a 2K x 8bit 2 RW-port synchronous SRAM.

```
module wrap_sram_2048x8_2rw(CLK,A0,A1,D0,D1,Q0,Q1,WE0,WE1,CE0,CE1,TESTMODE,POWERSAVE);
```

```
  input          CLK;
  input  [AW:0]  A0;
  input  [AW:0]  A1;
  input  [DW:0]  D0;
  input  [DW:0]  D1;
  output [DW:0]  Q0;
  output [DW:0]  Q1;
  input          WE0;
  input          WE1;
  input          TESTMODE;
  input          POWERSAVE;

  sram_2048x8_2rw ram( .QA(Q0),
                     .CLKA(CLK),
                     .CENA(CE0),
                     .WENA(WE0),
                     .AA(A0),
                     .DA(D0),
                     .QB(Q1),
                     .CLKB(CLK),
                     .CENB(CE1),
                     .WENB(WE1),
                     .AB(A1),
                     .DB(D1),
                     .TEST(TESTMODE),
                     .PWRSV(POWERSAVE));

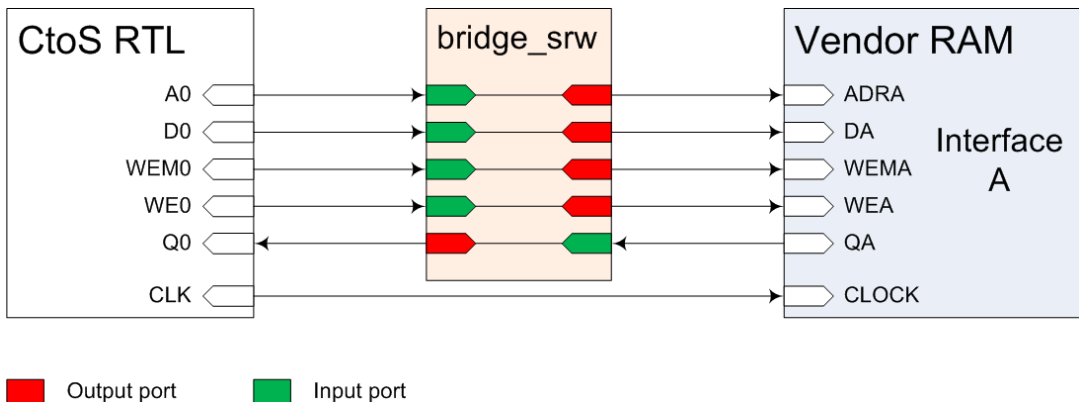
endmodule
```


H.2.2 Using Memory Bridges to Allocate Vendor RAMs

Important The new "memory bridge" method for allocating Vendor RAMs has been designed to deal with individual memory ports, as compared with the module-based memory definition provided by the existing "wrapper file" method. The "memory bridge" method is recommended only when a design requires multi-clocked memory ports. Although, this method has been successful for some designs, this preliminary feature can be problematic if a Vendor RAM exhibits complex auxiliary port settings.

Memory bridges are building blocks that connect CtoS RTL to a memory technology cell, one logical memory port at a time, as shown in [Figure H-3 on page H-9](#).

Figure H-3 Block Diagram of Memory Bridge Connecting RTL to Memory Technology Cell



Memory bridges are designed to be reusable and bit-width neutral. Each memory bridge corresponds to a particular access protocol (as listed in [Table H-3 on page H-21](#)) and implements only one protocol per bridge definition.

Since memory technology vendors often use the same port naming scheme, a memory bridge can be defined once for a particular vendor and reused many times, as long as both the access protocol and the technology cell naming are the same.

Additionally, CtoS supports logical expressions in the mapping definition, allowing assignments to constants or *use bitwise* operations in Verilog.

The following sections describe how to use the memory bridge method in detail:

- ["IP Definition File for Memory Bridges" on page H-10](#)
- ["Wildcards with Memory Bridge Definitions" on page H-14](#)

- “Memory Bridges for Vendor RAMs with Multiple Clock Ports” on page H-16

H.2.2.1 IP Definition File for Memory Bridges

In this section, an IP definition file that uses the memory bridge method, shown in [Figure H-4 on page H-10](#), is described in detail.

Figure H-4 IP Definition File for Vendor RAM (Memory Bridges) Example

```
<?xml version="1.0"?>
<ctos_ip_definitions>
  <RAMDef>
    <name>vendor_512x8_1srw</name>
    <liberty_filename>vendor512x8cell.lib</liberty_filename>
    <tech_cell_name>vendor512x8cell</tech_cell_name>
    <num_words>512</num_words>
    <width>8</width>
    <min_write_width>1</min_write_width>
    <interface_types>
      <elem>sync_read_write</elem>
    </interface_types>
    <bridges>
      <elem>bridge_srw</elem>
    </bridges>
    <clock_per_port>false</clock_per_port>
    <clock_port>CLOCK</clock_port>
    <clock_posedge>true</clock_posedge>
    <has_reset>false</has_reset>
  </RAMDef>
  <memory_bridge_def>
    <name>bridge_srw</name>
    <interface_type>sync_read_write</interface_type>
    <bridge_ports>
      <bridge_port>
        <name>ADR%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>A%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>D%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>D%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>WEM%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>WEM%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>WE%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>WE%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>ME%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>CE%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>Q%0</name>
        <is_mem_port>false</is_mem_port>
      </bridge_port>
    </bridge_ports>
  </memory_bridge_def>
</ctos_ip_definitions>
```

```
        <connected_to>Q%A</connected_to>
    </bridge_port>
</bridge_ports>
</memory_bridge_def>
</ctos_ip_definitions>
```

The IP definition file for the memory bridge method is described in the following subsections:

- “RAM Definition (<RAMDef>) for Memory Bridges” on page H-11
- “Technology Cell Name Definition (<tech_cell_name>) for Memory Bridges” on page H-12
- “Bridges Definition (<bridges>) for Memory Bridges” on page H-12
- “Memory Bridge Definition (<memory_bridge_def>)” on page H-12
- “Output Port Definition (<bridge_port>) for Memory Bridges” on page H-12

RAM Definition (<RAMDef>) for Memory Bridges

In the IP definition file for the memory bridge method (as shown in [Figure H-4 on page H-10](#)), the high-level properties of the RAM are specified in the <RAMDef>, while the individual connections are specified in the <memory_bridge_def> [described in the following section, “Memory Bridge Definition (<memory_bridge_def>)” on page H-12].

In this example, the <RAMDef> describes a 512x8 bit RAM with a single read/write port, as indicated by the <interface_types> of **sync_read_write** [see “Access Protocols (interface_types) in Vendor RAMs” on page H-21]. Inclusion of the <min_write_width> element tells you that writing each word in the RAM is bit-modifiable using the **WEM** port (see “Specifying Minimum Write Width in Vendor RAMs” on page H-18).

This RAM will be accessed by only one process in the design, so the <clock_per_port> element is set to *false*, and there is only one clock, which is defined using the <clock_port> element (for multiple clock ports, see “Memory Bridges for Vendor RAMs with Multiple Clock Ports” on page H-16).

When using memory bridges, CtoS must be able to uniquely identify the clock and reset ports on the memory technology cell. These ports are not mapped using memory bridges, but must be specified using the <clock_port> element and the <reset_port> element, respectively, and must not contain wildcards (use of wildcards is described in “Wildcards with Memory Bridge Definitions” on page H-14).

Since there is only one protocol port, only one <bridges> element is specified, named **bridge_srw**, when connecting this RAM [described in the following section, “Bridges Definition (<bridges>) for Memory Bridges” on page H-12].

Note The <RAMdef> specification is stored in the CtoS database as “Memory Definition Object Attributes (memory_defs)” on page D-30.

Technology Cell Name Definition (<tech_cell_name>) for Memory Bridges

The <tech_cell_name> element is the name of the memory technology cell being used to implement the memory – it must match the name of the <liberty_filename> (the .lib file).

Note The <tech_cell_name> element is stored in the CtoS database as “tech_cell_name” on page D-32.

Bridges Definition (<bridges>) for Memory Bridges

The <bridges> element lists all of the memory bridge names that will be used to connect each logical interface on the memory, one at a time. In the example shown in [Figure H-4 on page H-10](#), there is only one element, named **bridge_srw**.

If there is more than one element, the order of declaration in the IP definition file *must* match the order of declaration of each interface type (for example, a memory bridge defined for interface protocol **sync_read_write** must have the same position as the entry in the <interface_type> list).

Note The <bridges> element is stored in the CtoS database as “bridges” on page D-30.

Memory Bridge Definition (<memory_bridge_def>)

The <memory_bridge_def> specification in the IP definition file defines the connections from ports on the memory technology cell to a standardized set of port names used by CtoS.

For example, in [Figure H-4 on page H-10](#), the <memory_bridge_def> specifies connections for a port protocol of type **sync_read_write**, which is the only protocol used by this RAM.

After the <name> of the memory bridge and the <interface_type> it implements are defined, the remainder of the definition consists of a series of output port definitions – the <bridge_ports>. Each individual definition begins with <bridge_port> [which is described in “[Output Port Definition \(<bridge_port>\) for Memory Bridges](#)” on page H-12].

Note The <memory_bridge_def> specification is stored in the CtoS database as “Memory Bridge Definition Object Attributes (memory_bridge_defs)” on page D-29.

Output Port Definition (<bridge_port>) for Memory Bridges

Each of the output port definitions (<bridge_port>) specifies which port on the CtoS standard RTL connects to which port on the Vendor RAM.

These definitions are written from the perspective of the bridge, not the RAM. This means the output ports of the bridge are named (<name>), and they connect to (<connected_to>) inputs of either the CtoS RTL or the RAM.

Visualizing the connections (see [Figure H-3 on page H-9](#)) makes it easier to understand the required elements for each port belonging to this interface protocol. The output ports on the bridge instance are shown in **red**; input ports are shown in **green**. Each output port is fed by a corresponding output from either the CtoS RTL or the RAM. Since the RAM uses an alphabetic port suffix, all outputs driving the memory cell use the %A wildcard to match the memory suffix.

In this example (and in the corresponding IP definition file shown in [Figure H-4 on page H-10](#)), one output port definition is specified for each of the ports used by the **sync_read_write** protocol.

All CtoS per-port terminals (see [Table H-4 on page H-22](#)) must be meaningfully connected – since this interface protocol is for both reads and writes – and if any are not specifically defined, you will get an error.

The specific elements of each **<bridge_port>** include the following:

- The **<name>** must match the port name used by the load side of the output port. If **<is_mem_port>** is *true*, the **<name>** must match the input port on the memory. If **<is_mem_port>** is *false*, the **<name>** must match the CtoS standard port name.

For a proper match, this may require the use of a *wildcard* (see “[Wildcards with Memory Bridge Definitions](#)” on page H-14) since all CtoS standard ports use a numeric suffix equivalent to **%0** in a bridge definition. However, if the memory cell is the load, any naming is acceptable (and wildcards are not strictly required).

Naming mismatches will cause problems either during timing analysis (part of the scheduling step) or during simulation. It is recommended that you ensure all memory and bridge definitions are correct before the scheduling step in the design flow.

- The **<is_mem_port>** element simply indicates whether the port is a CtoS standard port name.

In this case, all ports, except **Q**, are outputs to the memory cell, so **<is_mem_port>** is set to *true*. **Q** is used for read-class protocols and goes to the CtoS RTL; therefore the **<is_mem_port>** for **Q** is set to *false*.

- The **<connected_to>** element specifies the assignment of the port. Any Verilog-legal expression can be used. Constants, including multi-bit constants (which must use one of the Verilog formats in [Table H-1 on page H-14](#)) and bitwise operations are allowed. CtoS will simply take this expression as a whole and use it when RTL is generated.

In the example in [Figure H-4 on page H-10](#), no constants or expressions are needed, since the memory ports closely match the CtoS standard port names. This will be a common case when connecting memories.

Any ports referenced in the **<connected_to>** element refer to the *driver* side of the connection. If **<is_mem_port>** is *true*, the driver side will be the CtoS RTL.

If **<is_mem_port>** is *false*, the driver side will be the memory technology cell. When the driving ports refer to CtoS standard ports, you must use proper naming to specify the correct port, that is, you must use CtoS per-port terminal names (see [Table H-4 on page H-22](#)), followed by the standard numeric suffix wildcard **%0**.

Any other ports are unknown to CtoS and will cause an error.

When the driving side is the memory technology cell, any legal port name can be used, with the understanding that it must match a port on your memory technology cell. Another way to think of the relationship between **<name>** and **<connected_to>** is that of an assignment statement in Verilog. The **<name>** becomes the LHS, and the value of **<connected_to>** is the RHS. The Verilog generated by CtoS closely follows this convention (although this is not precisely guaranteed).

Note The **<bridge_port>** specification is stored in the CtoS database as “Memory Bridge Port Def Object Attributes (memory_bridge_port_defs)” on page D-29.

Table H-1 Verilog Formats for Multi-Bit Constants

Radix	Format	Example
binary	N' bxxx	8'b00010001
decimal	N' dxxx	32'd43
hex	N' hxxx	32'h00008af9

H.2.2.2 Wildcards with Memory Bridge Definitions

Since memory bridges are not specific to a particular numbered interface on a memory technology cell, CtoS provides a way to specify port names using *wildcards*.

This ensures that the bridge can be re-used across interfaces with different suffix identifiers.

For example, if a dual-port memory has logical ports **A** and **B**, and if both ports implement the same interface protocol (**sync_read_write**), then the same bridge definition can be used for each port.

To abstract port naming, CtoS uses a *wildcard substitution* scheme similar to that used by the Unix **date** command. For each wildcard specified, CtoS substitutes a character representing the index of the memory interface.

Table H-2 on page H-15 shows the wildcards supported by CtoS, and the way they are interpreted.

Wildcards may be used in both the **<name>** and the **<connected_to>** elements; however, the use of wildcards is completely optional. Identifiers may use any number of wildcards. If a wildcard is not used in an expression, the literal name is used directly.

Note The **<clock_port>** element and the **<reset_port>** element must not contain wildcards.

Table H-2 CtoS-Supported Wildcards

Wildcard	Interpreted as	Example
%A	capital alphabetic suffix	CLK%A would become CLKA, CLKB, CLKC ...
%a	lowercase alphabetic suffix	WE%a would become WEa, WEb, WEc ...
%1	numeric suffix, 1-based	A%1 would become A1, A2, A3 ...
%0	numeric suffix, 0-based	Q%0 would become Q0, Q1, Q2 ...

H.2.2.3 Memory Bridges for Vendor RAMs with Multiple Clock Ports

CtoS also supports memories with separate clock ports per logical interface. This feature can be used in cases where multiple **SC_CTHREAD** processes access a single array, but are sensitive to different clocks. If a memory cell has a different clock port for each interface, CtoS can use this structure during allocation to bind processes with different clock sensitivities.

Unlike the single clock case described previously, specifying one clock per interface requires the addition of a clock port mapping in the **<memory_bridge_def>** element, as well as the following:

- You must have more than one process access an array, using different clock sensitivities.
- You must set the **<clock_per_port>** element to *true*.
- You must define a mapping to the CtoS port name **CLK%0** for each memory that uses **<clock_per_port>**.

A block diagram and an example of this feature are shown in [Figure H-5 on page H-16](#) and [Figure H-6 on page H-17](#), respectively.

Figure H-5 Block Diagram of Multiple Clock Ports in Memory Bridges

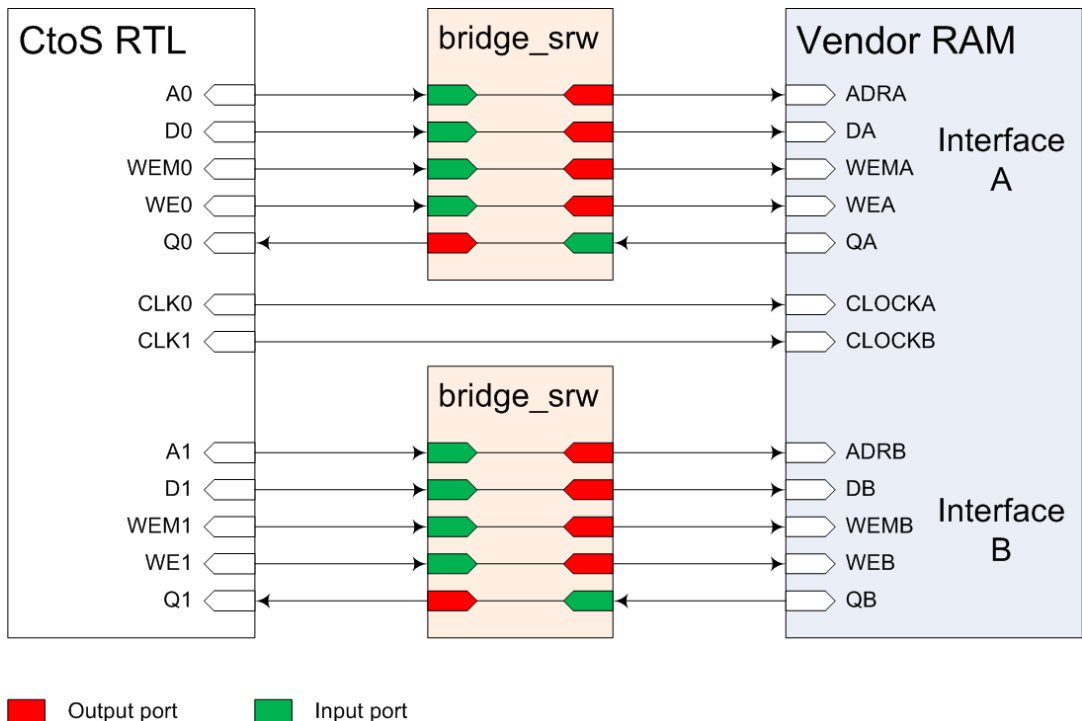


Figure H-6 IP Definition File for Multiple Clock Ports (Memory Bridges) Example

```
<?xml version="1.0"?>
<ctos_ip_definitions>
  <RAMDef>
    <name>vendor_512x8_1srw</name>
    <liberty_filename>vendor512x8cell.lib</liberty_filename>
    <tech_cell_name>vendor512x8cell</tech_cell_name>
    <num_words>512</num_words>
    <width>8</width>
    <min_write_width>1</min_write_width>
    <interface_types>
      <elem>sync_read_write</elem>
    </interface_types>
    <bridges>
      <elem>bridge_srw</elem>
    </bridges>
    <clock_per_port>true</clock_per_port>
    <clock_posedge>true</clock_posedge>
    <has_reset>>false</has_reset>
  </RAMDef>
  <memory_bridge_def>
    <name>bridge_srw</name>
    <interface_type>sync_read_write</interface_type>
    <bridge_ports>
      <bridge_port>
        <name>CLOCK%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>CLK%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>ADR%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>A%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>D%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>D%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>WEM%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>WEM%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>WE%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>WE%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>ME%A</name>
        <is_mem_port>true</is_mem_port>
        <connected_to>CE%0</connected_to>
      </bridge_port>
      <bridge_port>
        <name>Q%0</name>
        <is_mem_port>>false</is_mem_port>
        <connected_to>Q%A</connected_to>
      </bridge_port>
    </bridge_ports>
  </memory_bridge_def>
</ctos_ip_definitions>
```

H.2.3 Specifying Minimum Write Width in Vendor RAMs

For an example of this feature, look in the following directory:

```
install_directory/share/ctos/examples/features/arrays/partial_write
```

Note Before running any CtoS examples, first review “[Setup for Examples](#)” on page F-2.

The **<min_write_width>** element in a RAM definition represents the smallest number of bits that may be written into the RAM cell.

This is determined by the presence or availability of *write mask* bits on the RAM cell itself, which are determined by the RAM vendor.

If you specify a minimum write width value, CtoS will optimize writes more efficiently in your design.

This optimization is known as a *partial write* and, in general, is desirable because it lowers power consumption of the RAM and can alleviate scheduling pressure of RAM reads and writes.

Notes

- The data **<width>** must be a multiple of **<min_write_width>**.
- *Partial writes is a preliminary feature.*

H.2.4 Specifying Reset Behavior in Vendor RAMs

CtoS supports Vendor RAM cells with reset behavior.

This is relatively uncommon among Vendor RAMs, but can be desirable for small RAM cells or FPGA implementations.

CtoS assumes that, during reset, all RAM bits across the entire array will be reset to the value 0.

The three XML elements, **<has_reset>**, **<reset_async>**, and **<reset_high>**, specify the behavior of RAM cells regarding reset.

Note *Support for FPGA designs is a preliminary feature.*

H.2.5 Specifying Negative Edge Clock in Vendor RAMs

Vendor RAMs can be specified as sensitive to a negative edge clock.

You must ensure that the **<clock_posedge>** element matches the clock edges of processes in your design.

H.2.6 Specifying Vendor RAM Library File/Simulation Model

To schedule with the **Timing Analyzer** set to **simple** or **accurate**., you must include the *Vendor RAM library file* (specified with the `<liberty_filename>` element), and its corresponding *simulation model*.

The library file, which must be in Liberty format, is a technology-specific model that contains timing, power, and area information about the RAM cell. During timing analysis, CtoS uses this file as the implementation of the RAM cell. If this file is missing or flawed, you will get an error.

The library file should match pin-for-pin the Verilog simulation model, which is a behavioral Verilog file used to simulate the behavior of the RAM.

If you do supply these files, both must be matched to a specific RAM cell provided by your foundry or third-party IP vendor.

Note You can generate RTL without a simulation model. This is useful when an appropriate RAM cell is not yet available, but you would like a preliminary scheduling result.

H.2.7 Specifying Auxiliary Ports in Vendor RAMs

CtoS supports the instantiation of memories with auxiliary ports unrelated to the read or write access protocols on the memory. Common examples include power down, timing margin select, and scan and test pins. These ports are not connected to CtoS-generated RTL.

You declare these ports in the IP definition file, using the `<aux_port_def>` element, as shown in [Figure H-7 on page H-20](#), so the memory can be properly instantiated in the RTL.

Each port definition requires three elements, and you can optionally specify a fourth:

- a `<name>` (required)
- whether it `<is_input>` (required)
- the bit `<width>` (required)
- an initial `<value>` (optional), which must follow the format in [Table H-1 on page H-14](#).

If you do not specify the initial value of an auxiliary port using the `<value>` element, auxiliary *input* ports are grounded by a `'b0` statement in Verilog. You can either edit the RTL to connect them to something else or use the **connect** command (["connect" on page E-34](#)) to connect each terminal to another internal net.

Auxiliary *output* ports are not connected to anything. They appear in the port declaration, but as no-connects. You can either edit the RTL to connect them to something else or use the **connect** command to connect each terminal to another internal net.

Notes and Limitations

- Auxiliary memory port names *cannot* consist of these keywords, suffixed with zero or more digits:
CLK, RST, RST_N, A, D, Q, CE, WE, WEM

- The `<aux_port_def>` specification is stored in the CtoS database as “Memory Definition Auxiliary Port Object Attributes (aux_port_defs)” on page D-33.

Figure H-7 IP definition File Showing Declaration of Auxiliary Ports

```
<RAMDef>
  <name>vendor_512x8_1srw</name>
  <liberty_filename>vendor512x8cell.lib</liberty_filename>
  <tech_cell_name>vendor512x8cell</tech_cell_name>
  <num_words>512</num_words>
  <width>8</width>
  <min_write_width>1</min_write_width>
  <interface_types>
    <elem>sync_read_write</elem>
  </interface_types>
  <bridges>
    <elem>bridge_srw</elem>
  </bridges>
  <clock_per_port>true</clock_per_port>
  <clock_posedge>true</clock_posedge>
  <has_reset>false</has_reset>
  <aux_ports>
    <aux_port_def>
      <name>PD</name>
      <is_input>true</is_input>
      <width>1</width>
    </aux_port_def>
    <aux_port_def>
      <name>ALP</name>
      <is_input>true</is_input>
      <width>3</width>
    </aux_port_def>
    <aux_port_def>
      <name>AWT</name>
      <is_input>true</is_input>
      <width>2</width>
    </aux_port_def>
    <aux_port_def>
      <name>TEST_TIMEOUT</name>
      <is_input>true</is_input>
      <width>1</width>
    </aux_port_def>
    <aux_port_def>
      <name>TEST_WB</name>
      <is_input>false</is_input>
      <width>1</width>
    </aux_port_def>
  </aux_ports>
</RAMDef>
```

H.2.8 Using Vendor RAMs in FPGA Designs

Note *Support for FPGA designs is a preliminary feature.*

When using Vendor RAMs in FPGA designs, there are a few differences from ASIC designs:

- The `<verilog_filename>` element identifies the Verilog file that contains the FPGA memory model. It is used by the FPGA synthesis tool to synthesize the memory.
- The `<liberty_filename>` element is not used.

- For Xilinx, you can use the `<xilinx_search_dir>` element to specify the directory containing the built-in memory model. You may need to add the following line before a reference to a built-in model:

```
// synthesis translate_off
```

H.2.9 Access Protocols (interface_types) in Vendor RAMs

For each port specified in the `<interface_types>` element of the `<RAMDef>` specification, a corresponding set of signals is expected to exist in the Verilog ports of the Vendor RAM. CtoS has four access protocols (`interface_types`) available, as defined in [Table H-3 on page H-21](#).

Table H-3 Four Access Protocols (interface_types) Supported by CtoS

Access Protocol (interface_type)	Description
async_read	asynchronous read: <pre>if (CEi) Qi <= M[Ai];</pre>
sync_read	synchronous read: <pre>@(posedge CLK) if (CEi) Qi <= M[Ai];</pre>
sync_write	synchronous write: <pre>@(posedge CLK) if (CEi & WEi) M[Ai] <= Di;</pre>
sync_read_write	synchronous read/write: <pre>@(posedge CLK); if (CEi & WEi) M[Ai] <= Di; if (CEi & ~WEi) Qi <= M[Ai];¹</pre>

Where, ¹ indicates the read_latency of 1 (the default). If read_latency is greater than 1, then the assignment will be delayed additional cycle(s).

These four access protocols apply to a particular port on the RAM. You may have a different access protocol for every distinct port on the RAM, in any combination and in any number. However, at least one port must support a write operation, and at least one port must support a read operation. These may be combined – the simplest example is a RAM with a single port of type `sync_read_write`.

Note If you are using the wrapper method (“[Using Wrapper Files to Allocate Vendor RAMs](#)” on page H-7), CtoS specifies the naming of these Verilog ports, and every Vendor RAM will have a user-crafted Verilog wrapper to connect the CtoS ports to the actual ports on the RAM instance. This Verilog wrapper must exist in order to simulate after allocating memory.

Example

To specify two ports:

```
<interface_types>
  <elem>sync_write</elem>
  <elem>sync_read</elem>
</interface_types>
```

To specify a single port:

```
<interface_types>
  <elem>sync_read_write</elem>
</interface_types>
```

H.2.10 Verilog Interface Per-Port Terminals for Vendor RAM

A single interface on a Vendor RAM is defined as a collection of inputs, and possibly outputs. All interfaces use consistent naming within CtoS. [Table H-4 on page H-22](#) shows the signals composing one port in CtoS. Multiple ports use the same naming prefix, but with a numerical suffix corresponding to the port number, which begins with 0.

Note that single-bit inputs and outputs of the Verilog wrapper must be declared as scalars, such as:

```
input clk;           // correct
```

rather than as single-bit vectors:

```
input [0] clk;       // incorrect
input [0:0] we0;     // incorrect
```

Table H-4 CtoS Per-Port Terminals for Vendor RAM Verilog Interface

Name	Type	Width	Existence Condition	Description	Notes
CLK	input	1	always	clock, active on posedge	shared by all ports
RST	input	1	has_reset = true, reset_high = true	reset, active high, active during CtoS reset.	shared by all ports
RST_N	input	1	has_reset = true, reset_high = false	reset, active low, active during CtoS reset.	shared by all ports
Ai	input	AW	always	address	
CEi	input	1	always	chip enable, active high	
WEi	input	1	sync_read_write sync_write	write enable, active high	
Di	input	DW	sync_read_write sync_write	data in	

Table H-4 CtoS Per-Port Terminals for Vendor RAM Verilog Interface

Name	Type	Width	Existence Condition	Description	Notes
Qi	output	DW	async_read sync_read sync_read_write	data out	
WEMi	input	$DW \div MWW$	sync_read_write sync_write AND when $\text{min\_write\_width} > 0$	write enable mask bus, specifies partial writes	
note:	AW is the address width DW is the data width MWW is the minimum write width i is the index number of the interface, starting at 0				

H.2.11 How CtoS Binds Processes to Vendor RAMs

When allocating memory (see also [“Allocating Vendor RAM” on page 9-10](#)), CtoS binds processes to Vendor RAMs in the following manner:

- “Basic “Best-Case Matching” Algorithm” on page H-23
- “Handling of Residual Ports” on page H-24
- “Handling of Synchronous Vs. Asynchronous Ports” on page H-24
- “User-Specified Binding Not Supported” on page H-24
- “Multiple Processes Assigned Separate Ports” on page H-24

H.2.11.1 Basic “Best-Case Matching” Algorithm

Each process accessing a RAM infers a certain type of access protocol, based on whether it reads, writes, or reads and writes the RAM.

CtoS use a *best-case matching* algorithm to bind each process to the RAM using the available ports described in the RAM definition file.

For example, if a single process reads and writes a RAM, and an unbound **sync_read_write** port exists on the RAM, CtoS will bind that port first; however, if there is no **sync_read_write** port, CtoS will bind one read port and one write port to this process.

H.2.11.2 Handling of Residual Ports

If extra *read* ports remain, they are bound in round-robin fashion to processes requiring read access. If extra *write* ports remain, CtoS ties them to an appropriate constant value (for example, write enable pins are grounded).

H.2.11.3 Handling of Synchronous Vs. Asynchronous Ports

If both a *synchronous read* port and an *asynchronous read* port exist on a RAM, CtoS does not distinguish between the two when automatically binding ports to processes. These read ports are bound in an unspecified order to each process.

H.2.11.4 User-Specified Binding Not Supported

There is currently no mechanism to specify which Vendor RAM port gets bound to which process. If a Vendor RAM has more than one type of access protocol for its ports, there is no deterministic way to ensure their binding to a specific process.

H.2.11.5 Multiple Processes Assigned Separate Ports

An array in SystemC code can be accessed by many **SC_THREAD** processes.

When multiple processes access a single RAM, CtoS assigns a separate port to each process.

Note the following restrictions in this case:

- All processes must have the same clock edge sensitivity (positive edge/negative edge).
- All processes must match the clock edge sensitivity specified in the RAM definition file.
- If more than one process uses a separate clock, you must use the `<clock_per_port>` element.

H.2.12 Limitations Reported during Scheduling and Model Generation

During scheduling and model generation, CtoS will report the following problems, related to allocating Vendor RAM:

- “Asynchronous Path from RAM Input to Q Output Not Allowed” on page H-25
- “Concurrent Access to Same Address Not Allowed” on page H-25

H.2.12.1 Asynchronous Path from RAM Input to Q Output Not Allowed

CtoS assumes that any ports specified as **sync_read** or **sync_read_write** have sequential behavior on the **Qn** output port.

If a read port is specified as **sync_read** or **sync_read_write**, but has an *asynchronous* path from a RAM input port to a **Q** output, it is highly likely the resulting RTL will contain combinational loops, which will produce an error.

You must ensure that both your Verilog wrapper and the RAM cell itself are sequential across all timing arcs from the set of port-specific inputs to the port **Qn** outputs.

H.2.12.2 Concurrent Access to Same Address Not Allowed

Most Vendor RAMs do not allow the same address to be read and written by different ports, or written by two ports, in the same cycle.

During RTL generation, CtoS will check and report on cases in which this limitation is not met.

Note *This is a preliminary feature.*

H.3 Vendor ROMs

To use the *wrapper file* method for allocating Vendor ROM, you create two files for each Vendor ROM: the ROM Verilog wrapper file, as shown in [Figure H-8 on page H-26](#), and the IP definition file, as shown in [Figure H-9 on page H-27](#), including a **<wrapper_filename>** element.

The Verilog wrapper file must meet the following requirements:

- It must be synthesizable Verilog, supported by the required implementation tool [that is, Encounter RTL Compiler (RC), etc.]
- It must conform to SystemVerilog P1800.

Important You can also use the memory bridge method for allocating Vendor ROMs; see [“Using Memory Bridges to Allocate Vendor RAMs” on page H-9](#).

This section includes the following subsections:

- [“ROM Standard Verilog Ports” on page H-26](#)
- [“Vendor ROM Library File and Simulation Model” on page H-26](#)
- [“Vendor ROM Example” on page H-26](#)

H.3.1 ROM Standard Verilog Ports

The ROM standard Verilog ports are shown in [Table H-5 on page H-26](#).

The port access protocol to Vendor ROMs is fixed. It is assumed that each ROM has a single synchronous read port.

Limitation Currently, CtoS supports neither asynchronous ROMs nor multiport ROMs.

Table H-5 ROM Standard Verilog Ports

Name	Type	Width	Description
CLK	input	1	clock, active on positive edge
A0	input	AW	address
CE0	input	1	chip enable, active high
Q0	output	DW	data out

H.3.2 Vendor ROM Library File and Simulation Model

The *Vendor ROM library file*, and its corresponding *simulation model*, must both be matched to a specific ROM cell provided by your foundry or third-party IP vendor.

The *library file*, which must be in Liberty format, is a technology-specific model that contains timing, power, and area information.

The *simulation model* is a behavioral Verilog file that can be used to simulate the behavior of the ROM.

H.3.3 Vendor ROM Example

In this section are examples of a ROM Verilog wrapper file and IP definition file.

Figure H-8 ROM Verilog Wrapper File Example

```
module rom_16x4_1r(CLK, CE0, A0, Q0);
    input CLK;
    input CE0;
    input [3:0] A0;
    output [3:0] Q0;

    BRMA23P320103AB2ZZ rom(.CEN(CE0),
                           .CLK(CLK),
                           .FNE(1),
```

```
.A(A0) ,  
.Q(Q0)) ;
```

Figure H-9 IP Definition File for Vendor ROM (Wrapper Method) Example

```
<<?xml version="1.0"?>  
<ctos_ip_definitions>  
  <ROMDef>  
    <name>rom_128x8_1sr</name>  
    <wrapper_filename>rom_128x8_1sr_wrap.v</wrapper_filename>  
    <verilog_filename>rom_128x8_1sr.v</verilog_filename>  
    <num_words>128</num_words>  
    <width>8</width>  
    <clock_posedge>true</clock_posedge>  
    <format>readmemb</format>  
  </ROMDef>  
</ctos_ip_definitions>
```

Note The elements of the IP definition file are further described in “ROMdef XML Elements” on page H-4 and are stored in the CtoS database as “Memory Definition Object Attributes (memory_defs)” on page D-30.

H.4 RTL IP

To use RTL IP in the a CtoS design flow, you first create an IP definition file, which is a set of specific elements describing the external RTL.

This section includes the following subsections:

- “Restrictions of RTL IP Pipeline Depth” on page H-28
- “RTL IP Example” on page H-29

H.4.1 Restrictions of RTL IP Pipeline Depth

This section contains two tables related to restrictions of the RTL IP pipeline depth.

Table H-6 Restrictions of RTL IP pipeline_depth, Related to Pipelined Loop II

RTL IP pipeline_depth	Pipelined Loop II	
	1	>1
0	okay	okay
1	okay	okay
> 1	okay	no

Table H-7 Restrictions of Some Ports, Related to pipeline_depth

Name	pipeline_depth = 0	pipeline_depth > 0
clock_port	cannot be specified	a single clock
reset_port	cannot be specified	a single reset
stall_port	cannot be specified	a single stall
valid_in_port	cannot be specified	a single valid_in

H.4.2 RTL IP Example

This example shows a pipelined multiply-add (**muladd**), with pipeline depth 2, being added to a design.

- “IP Definition File for RTL IP” on page H-29
- “IP Definition File for CGIC IP” on page H-30
- “SystemC Function Implemented by RTL IP to Model the Multiply-Add Op” on page H-30
- “RTL (muladd.v)” on page H-30
- “SystemC (basic.cpp)” on page H-31

Note The elements of the IP definition file are further described in “rtl_ip_def XML Elements” on page H-5 and are stored in the CtoS database as “RTL IP Definition Object Attributes (rtl_ip_defs)” on page D-83.

Figure H-10 IP Definition File for RTL IP

```
<?xml version="1.0"?>
<ctos_ip_definitions>
  <rtl_ip_def>
    <name>muladd</name>
    <rtl_filename>muladd.v</rtl_filename>
    <rtl_language>verilog</rtl_language>
    <rtl_type>pipelined</rtl_type>
    <pipeline_depth>2</pipeline_depth>
    <ports>
      <clock_port>
        <name>clk</name>
        <active_edge>rise</active_edge>
      </clock_port>
      <reset_port>
        <name>rst</name>
        <type>synch</type>
        <active_level>high</active_level>
      </reset_port>
      <valid_in_port>
        <name>start</name>
      </valid_in_port>
      <stall_port>
        <name>stall</name>
      </stall_port>
      <input_port>
        <name>a_in</name>
        <width>8</width>
      </input_port>
      <output_port>
        <name>muladd_out</name>
        <width>16</width>
      </output_port>
    </ports>
  </rtl_ip_def>
</ctos_ip_definitions>
```

Figure H-11 IP Definition File for CGIC IP

```
<?xml version="1.0"?>
<ctos_ip_definitions>
  <cgic_ip_def>
    <name>cgic1</name>
    <rtl_filename>cgic1.v</rtl_filename>
    <ports>
      <clock_port>
        <name>clk_i</name>
        <active_edge>rise</active_edge>
      </clock_port>
      <enable_port>
        <name>en</name>
        <active_level>high</active_level>
      </enable_port>
      <aux_input_port>
        <name>test0</name>
        <width>1</width>
      </aux_input_port>
      <gated_clock_port>
        <name>gclk_o</name>
      </gated_clock_port>
    </ports>
  </cgic_ip_def>
</ctos_ip_definitions>
```

Figure H-12 SystemC Function Implemented by RTL IP to Model the Multiply-Add Op

```
sc_int<16>
basic::muladd(sc_int<8>    a,
              sc_int<8>    b,
              sc_int<8>    c)
{
    return a * b + c;
}
```

Figure H-13 RTL (muladd.v)

```
module muladd (clk, rst, a_in, b_in, c_in, muladd_out);
    input clk, rst;
    input signed [7:0] a_in;
    input signed [7:0] b_in;
    input signed [7:0] c_in;
    output signed [15:0] muladd_out;

    reg signed [7:0] a_reg;
    reg signed [7:0] b_reg;
    reg signed [7:0] c_reg;
    reg [15:0] muladd_reg;

    always @(posedge clk) begin
        if (rst) begin
```

```

        a_reg <= 0;
        b_reg <= 0;
        c_reg <= 0;
        muladd_reg <= 0;
    end else begin
        a_reg <= a_in;
        b_reg <= b_in;
        c_reg <= c_in;
        muladd_reg <= a_reg * b_reg + c_reg;
    end
end
assign muladd_out = muladd_reg;
endmodule // muladd

```

Figure H-14 SystemC (basic.cpp)

```

// *****
// basic
//
/// This is a test case for a pipelined rtl design to link onto a
/// combinational SystemC function.
// *****

```

```

#include <systemc.h>

class basic: public sc_module {
public:
    sc_in<bool>          m_clk;
    sc_in<bool>          m_rst;

    sc_in<sc_int<8> >    m_a;
    sc_in<sc_int<8> >    m_b;
    sc_in<sc_int<8> >    m_c;

    sc_out<sc_int<8> >    m_x;

```

SystemC (basic.cpp) – Continued

```

    sc_out<sc_int<8> >    m_y;

    basic(sc_module_name name)
    :   sc_module(name),
        m_clk("m_clk"),
        m_rst("m_rst"),
        m_a("m_a"),
        m_b("m_b"),
        m_c("m_c"),
        m_x("m_x"),
        m_y("m_y")
    {
        SC_CTHREAD(main,m_clk.pos());
        reset_signal_is(m_rst,true);
    }
    ~basic() {}

    void main();
    sc_int<16> muladd(sc_int<8> a,
                     sc_int<8> b,
                     sc_int<8> c);

    SC_HAS_PROCESS(basic);
};

```

```
// *****
// basic::main()
//
/// This SC_CTHREAD process exercises the member function.
// *****
void
basic::main()
{
    m_x = 0;
    m_y = 0;
    wait();

    while (true) {
        sc_int<16>      m_var;

        m_var = muladd(m_a.read(), m_b.read(), m_c.read());
        wait();
        wait();
        m_x = m_var.range(7,0);
        m_y = m_var.range(15,8);
    }
}

// *****
// basic::muladd()
//
// *****
sc_int<16>
basic::muladd(sc_int<8>    a,
              sc_int<8>    b,
              sc_int<8>    c)
{
    return a * b + c;
}

SC_MODULE_EXPORT(basic);
```


I Debugging Simulation Mismatches

This appendix describes a technique for speeding up the debugging of simulation mismatches.

The technique consists of adding **sc_signals** to a design and writing variables of interest to those signals.

CtoS can preserve the **sc_signals** defined in the design, throughout the synthesis process, so they can then be compared automatically in a side-by-side simulation using a verification wrapper.

This appendix contains the following sections:

- [“Introduction” on page I-2](#)
- [“Debug Flow” on page I-3](#)
- [“Hints for Adding Observation Points” on page I-4](#)
- [“Scheduling and RTL” on page I-5](#)
- [“Caveats” on page I-6](#)

I.1 Introduction

Mismatches between the behavior of a model produced by CtoS and the original SystemC model occur for many reasons, including:

- The compiler directives (for example, **#if**) that are active when CtoS is processing the design are not consistent when the simulator is processing the design (user error).
- The design reads from an uninitialized local variable or a module field that is not properly reset (user error).
- The design has a race condition (user error).
- The design was incorrectly elaborated (CtoS error).
- The design was incorrectly optimized (CtoS error).
- The specified latency constraints are incorrect (user error).

Side-by-side simulation is a powerful technique for debugging such simulation mismatches. The CtoS **write_wrapper** command supports this technique, as it generates a wrapper for instantiating any model generated by CtoS or the original model of the design underneath the testbench.

The wrapper can also instantiate two models:

1. a *reference model* that does not feed back into the testbench, but serves only as a reference, and
2. the model referred to as the *model under test*.

The wrapper can optionally compare the outputs of the *model under test* against those of the *reference model* and issue a message if they disagree. The configuration of the wrapper is specified via the arguments of its module constructor.

Side-by-side simulation is practical only if the two models being simulated have the same timing. Typically, the *reference model* is the original SystemC model, and the *model under test* is the **write_sim-birthday** model obtained before performing any timing-changing transforms.

To determine the root cause of a mismatch, you must trace back the logic driving the primary output (with the mismatch) starting from the first cycle on which the mismatch occurs. You must do this on both the *model under test* and the *reference model*. As the root becomes more distant from the primary output (with the mismatch), it becomes harder and more time-consuming to diagnose.

A common technique to speed up the diagnosis is to increase observability and to compare not just the primary outputs of both models, but also internal observation points. Since **sc_signals** are used for modeling communication between processes, they are natural candidates for adding observation points. The **-compare_signals** option of the **write_wrapper** command supports comparing of **sc_signals**.

To further increase observability, you can instrument the design. This consists of adding **sc_signals** to the design, initializing them during reset, and writing to them any variable of interest. Typically, one **sc_signal** is chosen per variable. Because these added signals are not read by any process, and are therefore nonessential, CtoS would normally optimize them out.

You can prevent CtoS from optimizing these **sc_signals** by defining the **keep_all_signals** design attribute.

Note See also “[write_wrapper](#)” on page E-176 and “[keep_all_signals](#)” on page D-17.

I.2 Debug Flow

To perform the debug flow, follow these steps:

1. Run CtoS, build the design, and immediately produce a Verilog simulation model and a verification wrapper that compares signals.

```
set_attr keep_all_signals true [get_design]
..
build
write_sim -birthday -o ${top}_elab.v [get_design]/modules/$top
write_wrapper -compare_signals -o ${top}_ctos_wrapper.v
               [get_design]/modules/$top
```

2. Modify the testbench to instantiate the wrapper instead of the original SystemC model. Configure the wrapper to the model generated in step 1 and the original SystemC model as a reference, and to compare the correspondence points of these models (primary outputs + **sc_signals**).

```
..
#if defined(CTOS_sim) || defined(CTOS_rtl)
#include "model/dut_ctos_wrapper.h"
#endif
..
#if defined(CTOS_sim) || defined(CTOS_rtl)
dut_ctos_wrapper      m_dut;
#else
dut                   m_dut;
#endif
..
#if defined(CTOS_sim)
m_dut("m_dut", "_sim", "", true), // ref. model=SystemC, compare=true
#elif defined CTOS_rtl
m_dut("m_dut", "_rtl"),
#else
m_dut("m_dut"),
#endif
..
```

3. Run the simulation.

If needed, determine the control state and variables that should be observed by:

- Adding **sc_signals** to the SystemC source for observing these variables.
- Initializing the new **sc_signals** in the reset section of the process that will own the signals.
- Adding writes for observing a variable at the point of interest.

Go back to step 1, and then to step 3. Note that both the simulation model and the wrapper must be re-generated; otherwise, the correspondences of the new **sc_signals** will not be performed.

I.3 Hints for Adding Observation Points

Here are some hints for adding observation points:

- [“Naming with a Specific Prefix” on page I-4](#)
- [“Observing the Control State of a Process” on page I-5](#)
- [“Observing Variables in Zero-Delay Loops” on page I-5](#)

I.3.1 Naming with a Specific Prefix

It is recommended that you name all added **sc_signals** with a specific prefix.

It will then be easier to spot these signals in the hierarchy browser when looking at the Verilog behavioral simulation model.

I.3.2 Observing the Control State of a Process

To track the control state of a process, you can add an **sc_signal**<short> (for example, **sig_state_p1**), then insert the following line immediately before every **wait** called by that process:

```
sig_state_p1.write(__LINE__);
```

The added signal will then have the value of the line number of the added line.

I.3.3 Observing Variables in Zero-Delay Loops

You cannot use one **sc_signal** to track a variable in a zero-delay loop, because only the last value would be visible.

As a workaround, you can use an array of **sc_signal** and use one **sc_signal** for each iteration.

Alternatively, you could insert a **wait** in the loop, provided that the testbench can tolerate this.

I.4 Scheduling and RTL

Instrumentation decreases scheduling freedom; however, if instrumentation has been carefully chosen, the design may still schedule, and you may be able to produce RTL. In this case, the added observation points may help you debug the RTL, but if CtoS has changed the timing of the design, you can no longer perform a side-by-side simulation with comparison.

Alternatively, you can manually add code to the original SystemC model and to the generated RTL to trace the observation points, run separate simulations of the original SystemC and the RTL, and compare the traces.

The name of the Verilog **reg** in the RTL corresponding to an **sc_signal** in the top-level module is simply the name of that **sc_signal**. For an array of **sc_signal**, the corresponding **regs** in the RTL take an index as a suffix.

If you did not add many observation points, an easy way to trace them in the RTL model is via a **\$monitor** statement:

```
initial begin
    $monitor($time,, sig_state_p1,, sig_acc);
end
```

I.5 Caveats

Note the following caveats:

- “Some Variables Should Not Be Observed Directly” on page I-6
- “Instrumentation Changes the Design” on page I-6
- “Instrumentation Deteriorates QoR” on page I-6

I.5.1 Some Variables Should Not Be Observed Directly

Pointers should not be observed directly via instrumentation, because address assignments done by CtoS and those done by the simulator are very different. If the pointer is known to address a given array, then you can convert the pointer to an array index by taking the difference with the address of element **0** of the array and observing the array index.

Similarly, objects of dynamic classes should not be observed directly via instrumentation, because these objects have a virtual table pointer, which cannot be compared.

Limitation Objects of user-defined types with base classes are currently not supported for automatic comparison via the wrapper. This is a limitation of the **write_wrapper** command.

I.5.2 Instrumentation Changes the Design

Instrumentation limits the optimizations that CtoS can perform. If the root cause of the mismatch happens to be an optimization bug, then instrumentation may cause that particular optimization not to occur, thereby hiding the bug.

Most of these issues can be avoided by judiciously and gradually adding instrumentation.

I.5.3 Instrumentation Deteriorates QoR

In addition to limiting optimizations, instrumentation can also reduce the freedom that normally exists when scheduling a design. Therefore, after simulation mismatches have been debugged, instrumentation should be removed or turned off (via compiler directives).

Note CtoS issues WARNING (CTOS-11252) if the **keep_all_signals** design attribute is set when you run any of the following commands: **build**, **schedule**, or **write_rtl**.

J Controlling CtoS-Generated Verilog

This appendix provides information about controlling certain aspects of CtoS-generated Verilog files, as follows:

- “Controlling Case Statement Generation” on page J-1
- “Controlling Mismatched Operand Bit Widths” on page J-3
- “Controlling pragma Keyword for Generated RTL” on page J-4
- “Controlling Use of Wire Arrays to Model Lookup Resources” on page J-4
- “Controlling Use of Indexed Part Selects” on page J-5
- “Controlling Use of Delay Control with Non-Blocking Assignments” on page J-6
- “Other Verilog-2001 Constructs Used in Model Generation” on page J-7

J.1 Controlling Case Statement Generation

You can control the following aspects of *case* statement generation:

- “Controlling Default Case” on page J-2
- “Controlling Select Expression, Case Label Generation” on page J-2

J.1.1 Controlling Select Expression, Case Label Generation

You can control how the *select expression and case labels will be generated in RTL models*.

In the **Output** tab of the **Design Properties** dialog, as shown in [Figure 6-21 on page 6-22](#), you can select the Mux Style to be used in RTL model generation. When set to **Reverse One Hot** (the default), the select expression will be a constant value, and *case* labels will be variables:

```
a = {a1, b1, c1};
case(1'b1) /* cadence parallel_case */
  a[0]: d = b;
  a[1]: d = c;
  a[2]: d = e;
  default: d = 3'bX;
endcase
```

If you set the value to **Case Z**, *casez* will be produced where the select expression will be a variable and case labels will be a constant value.

Note that for longer bit sequences, this may be unwieldy

```
casez ({a1, b1, c1}) /* cadence parallel_case */
  3'b??1: d = b;
  3'b?1?: d = c;
  3'b1??: d = e;
  default: d = 3'bX;
endcase
```

If you set the value to **One Hot**, *case* will be produced where the select expression will be a variable and case labels will be a constant value:

```
case ({a1, b1, c1}) /* cadence parallel_case */
  3'h1: d = b;
  3'h2: d = c;
  3'h4: d = e;
  default: d = 3'bX;
endcase
```

This sets the **verilog_mux_style** design attribute ([“verilog_mux_style” on page D-26](#)).

J.1.2 Controlling Default Case

You can control how the *default case will be generated*.

In the **Output** tab of the **Design Properties** dialog, as shown in [Figure 6-21 on page 6-22](#), if you leave **X** (the default) selected, the default *case* will be assigned a value of **X**, as follows:

```
case (1'b1) /* cadence parallel_case */
  a[0]: d = b;
  ...
  default: d = 1'bX;
endcase
```


If you select **0**, instead, the default *case* will be assigned a value of **0**:

```
case (1'b1) /* cadence parallel_case */
  a[0]: d = b;
  ...
  default: d = 1'b0;
endcase
```

This sets the **verilog_use_case_default_x** design attribute ("[verilog_use_case_default_x](#)" on page D-27).

J.2 Controlling Mismatched Operand Bit Widths

CtoS-generated RTL may contain operations that have operands of mismatched bit widths. This is a result of bit-trimming optimizations and is often necessary for getting the best QoR.

However, HDL lint checkers, such as HAL or Spyglass, will issue warnings for such occurrences due to differences in how simulators interpret these operations. Lint checkers will typically recommended that you modify such expressions to use operands of equal widths so the desired padding of the operands will be used to generate the expected operation result.

To prevent such warnings in HDL link chckers, you can direct CtoS to align widths. This option will pad inputs to certain arithmetic and all comparison/equality operations to the same matching bit width.

In the **Output** tab of the **Design Properties** dialog, as shown in [Figure 6-21 on page 6-22](#), check the **Use Aligned Widths** checkbox to enable this feature. If you check this box, the alignment will happen to all operands on LHS, as follows:

```
reg [13:0] add_ln617_8_0_z;
reg [13:0] add_ln617_8_1_z;
reg [14:0] add_ln617_8_10_z;
...
add_ln617_8_10_z = {1'b0, add_ln617_8_0_z} + {1'b0, add_ln617_8_1_z};
```

If you uncheck this checkbox, all operands may not be aligned properly:

```
reg [13:0] add_ln617_8_0_z;
reg [13:0] add_ln617_8_1_z;
reg [14:0] add_ln617_8_10_z;
...
add_ln617_8_10_z = {1'b0, add_ln617_8_0_z} + add_ln617_8_1_z;
```

This sets the design attribute **verilog_use_aligned_widths** ("[verilog_use_aligned_widths](#)" on page D-27).

J.3 Controlling pragma Keyword for Generated RTL

Depending upon the implementation target type (ASIC or FPGA), the following pragma keywords are automatically selected by CtoS:

- For an ASIC implementation target type, **pragma** is the default pragma keyword.
- For an FPGA implementation target type, **synthesis** is the default pragma keyword.

CtoS lets you set a customized **Logic Synthesis Pragma Keyword** for generated RTL in the **Output** tab of the **Design Properties** dialog, as shown in [Figure 6-21](#) on [page 6-22](#). This is useful when the CtoS-generated RTL is input to different synthesis back-end tools that support different pragma keywords.

The CtoS model generator will use the name that you provide as the pragma keyword for all pragmas generated in the RTL.

Note that CtoS also uses the following pragmas in generated RTL:

- **parallel_case**
- **translate_on/translate_off**
- **async_set_reset_local**
- **sync_set_reset_local**

For example:

- Using the default keyword of *pragma*: `case (1'b1) // pragma parallel_case`
- Setting the keyword to *synthesis*: `case (1'b1) // synthesis parallel_case`

This sets the **verilog_pragma_keyword** design attribute ("[verilog_pragma_keyword](#)" on [page D-26](#)).

J.4 Controlling Use of Wire Arrays to Model Lookup Resources

You can control whether wire arrays will be used to model lookup resources by setting the **verilog_use_wire_array** design attribute ("[verilog_use_wire_array](#)" on [page D-28](#)).

The default is *false*, which means the lookup resource will be modeled as a case statement within a function as shown here:

```
function [9:0] array(input reg [6:0] addr);
case (addr)
7'h0: array = 10'ha;
7'h1: array = 10'h59;
7'h2: array = 10'h3;
7'h3: array = 10'h5;
7'h4: array = 10'h0;
7'h5: array = 10'h0;
7'h6: array = 10'h0;
```

Setting the attribute to *true* will model a lookup resource as a wire array as shown here:

```
assign array[0] = 10'ha;
assign array[1] = 10'h59;
assign array[2] = 10'h3;
assign array[3] = 10'h5;
assign array[4] = 10'h0;
assign array[5] = 10'h0;
assign array[6] = 10'h0;
```

Important The default for `verilog_use_wire_array` is *false* because the INCISIV tools (`ncelab` and `ncsim`) will issue the following warning message if wire arrays are found inside an **always @*** process:

```
nchelp: 08.10-s003: (c) Copyright 1995-2008 Cadence Design Systems, Inc.
ncelab/MRSTAR =
This array or unpacked structure reference is in a statement with @*
attached. By section 9.7.5 of IEEE Std 1364-2001, this means that the array
or unpacked structure identifier should be added to the implicit event list.
The behavior of this is not covered by the IEEE standard. NC-Verilog will
handle this situation by inferring sensitivity to all elements of the array
(or all members of the unpacked structure), which is the most conservative
interpretation of such constructs. This should give correct behavior for
combinational logic. While this is a common interpretation, this behavior
may not be portable to other tools, or to future versions of NC-Verilog if
the IEEE standardizes different behavior.
```

J.5 Controlling Use of Indexed Part Selects

Indexed vector part selects, a feature of Verilog-2001, are used in model generation.

For example, in the following code excerpt, if `byteNum` has a value of 4, then the value of `word[39:32]` is assigned to `byteN`.

```
Format: [base_expr +: width_expr] // positive offset
        [base_expr -: width_expr] // negative offset

reg [63:0] word;
reg [3:0] byteNum;
wire [7:0] byteN = word[byteNum*8 +: 8];
```

You can control the use of this feature by setting the `verilog_use_indexed_part_select` design attribute ([“verilog_use_indexed_part_select” on page D-27](#)).

The default is *true* and is set as follows:

```
set_attr verilog_use_indexed_part_select true [get_design]
```

Set this attribute to *false* to specify that *case* statements be used, instead:

```
set_attr verilog_use_indexed_part_select false [get_design]
```

J.6 Controlling Use of Delay Control with Non-Blocking Assignments

You can control whether a *delay control* (a #1) will be used for every non-blocking assignment by setting the **verilog_use_non_blocking_delay_control** design attribute (“[verilog_use_non_blocking_delay_control](#)” on page D-28).

The default is *false*:

```
always @(posedge clk)
  if(rst_n == 0)
    q <= 1'b0;
  else
    q <= d;
```

However, if you set this attribute to *true*, CtoS will use a #1 intra-assignment delay for clocked **always** blocks.

This can make debugging easier by having **clk** and **data** changes separated in the waveforms. For example:

```
always @(posedge clk)
  if(rst_n == 0)
    q <= #1 1'b0;
  else
    q <= #1 d;
```

Warning Contrary to some designers’ opinions, this coding style will *not* help avoid Verilog race conditions.

Furthermore, since logic synthesis tools, such as Encounter RTL Compiler (RC), will ignore these added intra-assignment delays, the use of this design attribute is not recommended for final RTL.

J.7 Other Verilog-2001 Constructs Used in Model Generation

In addition to indexed vector part selects, as described in “Controlling Use of Indexed Part Selects” on page J-5, the following Verilog-2001 constructs are used in model generation:

- “Signed Arithmetic Extensions” on page J-7
- “Combinational Logic Sensitivity” on page J-7
- “Combined Port and Data Type Declarations” on page J-8

J.7.1 Signed Arithmetic Extensions

Signed arithmetic extensions are used in model generation, as follows:

- **reg** and **net** data types can be declared as signed.
- Operands can be converted from signed to unsigned using **\$signed** and **\$unsigned** system functions.
- An additional specifier ('s') can be used with the radix specifier to indicate that a literal number is a signed value.
- Arithmetic right shift operators (>>>).

The following is *not* used in model generation:

- Arithmetic left shift operators (<<<).

For example:

```
16'hC501          // unsigned 16-bit hex value
16'shC501         // signed 16-bit hex value
reg [63:0] a;      // unsigned data type
b1 = a / 2 ;       // unsigned arithmetic
b2 = $signed(a) / 2; // signed arithmetic
```

J.7.2 Combinational Logic Sensitivity

Combinational logic sensitivity is used in model generation. In order to model combinational logic using the Verilog **always** construct, you must include all input signals to the combinational logic block in the sensitivity list for the combinational process.

The CtoS model generator uses the Verilog-2001 wildcard token, **@*** (or the combinational logic sensitivity token), that represents the combinational logic sensitivity list for this purpose.

For example:

```
always @(*)
```

J.7.3 Combined Port and Data Type Declarations

Combined port and data type declarations are used in model generation.

In Verilog-1995, you were required to declare the signals connected to the input or outputs of a module with two separate statements: the direction of the port and the data type of the signal.

Verilog-2001 specifies a simpler syntax, which combines the two declarations into one statement, and the CtoS model generator uses this new form.

For example:

```
module testMod(clk, a, b, c);  
    input clk;  
    input [7:0] a, b;  
    output reg [7:0] c;
```

K Logic Synthesis Steps

During logic synthesis, a design is optimized and mapped to the target technology, using a series of steps.

After each of these steps, a database file is saved. In addition, basic reports are generated, and a timestamp is put in the logfile.

You can see a listing of these steps in [Table K-1 on page K-2](#).

Notes

- For more information on logic synthesis in CtoS, see [“Generating Gates in CtoS” on page 13-42](#).
- For more information on the RC synthesis flow and steps, please see *Performing Synthesis*, in the *RTL Compiler User Guide*.

Tip To perform zero wireload synthesis, set these variables:

```
set vars(wireload_mode) top
set vars(force_wireload) none
```

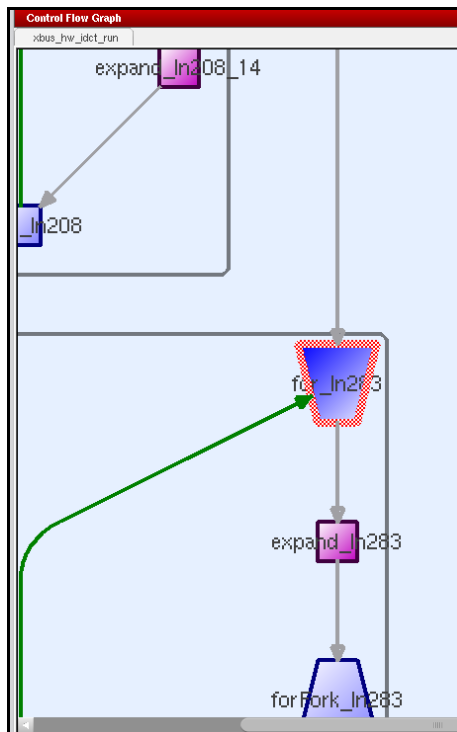
Table K-1 Logic synthesis steps

Step Description	Step Name
Configure RC for the particular flow (set root attributes and RC-specific variables). Then, source or execute the plugin post_setup at the end of this step	ff_root_attributes
Load libraries	ff_load_library
Set dont_use cell settings	ff_setup_dont_use_cells
Set up use cell settings	ff_setup_use_cells
Load HDL (read_hdl)	ff_load_hdl
Elaborate the design	ff_elab
Set design-level attributes	ff_design_attributes
Load SDC (timing) constraints	ff_load_constraints
Synthesize to generic structures	ff_syn_gen
Synthesize to technology-mapped gates	ff_syn_map
Optionally insert LP (low power)	ff_insert_lp
Optimize incrementally	ff_syn_incr
Perform uniquification	ff_uniquify
Generate reports (area, timing, power, clock gating)	ff_final_reports
Generate outputs (write_hdl , database)	ff_write_final_outputs
Generate <i>QoR Metrics File</i>	ff_write_xml_metrics
Exit	ff_exit

L Control Flow Graph (CFG) Viewer

The **Control Flow Graph (CFG)** viewer, as shown in [Figure L-1 on page L-1](#), is displayed when you select **View -> CFG**, or right click on a loop or state and select **Show in Control Flow**.

Figure L-1 Control Flow Graph Viewer



This appendix describes the CFG viewer in detail, as follows:

- “Navigating the CFG Viewer” on page L-2
- “Exploring with Annotation in the CFG Viewer” on page L-3
- ““Detail Bubbles” for Objects in the CFG Viewer” on page L-4
- “Canvas, Loops, Function Call Loops Supported” on page L-7
- “Tooltips Provide Helpful Information” on page L-8
- “Pipelined Loops Have Distinct Glyph” on page L-8

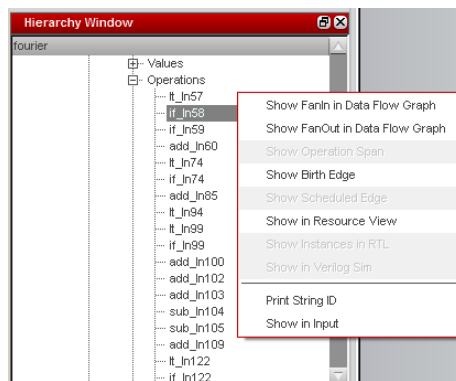
L.1 Navigating the CFG Viewer

In addition to regular tool tips and context menus, the CFG viewer offers the following methods of control to make exploring easier:

- **Vertical Scrolling** – mouse wheel
- **Horizontal Scrolling** – **Alt** + mouse wheel
- **Direct Zoom In/Out** – In addition to using the zoom control in the toolbar, you can select the **Ctrl** button plus the mouse wheel to zoom in or out directly at the position of the mouse cursor.
- **Panning** – When the graph is larger than the viewing area, as an alternative to the scroll bars, you can grab the graph (pressing and holding the left mouse button) and pan it directly in any direction.
- **Default action for context menu** (if one is defined) – double-click

You also have the ability to right-click on a CFG edge and insert a state right there if the latency constraints allow it. Similarly, for any given operation, in addition to showing its op span, you can also visually determine the location of its birthday edge and/or scheduled edge (available after scheduling) on the CFG by right-clicking the mouse in the **Hierarchy Window**, as shown in [Figure L-2 on page L-2](#).

Figure L-2 Menu for Determining Birthday and/or Scheduled Edge in CFG Viewer



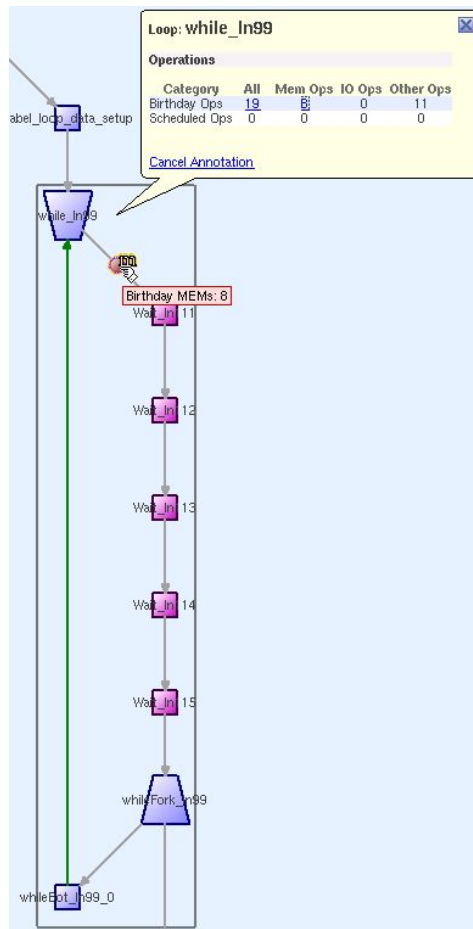
L.2 Exploring with Annotation in the CFG Viewer

The CFG viewer also has *annotation*, which can bring out information in a graph that is not easily accessed, for example:

- ops on an edge
- the types of the ops on an edge
- distribution of these ops, array accesses and/or I/Os across the edges of a loop or of the entire CFG

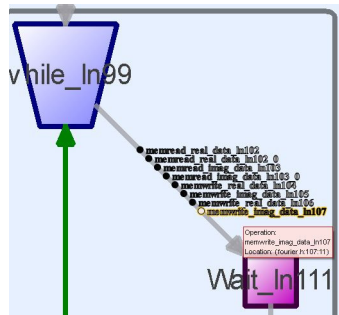
Annotation can also help to solve exploration problems; for example, [Figure L-3 on page L-3](#) shows a loop with 19 birthday ops, of which eight access memories and are all located on the same edge.

Figure L-3 Loop with 19 Birthday ops on Same Edge



If you zoom in, as shown in [Figure L-4 on page L-4](#), you can actually see the eight individual memory ops, order preserved. These ops are full-fledged objects with tooltip and context menu support, so you can interact with them just as you would if you were in the **Hierarchy Window** or **CDFG Viewer**.

Figure L-4 Zoom in on Loop - Showing Eight Memory Ops



If you cross-highlight an object, for example, the op span of an operation, as shown in [Figure L-5 on page L-4](#), not only will you be taken to the op span edges, the edges themselves will also *glow* briefly so you can easily identify the object you selected.

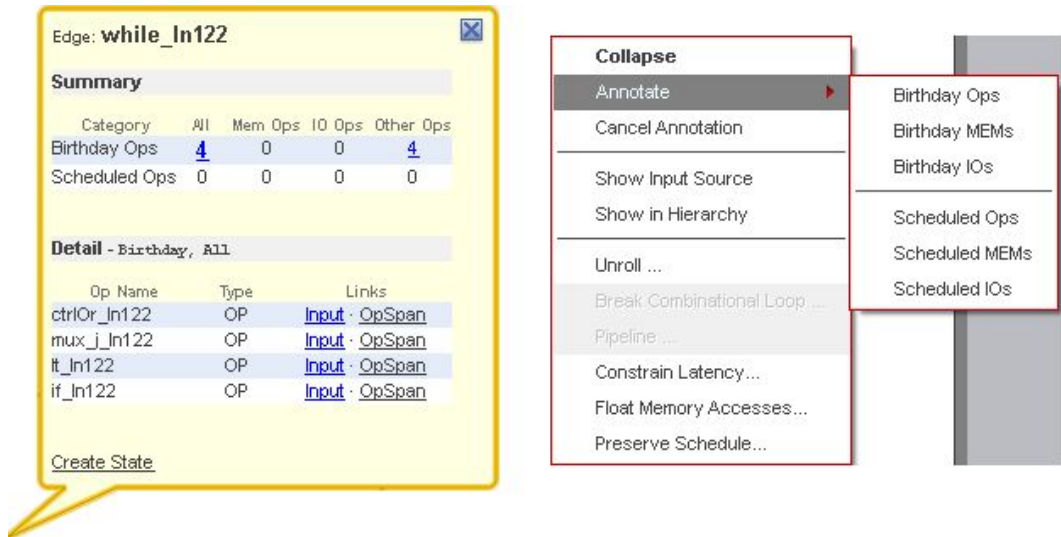
Figure L-5 Edges of selected object “glow”



L.3 “Detail Bubbles” for Objects in the CFG Viewer

In general, if you want to learn more about a particular object, simply clicking on it, or on its information icon, will bring up a *Detail Bubble*, as shown in [Figure L-6 on page L-5](#) for an edge. You can activate additional annotation from this *Detail Bubble*, or you can do it through context menus.

Figure L-6 “Detail Bubble” for an Edge in the CFG Viewer

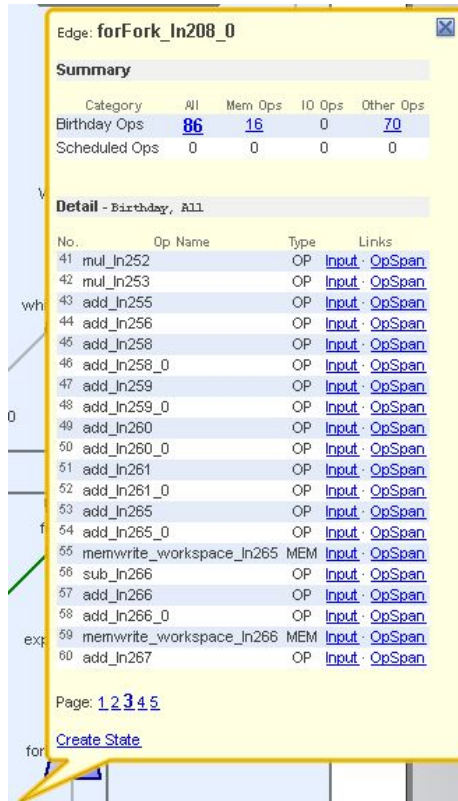


To help the *Detail Bubble* of an edge cope with a large number of ops:

- Ops are numbered.
- Large numbers of ops are automatically split into pages, with page links at the bottom to control which page gets viewed, as shown in [Figure L-7 on page L-6](#).

The page size is currently set at 20.

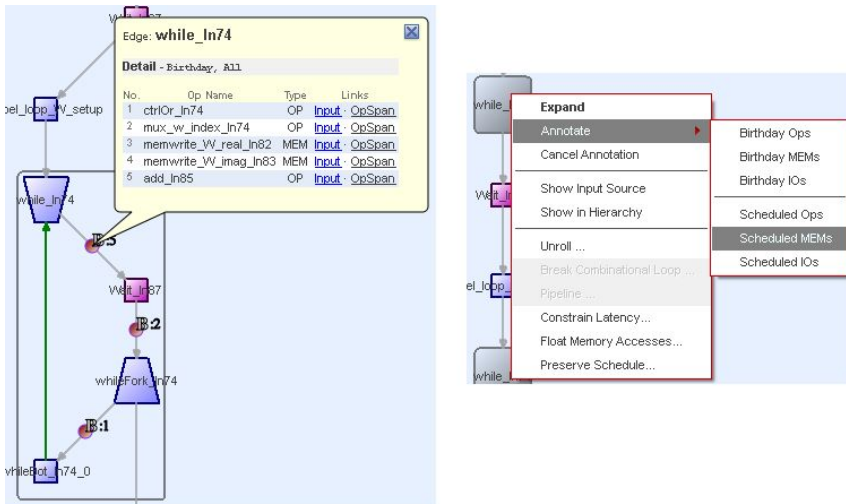
Figure L-7 CFG Viewer Page Feature for Large Numbers of Ops



When you click on a grouped marker in the CFG, you also get a *Detail Bubble*, as shown in [Figure L-8](#) on page L-7.

Also, if you try to annotate a loop that is collapsed, it will automatically be expanded so you can see the annotation result.

Figure L-8 Detail Bubble for a Grouped Marker in CFG Viewer



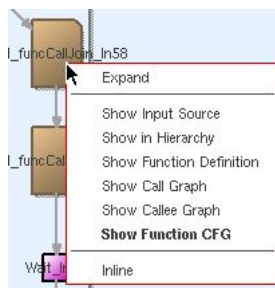
L.4 Canvas, Loops, Function Call Loops Supported

Context menus and default actions support *canvas*, *loops*, and *function call loops*.

Function call loops are visualized as distinct entities on the CFG. The context menu for a sequential custom op in the **Hierarchy Window** includes *Show Function Call Loop*, to display it in the CFG.

Show Function CFG, as shown in Figure L-9 on page L-7, is the default action for the context menu of a *function call loop* (as contrasted with expand/collapse in the case of a normal loop).

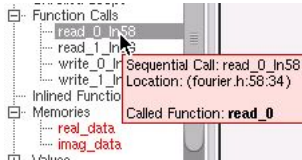
Figure L-9 Show Function CFG in the CFG Viewer



L.5 Tooltips Provide Helpful Information

Tooltips, especially those for *functions*, *function calls*, *normal loops*, and *function call loops*, carry helpful information, as shown in [Figure L-10 on page L-8](#).

Figure L-10 Tooltip in the CFG Viewer

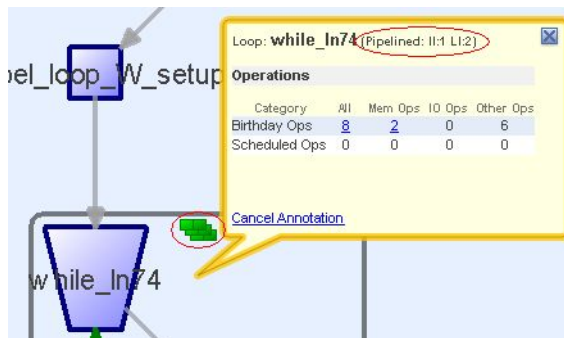


L.6 Pipelined Loops Have Distinct Glyph

Pipelined loops in the CFG have a distinctive glyph – a visual cue to help you quickly discern pipelined loops from all others at a glance, as shown in [Figure L-11 on page L-8](#).

The *Detail Bubble* of the loop also shows high-level information on pipelining, such as initiation interval (II) and latency interval (LI).

Figure L-11 Pipelined Loops in the CFG Viewer



M Release Notes for 14.10

This chapter describes the new features and enhancements of **Cadence C-to-Silicon Compiler (CtoS)**, version **14.10**. It also includes the known limitations and the problems that are resolved for this release. The following features and enhancements are part of the **14.10** release:

- **What's New**
 - **Ease of Use**
 - **Expanded SystemC Support for Array of Module and TLM Ports**
 - **Enhanced MicroArch Manager Enables You to Focus on a Process**
 - **Changing Pipeline Loop Latency**
 - **Better Handling of Hierarchical Designs**
 - **Improved Output of Create Required States**
 - **Improved Output of Create Required States**
 - **Extra Timing Effort Option to Pipeline with Minimal Latency**
 - **New Strongly Connected Component (SCC) Report**
 - **Improved Flow Integration**
 - **Support Simulation of SystemC within Verilog Testbench**
 - **Reduce Lint Issues**
 - **Reduce ECO Differences from New Resources**
- **Deprecated Items**
 - **schedule -useBirthday Option Renamed**

- **Changes to CtoS-Supported OS Platforms and Software Versions**
- **Known Limitations**
- **Known Problems with Workarounds in 14.10**
- **Problems Resolved/Enhancements Made in 14.10**

M.1 What's New

The **Cadence C-to-Silicon Compiler (CtoS)**, production version **14.10**, includes the following new features and enhancements:

- “Ease of Use” on page M-2
- “Improved Output of Create Required States” on page M-7
- “Improved Flow Integration” on page M-9

M.1.1 Ease of Use

- “Expanded SystemC Support for Array of Module and TLM Ports” on page M-2
- “Enhanced MicroArch Manager Enables You to Focus on a Process” on page M-4
- “Changing Pipeline Loop Latency” on page M-6
- “Better Handling of Hierarchical Designs” on page M-6
- “Improved Output of Create Required States” on page M-7

M.1.1.1 Expanded SystemC Support for Array of Module and TLM Ports

CtoS 14.1 provides better support for arrays of modules, arrays of **sc_port** and arrays of **sc_export**, as follows:

- You can now call a function on an element of an array module, **sc_port**, or **sc_export**.
- You can now select a field from an element of an array of module.
- The array index used for selecting the element of the array module, **sc_port**, or **sc_export** does not need to be a compile-time constant.

Example

Consider the example below:

- Module DUT has a field **m_ports**, which is of type array of **sc_port<IF1>** (line 30)

- The ports in this array are connected in the constructor of DUT (line 24)
- Process **main()** of DUT calls the interface methods exposed through an element of **m_ports** (line 34)

CtoS 13.2 had limited support for array of **sc_port** and **sc_export**:

- It allowed declaration and binding of such ports in the module constructor (that is, lines 30, 24 are supported)
- It did *not* support interface method calls on an element of an array of **sc_port** and **sc_export**
- Consequently, CtoS 13.2 allowed use of **sc_port** and **sc_export** only for defining the structure of the connectivity; by itself this is not very useful.

CtoS 13.2 had limited support for array of (pointer-to) **sc_module**

- It allowed for declaration of such arrays (line 29)
- It did not allow for access to the elements of an array of (pointer-to) **sc_module** from a process unless the index expression is a compile-time constant (line 34).
- Again, the limited support for array of (pointer-to) **sc_module** was not very useful in practice.

```
10 struct IF1: public virtual sc_interface {
11     virtual int func1(int v) = 0;
12 };
13
14 SC_MODULE(SUB), public virtual IF1 {
15     virtual int func1(int v) { .. }
16 };
17
18 SC_MODULE(DUT) {
19     sc_in<bool> m_clk;
20     sc_in<bool> m_rst;
21     SC_CTOR(DUT) {
22         for (int i = 0; i < 2; i++) {
23             m_subs[i] = new SUB("sub");
24             m_ports[i].bind(*m_subs[i]);
25         }
26         SC_CTHREAD(main, m_clk.pos());
27         reset_signal_is(m_rst, true);
28     }
29     SUB                *m_subs[2];
30     sc_port<IF1>        m_ports[2];
31     int func2(SUB *sub) {return sub->func1();} // Not supported
32     void main() {
33         ..
34         m_ports[i]->func1(..); // access through port: supported in 14.1
35     }
```

```
36      m_subs[i]->func1(..); // port-less access: supported in 14.1
37      ..
38      func2(&m_subs[i]); // not supported
39    }
40  };
```

Newly Supported Constructs with this Feature

Constructs that CtoS 14.1 will support (but CtoS 13.2 will reject):

- Interface method call on element of module field of type array of **sc_port<T>** (see line 34 in above example).
- Interface method call on element of module field of type array of **sc_export<T>**
- Call of member function of element of module field of type array of **sc_module**
- Call of member function of de-reference of element of module field of type array of pointer-to-**sc_module** (see line 36 in above example).

Limitations That Still Exist

CtoS 14.1 will not support any of the following (no change from CtoS 13.2):

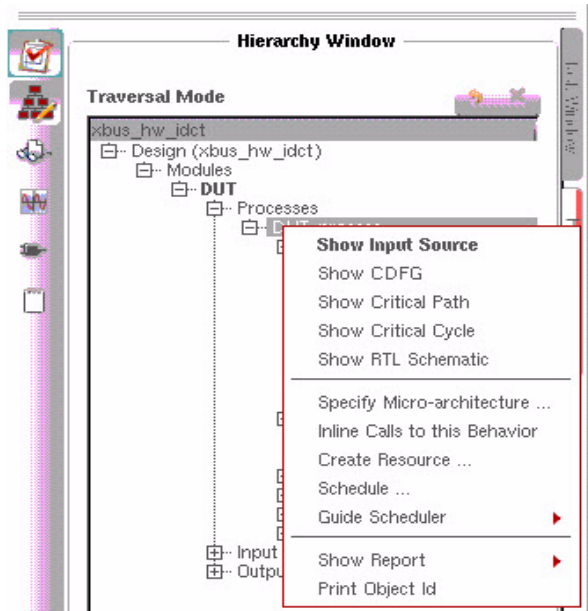
- Local variable of type pointer to **sc_port**, **sc_export**, or **sc_module**
- Formal argument of type pointer to **sc_port**, **sc_export**, or **sc_module** in function called from a process of the design:
 - Example function func2 line 31 has a formal argument of type pointer to **sc_module**
 - func2 is called at line 38 from process **DUT::main()**

M.1.1.2 Enhanced MicroArch Manager Enables You to Focus on a Process

The MicroArch Manager has been enhanced with a new menu option to *Specify Micro-architecture...* which enables you to focus on a specific behavior when specifying actions on loops, arrays and called functions.

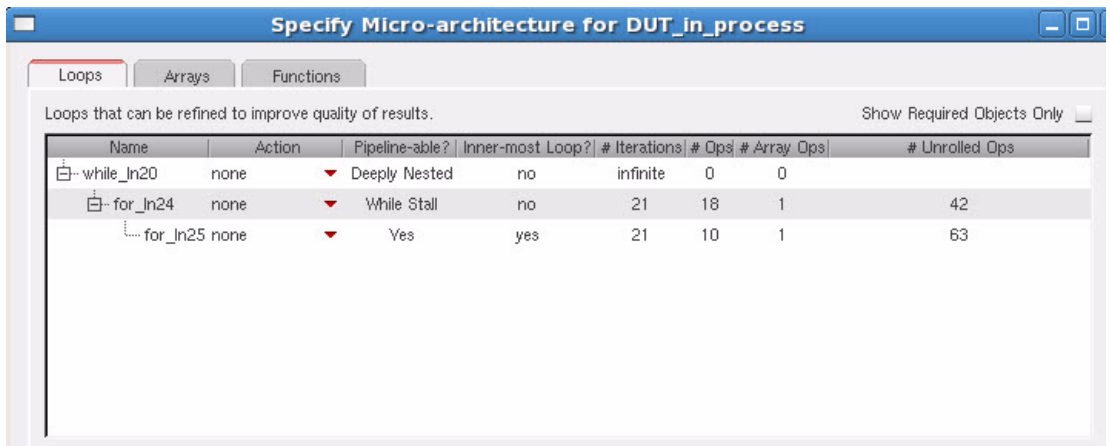
Right-click on a process to access and choose the *Specify Micro-architecture...* option.

Figure M-1 MicroArch Manager “Specify Micro-architecture” Menu Option



The Specify Micro-architecture window appears for your chosen process and lists the loops contained in that specified behavior.

Figure M-2 Specify Micro-architecture Window



The Loops Tab, when "Show Required Objects Only" is false, will display the loops as a tree of nested loops.

M.1.1.3 Changing Pipeline Loop Latency

The following node attributes are now writable, enabling you to change the latency of a pipelined loop:

- **init_interval**
- **min_lat_interval**
- **max_lat_interval**

You can no longer call **pipeline_loop** on a join node that has already been pipelined. Instead, you would use the **set_attr** command to change the **init_interval**, and minimum or maximum latency of the pipelined loop. If multiple attributes are being changed, change the max before the min to avoid range errors.

- New value for **init_interval** must be less than **min_lat_interval**
- New value for **min_lat_interval** must be greater than or equal to the old value
- New value for **max_lat_interval** must be greater than or equal to **min_lat_interval**

These attributes are 0 if the join node is not a pipelined loop. In addition:

- **init_interval** < **min_lat_interval**
- **min_lat_interval** <= **max_lat_interval**
- **lat_interval** starts out at **min_lat_interval**, but may be increased up to and including **max_lat_interval** by the **create_required_states** and **schedule**

When the **min_lat_interval** is increased, the necessary number of states is added to the pipelined loop at the expansion point, and the **lat_interval** attribute is set to the new value.

When the **init_interval** is increased, the pipeline is reconfigured and all existing op constraints are removed. A message to this effect is issued. In the GUI, a message box is displayed regarding ops being reset.

Note These attributes can be set during the **micro_architecture** step and become read-only during the allocate IP step.

For more information, see “[Node Object Attributes \(Nodes\)](#)” on page D-50 and “[pipeline_loop](#)” on page E-92.

M.1.1.4 Better Handling of Hierarchical Designs

We have improved handling of hierarchical designs as follows:

- Added a new design attribute **name_module_prefix_policy** to help avoid module naming conflicts.

- Added a new **-file_policy** option to **write_rtl** to ensure RTL file naming matches SystemC filenames.

Avoid Module Naming Conflict

A new design attribute **name_module_prefix_policy** enables you to control the naming policy of non-user-defined modules. The modules are prefixed with, as follows:

- The value of **name_module_prefix_attr** design attribute (previously supported) OR
- The name of the user-defined module that instantiates the non-user-defined module.

The Design Property Dialog (Naming Tab) has also been updated to let user specify the Verilog module naming policy.

For more information, see [“Design Object Attributes \(Designs\)” on page D-9](#).

RTL File Naming to Match SystemC Filenames

A new option **file_policy** has been added to the **write_rtl** command. The **file_policy** option enables you to specify if files are generated for each Verilog module or SystemC module. Previously CtoS generated a separate file for each Verilog module. This enhancement provides an alternative to create separate file for each SystemC module in the design. In this scenario, CtoS generated modules (for example memory bridges) are declared in the same file as the parent module.

This policy is only applicable when ‘-dir’ option is also specified. There is no change in behavior when this option is used in conjunction with ‘-file’, because all the modules are placed in a single file. If specified with -file then a warning is issued and it is ignored.

The Write RTL Dialog has also been enhanced to prompt user to write file per Verilog Module or SystemC module in user specified directory.

For more information, see [“write_rtl” on page E-161](#).

M.1.1.5 Improved Output of Create Required States

The new format of output from Create Required States identifies the edge as well as the op that is the reason for the state addition. This is helpful in illustrating to the user the conditions where state addition is required.

Here is an example of the output:

```
=====
Creating required states for behavior: 'always_block'
    Sequential ops = 4   Array ops = 4
```

Edge	Add State Reason
expand_ln35_0	Increasing latency of pipeline loop 'forT_ln29' due to dependency (add_ln33)
expand_ln35_1	Increasing latency of pipeline loop 'forT_ln29' due to dependency (or_ln34)

Create Required States adds states for the following cases:

- States are added to de-couple loops and to sequentially separate pipelined functions born on the same edge in relaxed latency mode.
- States are added to resolve memory dependencies (both in pipelines and outside of pipelines)
- States are added to resolve memory contention (both in pipelines and outside of pipelines)

M.1.2 QoR Improvements

- [“Extra Timing Effort Option to Pipeline with Minimal Latency” on page M-8](#)
- [“New Strongly Connected Component \(SCC\) Report” on page M-8](#)

M.1.2.1 Extra Timing Effort Option to Pipeline with Minimal Latency

A new **-extra_timing_effort** option is added to the **pipeline_loop** and **pipeline_function** commands. When this option is specified the scheduler will do an exhaustive search for the minimal latency that closes timing within the user-specified minimum and maximum latency intervals. The default value is **false**.

This option may increase run time of the scheduler. The recommendation is to enable this option on a second run of the scheduler with a smaller latency interval narrowed down around latency value by the first run. For example, if latency *lat* is found, try +/- 5 states around *lat*.

For more information, see [“pipeline_loop” on page E-92](#) and [“pipeline_function” on page E-91](#).

M.1.2.2 New Strongly Connected Component (SCC) Report

You can use the new **report_sccs** command to help identify Strongly Connected Components that can inhibit pipeline scheduling. This report lists all the SCCs in design, behavior or loop. For each SCC the report specifies the variable which is most critical to this SCC helping you to understand why these ops are strongly connected.

Here are common reasons for the formation of an SCC:

- The value of the variable is being modified in each iteration of the loop and needs to be stored for the next iteration. These kind of SCCs are common on variables used as a counter for a loop.

- The value of the variable to be written to the port/signal depends on its value from previous iteration. These kind of SCCs are common on module ports or signals that are used implement a handshake protocol.
- Loop carried dependencies amongst array accesses.

For more information, see [“report_sccs”](#) on page E-121.

M.1.3 Improved Flow Integration

- [“Support Simulation of SystemC within Verilog Testbench”](#) on page M-9
- [“Reduce Lint Issues”](#) on page M-9
- [“Reduce ECO Differences from New Resources”](#) on page M-9

M.1.3.1 Support Simulation of SystemC within Verilog Testbench

New command **write_verilog_wrapper** improves verification flow enabling you to instantiate SystemC input in your Verilog testbench.

For more information, see [“write_verilog_wrapper”](#) on page E-175.

M.1.3.2 Reduce Lint Issues

Reduce lint issues related to:

- unused variables
- bit-width mismatch
- **align_widths**
- **\$signed casting**

M.1.3.3 Reduce ECO Differences from New Resources

The new design attribute **eco_sharing_policy** enables you to control resource sharing between the baseline and ECO resources, to reduce overall ECO differences.

You can set the following values:

- **reuse_existing_resources** (default)
- **create_new_resources**

The Design Property Dialog (Incremental tab) has been enhanced to let user select sharing policy for new ops.

For more information, see [“Design Object Attributes \(Designs\)”](#) on page D-9.

M.2 Deprecated Items

M.2.1 `schedule -useBirthday` Option Renamed

The option **useBirthday** of the **schedule** command has been renamed to **use_birthday**. Support of **useBirthday** is still available in the CtoS 14.1 release, but it will be completely removed from future releases. Instead, use **use_birthday**.

For more information, see [“schedule”](#) on page E-138.

M.3 Changes to CtoS-Supported OS Platforms and Software Versions

The following changes have been made, in this release, to the CtoS-supported OS platforms and software versions:

- Dropped support for SUSE Linux Enterprise Server 10. You must upgrade to SUSE Linux Enterprise Server 11.
- Cadence's Encounter RTL Compiler (RC) version upgraded from version 12.2 to 13.1.
 - Improved technology library support.
 - CtoS no longer supports the 32-bit version. Instead, you can use the 64-bit version.
- Cadence's Incisive Enterprise Simulator (INCISIV) upgraded from version 12.2 to 13.1.

M.4 Known Limitations

Here are the known limitations, at present:

- [“Pipelined Loops with Reads and Writes to Same Array”](#) on page M-11
- [“Limited Support for Multiple Clocks”](#) on page M-12

M.4.1 Pipelined Loops with Reads and Writes to Same Array

If a pipelined loop contains reads and writes to the same array, the CtoS scheduler will ensure that the order of write operations, with respect to reads or other writes, is not affected by pipelining.

To make sure this is handled correctly, CtoS schedules writes so they are not separated by *more than one Π (initiation interval)* from related reads and writes. Thus, reads and writes from future iterations are not folded in between reads and writes of this iteration.

The scheduler, however, may be overly conservative in determining which reads and writes are *related*.

For example, consider this loop:

```
while (c) {
    mem[i] = x;
    wait();
    wait();
    y = mem[j];
}
```

Unless the scheduler can ensure that **mem[i]=x** could never write to the same address – in a future iteration – as is read by **y=mem[j]** – in this iteration – the read and write will be scheduled within one Π (initiation interval) of each other.

You may *know* that two operations are *not related* and find the scheduler too restrictive, that is, if you knew that **i** – in a future iteration – could never equal **j** – in this one – this restriction would be unnecessary.

Consider the following example:

```
LOOP: for (unsigned i = 0; i < n; i++) {
    m[i] = f(m[i]);
    W: wait();
}
```

In this case, CtoS *can* schedule the loop if the memory has a read and write port. There is no dependency between the write of the first iteration (**m[0]**) and the read of the next iteration (**m[1]**). Thus, the CtoS scheduler is able to prove that no two iterations of the loop are dependent on each other and is free to schedule the write at any stage after the read.

However, in more complex cases, manual analysis is required.

Consider the following example:

```
LOOP: for (unsigned i = 0; i < n; i++) {
    m[i+2] = f(m[i]);
    W: wait();
}
```

In this case, it is possible to prove that iteration **i** writes to a location that will be read only two iterations later. Hence, the write at a given iteration can be scheduled *at most* two stages after the read – otherwise, the read two iterations later would read the original value.

You can override this restriction by using the **-type hard** option of the **constrain_op** command to manually schedule read and write operations within the loop. If two memory operations both have fixed constraints, the scheduler will treat them as unrelated.

Here are two examples of using the **-type hard** option of the **constrain_op** command, depending on whether the loop is pipelined:

- If the loop is *not* pipelined, and the memory ops have been constrained using the **constrain_op** command, as shown below, the memory dependency between the write and the next read is ignored:

```
constrain_op -type hard -edge .../edges/LOOP_for_begin .../ops/memread_m_ln99
constrain_op -type hard -edge .../edges/W .../ops/memwrite_m_ln99
```

- If the loop *is* pipelined and the memory ops have been constrained using the **constrain_op** command, as shown below, the memory dependency between the write and the next read is ignored:

```
constrain_op -type hard -stage 1 .../ops/memread_m_ln99
constrain_op -type hard -stage 2 .../ops/memwrite_m_ln99
```

This eliminates the following potential problems, related to scheduling a design with such a loop:

- A memory read is forced to be later, or a memory write to be earlier, thereby reducing the scheduling opportunities for other ops in the loop.
- In extreme cases, the design cannot be scheduled without adding a state.

Note It is the designer's responsibility to ensure that array accesses are never to the same address.

M.4.2 Limited Support for Multiple Clocks

CtoS provides limited support for designs with multiple clocks.

If a design has more than one clock, one must be the *base*, and its frequency must be a multiple of all other frequencies.

Also, CtoS will not create circuitry to ensure clock domains are properly interfaced; defining such circuitry is your responsibility.

Note See also [“Create New Design Wizard” on page 6-10](#).

M.5 Known Problems with Workarounds in 14.10

Here are the known problems, with workarounds, at the time of this release.

CCR Number	Known Problem	Workaround
<i>Problems related to Opening and Saving Designs</i>		
534693	When you open a saved design, the working directory might be different from the directory at the time of the save, and you cannot rebuild or use the CtoS GUI's <i>Show in Source</i> functionality.	Use the lcd command to change the working directory.
<i>Problems related to SystemC/C++ Support</i>		
487877	CtoS could issue this error during the build process for designs with internal communication via a user-defined sc_port<IF> whose channel methods and methods that invoke the port method are defined in different compile units (different .cpp files): ERROR (CTOS-11115): undefined port method.	For a workaround, see “Coding Requirements for Multi-Source Designs” on page 6-30 .

CCR Number	Known Problem	Workaround
<i>Problems related to SystemC/C++ Support (cont'd)</i>		
1275630	An internal error may be issued during the build command like the following: <pre>ERROR (CTOS-13112): Internal error while executing inline: bstBehaviorValidator.cpp:674: srcEdge->dominates(useEdge), 'get_lnXX', 'mux_call_lnXX_lnXX'</pre> The problem occurs if the design has all the following: • a function call on an array of (pointer-to) sc_module or sc_port • the function returns a struct (that is not packed or monolithic) • the function is sequential. For example, if you have an array of tlm_fifo<T> where T is a struct and you call my_fifos[i].get(), you will run into this problem.	<i>Workaround 1:</i> Add #pragma ctos packed to the definition of struct T. (undesirable). <i>Workaround 2:</i> Use a non-blocking call instead and do the waiting in the caller. For example rewrite: <pre>data = my_fifos[i].get();</pre> with the following: <pre>while (!my_fifos[i].nb_can_get()) {wait();} my_fifos[i].nb_get(data);</pre>
<i>Problems related to Specifying Micro-architecture</i>		
814763	CtoS could issue this error if you try to convert the implementation of a combinational function into a <i>table lookup</i>, and that function has a multiply, divide, or modulo op with inputs or outputs with more than 64 bits (63 bits if unsigned): <pre>ERROR (CTOS-17031): Cannot convert function '<FUNCTION>' to a lookup table because CtoS had a problem evaluating the function. Error in processing command convert_to_lookup.</pre>	<i>Workaround 1:</i> Decrease the width of the variable declarations (if it will not affect the precision of your result). <i>Workaround 2:</i> Use the split_op command ("split_op" on page E-148) to split large multiplies into smaller multiplies. <i>Workaround 3:</i> Rewrite divides as multiplies by the reciprocal.

CCR Number	Known Problem	Workaround
<i>Problems related to Specifying Micro-architecture (cont'd)</i>		
954405	CtoS could issue this error when trying to unroll loops where the starting point for the iterations is a variable: <pre>ERROR (CTOS-17014): Cannot unroll loop 'loop1_for_begin' because it does not terminate after 512 iterations ...</pre>	Rewrite the loop so the initial value of the iterator is a constant.
<i>Problems related to Allocating Memory and RTL IP</i>		
685020	Only one function per thread should access a memory. If multiple functions access arrays stored in the same memory, the memory access order may not be preserved if two functions in a thread access the same memory.	If multiple functions access arrays stored in the same memory, you must inline these functions until only one function accesses the memory. At this point, the memory accesses will be well-ordered.
685527	Generated RC scripts (File -> Generate -> RTL or write_rc_script command) may not have input and output delays for ports added to interface to memory external to an RTL module using the Exported Memories feature. The delays should be based on the library specified in the IP definitions file.	Add these delay constraints manually to synthesize the RTL separately from the memory.
<i>Problems related to the CtoS Optimizer</i>		
623328	When two unaligned arrays in the same structure are passed to a function, CtoS does not perform sufficient optimizations. Address calculations are not optimized, which may generate dead code in the function, especially if the size of each array is not a <i>power of two</i>.	Add members to align the arrays to a common <i>power of two</i> boundary to simplify the address calculation.

CCR Number	Known Problem	Workaround
<i>Problems related to the CtoS Optimizer (cont'd)</i>		
662227	CtoS sometimes generates larger than expected resources when there are signed array index expressions for a multiple-dimension array.	The problem is subtraction in the array indexing: <pre>int mem[MEM_SIZE_2][MEM_SIZE]; x = mem[i][j-1];</pre> If you make sure the index is positive (assuming <code>MEM_SIZE</code> is a <i>power of two</i>), QoR will improve: <pre>x = mem[i][(j-1) & (MEM_SIZE - 1)];</pre>

CCR Number	Known Problem	Workaround
<i>Problems related to the CtoS Scheduler</i>		
865042	The CtoS scheduler merges two 2-1 muxes to create a 3-1 mux to improve timing (and, to a lesser degree, area). The problem is that it causes CtoS to allocate nine extra registers.	By re-timing the loop, you can prevent CtoS from merging these muxes. The following code is a functionally equivalent form in which CtoS cannot merge the muxes: <pre>// layer 1 void sample::thread0() { rw0 = 0; rd.write(0); while(1){ ////////// WAIT wait(); ////////// register clear if (clear.read()){ rw0 = 0; } ////////// register read if (re.read()){ rd.write(rw0); }else{ rd.write(0); } ////////// register write if (we.read()){ rw0 = wd.read(); } } }</pre>

CCR Number	Known Problem	Workaround
<i>Problems related to Register Allocation/Netlisting</i>		
958054	Extra (unnecessary) registers are allocated to store values computed in an early stage of a pipelined loop, but are used only after the pipelined loop exits.	Rewrite the code for the design with the array as an sc_signal. This change is not simply a matter of changing the declaration, because values written to the array are not visible until the next cycle, so you must ensure there is a wait between a write and a read.
1071739	In some cases, CtoS creates data flow registers instead of using equivalent output register.	If register sharing is required to reduce the number of registers in the design, then following modeling should be used (It may be difficult to achieve in complex code). And, always load and save the value to the output: <pre>for (...) { sc_uint<4> data; wait(); while (...) { data = <expression>; out.write(data); wait(); data = out.read(); } wait(); <another use of data> }</pre> By assigning to 'data' after every wait() CtoS will be able to recognize that 'data' has the same value as 'out' and share the output register.

CCR Number	Known Problem	Workaround
<i>Problems related to Model Generation</i>		
1073840	CtoS may generate an incorrect model when input port name and array name are the same in SystemC. This may occur when the array name is declared in a function and the port is declared at the calling module. This is valid SystemC because the scopes are different. In Verilog, the arrays/memories are declared at the module scope and thus conflict with other names at the module scope.	Change the name of the port or array in the SystemC so that they are not the same.

M.6 Problems Resolved/Enhancements Made in 14.10

The following are the problems resolved and enhancements made, at the time of this release:

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the SystemC/C++ Support</i>		
1073129	Enhancement request for CtoS to support designs that have an array of <code>sc_port<IF></code> or <code>sc_export<IF></code> .	Enhancement request for CtoS to support designs that have an array of <code>sc_port<IF></code> or <code>sc_export<IF></code> .
1133530 1174981 1205224	There are situations where CtoS does not exploit the limited lifetime of local variables and this may lead to unnecessary registers and logic. In the example below, the array inferred from local variable 'vec1' is flattened. <pre> MAIN_LOOP: while (true) { wait(); int vec1[VEC_SIZE]; // Code using vec1 here. .. } </pre> There are situations where CtoS will allocate registers to store the content of the 'vec1' in the state at the top of the body of MAIN_LOOP.	CtoS has been enhanced to analyze the lifetime of local variables. This enables the optimizer to reduce the number of registers needed.
1175631	Enhancement request to support calling a function on an array of module where the array access uses an index that is not a compile-time constant.	CtoS has been enhanced to support a design containing an array of module or array of pointer-to-module, where the design calls functions on an element of the array of (pointer-to) module and the index is not a compile-time constant. For more information see: “Expanded SystemC Support for Array of Module and TLM Ports,” on page 2

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the SystemC/C++ Support (cont'd)</i>		
1217079	According to the CtoS User Guide, initialization of fields of <code>sc_modules</code> are ignored, unless these fields are modules or pointers to modules, or classes containing ports. CtoS issues warning 11130 if it encounters a pointer de-reference for which it cannot determine the variable pointed to by the pointer. However, in case of a de-reference of a pointer variable, where the only assignment to that variable is in the module constructor, message 11130 is not emitted.	The issue has been fixed.
1217532	The build command may crash with the following information: The source context in which this error occurred is as follows: ../src/top.h:37:5: Resolving pointer for construct. This may occur if the design contains a class/struct that - is not an <code>sc_module</code>, and - contains a pointer to a module, and - contains a pointer to a non-module. and the design calls a member function of this struct.	The issue has been fixed.
1219566	The build command with <i>-verbose</i> option produces an info message when CtoS infers a module terminal from an unconnected <code>sc_in/sc_out</code> of a sub module. This information is really useful, and request that this message be printed even without <i>verbose</i> option specified.	CtoS has been enhanced to produce a warning message for such situations regardless of whether the <i>-verbose</i> options of build is specified.
1221539	The build command may issue an internal error if the design contains post-increment or post-decrement expression of a bit-field. Here is an example of struct 'my_struct' that has a bit-field "count" being incremented: <pre>my_struct.count++;</pre>	The issue has been fixed.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the SystemC/C++ Support (cont'd)</i>		
1234291	The build command may issue an internal if the design accesses an enum value of an enumerated type whose definition is nested in a class and the access is through an object of that class. Example: <pre>struct MyS { enum MyE {E1,E2,E3}; .. }; void func() { MyS::MyE e; MyS s; e = MyS::E1; // OK e = s.E1; // Internal error .. }</pre>	The issue has been fixed.
1236121	The header files of the ctos fixed point library contains two definitions of the IWL1 macro which differ only in parentheses.	The definitions of IWL1 macro have been changed to be consistent.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the SystemC/C++ Support (cont'd)</i>		
1244109	The build command issues unclear error message for out-of-range index in static sensitivity involving array of <code>sc_in</code>. <pre>./test.h:11:10: ERROR (CTOS-11253): [SystemC Elaborator] Unsupported event expression.</pre> This may occur when the user has specified an invalid static sensitivity in the SystemC source. Here is an example where ('in[5]') is specified in static sensitivity list and array index '5' is out of range. The error message points at line 11 which is confusing. <pre>1:#include <systemc.h> 2:SC_MODULE(test) { 3: 4: 5: sc_in < bool > in[5]; 6: sc_out < bool > out; 7: SC_CTOR(test) 8: { 9: SC_METHOD(method); 10: sensitive 11: << in[0] 12: << in[1] 13: << in[2] 14: << in[3] 15: << in[4] 16: << in[5] 17: ;</pre>	The build command has been improved to point to the problematic event expression. In the example, the following error will reference line 16. <pre>./test.h:16:8: ERROR (CTOS-11253): [SystemC Elaborator] Unsupported event expression</pre>

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the SystemC/C++ Support (cont'd)</i>		
1258779	The build command may issue the following internal error: <pre>ERROR (CTOS-13110): Internal error while executing build: optPropConst.cpp:2142: widthA == 8 * sizeof(switchVal) (switchVal >> widthA) == 0ull</pre> This may occur if the design has an array of sc_signal/sc_in/sc_out of more than two elements and it uses an expression of type 'bool' to index into the array. Example: <pre>SC_MODULE(DUT) { sc_in<int> ins[4]; void proc() { .. bool ind = ..; .. = ins[ind].read(); } };</pre>	The issue has been fixed.
<i>Resolved Problems and Enhancements Made, related to the Optimizer</i>		
553623 942488 1191320	CtoS does not convert a 'FOR' loop to a DoWhile loop when the 'FOR' statement is followed by a conditional 'IF' statement.	The issue has been fixed. Loops of "FOR" or "WHILE" type which are executed at least once are converted to DoWhile loops.
821415 1122684 1123651	If a pipelined loop reads a signal that is also read in a stall loop of the pipelined loop, then CtoS may not be able to pipeline the loop. For example, error 20137 is generated when pipelining the loop below: <pre>pipe: while(1) { ... stall: while (x.read() && y.read()) wait(); // stall loop if(!(x.read() && y.read())) out.write(...); }</pre>	This restriction has been relaxed.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the Optimizer (cont'd)</i>		
1118643	When merge loops transform is applied to two nested loops, the surviving loop might have the wrong number of unrolled iterations.	The issue has been fixed.
1231245	The allocate_prototype_memory command may fail with the following error message: <pre>ERROR (CTOS-20397): Array 'rmb' is read by 2 processes, and only 1 read and 0 read-write interfaces were specified as command options.</pre> This can happen when partial write is used and there is a mux op between the memread op and the partial write op.	The memread is optimized out when the mux fully covers all partial writes options (for example, the mask width equal the control width of the mux).
1250802	CtoS shows the following WARNING message. <pre>WARNING (CTOS-17143): Combinational method 'sample_proc0' has a non-trivial CFG. Ensure that nets are written under all conditions.</pre> This warning message is issued for a combinational method which has a non-trivial CFG. In this particular case, CtoS might not be able to reason that all writes to nest are covered for all conditional paths. Therefore, the user should ensure that writes to nets are covered for all conditional paths.	CtoS is now able to reason that nets are written on all conditional paths.
<i>Resolved Problems and Enhancements Made, related to the Timing Analyzer</i>		
1081150	Enhancement request to upgrade CtoS to use RC 13.1. This RC version supports noise characterization in cells in technology library.	RC has been upgraded to 13.1. For more information see: Chapter 3 “Prerequisites for CtoS and for this User Guide”.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the Timing Analyzer (cont'd)</i>		
1204892	The report_area, report_timing and report_summary commands may result in the following internal error: <pre>ERROR (CTOS-13110): Internal error while executing report_area: Stream.cpp:128: OS error 'Bad file descriptor' (code 9)</pre> This may occur when machine has resource issues preventing CtoS from invoking another process.	Instead of an internal error, CtoS issues error 1071 which shows the executable that could not be started and the OS error.
<i>Resolved Problems and Enhancements Made, related to Pipelining</i>		
903820	An enhancement request to add a warning when CtoS scheduler reaches the user-specified max latency of a pipelined loop.	A new warning, CTOS-20573, has been added to inform the user when CtoS scheduler reaches the user-specified max latency of a pipelined loop.
1189480 1208037	The scheduler may complete with negative slack on a pipeline loop. This may occur even if the user has specified a large <i>max_lat_interval</i> for the pipelined loop.	The pipeline_loop command has a new option <i>-extra_timing_effort</i> that causes the scheduler to perform an exhaustive search within the user-specified latency range. For more information see: “Extra Timing Effort Option to Pipeline with Minimal Latency,” on page 8.
1238735	The scheduler may issue the following internal error: <pre>ERROR (CTOS-13111): Internal error while executing schedule: cfgPipelineValidator.cpp:772: srcOp->getResBinding()->getStage() == 1, 'read_myarr_lnXX'</pre> This error occurs if you specify a net to both float and be part of the expand_before for a pipeline loop. CtoS should issue a user-level error that this is an invalid combination.	The problem has been resolved. CtoS issues user error when floating nets overlap with the expansion nets of a pipeline loop.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Pipelining (cont'd)</i>		
1249257	The pipeline_loop command issues an internal error: <pre>ERROR (CTOS-13110): Internal error while executing pipeline_loop: ../util/HashTbl.h:490: !E() (m_recs[loc].key(), key)</pre> This may occur on a loop with deeply nested non-stall loops. Instead of issuing an error that this is not supported, it crashes.	The pipeline_loop command has been improved to issue an error message: <pre>ERROR (CTOS-20141): Loop 'main_loop_while_begin ' because it has a deeply nested stall loop 'WAIT_B_GET_do_begin'.</pre>
1252591	CtoS is not able to pipeline a loop that has the exit branch (k<lat) immediately adjacent to an internal branch (k<fsize). <pre>foreach_pixel: for (unsigned k = 0; k<lat; ++k, ++i) { if (k<fsize) { advance(); } }</pre>	The problem has been fixed.
1254200	The schedule command may issue the following internal error: <pre>ERROR (CTOS-13110): Internal error while executing schedule: cfgPipelineValidator.cpp:314: numStallEnableNets == numStages</pre> This may occur when scheduling a design with multiple behaviors. If one of the behaviors fails to schedule and another behavior has a pipelined loop, then the schedule command may get an internal error while folding the pipelined loop.	The problem has been fixed.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Pipelining (cont'd)</i>		
1277011	The schedule command may issue an internal error: <pre>ERROR (CTOS-13110): Internal error while executing schedule: cfgEdge.cpp:1289: edgeBelow->m_ctlVal->getSrc() == NULL</pre> This may occur if there is no wait after the last stall loop. <pre>PIPE: while (true) { ... wait(); while (stall.read() == false) { STALL: wait(); } }</pre>	If there is no wait after the last stall loop, CtoS will add a state at the loop end if possible. Otherwise, CtoS issues a user error that there are not enough states.
<i>Resolved Problems and Enhancements Made, related to the Scheduler</i>		
935695	The CtoS scheduler with option <i>-post_optimize advanced</i> may produce worse QoR then without specifying the advanced option.	The issue has been fixed.
1210376	In relaxed latency mode, the create_required_states command may fail to add the required number of states when two ops with non-zero latency are born on the same edge. In this particular case, two pipelined function calls each with latency greater than 1 are born on the same edge.	CtoS now recognizes this situation and automatically adds a state between these ops to allow the create_required_states command to complete successfully.
1221511	The schedule command may issue the following internal error: <pre>ERROR (CTOS-13110): Internal error while executing schedule: dfgOp.cpp:1536: originEdge->canReach(edge, true, true)</pre> This may occur when reset ops are combinationally reachable from the origin.	The issue has been fixed.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the Scheduler (cont'd)</i>		
1223758	The schedule command may issue the following internal error: <pre>ERROR (CTOS-13110): Internal error while executing schedule: uarchPipeliner.cpp:153: nAdded</pre> This may occur on a design with a pipeline loop with a tight upper bound on latency and $ii > 1$. This error happens when the scheduler tries to add a stage to a pipeline loop but it cannot do this because adding a stage would exceed the user-specified <i>max_lat_interval</i>.	The problem has been resolved.
1229311	The schedule command issues the following warning message when no ops are scheduled in a state: <pre>WARNING (CTOS-20406): A user defined state 'Wait_ln13' at sample.cpp:13:7 does not contain any scheduled operations and is redundant.</pre> This is useful to identify redundant states but should be skipped for states in a protocol region.	The warning is not issued if the empty state is within a protocol region.
1230472	The schedule command when invoked with <i>scheduling_effort=medium</i> might result in slack degrading to negative when instances are shared.	The scheduler was improved to keep the slack degradation at minimum. The degradation still happens because the scheduler is not modelling muxes faithfully (assumes 2-input mux to be a fair representation of the sharing mux).
1235246	The schedule command issues an error message when using an external memory with a delay larger than a clock cycle even though the memory is registered at the input and output.	The scheduler was enhanced to identify the case of registered external memories with large delays and ignoring them when considering the delay of resources

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to the Scheduler (cont'd)</i>		
1251142	The schedule command may fail with the following error messages: <pre>ERROR (CTOS-20324): Scheduling failed because 'myfunc_ln16' has delay 3519 which exceeds the clock cycle 3200 minus setup and launch delay of the flip-flop. Consider splitting the op or increasing clock cycle to 3618. ERROR (CTOS-13112): Internal error while executing schedule: cfgPipelineValidator.cpp:727: srcOutStage == stage, 'lt_ln444', 'add_ln444'</pre> When the scheduler finds an operation with a delay larger than a clock cycle it stops and leaves the database in inconsistent state leading to the internal error.	The problem has been resolved. The scheduler was improved so that issuing error 20324 does not result in an inconsistent state.
1252014	The timing for sequential function is not closed in relaxed latency mode.	The scheduler was improved to add states to the final edge of the function when it sees the potential timing violations due to back to back calls of the same function.
1253331	The schedule command should display the total number of ops in a behavior in the output messages when scheduling a behavior.	The scheduler now reports the total number of ops in a behavior as part of the behavior settings portion of the scheduler output messages.
<i>Resolved Problems and Enhancements Made, related to Allocating Registers</i>		
1235912	The allocate_registers command may issue internal error: <pre>ERROR (CTOS-13111): Internal error while executing allocate_registers: nbModuleNetlist.cpp:445: !enable->isLogic1() && !enable->isLogic0(), 'PPFtoPC_MDEN'</pre> This may occur when a signal is reset by a process that does not write to this signal during normal operation.	The problem has been resolved.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Allocating Registers (cont'd)</i>		
1240058	The allocate_register command may issue the following internal error: <pre> ERROR (CTOS-13111): Internal error while executing allocate_registers: nbCombProcNetlist.cpp:533: edge->getSrc()->isState(), 'FLEX_NB_GET_START' </pre> This may occur when a clocked SC_METHOD includes protocol region.	The problem has been resolved.
1263965	Register allocation may take a very long time for designs with a write op that is scheduled on an edge that has many combinationally adjacent edges. This can be the result of unrolling loops with complex control structures.	The problem has been resolved.
<i>Resolved Problems and Enhancements Made, related to Reports</i>		
1076320	Enhancement request to improve reporting of Strongly Connected Components (SCCs).	A new command 'report_sccs' has been added to CTOS. The report specifies the variable that is critical for a given SCC, along with all the ops in the SCC. For more information see: “New Strongly Connected Component (SCC) Report,” on page 8.
<i>Resolved Problems and Enhancements Made, related to Model Generation</i>		
1096733	CtoS generates un-referenced variable <code>CtoS_rtl_<module_name></code> This variable is generated when the <code>-tcf</code> option is used in the write_rtl command. The presence of this variable is used by the read_tcf command to determine whether the TCF is from the simulation model or the RTL model.	The RTL netlist that is generated by CtoS no longer contains unused variables that are related to variable for the state machine or unused variables that originate from control values for tags.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Model Generation (cont'd)</i>		
1150685	Enhancement request to generate a verification wrapper in verilog that enables verification of the Input SystemC model in a Verilog testbench.	CtoS has been enhanced to generate a verilog verification wrapper. For more information see, “Support Simulation of SystemC within Verilog Testbench” on page M-9.
1199386	The generated RTL may unnecessarily cast the signed argument with \$signed cast. Lint complains about this unnecessary casting.	The issue has been resolved to apply the \$signed casting only when required.
1237183	The generated RTL may contain unused registers with names similar to pattern Wait_In*.	The unused registers are not included in the generated RTL.
1243583	A linting tool may complain about a bit-width mismatch. This may occur when design contains constant arithmetic ops because an extra bit is inadvertently used to represent the signedness of the op.	The issue has been resolved
1252022	The write_rtl command can result in the following internal error <pre>ERROR (CTOS-13110): Internal error while executing write_rtl: mgVlBuilder.cpp:342: name != "Const_0" && name != "Const_1"</pre> This may occur when the design has nets that are reset but not written by one process and written by other processes.	The issue has been resolved.
1276429	The write_rtl command may issue the following internal error: <pre>ERROR (CTOS-13110): Internal error while executing write_rtl: ../util/HashTbl.h:708: find(key, data)</pre> This issue is related to writing contention checker for arrays that are shared between modules.	The issue has been resolved.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to ECO</i>		
1051764	Enhancement request for ECO mode to not reuse existing resources for new ops. Instead new ops should be mapped to new resources.	A new design attribute, eco_sharing_policy, controls the use of resources for new ops in ECO mode. For more information see: "Reduce ECO Differences from New Resources," on page 9.
1225172	When running ECO mode, CtoS may show differences even when the SystemC input did not change. There may be some differences in order register declaration and n statements in process bodies. The statement order may occur when there are multiple behaviors in a module and the order of scheduling these behaviors is different.	The problem has been resolved reducing the differences after scheduling with ECO mode.
1226421	Register allocation reduces the number of registers by sharing a tag control register with equivalent non-tag control values. However, in an ECO this sharing is not done, resulting in duplicate registers.	This problem has been resolved by having ECO register allocation perform the same sharing as the original.
1226930	If an op generated multiple tag control values, they would be stored in one multi-bit register. During ECO, the register would be copied, but renamed with suffix like '%0', which would cause RTL syntax errors because the variable name would be 'var%0'.	The problem has been resolved.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to GUI</i>		
1157484	Enhancement request to focus Micro Arch Manager on a single process.	The Micro Architecture Manager can now be invoked on a single behavior. Using the Hierarchy Window, click right-mouse-button on a behavior there is a new menu item 'Specify Micro-Architecture'. For more information see: “Enhanced MicroArch Manager Enables You to Focus on a Process,” on page 4.
1239199	Arrays in SC_METHOD must be flattened. Micro Architecture Manager doesn't show this array as requiring action.	Arrays in SC_METHODS will be marked as required in MicroArch Manager because they need to be flattened.

CCR Number	Problem/Enhancement	Resolution
<i>Resolved Problems and Enhancements Made, related to Libraries</i>		
1214835	CtoS is unable to pipeline a loop that uses the non-blocking interfaces of a <code>tlm_fifo</code> .	The four fifos in <code>ctos/include/ctos_tlm</code> have been enhanced to allow pipelining loops that use the non-blocking interfaces. As part of the rewrite, there are now four more functions that need to be inlined: <code>_reset_r</code>, <code>_reset_w</code>, <code>is_full_nb_can</code>, <code>is_empty_nb_can</code>. The <code>tlm_fifo_1t</code> and <code>tlm_fifo_reg_1t</code> versions require inlining the following 2 additional functions: <code>_is_busy_w_nb_can</code>, <code>_is_busy_r_nb_can</code>. There is a small area overhead for designs that use the non-blocking interfaces. For designs that use only the blocking interfaces there is no QoR impact.
<i>Resolved Problems and Enhancements Made, related to Extensions</i>		
1239218	The TCL extension <code>write_nonbus_rtl</code> does not generate the nonbus RTL when the top module has the flexChannel ports.	The problem has been fixed.
1250586 1250588	The TCL extension <code>write_nonbus_rtl</code> producing ports in different order than that produced by ' <code>write_rtl</code> ' command.	The port order is now consistent with that generated by ' <code>write_rtl</code> ' command.

Release Notes for 14.10

Problems Resolved/Enhancements Made in 14.10

N Glossary

This glossary contains terms and acronyms found in this book.

Wikipedia, The Free Encyclopedia, and *Understanding Behavioral Synthesis*, by John Elliott, were both used as references for some of these terms.

Note about SystemC Functions

One of the focuses of this glossary is the various types of processes in SystemC.

Processes may call different types of functions in SystemC, as follows:

Functions

Process : declared with macro `SC_THREAD`, `SC_CTHREAD` or `SC_METHOD`

Thread Process - `SC_THREAD` : process with waits

Clocked Thread Process - `SC_CTHREAD` -> thread process sensitive only to clock

Method Process - `SC_METHOD` : process without waits

Clocked Method Process -> method process sensitive only to clock

Combinational Function

Term	Expanded or Additional Terms	Definition
architecture		An <i>architecture</i> is a high-level view of a computer design – essentially the conceptual view of how a design will be implemented in hardware.
array		In CtoS, an <i>array</i> is defined as an object in the database (as contrasted with a “ memory ”, which is defined as a resource). Arrays appear in the input source and in the “ CFG ”.
array dependency		In CtoS, <i>array dependencies</i> describe reads and writes that must be <i>ordered</i> . The “ create_array_dependencies ” command generates initial dependencies (read-write, write-write), but you can break them with the “ break_array_dependency ” command. See also “ Resolving Arrays (Memories) ”.
backward edge		In CtoS, a <i>backward edge</i> represents the transition of a loop to the next iteration in the “ CFG ” (the opposite of a backward “ edge ” is a “ forward edge ”).
behavior		In CtoS, a <i>behavior</i> is the internal representation of a member function (or function). A behavior can be either a “ process behavior ” (“ method process ” or “ thread process ”) or a non-process behavior (“ combinational function ” or “ sequential function ”).
birthday		In CtoS, <i>birthday</i> refers to the original state of a design – how it is defined in the original source code.
body unrolling		In CtoS, <i>body unrolling</i> occurs when the body of a loop is copied for a specified number of times while execution is still kept in a loop. See also “ Unrolling Loops ” on page 8-17.
call-busy loop		See “ function call loop ”.
CAM	content-addressable memory	<i>CAM (content-addressable memory)</i> is a type of storage device with comparison logic in each bit of storage. A data value is broadcast to all words of storage and compared with stored values. Words that match are flagged, so subsequent operations can work on them. For example, they can be read out individually or the bit positions of all of them can be written to. A CAM can thus operate as a data parallel (SIMD) processor and is often used in high-speed search applications.

Term	Expanded or Additional Terms	Definition
CDFG	control and data flow graph	A <i>CDFG (control and data flow graph)</i> is the convergence of a “CFG” and a “DFG”. Like a CFG, a CDFG is a directed graph, similar to a flowchart, containing “node”s, which are “state”s and control points in the source code, and “edge”s, which are transitions between nodes. However, in a <i>CDFG</i> , edges can be expanded to show the DFG on that edge.
CFG	control flow graph	A <i>CFG (control flow graph)</i> is a directed graph, similar to a flowchart, containing “node”s, which are “state”s and control points in the source code, and “edge”s, which are transitions between nodes.
CGIC	clock gating integrated cell	A clock-gating integrated cell (CGIC) integrates the various combinational and sequential elements of a clock gate into a single library cell. For more details on CGIC, refer to “ <i>Low Power in Encounter RTL Compiler</i> .”
CLB	configurable logic block	A <i>configurable logic block</i> is the basic logic unit in an FPGA. Exact numbers and features vary from device to device, but every CLB consists of a configurable switch matrix with 4 or 6 inputs, some selection circuitry (MUX, etc.), and flip-flops.
clock		A <i>clock</i> is a “signal” that represents the time a wave stays at a high or low state. The rising and falling edges of a clock square wave trigger the activity of the circuits.
clocked method process (SystemC)	<i>also referred to as an RTL process or a clocked SC_METHOD</i>	A <i>clocked SystemC method process</i> is a “method process” whose sensitivity list consists of either one or two edge events, one of which is interpreted by CtoS to be the “clock” event. CtoS interprets such a process to model “sequential logic”.
clocked SC_METHOD (SystemC)		See “clocked method process”.

Term	Expanded or Additional Terms	Definition
clocked thread process (SystemC)		A <i>clocked SystemC thread process</i> is a “ thread process ” whose static sensitivity consists of a single edge event. It can be created using either the SC_CTHREAD macro, or else the SC_THREAD macro together with the sensitive construct for specifying the “ clock ” event.
combinational function		A <i>combinational function</i> is a function that contains only “ combinational logic ”.
combinational logic		<i>Combinational logic</i> is a type of logic circuit whose output is a pure function of the present input only. This is in contrast to “ sequential logic ”, in which the output depends not only on the present input, but also on the history of the input.
combinational method process (SystemC)	<i>also referred to as a combinational process or a combinational SC_METHOD</i>	A <i>combinational SystemC method process</i> is a “ method process ” whose sensitivity list does not contain any positive edge or negative edge events. CtoS interprets such processes to model purely “ combinational logic ”.
combinational process (SystemC)		See “ combinational method process ”.
combinational SC_METHOD (SystemC)		See “ combinational method process ”.
combinationally adjacent		Two “ CDFG ” “ edge ”s are considered <i>combinationally adjacent</i> if there is a path from one edge to the other that does not include a state “ node ”.
combinatorial logic		See “ combinational logic ”.
constraint		A <i>constraint</i> is a user-specified synthesis control expressed as a “ value ” or a set of values for a particular parameter, such as maximum area or delay.
control op	control operation	In CtoS, a <i>control op</i> is an “ op ” that is part of the control “ state machine ” generated during synthesis.

Term	Expanded or Additional Terms	Definition
control state		A <i>control state</i> is a “state” in the “state machine” generated during synthesis.
critical endpoint		A <i>critical endpoint</i> is the endpoint with the least slack.
critical path		A <i>critical path</i> is the path within a design that dictates the fastest time at which the design can run.
custom op		In CtoS, a custom_op is an “op” that represents the call to a user-defined function in the SystemC code.
dereference		A <i>dereference</i> is an access to something to which a pointer points, that is, you are following the pointer.
design		In CtoS, a <i>design</i> corresponds to a container used to encapsulate “module”s in the source file(s).
DFG	data flow graph	A <i>DFG</i> (<i>data flow graph</i>) is a directed graph containing “node”s that are “op”s, and “edge”s, which are dependencies (for example, “value”s) transferred between ops.
DUT	design under test	A <i>DUT</i> (<i>design under test</i>) is the particular piece of source code on which testing is focused. This could be in contrast to a “REF” (reference model), used for comparison purposes.
dynamic class		<i>Dynamic class</i> is a class requiring a virtual table pointer, because it or its bases have one or more virtual member functions or virtual base classes. See also “multiple inheritance”, “single inheritance”, “virtual inheritance”.
ECO	engineering change order	An <i>ECO</i> (<i>engineering change order</i>) is a late change in a design specification after implementation has begun. <i>Incremental Synthesis</i> can minimize the size of an ECO. See also “Incremental Synthesis” on page 17-1.

Term	Expanded or Additional Terms	Definition
edge		In CtoS, an <i>edge</i> represents different things: • In the “CFG”, an <i>edge</i> represents a transition between “node”s. • In the “DFG”, an <i>edge</i> represents a dependency (for example, a “value”) transferred between “op”s. Unless otherwise indicated, the term, <i>edge</i>, will usually refer to a <i>CFG edge</i> in CtoS. See also “backward edge” and “forward edge”.
fan-in		<i>Fan-in</i> is the number of inputs of an electronic logic gate. Logic gates with a large fan-in can be slower than those with a small fan-in, as the complexity of the input circuitry increases input capacitance of the device. See also “fan-out”.
fan-out		<i>Fan-out</i> is a measure of the ability of a logic gate output, implemented electronically, to drive a number of inputs of other logic gates of the same type. Logic gates are usually connected to form more complex circuits; often, one logic gate output is connected to several logic gate inputs. See also “fan-in”.
Flop Map		A <i>Flop Map</i> is a set of <i>hints</i> that indicate to Calypto’s “SLEC” which flip-flops in the specification are equivalent to which flip-flops in the implementation. Providing such hints usually dramatically improves SLEC performance.
forward edge		In CtoS, a <i>forward edge</i> is any “edge” in the “CFG” that is <i>not</i> a “backward edge”; in other words, a forward edge does <i>not</i> represent the transition of a loop to the next iteration in the CFG.
fork node		A <i>fork node</i> is a “node” that splits an incoming control flow or object flow into multiple control or object flows.
FSM	finite state machine	See “state machine”.
function call from a thread process		See “thread process”.
function call loop		In CtoS, a <i>function call loop</i> is a loop that CtoS adds to the “CFG” of the calling behavior to manage a call to a “thread process”.

Term	Expanded or Additional Terms	Definition
HDL	hardware description language	An <i>HDL (hardware description language)</i> is a high-level language used to describe the features and functionality of chips and systems prior to handoff to the IC layout process. HDL descriptions are used in both design implementation and verification flows. Examples of HDLs are “ Verilog ” and “ SystemC ”.
INCISIV	Incisive Enterprise Simulator	<i>INCISIV (Incisive Enterprise Simulator)</i> combines Specman testbench automation, the multi-language Incisive Design Team Simulator, SimVision, Incisive Desktop Manager, Verification Builder, Scenario Builder, eAnalyzer, and Plan-to-Closure Methodology in a single optimized package.
II	initiation interval	<i>II (initiation interval)</i> is the number of cycles required for one “ pipeline stage ” to complete (except the last).
inlining		<i>Inlining</i> is a process by which a function’s contents are synthesized as if the contents had been copied into the body of a “ process ” in which the function was used. See also “ Inlining Functions ” on page 8-3.
instance		In CtoS, an <i>instance</i> corresponds to a reference of another “ module ” that is instantiated in this module.
instance terminal		In CtoS, an <i>instance terminal</i> corresponds to a reference of a master “ terminal ” in an “ instance ”.
integral type		<i>Integral types</i> collectively refer to these C++ types: - signed integer types (signed char, short, int, long) - unsigned integer types (unsigned char, unsigned short, unsigned int, unsigned long) char and wchar_t bool
join node		A <i>join node</i> is a “ node ” in the “ CFG ” at which “ edge ”s are coming together. CtoS uses these nodes as the <i>signature</i> of a combinational loop. See also “ loop join node ”.
LI	latency interval	<i>LI (latency interval)</i> is the total number of cycles required for information to pass the pipeline.

Term	Expanded or Additional Terms	Definition
lifetime		In CtoS, <i>lifetime</i> refers to the interval between the generation and the last usage of a “ value ”.
loop body		The <i>loop body</i> is the set of statements to be performed iteratively with the range of a loop.
<i>loop_join_id</i>		In CtoS, for the command argument <i>loop_join_id</i> , you may insert the object id of any node whose type is join and whose is_loop value is true . See also “ Node Object Attributes (Nodes) ” on page D-50.
loop join node		A <i>loop join node</i> is a “ join node ” that represents a loop.
loop unrolling		<i>Loop unrolling</i> is a process by which the statements in a loop are copied for as many times as they would have been executed in a design. See also “ Unrolling Loops ” on page 8-17.
loop variable		In CtoS, a <i>loop variable</i> is a “ variable ” declared in the init expression of a for loop.
LUT	lookup table	An <i>LUT (lookup table)</i> is a data structure, usually an “ array ” or associative array, which replaces a runtime computation with a simpler array indexing operation.
memory		In CtoS, a <i>memory</i> is defined as a “ resource ” (as contrasted with an “ array ”, which is an object in the database). A memory is an <i>implementation</i> of an array.
method process (SystemC)	declared as SC_METHOD	A <i>SystemC method process</i> is a “ process ” that, when activated, executes from the function associated with the method process from the beginning until the end (return) without interruption. Method processes do not maintain state between activations. There are two types of method processes: “ combinational method process ” and “ clocked method process ”.
micro-architecture		The <i>micro-architecture</i> is a lower-level, more detailed description of a “ design ” (as contrasted with “ architecture ”) that starts to take into effect how the parts of a design will work with each other.
mobility		See “ op span ” (this is how CtoS refers to <i>mobility</i>).

Term	Expanded or Additional Terms	Definition
module		In CtoS, a <i>module</i> corresponds to a “Verilog” module or an SC_MODULE class in SystemC.
multi-cycle op		A <i>multi-cycle op</i> is an “op” that takes more than one cycle to execute, for example, a synchronous read or a pipelined RTL IP op.
multiple inheritance		<i>Multiple inheritance</i> is class inheritance where a class may have more than one direct base class. See also “dynamic class”, “single inheritance”, “virtual inheritance”.
multiple writer variable or net		A <i>multiple writer variable or net</i> is a “variable” or “net” that is written by more than one “process”, which requires CtoS to generate special arbitration logic.
NC-SC	native-compiled SystemC	NC-SC is the Cadence native-compiled “SystemC” simulator.
NC-Verilog	native-compiled Verilog	NC-Verilog is the Cadence native-compiled “Verilog” simulator.
net		In CtoS, a <i>net</i> is an object in the “RTL” and the CtoS database. Some nets, but not all, are inferred from a “signal” in the source code. A net may have reads and/or writes. See also “shared variable or shared net”, “multiple writer variable or net”, “value”, “variable”.
netlist		A <i>netlist</i> is a list of logic gates and their interconnections, which make up a circuit.
node		In CtoS, there are two definitions for a <i>node</i> : - In the “CFG”, a <i>node</i> represents a “state” or control point in the source code. - In the “DFG”, a <i>node</i> represents an “op”. CtoS has several distinct types of nodes: “fork node”, “loop join node”, “origin node”, and “simple node”.
op	operation	In CtoS, an <i>op</i> or <i>operation</i> is an operator in the source code (+, -, *, ~, &, etc.) represented as a “node” in the “DFG”. Ops are the instruction set used by CtoS and, during the synthesis process, are generally mapped to hardware “resource”s (adder, multiplier, mux, etc.).

Term	Expanded or Additional Terms	Definition
op slack		See “ op span ” (this is how CtoS refers to <i>op slack</i>).
op span	operation span	In CtoS, an <i>op span</i> is a set of “ edge ”s on which an “ op ” can be scheduled without violating any causality constraints.
operand		In CtoS, an <i>operand</i> corresponds to a set of “ value ”s connected to an input “ port ” of an “ op ” in a “ DFG ”.
origin node		The <i>origin node</i> is the top “ node ” in the “ CFG ”.
partial unrolling		In CtoS, <i>partial unrolling</i> occurs when a loop is unrolled only partially, and the remaining executions are kept in the loop. See also “ Unrolling Loops ” on page 8-17.
path		A <i>path</i> is defined in terms of the start and end “ instance ”s. The <i>number of paths</i> is reported in terms of the number of distinct pairs of start and end instances of the paths (as contrasted with “ region ”, which is a set of “ edge ”s between two designated “ CFG ” “ node ”s, called a <i>start</i> and an <i>end</i>).
performance model		See “ timed functional model ”.
phase of a pipeline		Each cycle of a “ pipeline stage ” is known as a <i>phase of a pipeline</i> stage. The number of phases in a pipeline is given by the pipeline's “ II ” (initiation interval).
pin		In CtoS, a <i>pin</i> corresponds to a pin in a “ port ” of an “ op ”.
pipeline depth of RTL IP		The <i>pipeline depth of RTL IP</i> is the total number of cycles after which output is valid, after input is available. For example: depth 0 means combinational; depth 1 means output is available in the next cycle; depth 2 means output is available in the cycle after next.
pipeline stage		A <i>pipeline stage</i> is a set of logic “ op ”s considered as one logical processing step of a pipeline.
pipelining		Loop <i>pipelining</i> is a technique in which the executions of the “ op ”s in a loop are overlapped in the resulting implementation to improve performance.

Term	Expanded or Additional Terms	Definition
POD, PODS	plain old data, plain old data structures	<i>POD (plain old data)</i> or <i>PODS (plain old data structures)</i> are data structures represented only as passive collections of field values, without using encapsulation or other object-oriented features. See also en.wikipedia.org/wiki/Plain_Old_Data_Structures .
POD-struct		A <i>POD-struct</i> is an aggregate class that has no non-static data members of type non-POD-struct, non-POD-union (or array of such types) or reference, and has no user-defined copy assignment operator and no user-defined destructor.
port		In CtoS, a <i>port</i> corresponds to a port of an “ op ”, as contrasted with a “ terminal ”, which corresponds to a terminal in a “ module ”.
process (SystemC)		A <i>SystemC process</i> is a function declared with macro SC_THREAD or SC_CTHREAD (“ thread process ”) or SC_METHOD (“ method process ”). A process is activated (by the SystemC kernel) when an event in the sensitivity of that process occurs.
process behavior		In CtoS, a <i>process behavior</i> is a “ behavior ” that represents a C++ member function (or function) declared as a SystemC “ method process ” or “ thread process ”.
PROM	programmable read-only memory	<i>PROM (programmable read-only memory)</i> is a type of ROM that can be written using a PROM programmer. The memory can be programmed only <i>once</i> after manufacturing by <i>blowing</i> the fuses, an irreversible process. Blowing a fuse opens a connection, while blowing an anti-fuse closes a connection. Programming is done by applying high-voltage pulses not encountered during normal operation.
PSV	pipeline stage vector	A <i>PSV (pipeline stage vector)</i> is the vector that keeps track of the active stages in a pipeline. The suffix psv is used to name this type of pipelined object.
QoR	quality of results	<i>QoR (quality of results)</i> is a term used in evaluating technological processes. It is generally represented as an aggregate of weighted factors, such as area, time, etc.

Term	Expanded or Additional Terms	Definition
RAM	random access memory	<i>RAM (random access memory)</i> is a type of computer data storage. The word <i>random</i> refers to the fact that any piece of data can be returned in a constant time, regardless of its physical location and whether it is related to the previous piece of data. In CtoS, a RAM is a “memory” that can be read and written.
REF	reference model	A <i>REF (reference model)</i> is used for comparison purposes with a “DUT”.
RC	Encounter RTL Compiler	<i>RC (Encounter RTL Compiler)</i> is Cadence’s synthesis tool. RC offers a unique set of patented global-focus algorithms to enable optimization of timing, area, and power.
region		In CtoS, a <i>region</i> is a set of “edge”s between two designated “CFG” “node”s, called a <i>start</i> and an <i>end</i>, (as contrasted with “path”, which is defined in terms of the start and end “instance”s, and the <i>number of paths</i> is reported in terms of the number of distinct pairs of start and end instances of the paths).
register		A <i>register</i> stores bits of information so that all bits can be written to or read out simultaneously. Signals from a “state machine” to the register control when registers transmit to, or accept information from, other registers. State machines can route information from one register, through a functional transform, such as an adder unit, to another register that stores the results.
register allocation		<i>Register allocation</i> is the process of multiplexing many target program “variable”s onto a small number of CPU “register”s, with the goal of keeping as many operands as possible in registers to maximize execution speed. Register allocation can occur over a basic block (<i>local register allocation</i>), over a whole function/procedure (<i>global register allocation</i>), or in-between functions as a calling convention (<i>interprocedural register allocation</i>).
register binding		In CtoS, a register binding identifies the use of a “register” in a “behavior”, to store a “value” in a specific “state” under a given “tag”.

Term	Expanded or Additional Terms	Definition
registered memory		A <i>registered memory</i> is a “memory” whose outputs on reads are buffered using one or more “register”s.
reset behavior		In CtoS, the <i>reset behavior</i> of a “process” consists of the fragment of the “DFG” that is born on a “reset path”. The reset behavior of a process specifies what happens when the process undergoes reset.
reset condition		In CtoS, a <i>reset condition</i> of a “process” consists of three elements: a “reset signal”, the active level of the reset signal, and whether the reset condition is synchronous or asynchronous. A reset condition of a process specifies when a process undergoes reset. A process can have more than one reset condition.
reset path		In CtoS, a <i>reset path</i> of a “process” is a path in the “CFG” of that process from the “origin node” to a “reset state”.
reset signal		In CtoS, a <i>reset signal</i> of a “process” is a “signal” that appears in one of the “reset condition”s of that process.
reset specification		In CtoS, a <i>reset specification</i> of a “process” consists of a set of “reset condition”s and a “reset behavior”.
reset state		In CtoS, a <i>reset state</i> of a “process” is a “state” in the “CFG” of the process that is reachable from the “origin node” via a combinational path.
reset value		See “start value” (CtoS has no concept of a <i>reset value</i>).
resource		In CtoS, a <i>resource</i> is any collection of gates that perform a well-defined task, such as multipliers, adders, and subtractors. Additional resources include “register”s for storing commonly needed “value”s; “memory”s to hold larger numbers of values; as well as individual gates of a library, such as NANDs, NORs, or multiplexers, that implement control logic.
resource binding		In CtoS, a <i>resource binding</i> is the binding of a “resource” in a “behavior”, which is the binding of a resource to a specific “op” for a specific “state”.

Term	Expanded or Additional Terms	Definition
ROM	read-only memory	<i>ROM (read-only memory)</i> is a type of computer data storage in which program instructions, operating procedures, or other data are permanently stored, generally on electronic chips during manufacture, and that ordinarily cannot be changed by a designer.
RTL	register-transfer level	<i>RTL (register-transfer level)</i> is a type of “HDL” in which a circuit is modeled by specifying the data flowing between a set of “register”s, which are elements of a “design” that transition between “state”s based on an event (a high or low “edge”) occurring on a “clock” “signal”. The level of abstraction of RTL is above the gate level and below the behavioral level.
RTL process		See “clocked method process”.
SCC	strongly connected component	An <i>SCC (strongly connected component)</i> is a member of a set of “op”s for which it holds true that you can reach every op from any other op of the set.
scheduling		<i>Scheduling</i> is the process by which CtoS assigns data “op”s to “clock” cycles so “constraint”s and other requirements are satisfied.
sequential function		A <i>sequential function</i> is a function in which “state”s were added during the synthesis process.
sequential logic		<i>Sequential logic</i> is a type of logic circuit in which the output depends not only on the present input, but also on the history of the input. This is in contrast to “combinational logic”, whose output is a pure function of the present input only.
shared variable or shared net		A <i>shared variable</i> or <i>shared net</i> is a “variable” or “net” that is read and/or written by more than one “process”. Shared variables or shared nets are data members of a “module”. A special case of a shared variable or shared net is a “multiple writer variable or net”.
signal		In CtoS, a <i>signal</i> is a “variable” defined as an sc_signal , sc_in , or sc_out . A signal is implemented as a “net” in the final design. See also “shared variable or shared net”, “value”.

Term	Expanded or Additional Terms	Definition
single inheritance		<i>Single inheritance</i> is class inheritance with the restriction that every class has at most one direct base class. See also “dynamic class”, “multiple inheritance”, “virtual inheritance”.
simple node		A <i>simple node</i> is a “node” that has a single input and a single output.
SC_METHOD (SystemC)		See “method process”.
SC_THREAD or SC_THREAD (SystemC)		See “thread process”.
slack		The <i>slack</i> for a “control state” is defined as the difference between the “clock” period and the maximum delay of the paths that are sensitizable at the “state”.
SLEC	Sequential Logic Equivalence Checker	Calypto’s <i>SLEC (Sequential Logic Equivalence Checker)</i> formally verifies the equivalence of system-level models and RTL designs.
speed grade		In CtoS, <i>speed grade</i> is a way to trade off area with respect to timing of individual “resource”s. A speed grade of 100 (the default) forces CtoS to utilize the <i>fastest</i> possible resources (which will be area-expensive) to implement the given “op”. Conversely, a speed grade of 0 instructs CtoS to choose the <i>slowest</i> implementation, which would probably result in the smallest area requirement. Note: The CtoS speed grade concept is not the same as a device’s speed grade.

Term	Expanded or Additional Terms	Definition
SRAM	static random access memory	<i>SRAM (static random access memory)</i> is a type of semiconductor “ memory ”. The word <i>static</i> indicates that, unlike <i>dynamic</i> “ RAM ” (DRAM), it does not need to be periodically refreshed, because SRAM uses bistable latching circuitry to store each bit. SRAM exhibits data remanence, but is still volatile in the conventional sense that data is eventually lost when the memory is not powered. Note: The term <i>SDRAM</i> , which stands for <i>synchronous DRAM</i> , should not be confused with SRAM.
start value		In CtoS, the <i>start value</i> of a “ DFG ” is the “ value ” that controls the (unique) output “ edge ” of the “ origin node ”. This value does not have a driver.
state		A <i>state</i> corresponds to a wait function call in the SystemC code. “ value ”s generated prior to the wait function call, and used after it, are saved in “ register ”s.
state machine	<i>also referred to as an FSM (finite state machine)</i>	A <i>state machine</i> is a model composed of a finite number of “ state ”s, transitions between those states, and actions. A state machine is an abstract model of a machine with a primitive internal “ memory ”.
STL	Standard Template Library	The <i>STL (Standard Template Library)</i> is a C++ library of container classes, algorithms, and iterators. It provides many of the basic algorithms and data structures of computer science. The STL is a <i>generic</i> library – its components are heavily parameterized, and almost every component in the STL is a template.

Term	Expanded or Additional Terms	Definition
SystemC		<i>SystemC</i> is a single, unified design and verification language that expresses architectural and other system-level attributes in the form of open-source C++ classes. SystemC provides an even higher level of abstraction modeling capability than other “HDL”s and a synthesizable subset supported by CtoS, as well. Ratified as <i>IEEE Standard 1666-2005 for SystemC</i>, SystemC is a language built in standard C++ by extending the language with the use of class libraries. SystemC addresses the need for a system design and verification language that spans hardware and software. The language is particularly suited to model system partitioning, to evaluate and verify the assignment of blocks to either hardware or software implementations, and to architect and measure the interactions between and among functional blocks. The Open SystemC Initiative (OSCI) is an independent, not-for-profit association composed of a broad range of organizations dedicated to defining and advancing SystemC as an open industry standard for system-level modeling, design, and verification. See also www.systemc.org.
tag		In CtoS, a <i>tag</i> is a stack of Boolean “value”s (predicates), which must all be true for an “op” to be scheduled. The stack lets CtoS represent the conditions of <i>nested if</i> statements. Tags are represented as a tree with a root tag that is always true and alternating levels in the tree. The root tag and every other level from the root tag are exclusive.
TCF	toggle count format	<i>TCF (toggle count format)</i> is a file format typically used by power calculation tools. TCF files include the toggle frequency and duty cycle of each and every “node” in a “design”, for a given period of time.
terminal		In CtoS, a <i>terminal</i> corresponds to a terminal in a “module”, whereas a “port” is a port of an “op”.

Term	Expanded or Additional Terms	Definition
thread process (SystemC)	declared as SC_CTHREAD or SC_THREAD	A <i>SystemC thread process</i> is a “ process ” that, in each activation, executes from a wait() statement (or the beginning of the function associated with that process) to another wait() statement (or the end of the function associated with that process). The state of variables prior to a wait() statement is retained to the next activation. CtoS supports only the “ clocked thread process ”.
throughput-accurate		<i>Throughput-accurate</i> means that the number of “ clock ” cycles required to execute one iteration of a loop is the same as the number of cycles required for “ RTL ” simulation. However, the clock cycles at which inputs are accepted and outputs are produced may <i>not</i> be the same.
TLM	Transaction-Level Modeling	<i>TLM (Transaction-Level Modeling)</i> is a standard set of C++ class libraries that have been defined by OSCI (www.systemc.org). The libraries sit on top of SystemC and implement a set of interfaces between models defined at the transaction-level.
timed functional model	<i>also referred to as a performance model</i>	A <i>timed functional model</i> is a direct translation of a design specification into “ SystemC ” for which timing delays have been added to “ process ”es within a design to reflect the timing constraints of the specification and processing delays of a particular target implementation.
untimed functional model		An <i>untimed functional model</i> is a direct translation of a design specification into “ SystemC ” for which there are no timing delays at all.
value		In CtoS, a <i>value</i> is a single bit in the “ DFG ” of a program; when a program is converted to a data flow, a “ variable ” may then have different values. See also “ net ”, “ shared variable or shared net ”, “ signal ”.
variable		In CtoS, a <i>variable</i> is an automatic variable declared in a function or member variable of an SC_MODULE . A variable may be a vector or a scalar. CtoS does not track variables, only “ value ”s of expressions in the source code. See also “ net ”, “ shared variable or shared net ”, “ signal ”.

Term	Expanded or Additional Terms	Definition
Vendor RAM	vendor random access memory	<i>Vendor</i> “RAM”s are IP blocks that implement standard “SRAM” protocols and that come with area, timing, power, and simulation models.
Verilog	<i>also referred to as Verilog HDL</i>	<i>Verilog</i> is a standardized “HDL” for specifying the structure and behavior of electronic systems in textual format. Developed in the mid-1980s as a proprietary language and acquired by Cadence Design Systems, it became a <i>de facto</i> industry standard. In the mid-90s, Cadence placed it in the public domain, and it became a <i>de jure</i> standard promulgated by the IEEE. “NC-Verilog” is also the name of a legacy simulation tool offered by Cadence. A subset of the statements in the language is synthesizable. If the modules in a design contain only synthesizable statements, the design can often be transformed into the circuit layout of a computer chip using appropriate software.
virtual inheritance		<i>Virtual inheritance</i> is class inheritance where a class has a virtual base class. See also “dynamic class”, “multiple inheritance”, “single inheritance”.

Index

Numerics

#1 (delay control) D-28, E-161, E-165, J-6
53036
 sh2_syntaxhead2
 5.10.1 flatten_array E-68

A

active_edge
 RTL IP definition port attribute D-84
active_level
 RTL IP definition port attribute D-84
add
 resource type 11-22
add_command
 syntax E-5
add_message
 syntax E-6
addr_width
 array attribute D-36
addsub
 controlling use of 11-23
 resource type 11-22
Adobe Reader (acroread)
 currently supported version 3-2
alignment holes 14-109
allocate_builtin_ram
 syntax E-7
allocate_memory

 syntax E-9
allocate_memory_interfaces
 syntax E-11
allocate_prototype_memory
 syntax E-14
allocate_registers
 syntax E-18
allow_op_delays_exceeding_clock_period
 design attribute
 defined D-11
 when modified D-10
Altera Quartus II
 currently supported version 3-2
ALTERNATIVE_EDITOR
 setting for external editors 6-36
ambiguous operator
 with fixed-point library 15-9
analyze (command)
 syntax E-20
annotation_path
 node attribute D-50
apply_directive
 syntax E-22
apply_uarch_action
 syntax E-25
architecture
 defined N-2
area
 instance attribute D-69
 module attribute D-34

- array
 - defined N-2
- array dependency
 - defined N-2
- array_constraints
 - behavior child scope D-45
- arrays
 - module child scope D-35
- asm directive
 - not supported 14-111
- async_read
 - access protocol H-21
- async_reset_signal_is 14-14
- auto_save_dir
 - design attribute
 - defined D-11
 - in Design Property 6-26
 - when modified D-9
- auto_write_models
 - design attribute
 - defined D-11
 - in CNDW 6-11
 - when modified D-10
 - with define_sim_config E-30
 - with launch_sim E-80
- aux_port_def
 - memory definition child scope D-33

B

- Backus-Naur Form (BNF) 3-3
- backward edge 11-9, 11-12, 11-18, D-47, E-37
 - defined N-2
 - with break_combinational_loop E-29
- baseline_dir
 - design attribute
 - defined D-12
 - in Design Property 6-26
 - when modified D-9
- behavior
 - (behavior_)term attribute D-67
 - defined N-2
 - edge attribute D-46
 - instance attribute D-69

- latency constraint attribute D-49
- node attribute D-50
- operation attribute D-57
- operation constraint attribute D-56
- pin attribute D-62
- port attribute D-60
- register binding attribute D-63
- resource binding attribute D-65
- tag attribute D-66
- value attribute D-68
- behaviors
 - module child scope D-35
- bind_value 12-18
 - syntax E-26
- birthday
 - defined N-2
- birth_edge
 - operation attribute D-57
- BitInstTerms D-80
- bits
 - instance terminal child scope D-74
 - module terminal child scope D-80
 - net child scope D-77
- body unrolling
 - defined N-2
- boolean
 - attribute value type D-5
- born_ops
 - edge attribute D-46
- break_array_dependency
 - syntax E-27
- break_array_inter_iteration_dependencies
 - syntax E-28
- break_combinational_loop
 - syntax E-29
- bridges
 - memory definition attribute D-30
- build (command)
 - syntax E-30
- build errors
 - reporting source context for 6-33
- Build icon (CtoS GUI) 6-28
- build_flat

- design attribute 5-15, 16-3, 16-4
 - defined D-12
 - in CNDW 6-12
 - when modified D-9
- build_monolithic_structs
 - design attribute
 - defined D-12
 - when modified D-9
- byte alignment 14-109

C

- C++ labels 14-94
- Call Graph viewer
 - of CtoS GUI 8-8
- call-busy loop. See function call loop.
- Calypto Sequential Logic Equivalence Checker (SLEC)
 - flop map N-6
- CAM (content-addressable memory)
 - defined N-2
- Cancel (Interrupt button) 6-32
- cd D-5
 - syntax E-31
 - using with CtoS 6-47
- CDFG (control and data flow graph)
 - defined N-3
- CDFG viewer (CtoS GUI) 6-41
- cerr 14-107
- CFG (control flow graph)
 - defined N-3
 - viewer
 - of CtoS GUI L-1
- CGIC 18-11
- CGIC (clock gating integrated cell)
 - defined N-3
- check_design
 - syntax E-32
- children
 - tag attribute D-66
- CLB (configurable logic block)
 - defined N-3
- clock
 - defined N-3

- generation 14-36
 - hierarchical mode 14-36
- clock gating integrated cell. See CGIC.
- clocked method process (SystemC)
 - defined N-3
- clocked SC_METHOD (SystemC). See clocked method process.
- clocked thread process (SystemC)
 - defined N-4
- clock_per_port
 - memory definition attribute D-30
- clock_port
 - memory definition attribute D-30
- clock_posedge
 - memory definition attribute D-30
- clocks
 - design attribute
 - defined D-12
 - limited support for designs with multiple 6-13
- close_design
 - syntax E-33
- combinational function
 - defined N-4
- combinational logic
 - defined N-4
- combinational loop
 - breaking 8-22
 - eliminating 8-17
- combinational method process (SystemC)
 - defined N-4
- combinational process (SystemC). See combinational method process.
- combinational SC_METHOD (SystemC). See combinational method process.
- combinationally adjacent
 - defined N-4
- combinatorial logic. See combinational logic.
- combine_sources
 - deprecated option of build command 6-29
- command
 - directive attribute D-61
- Command Window (CtoS GUI) 6-8

- compile_flags
 - design attribute
 - defined D-13
 - in CNDW 6-12
 - when modified D-9
- compiler flags supported in CtoS 6-12
- compiling multi-source designs 6-29
- config_file_name
 - synthesis config attribute D-86
- configurable logic block. See CLB.
- connect (command)
 - syntax E-34
- connected_to
 - memory bridge port def attribute D-29
- constrain_latency 11-13
 - syntax E-36
- constrain_op
 - syntax E-38
- constraint
 - defined N-4
- constraint_type
 - operation constraint attribute D-56
- container classes 5-10
- content-addressable memory. See CAM.
- control and data flow graph. See CDFG.
- control flow graph. See CFG.
- control op
 - defined N-4
- control operation. See control op.
- control state
 - defined N-5
- control_delay
 - instance attribute D-70
- controlled_edges
 - value attribute D-68
- controlled_tags
 - value attribute D-69
- control_value
 - edge attribute D-47
 - tag attribute D-67
- conventions 3-3
- convert_to_lookup 8-16
 - syntax E-40
- core channel 15-72
- core instance
 - defined E-170
- cout 14-107, 14-108
- create_array_dependencies
 - syntax E-41
- created_by
 - operation constraint attribute D-56
- create_initial_resources E-153
 - syntax E-42
- create_protocol_region 11-17
 - syntax E-43
- create_register
 - syntax E-45
- create_required_states
 - syntax E-46
- create_resource 11-21
 - syntax E-47
- create_rom_program
 - allocating Vendor ROM 9-15
 - syntax E-49
- create_state
 - syntax E-50
- critical endpoint
 - defined N-5
- critical path
 - defined N-5
- __CTOS__ 14-102
- ctos (executable)
 - syntax E-51
- CtoS TLM Library 15-68
- ctos_combined_source.cpp
 - combined source file 6-29
- ctos_compare_outputs G-6
- CTOS_EXE
 - overriding at command line 7-21
- ctos_external_array 9-38
- CTOS_FX_ASSIGN_BIT 15-28
- CTOS_FX_ASSIGN_RANGE 15-8, 15-28
- CTOS_FX_WITH_FLAGS 15-8, 15-30
- ctosgui (executable)
 - syntax E-52
- ctos.gui (preferences file) 6-4

ctosgui.log (command log file) 6-8
 .ctos_init (initialization file) 6-3
 CTOS_MODEL 7-19
 CTOS_TARGET_NAME 7-21
 CTOS_TARGET_SUFFIX macro 7-14
 ctos_tlm 15-68
 custom op
 defined N-5
 custom_flow D-86
 Cycle Analysis viewer 13-20

D

data flow graph. See DFG.
 data_delay
 instance attribute D-70
 data_width
 array attribute D-36
 data_words
 array attribute D-36
 default_clock
 design attribute
 defined D-13
 used with RC 6-13
 when modified D-10
 default_export_memories
 design attribute
 defined D-13
 in CNDW 6-15
 when modified D-10
 default_scheduling_effort
 design attribute
 defined D-14
 when modified D-9
 default_setup.tcl 13-48
 default_speed_grade
 design attribute
 defined D-14
 when modified D-10
 default_synthesis_flow 13-46, D-86
 default_toggle_probability
 design attribute
 defined D-14
 in power estimation 18-5
 when modified D-10
 default_value_1_probability
 design attribute
 defined D-15
 in power estimation 18-5
 when modified D-10
 define_clock 12-2, E-59
 called by CNDW 6-13
 syntax E-53
 define_control_error_terminal 13-38, E-163
 syntax E-54
 --define_macro (compiler flag) 6-12
 define_sim_config
 syntax E-55
 with the launch_sim command E-80
 define_synth_config
 syntax E-56
 define_tlm_transactor_pair E-170
 syntax E-57
 delay control E-161, E-165
 delay control (#1) D-28, J-6
 dereference
 defined N-5
 design
 array attribute D-36
 defined N-5
 module attribute D-34
 simulation config attribute D-85
 Design Property dialog
 described 6-18
 design under test. See DUT.
 design_dir
 design attribute
 defined D-15
 in CNDW 6-11
 when modified D-9
 design_sim_config
 called by CNDW 6-11
 DFG (data flow graph)
 defined N-5
 DFTL (detectable fixed throughput and latency)
 13-40
 directed test patterns 7-3

- direction
 - (behavior_)term attribute D-68
 - instance terminal attribute D-73
 - instance terminal bit attribute D-74
 - module terminal attribute D-79
 - module terminal bit attribute D-81
 - pin attribute D-62
 - port attribute D-60
 - RTL IP definition port attribute D-84
- directives
 - array child scope D-38
 - behavior child scope D-45
 - edge child scope D-48
 - node child scope D-55
 - operation child scope D-59
- direct_rc_script D-86
- Dname (compiler flag) 6-12
- do loop 8-56
- do/while loop 8-56
- doc option
 - help command 3-2
- dont_initialize()
 - ignored by CtoS 14-36
- dont_touch
 - array attribute D-36
 - with Vendor ROMs 9-15
 - behavior attribute D-41
 - with RTL IP 9-41
 - controlling with startup script 6-16
- dont_use
 - controlling with startup script 6-16
- double
 - attribute value type D-5
- driver
 - net attribute D-76
 - net bit attribute D-78
- dumptcf
 - ncsim command 18-4
- DUT (design under test)
 - defined N-5
- dynamic class
 - defined N-5
- dynamically allocated arrays

- not supported 14-111

E

- ECO (engineering change order)
 - defined N-5
 - using Incremental Synthesis for 17-3
 - with save/open 6-51
- eco_sharing_policy
 - design attribute
 - defined D-15
 - when modified D-9
- edge
 - backward D-47
 - defined N-6
 - operation constraint attribute D-56
 - resource binding attribute D-65
- edges
 - behavior child scope D-45
- \$EDITOR
 - with external editors 6-36
- elaborated
 - subdirectory 17-5, D-11
 - subdirectory for ECO mode 6-26
- .emacs 6-36
- emacsclient 6-36
- enable_addsubs
 - behavior attribute 11-23, D-41, E-42
 - before schedule command E-138
- enable_const_resources
 - behavior attribute D-41
- enable_multiple_pipeline_stalls
 - design attribute
 - defined D-15
 - when modified D-9
- enable_slec_verification
 - design attribute
 - defined D-15
 - when modified D-9
- Encounter Conformal ECO 17-3
- Encounter RTL Compiler (RC)
 - called by Timing Analyzer 12-4
 - characterizing resources in synthesis A-14, B-15

- clock period defined by 6-13, E-53
 - currently supported version 3-2
 - defined N-12
 - intra-assignment delays ignored by E-161, E-165, J-6
 - rc_startup_script attribute with 6-16, D-22
 - rc_work_dir attribute with D-22
 - report_timing command with E-131
 - RTL IP with 9-43, H-7, H-25
 - switching and level probability propagation capabilities of 18-2
 - technology library for E-103
 - verilog_use_non_blocking_delay_control attribute with D-28
 - write_rc_script command with E-159
 - endianness 14-109
 - end_node
 - array constraint attribute D-46
 - floating constraint attribute D-48
 - latency constraint attribute D-49
 - protocol constraint attribute D-63
 - engineering change order. See ECO.
 - ERROR (CTOS-11112) 6-33
 - ERROR (CTOS-11115) 1-9, M-13
 - ERROR (CTOS-14035) E-172
 - ERROR (CTOS-14036) E-172
 - ERROR (CTOS-14037) E-172
 - ERROR (CTOS-14038) E-172
 - ERROR (CTOS-14039) E-172
 - ERROR (CTOS-14040) E-172
 - ERROR (CTOS-14041) E-172
 - ERROR (CTOS-14042) E-172
 - ERROR (CTOS-15089) E-154
 - ERROR (CTOS-17014) 1-9, M-15
 - ERROR (CTOS-17019) 8-62
 - ERROR (CTOS-17031) 1-9, M-14
 - ERROR (CTOS-19047) 10-2
 - ERROR (CTOS-19097) 10-2
 - ERROR (CTOS-20051) 8-43
 - ERROR (CTOS-20080) 8-43
 - ERROR (CTOS-20085) 8-43
 - ERROR (CTOS-20087) 8-43
 - ERROR (CTOS-20233) 8-43
 - ERROR (CTOS-20289) 8-45
 - ERROR (CTOS-20320) 8-42
 - ERROR (CTOS-20322) 8-42
 - ERROR (CTOS-7026) 10-2
 - ERROR (CTOS-7102) 10-2
 - error_recovery_terminal
 - behavior attribute D-42
 - with define_control_error_terminal command 13-38, E-54
 - with undefine_control_error_terminal command E-149
 - explicit reference
 - not supported 14-112
 - export_memories
 - design attribute
 - when modified D-10
 - extern global variable
 - not supported 14-111
 - external editor
 - using in CtoS GUI 6-36
 - external_array_bindings
 - array attribute D-36
 - array child scope D-38
 - external_delay 11-11, 12-2
 - syntax E-58
 - external_input_delay
 - design attribute
 - defined D-42
 - net attribute D-76
 - external_output_delay
 - design attribute
 - defined D-42
 - net attribute D-76
- ## F
- fan-in
 - defined N-6
 - fan-out
 - defined N-6
 - ff_design_attributes K-2
 - ff_elab K-2
 - ff_exit K-2
 - ff_final_reports K-2

- ff_insert_lp K-2
- ff_load_constraints K-2
- ff_load_hdl K-2
- ff_load_library K-2
- ff_root_attributes K-2
- ff_setup_dont_use_cells K-2
- ff_setup_use_cells K-2
- ff_syn_gen K-2
- ff_syn_incr K-2
- ff_syn_map K-2
- ff_uniquify K-2
- ff_write_final_outputs K-2
- ff_write_xml_metrics K-2
- field-fragmented object model 14-45
 - assumptions on data access 14-50
 - described 14-48
 - setting with build_monolithic_structs D-12
- find (command)
 - syntax E-60
- findbyannotationpath (command)
 - syntax E-62
- find_combinational_loops
 - syntax E-63
- find_from_rtl
 - syntax E-64
- find_in_rtl
 - syntax E-65
- find_source
 - syntax E-66
- finite state machine. See state machine.
- flatten_array
 - syntax E-68
- flattening designs
 - during build process 6-28
- Flex Channels Library 15-33
- flex_channels_nets
 - syntax E-69
- float_array_accesses
 - syntax E-70
- float_io_accesses
 - syntax E-71
- Flop Map
 - defined N-6
- for loop
 - transformed to do/while loop 8-56
- fork node
 - defined N-6
- format
 - memory definition attribute D-30
- forward edge
 - defined N-6
- fpga_dsp_cost
 - design attribute
 - defined D-16
 - in CNDW 6-18
 - when modified D-10
- fpga_install_path
 - design attribute
 - defined D-16
 - in CNDW 6-17
 - when modified D-10
- fpga_target
 - design attribute
 - defined D-16
 - in CNDW 6-17
 - when modified D-10
- fpga_work
 - default FPGA working directory 6-17
- fpga_work_dir
 - design attribute
 - defined D-16
 - in CNDW 6-17
 - when modified D-10
- FSM. See state machine.
- full_name
 - module terminal attribute D-79
 - module terminal bit attribute D-81
 - net attribute D-76
 - net bit attribute D-78
- full_time_info.rpt 13-52
- function call from a thread process. See thread process.
- function call loop
 - defined N-6

G

GCC. See GNU Compiler Collection (GCC).
 gedit 6-37
 get_attr
 syntax E-73
 with object attributes D-5
 get_child_objects 14-107, 14-108
 get_design
 syntax E-74
 get_install_path
 syntax E-75
 get_version
 syntax E-76
 global variables
 not supported in CtoS 14-44
 GNU Compiler Collection (GCC)
 version currently supported by INCISIV 3-2
 GNU extensions
 not supported 14-111
 gnuclient 6-37
 gtle
 resource type 11-22

H

hardware description language. See HDL.
 has_reset
 memory definition attribute D-31
 has_stall
 memory definition attribute D-31
 HDL (hardware description language)
 defined N-7
 header_files
 design attribute
 defined D-16
 in CNDW 6-11
 when modified D-9
 help (command)
 syntax E-77
 Hierarchy Window (CtoS GUI) 6-44

I

-Idir (compiler flag) 6-12
 IEEE Standard 1364-1995/2001 for Verilog HDL
 3-2, E-165
 IEEE Standard 1666-2005 for SystemC D-24,
 N-17
 Language Reference Manual 3-2, 14-4,
 14-110, 15-1
 IEEE Standard 1850 for Property Specification
 Language (PSL) 3-3
 II (initiation interval)
 defined N-7
 impl_behavior
 pipeline function object attribute D-40
 implementation_target
 design attribute
 defined D-17
 in CNDW 6-14
 when modified D-9
 INCISIV. See Incisive Enterprise Simulator
 (INCISIV).
 Incisive Enterprise Simulator (INCISIV)
 currently supported version 3-2
 defined N-7
 --include_directory (compiler flag) 6-12
 Incremental Synthesis 17-1
 index
 pin attribute D-62
 register binding attribute D-64
 indexed part select
 feature of Verilog 2001 D-27
 in_edges
 node attribute D-50
 initiation interval. See II.
 init_interval
 node attribute D-51
 inline 8-11, 8-21, 8-22
 inline (command)
 syntax E-78
 inline_calls 8-11
 syntax E-79
 inlining
 defined N-7

- methods described 8-3
- in_ports
 - operation attribute D-57
- Input Source viewer (CtoS GUI) 6-38
- input_delay
 - instance terminal attribute D-73
 - instance terminal bit attribute D-74
 - module terminal attribute D-79
 - module terminal bit attribute D-81
- inst
 - array attribute D-37
 - instance terminal attribute D-73
 - instance terminal bit attribute D-74
 - operation constraint attribute D-56
 - resource binding attribute D-66
- instance
 - defined N-7
- instance terminal
 - defined N-7
- insts
 - module child scope D-35
- inst_terms
 - instance child scope D-72
- integer
 - attribute value type D-5
- integer data type
 - with fixed-point library 15-9
- integral type
 - defined N-7
- interface_type
 - memory bridge def attribute D-29
- interface_types
 - memory definition attribute D-31
- Interrupt button (CtoS GUI) 6-32
- intra-assignment delay D-28, J-6
- io_ops
 - net attribute D-76
 - net bit attribute D-78
- is_active_high
 - reset condition attribute D-65
- is_allocated
 - array attribute D-37
- is_async
 - reset condition attribute D-65
- is_backward
 - edge attribute D-47
- is_clock
 - instance terminal attribute D-73
 - instance terminal bit attribute D-75
 - module terminal attribute D-79
 - module terminal bit attribute D-81
- is_combinational
 - behavior attribute D-42
- is_exported
 - array attribute D-37
- is_external
 - array attribute D-37
- is_failure
 - resource binding attribute D-66
- is_function_call_loop
 - node attribute D-51
- is_input
 - memory def auxiliary port attribute D-33
- is_inverted
 - register binding attribute D-63
- is_loop
 - node attribute D-51
- is_mem_port
 - memory bridge port def attribute D-30
- is_optimized
 - module attribute D-34
- is_pipeline_loop
 - node attribute D-51
- is_primary
 - register binding attribute D-64
- is_process
 - behavior attribute D-42
- is_respected
 - operation constraint attribute D-56
- is_root
 - tag attribute D-67
- is_scheduled
 - design attribute
 - defined D-17
- is_shared
 - array attribute D-37

is_stall_loop
 node attribute D-51
 is_state
 node attribute D-51
 is_user_created
 instance attribute D-70
 is_user_defined
 module attribute D-34

J

join node
 defined N-7

K

keep_all_signals
 design attribute
 defined D-17
 when modified D-9
 keep_signals
 design attribute
 defined D-76

L

latency
 pipeline function object attribute D-40
 latency interval. See LI.
 latency_constraints
 behavior child scope D-45
 lat_interval
 node attribute D-52
 launch_delay
 instance attribute D-70
 launch_sim
 syntax E-80
 lcd
 syntax E-81
 level
 tag attribute D-67
 LI (latency interval)
 defined N-7
 liberty_filename

memory definition attribute D-31
 lifetime
 defined N-8
 list of strings
 attribute value type D-5
 list_attr
 syntax E-82
 with object attributes D-5
 lls
 syntax E-83
 loads
 net attribute D-76
 net bit attribute D-78
 LOG_DIR 13-54
 lookup table. See LUT.
 loop body
 defined N-8
 loop join node
 defined N-8
 loop pipelining
 described 8-26
 loop unrolling
 defined N-8
 loop variable
 defined N-8
 loop_join_id
 defined N-8
 loop_level
 node attribute D-52
 -low_power
 option for schedule command 12-16
 low_power_clock_gating
 design attribute
 defined D-17
 in CNDW 6-16
 using 18-10
 when modified D-10
 lpwd
 syntax E-84
 LRM. See IEEE Standard 1666-2005 for
 SystemC Language Reference Manual.
 lrot
 resource type 11-22

- ls D-5
 - syntax E-85
 - using with CtoS 6-47

- lsh
 - resource type 11-22

- LUT (lookup table)
 - defined N-8

M

- Main View (CtoS GUI) 6-2

- makefile
 - simulation config attribute D-85

- master
 - external array binding object attribute D-39
 - instance attribute D-70

- max_auto_flatten_array_size
 - design attribute
 - defined D-18

- max_delay
 - instance attribute D-70
 - module attribute D-34

- max_latency
 - latency constraint attribute D-49
 - pipeline function object attribute D-40

- max_lat_interval
 - node attribute D-52

- max_output_line_length
 - design attribute
 - defined D-18
 - in Design Property 6-22
 - when modified D-10

- member function 14-61

- member operator of structure 14-61

- memory
 - defined N-8

- memory_bridge_port_defs
 - memory_bridge_defs child scope D-29

- memory_defs
 - design child scope D-29

- memory_generators (design child scope reserved for future use) D-29

- Menu Bar (CtoS GUI) 6-5

- merge_arrays 8-63, 8-78, 8-86

- syntax E-86
 - using with object models 14-51

- merged_join_states
 - node attribute D-52

- method process (SystemC)
 - defined N-8

- micro-architecture
 - defined N-8

- min_latency
 - pipeline function object attribute D-40

- min_lat_interval
 - node attribute D-53

- min_speed_grade
 - operation attribute D-57

- min_write_width
 - array attribute D-37
 - memory definition attribute D-31

- mobility. See op span.

- model_dir
 - default_sim_config attribute
 - in CNDW 6-11
 - simulation config attribute D-85

- mod_type
 - module attribute D-34

- module
 - array attribute D-37
 - behavior attribute D-42
 - defined N-9
 - instance attribute D-70
 - module terminal attribute D-79
 - module terminal bit attribute D-81
 - net attribute D-77
 - net bit attribute D-78

- module constructor 14-5

- modules

- design child scope D-29

- monolithic object model 14-45
 - assumptions on data access 14-50
 - described 14-47
 - setting with build_monolithic_structs D-12

- multi-cycle op
 - defined N-9

- multiple inheritance 14-61

- defined N-9
- for specifying TLM interface 14-61
- multiple writer variable or net
 - defined N-9
- multiports
 - not supported 14-112
- mux
 - setting purpose for an instance D-71
- mux_purpose
 - instance attribute D-71

N

- name
 - (behavior_)term attribute D-68
 - array attribute D-38
 - array constraint attribute D-46
 - behavior attribute D-42
 - design attribute
 - defined D-18
 - when modified D-9
 - directive attribute D-61
 - edge attribute D-47
 - external array binding object attribute D-39
 - floating constraint attribute D-48
 - instance attribute D-71
 - memory bridge def attribute D-29
 - memory bridge port def attribute D-30
 - memory def auxiliary port attribute D-33
 - memory definition attribute D-31
 - module attribute D-34
 - module terminal attribute D-79
 - module terminal bit attribute D-81
 - net attribute D-77
 - net bit attribute D-78
 - node attribute D-53
 - operation attribute D-57
 - operation constraint attribute D-56
 - protocol constraint attribute D-63
 - register binding attribute D-64
 - resource binding attribute D-66
 - RTL IP definition attribute D-83
 - RTL IP definition port attribute D-84
 - simulation config attribute D-85
 - tag attribute D-67
 - value attribute D-69
- name_max_length
 - design attribute
 - defined D-18
 - in Design Property 6-25
 - when modified D-9
- name_module_prefix
 - design attribute
 - defined D-18
 - when modified D-9
- name_module_prefix_policy
 - design attribute
 - defined D-19
 - when modified D-9
- name_use_class_name
 - design attribute
 - defined D-19
 - in Design Property 6-25
 - when modified D-9
- name_use_file_name
 - design attribute
 - defined D-19
 - in Design Property 6-25
 - when modified D-9
- name_use_inline_location
 - design attribute
 - defined D-19
 - in Design Property 6-25
 - when modified D-9
- name_use_line_number
 - design attribute
 - defined D-19
 - in Design Property 6-25
 - when modified D-9
- name_use_node_index
 - design attribute
 - defined D-20
 - in Design Property 6-25
 - when modified D-9
- name_use_template_parameters
 - design attribute
 - defined D-20

- name_use_unrolling_index
 - design attribute
 - defined D-20
 - in Design Property 6-25
 - when modified D-9
- name_use_var_line_number
 - design attribute
 - defined D-20
 - in Design Property 6-25
 - when modified D-9
- native-compiled SystemC. See NC-SC.
- native-compiled Verilog. See NC-Verilog.
- nc (Client for Nirvana Editor) 6-37
- ncelab J-5
- NC-SC (native-compiled SystemC)
 - defined N-9
- ncsim 18-1, 18-4, J-5
- NC-Verilog (native-compiled Verilog)
 - defined N-9
- nedit 6-37
- neg event finder 14-112
- nested side effects
 - not supported 14-111
- net
 - (behavior_)term attribute D-68
 - defined N-9
 - instance terminal attribute D-73
 - instance terminal bit attribute D-75
 - module terminal bit attribute D-81
 - reset condition attribute D-65
- netcat 6-37
- netlist
 - defined N-9
- nets
 - module child scope D-35
- new_design
 - called by CNDW 6-10
 - syntax E-88
- node
 - defined N-9
 - register binding attribute D-64
- nodes
 - behavior child scope D-45

- non-blocking assignment D-28, E-161, E-165,
 - J-6
- +nospecify Verilog option 9-14
- no_warnings (compiler flag) 6-12
- number_of_schedule_passes
 - behavior attribute D-42
- num_words
 - memory definition attribute D-31

O

- object
 - directive attribute D-61
- object model
 - using in CtoS 14-45
- one_value
 - behavior attribute D-42
- op
 - defined N-9
 - operation constraint attribute D-56
 - pin attribute D-62
 - port attribute D-60
 - resource binding attribute D-66
- op slack. See op span.
- op span 12-2, E-138, L-2
 - defined N-10
- op_constraint
 - operation attribute D-57
- op_constraints
 - behavior child scope D-45
 - edge attribute D-47
 - instance attribute D-71
- Open SystemC Initiative (OSCI)
 - 1.0 library 15-68
 - discrepancies with IEEE 14-110
 - TLM interfaces and channels 15-65
- open_design
 - syntax E-89
- operand
 - defined N-10
- operation
 - defined N-9
- operation span. See op span.
- ops

- behavior child scope D-45
- optimize (command)
 - syntax E-90
- optimize_enable_arithmetic_trees
 - behavior attribute D-43
- optimize_enable_if_loops_transform
 - design attribute
 - defined D-21
 - when modified D-9
- optimize_enable_merge_join_states
 - design attribute
 - defined D-21
 - when modified D-10
- op_type
 - operation attribute D-57
- origin
 - behavior attribute D-43
- origin node
 - defined N-10
- OSCI
 - see Open SystemC Initiative (OSCI)
- out_edges
 - node attribute D-53
- out_ports
 - operation attribute D-58
- output_delay
 - instance terminal attribute D-73
 - instance terminal bit attribute D-75
 - module terminal attribute D-79
 - module terminal bit attribute D-81
- overflow mode
 - fixed-point conversion with 15-6

P

- packed object model 14-45
- parent
 - tag attribute D-67
- partial unrolling
 - defined N-10
- path
 - defined N-10
- performance model. See timed functional model.
- phase of a pipeline
 - defined N-10
- pin
 - defined N-10
- pins
 - port child scope D-60
- pipeline depth of RTL IP
 - defined N-10
- pipeline stage
 - defined N-10
- pipeline stage viewer. See PSV.
- pipeline stall 8-49
- Pipeline View 8-36
- pipeline_depth
 - RTL IP definition attribute D-83
- pipeline_function
 - design attribute
 - defined D-43
- pipeline_loop
 - syntax E-91, E-92
- pipeline_stage_mux
 - node attribute D-53
- pipelining
 - defined N-10
- plain old data. See POD.
- plain old data structures. See PODS.
- plugins
 - customizing Foundation Flows with 13-50
- POD (plain old data)
 - defined N-11
- PODS (plain old data structures)
 - defined N-11
- POD-struct
 - defined N-11
- pointer-to-member types
 - not supported 14-112
- port
 - defined E-170, N-11
 - pin attribute D-62
- ports
 - operation child scope D-59
 - RTL IP definition child scope D-83
- port_type
 - RTL IP definition port attribute D-84

- pos event finder 14-112
- power
 - behavior attribute D-43
 - module attribute D-35
 - operation attribute D-58
- pragma
 - changing default keyword 6-23
 - ctos dont_touch E-79
 - RTL IP 9-41
 - ctos monolithic
 - controlling object model with 14-52
 - ctos_dont_touch
 - setting dont_touch array attribute D-36
 - setting dont_touch behavior attribute D-41
 - with divide operators 15-10
 - with Vendor ROMs 9-15
 - pragma (default for ASIC) J-4
 - synthesis (default for FPGA) J-4
 - verilog_pragma_keyword for changing names D-26
- Preferences dialog (CtoS GUI) 6-4
- preserve
 - node attribute 14-97, D-54
- primary_binding
 - register binding attribute D-64
- print_message
 - syntax E-95
- process (SystemC)
 - defined N-11
- process behavior
 - defined N-11
- programmable read-only memory. See PROM.
- PROM (programmable read-only memory)
 - defined N-11
- prototype_memory_launch_delay
 - design attribute
 - defined D-21
 - in CNDW 6-15
 - when modified D-10
- prototype_memory_setup_delay
 - design attribute
 - defined D-21

- in CNDW 6-15
 - when modified D-10
- PSL. See IEEE Standard 1850 for Property Specification Language (PSL).
- psv
 - suffix for pipeline stage vector N-11
- PSV (pipeline stage vector)
 - defined N-11
 - naming convention 8-60
- pwd D-5
 - syntax E-96
 - using with CtoS 6-47

Q

- QoR (quality of results)
 - affect of multi-dimensional arrays access on 14-100
 - affect of sharing mode 12-13
 - affect when container class abstraction kept intact 5-16
 - defined N-11
 - effect of large container classes on 5-11
 - fixed-point library impact on 15-9
 - how to affect with incremental synthesis 17-2
 - impact of optimization from pointer references 5-14
 - impact of speed grade on 12-15
 - impact of unneeded variability on 14-67
 - incremental synthesis and binding decisions
 - impact on 17-1
 - inlining behaviors impact on 15-7, 15-74
 - multi-dimensional read-only arrays impact on 14-40
 - multiple resets impact on 14-12, 14-16, 14-29
 - overflow and quantization flags impact on 15-7
 - shared design components impact on 5-3
 - use of functions for common code patterns 5-3
 - use of pointers impact on 14-40
- QoR Metrics File 13-52
- quality of results. See QoR.

quantization mode
 fixed-point conversion with 15-6
 Quick Launch (CtoS GUI) 6-8

R

RAM (random access memory)
 defined N-12
 ram_def_files
 design attribute
 when modified D-10
 random access memory. See RAM.
 RC. See Encounter RTL Compiler (RC).
 RC_INSTALL_DIR 13-54
 rc_run.tcl 13-51
 rc_startup_script
 design attribute
 defined D-22
 in CNDW 6-16
 when modified D-10
 rc_work_dir
 design attribute
 defined D-22
 in CNDW 6-16
 when modified D-10
 read_ip_defs
 syntax E-97
 read_latency
 memory definition attribute D-32
 read_only
 array attribute D-38
 read-only memory. See ROM.
 read_ops
 array attribute D-38
 read_tcf
 syntax E-98
 Red Hat Enterprise Linux (RHEL)
 currently supported versions 3-2
 redirecting commands 6-48
 REF (reference model)
 defined N-12
 ref_array
 external array binding object attribute D-39
 reference model. See REF.
 ref_inst
 external array binding object attribute D-39
 reg
 register binding attribute D-64
 reg_bindings
 behavior child scope D-45
 instance attribute D-71
 node attribute D-54
 tag attribute D-67
 value attribute D-69
 region
 defined N-12
 register
 defined N-12
 register allocation
 defined N-12
 register binding
 defined N-12
 registered memory
 defined N-13
 register-transfer level. See RTL.
 relax_latency
 behavior attribute D-43
 remove_clock
 syntax E-100
 remove_directive
 syntax E-101
 rename_state
 syntax E-102
 report_area
 syntax E-103
 report_array_dependencies 11-3
 syntax E-105
 report_behavioral_power
 syntax E-106
 report_incremental
 syntax E-107
 report_latency 11-14
 syntax E-108
 report_opspan
 syntax E-109
 report_paths
 syntax E-114

- report_power
 - syntax E-116
- report_registers
 - syntax E-117
- report_resources 8-7
 - syntax E-119
- report_schedule
 - syntax E-121, E-125
- report_slack
 - syntax E-126
- report_states
 - syntax E-127
- report_summary 8-7
 - syntax E-128
- report_timing
 - syntax E-130
- report_tlm_transactor_pairs
 - syntax E-135
- required_match_percentage
 - design attribute
 - defined D-22
 - in Design Property 6-26
 - when modified D-9
- res_binding
 - operation attribute D-58
- res_bindings
 - edge attribute D-47
 - instance attribute D-72
- reset behavior
 - defined N-13
- reset condition
 - defined N-13
- reset path
 - defined N-13
- reset signal
 - defined N-13
- reset specification
 - defined N-13
 - of a process 14-12
- reset state
 - defined N-13
- reset value. See start value.
- reset_all_registers
 - design attribute
 - defined D-23
 - in CNDW 6-16
 - when modified D-9
 - with allocate_registers command E-18
- reset_async
 - memory definition attribute D-32
- reset_conditions
 - behavior child scope D-45
- reset_get() 15-72
- reset_high
 - memory definition attribute D-32
- reset_port
 - memory definition attribute D-32
- reset_put() 15-72
- reset_signal_is 14-13
- reset_type
 - RTL IP definition port attribute D-84
- reset_value
 - behavior attribute D-43
- resource
 - defined N-13
- resource binding
 - defined N-13
- restructure_array 8-78
 - syntax E-136
- RHEL. See Red Hat Enterprise Linux (RHEL).
- ROM (read-only memory)
 - defined N-14
- rrot
 - resource type 11-22
- RTL (register-transfer level)
 - defined N-14
- RTL Compiler. See Encounter RTL Compiler (RC).
- RTL process. See clocked method process.
- rtl_filename
 - RTL IP definition attribute D-83
- rtl_ip_defs
 - design child scope D-29
- rtl_language
 - RTL IP definition attribute D-83
- rtl_ready

- subdirectory D-11
- subdirectory for ECO mode 6-26
- rtl_type
 - RTL IP definition attribute D-83
- run_command_scripts (design attribute reserved for future use) D-23
- run_dir
 - synthesis config attribute D-86
- run_synth_gates 13-46, D-86

S

- save_design
 - syntax E-137
- scalar datum 14-109
- SCC (strongly connected component)
 - defined N-14
- sc_clock E-171
- sc_export E-171
 - with TLM 15-66
- SC_FX_EXCLUDE_OTHER 15-9
- schedule (command)
 - syntax E-22, E-25, E-138, E-145, E-146
- scheduled_edge
 - operation attribute D-58
- scheduled_ops
 - edge attribute D-47
- scheduled_phase
 - operation attribute D-58
- scheduled_stage
 - operation attribute D-58
- schedule_failure_edge
 - behavior attribute D-43
- schedule_slack
 - behavior attribute D-44
 - instance attribute D-72
 - module attribute D-35
- schedule_slack_margin
 - design attribute
 - defined D-23
 - when modified D-10
- scheduling
 - defined N-14
- scheduling_effort
 - behavior attribute D-44
 - design attribute
 - with analyze command E-20
- sc_in E-171
- sc_inout
 - not supported 14-112
- sc_interface
 - with TLM 15-65
- SC_METHOD
 - clocked 14-25
 - combinational 14-19
 - simple example 14-20
- SC_METHOD. See method process for definition.
- sc_module
 - with TLM 15-66
- sc_out E-171
- sc_port E-171
 - with TLM 15-66
- sc_prim_channel 15-73
- script
 - synthesis config attribute D-86
- script_type
 - synthesis config attribute D-86
- sc_sensitive_neg
 - not supported 14-112
- sc_sensitive_pos
 - not supported 14-112
- SC_THREAD 14-19
- sc_uint 14-61
- sdiv
 - resource type 11-22
- secondary register binding
 - reporting of 12-21, E-117
- secondary_binding
 - register binding attribute D-64
- secondary_object
 - directive attribute D-61
- seq_slack
 - operation attribute D-58
- sequential function
 - defined N-14
- sequential logic

- defined N-14
- Sequential Logic Equivalence Checker. See SLEC.
- set_attr
 - syntax E-140
 - with object attributes D-5
- set_baseline
 - syntax E-141
- set_design
 - syntax E-143
- set_stall_value
 - syntax E-144
- set_synthesis_mode
 - syntax E-145
- set_uarch_action
 - syntax E-146
- setup_delay
 - instance attribute D-72
- shared net
 - defined N-14
- shared variable
 - defined N-14
- shortcuts
 - Find Pattern box 6-7
 - general CtoS GUI 6-9
 - Preferences dialog 6-4
- signal
 - defined N-14
- signed
 - (behavior_)term attribute D-68
 - instance terminal attribute D-73
 - instance terminal bit attribute D-75
 - module terminal attribute D-80
 - module terminal bit attribute D-82
 - net attribute D-77
 - net bit attribute D-78
 - port attribute D-60
- SIM_GUI
 - overriding at command line 7-21
- simple node
 - defined N-15
- simulation configuration
 - how to define 7-18
- simulation_configs
 - design child scope D-29
- simulator_args
 - simulation config attribute D-85
- sim_wrapper_filename
 - design attribute
 - defined D-23
 - when modified D-9
 - with write_wrapper command E-176
- single inheritance 14-61
 - defined N-15
- sink
 - edge attribute D-47
- sinks
 - value attribute D-69
- sizeof argument
 - must be statically determined 14-111
- slack
 - behavior attribute D-44
 - defined N-15
 - instance attribute D-72
 - instance terminal attribute D-73
 - instance terminal bit attribute D-75
 - module attribute D-35
 - module terminal attribute D-80
 - module terminal bit attribute D-82
 - operation attribute D-59
- SLEC (Sequential Logic Equivalence Checker)
 - defined N-15
- smod
 - resource type 11-22
- smul
 - resource type 11-22
- SoC Encounter RTL-to-GDSII System 18-10
- source
 - edge attribute D-47
 - value attribute D-69
- source_files
 - design attribute
 - defined D-24
 - in CNDW 6-11
 - minimum requirement 14-2
 - setting for multi-source designs 6-30

- when modified D-10
- spec_behavior
 - pipeline function object attribute D-40
- speed grade
 - defined N-15
- speed_grade
 - instance attribute D-72
- split_array 8-65
 - syntax E-147
- split_op
 - syntax E-148
- SRAM (static random access memory)
 - defined N-16
- src_links
 - array attribute D-38
 - behavior attribute D-44
 - instance attribute D-72
 - instance terminal attribute D-74
 - instance terminal bit attribute D-75
 - module terminal attribute D-80
 - net attribute D-77
 - node attribute D-54
 - operation attribute D-59
 - port attribute D-60
- srsh
 - resource type 11-22
- stall loop 9-44
- Standard Template Library. See STL.
- standard_flow D-86
- start value
 - defined N-16
- start_node
 - array constraint attribute D-46
 - floating constraint attribute D-48
 - latency constraint attribute D-49
 - protocol constraint attribute D-63
- state
 - defined N-16
- state machine
 - defined N-16
- static global variable
 - not supported 14-111
- static random access memory. See SRAM.
- STL (Standard Template Library) 5-10
 - defined N-16
- Stopping (Interrupt button) 6-32
- streams
 - ignoring of 14-107
- string
 - attribute value type D-5
- strongly connected component. See SCC.
- sub
 - resource type 11-22
- success_msg
 - simulation config attribute D-85
- SUSE Linux Enterprise Server
 - currently supported versions 3-2
- syn2gen.qor 13-52
- syn2map_incr.qor 13-52
- syn2map.qor 13-52
- sync_read
 - access protocol H-21
- sync_read_write
 - access protocol H-21
- sync_write
 - access protocol H-21
- syntax 3-3
- Synthesis Monitor 13-42
- synthesis_mode
 - behavior attribute D-44
- synthesis.tcl 13-53
- SYNTH_EXIT 13-54
- synth_gates target 13-54
- synth_gates.cmd 13-54
- synth_gates.log 13-54
- SystemC
 - currently supported version 3-2
 - defined N-17
 - header D-24
- systemc_out_header_ext
 - design attribute
 - defined D-24
 - in Design Property 6-22
 - when modified D-10
- systemc_out_source_ext
 - design attribute

- defined D-24
- in Design Property 6-22
- when modified D-10

systemc_version

- design attribute
- defined D-24
- when modified D-10

T

table lookup

- conversion to 1-9, 8-15, M-14

tag

- defined N-17
- operation attribute D-59
- register binding attribute D-64
- resource binding attribute D-66

tags

- behavior child scope D-45

Task Window (CtoS GUI) 6-43

TCF (toggle count format) 18-1

- defined N-17

Tcl (Tool Control Language) 3-3

- currently supported version 3-2

tech_cell_name

- memory definition attribute D-32

tech_lib_names

- design attribute
- defined D-24
- in CNDW 6-16
- when modified D-10

term

- instance terminal attribute D-74
- instance terminal bit attribute D-75

terminal

- defined N-17

terms

- behavior child scope D-45
- module child scope D-35

testbench_files

- simulation config attribute D-85

testbench_kind

- simulation config attribute D-85

text

- directive attribute D-61

thread process (SystemC)

- defined N-18

throughput-accurate

- defined N-18

timed functional model

- defined N-18

timing analysis

- pre-scheduling 12-16

timing_criticality

- behavior attribute D-44

TLM (Transaction-Level Modeling)

- defined N-18

tlm_fifo 15-72

tlm_fifo_1t 15-72

tlm_fifo_reg 15-72

tlm_fifo_reg_1t 15-72

tlm_req_rsp_channel 15-73

tlm_transport_channel 15-73

toggle count format. See TCF.

Tool Bar (CtoS GUI) 6-7

top module

- defined E-170

top_module

- design attribute
- defined D-25

top_module_path

- design attribute
- defined D-25
- in CNDW 6-12
- minimum requirement 14-2
- when modified D-10

_top_wrapper

- default top wrapper suffix 6-15

Transaction-Level Modeling. See TLM.

transactor instance

- defined E-170

transformed_do_while

- node attribute D-54

transformed_if_loops

- node attribute D-54

Tree Map

- of CtoS GUI 13-29

try-catch statements
 not supported 14-112

type
 (behavior_)term attribute D-68
 array attribute D-38
 array constraint attribute D-46
 behavior attribute D-44
 design attribute
 defined D-25
 edge attribute D-47
 floating constraint attribute D-48
 instance attribute D-72
 instance terminal attribute D-74
 instance terminal bit attribute D-75
 latency constraint attribute D-49
 module attribute D-35
 module terminal attribute D-80
 module terminal bit attribute D-82
 net attribute D-77
 net bit attribute D-78
 node attribute D-54
 operation attribute D-59
 operation constraint attribute D-56
 pin attribute D-62
 port attribute D-60
 protocol constraint attribute D-63
 register binding attribute D-64
 resource binding attribute D-66
 RTL IP definition attribute D-83
 RTL IP definition port attribute D-84
 simulation config attribute D-85
 tag attribute D-67
 value attribute D-69

type reference
 not supported 14-112

U

uarch_action
 array attribute D-38, D-55
 behavior attribute D-45
 operation attribute D-59

udiv
 resource type 11-22

umod
 resource type 11-22

umul
 resource type 11-22

-Uname (compiler flag) 6-12

undefine_control_error_terminal
 syntax E-149

--undefine_macro (compiler flag) 6-12

undefine_tlm_transactor_pair
 syntax E-150

unions
 not supported 14-112

Universal Verification Methodology (UVM) 7-3

unrolling (loops)
 body unrolling 8-21
 complete 8-18
 partial 8-19

unroll_loop
 syntax E-151

unschedule (command)
 syntax E-153

untimed functional model
 defined N-18

ursh
 resource type 11-22

use_dedicated_resource
 operation attribute D-59

use_dsp
 syntax E-154

use_ip
 syntax E-156

USER_ARGS
 overriding at command line 7-21

user-defined function argument
 using primitive types for 9-46
 Verilog representation in SystemC 9-46

V

value
 defined N-18
 pin attribute D-62
 register binding attribute D-65

values

- behavior child scope D-45
- value_type
 - value attribute D-69
- variable
 - defined N-18
- variable-length array
 - not supported 14-111
- Vendor RAM
 - defined N-19
- Vendor random access memory. See Vendor RAM.
- Verilog
 - defined N-19
- Verilog behavioral mode
 - generating 7-9
- verilog_filename
 - memory definition attribute D-32
- verilog_mux_style
 - design attribute
 - defined D-26
 - in Design Property 6-23
 - to control case statement generation J-2
- verilog_out_file_ext
 - design attribute E-167
 - defined D-26
 - in Design Property 6-23
 - when modified D-10
- verilog_pragma_keyword
 - design attribute
 - defined D-26
 - in Design Property 6-23
 - when modified D-11
- verilog_rtl_model_suffix
 - design attribute
 - defined D-26
 - in Design Property 6-23
 - when modified D-10
- verilog_top_wrapper_suffix
 - design attribute
 - defined D-27
 - in CNDW 6-15
 - when modified D-10
- verilog_use_aligned_widths
 - design attribute
 - defined D-27
 - in Design Property 6-23
 - to control bit widths J-3
 - when modified D-11
- verilog_use_case_default_x
 - design attribute
 - defined D-27
 - in Design Property 6-23
 - to control case statement generation J-3
 - when modified D-11
- verilog_use_indexed_part_select
 - design attribute
 - defined D-27
 - to control part select vs. case statements J-5
 - when modified D-11
- verilog_use_non_blocking_delay_control
 - design attribute
 - defined D-28
 - to control #1 intra-assignment delay J-6
 - when modified D-11
 - with write_rtl E-161
 - with write_sim E-165
- verilog_use_reverse_one_hot
 - design attribute
 - to control select expression and case labels J-2
 - when modified D-11
- verilog_use_state_names
 - design attribute
 - defined D-28
- verilog_use_wire_array
 - design attribute
 - defined D-28
 - to model lookup resources J-4
 - when modified D-11
- virtual function 14-62
- virtual inheritance 14-63
 - defined N-19
 - for specifying TLM interface 14-63
- \$VISUAL
 - with external editors 6-36

W

- w (compiler flag) 6-12
- wait_until
 - not supported 14-111
- WARNING (CTOS-11141) 14-43
- WARNING (CTOS-11252) I-6
- WARNING (CTOS-19026) 10-2
- WARNING (CTOS-19027) 10-2
- WARNING (CTOS-19029) 10-2
- WARNING (CTOS-19030) 10-2
- WARNING (CTOS-19046) 10-2
- WARNING (CTOS-19110) 10-2
- WARNING (CTOS-7027) 10-2
- well-formedness E-170
- while loop
 - transformed to do/while loop 8-56
- width
 - (behavior_)term attribute D-68
 - instance terminal attribute D-74
 - instance terminal bit attribute D-75
 - memory def auxiliary port attribute D-34
 - memory definition attribute D-33
 - module terminal attribute D-80
 - module terminal bit attribute D-82
 - net attribute D-77
 - net bit attribute D-79
 - port attribute D-60
 - RTL IP definition port attribute D-84
- wire array
 - to model lookup resources J-4
- wire array to model lookup resources
 - feature of Verilog 2001 D-28
- wrapper_filename
 - memory definition attribute D-33
- write_foundation_template 13-53
- write_gates
 - syntax E-157
- write_hdl
 - RC command 13-52
- write_ops
 - array attribute D-38
- write_rc_run_script
 - syntax E-158

- write_rc_script
 - syntax E-159
- write_rename_states_script
 - syntax E-160
- write_rtl
 - syntax E-161
 - tcf option 18-6
- write_setup
 - syntax E-164
- write_sim
 - syntax E-165
 - tcf option 18-6
- write_sim_makefile
 - syntax E-168
- write_synth_makefile
 - syntax E-169
- write_tlm_wrapper
 - syntax E-170
- write_top_wrapper
 - syntax E-173
- write_verilog_wrapper
 - syntax E-175
- write_wrapper
 - syntax E-176

X

- .xemacs 6-37
- xemacs 6-37
- Xilinx ISE Design Suite
 - currently supported version 3-2
- xilinx_search_dir
 - memory definition attribute D-33

Z

- Z value
 - not supported 14-112
- zero_value
 - behavior attribute D-45

