SysWip

Alternative Verification

Wishbone Bus Verification IP User Manual

Release 1.0

Table of Contents

1. Introduction

The Wishbone Bus is an open source hardware computer bus which is used by many designs in the OpenCores project.

The Wishbone Verification IP described in this document is a solution for verification of Wishbone master and slave devices. The provided Wishbone verification package includes master and slave verification IPs and examples. It will help engineers to quickly create verification environment and test their Wishbone master and slave devices.

Package Hierarchy

After downloading and unpacking package you will have the following folder hierarchy:

- wishbone_vip
  - docs
  - examples
    - sim
    - testbench
  - verification_ip
    - master
    - slave

The Verification IP is located in the *verification_ip* folder. Just copy the content of this folder to somewhere in your verification environment.

Features

- Easy integration and usage
- Free SystemVerilog source code
- Compliant to Wishbone B3 Protocol
- Operates as a Master or Slave
- Supports 1, 2, 4 and 8 bytes data block size
- Supports single cycle transfers
- Supports wait states injection
- Supports programmable retry and error insertion
- Supports full random timings
- Supports misaligned transfers

Limitations

- Doesn't support TAGs
- Doesn't support Lock signal

2. Wishbone Master

The Wishbone Master Verification IP (VIP) initiates transfers on the Wishbone bus.

Commands

Commands marked as a *queued* will be put in the one queue and executed sequentially.

Commands marked as a *blocking* will block other commands execution until finishing its own process.

- **Configuration Commands**
  - **setRndDelay(): -** *queued, non blocking*
  - **setTimeOut(): -** *queued, non blocking*
- **Data Transfer Commands**
  - **writeData(): -** *queued, non blocking*
  - **readData(): -** *queued, blocking*
  - **busIdle(): -** *queued, non blocking*
  - **pollData(): -** *queued, blocking*
- **Other Commands**
  - **startEnv(): -** *non queued, non blocking*
  - **waitCommandDone():-** *queued, blocking*
  - **printStatus():-** *non queued, non blocking*

Commands Description

All commands are *WSHB_m_env* class methods.

- **setRndDelay()**
  - **Syntax**
    - *setRndDelay(minBurst, maxBurst, minWait, maxWait)*
  - **Arguments**
    - *minBurst:* An *int* variable which specifies minimum value for burst length
    - *maxBurst:* An *int* variable which specifies maximum value for burst length
    - *minWait:* An *int* variable which specifies the minimum value for wait cycles
    - *maxWait:* An *int* variable which specifies the maximum value for wait cycles
  - **Description**
    - Enables/Disables bus random timings. To disable random timing all arguments should be set to zero.

- **setTimeOut()**
  - **Syntax**
    - *setTimeOut(ackTimeOut, rtyTimeOut)*
  - **Arguments**
    - *ackTimeOut:* An *int* variable which specifies the maximum wait clock cycles for slave acknowledge signal.
    - *rtyTimeOut:* An *int* variable which specifies maximum number of retry transfers.
  - **Description**
    - Sets the maximum clock cycles for slave acknowledge signal and maximum number of retry transfers. If slave delays acknowledge or generates retry more then specified, error message will be generated.
- **writeData()**
  - **Syntax**
    - *writeData(addr, inBuff)*
  - **Arguments**
    - *addr:* An *int* variable that specifies the start byte address
    - *inBuff:* 8 bit vector queue that contains data buffer which should be transferred
  - **Description**
    - Writes data buffer.
- **readData()**
  - **Syntax**
    - *readData(addr, outBuff, dataLength)*
  - **Arguments**
    - *addr:* An *int* variable that specifies the start byte address
    - *outBuff:* 8 bit vector queue that contains read data buffer
    - *dataLength:* An *int* variable that specifies the amount of bytes which should be read.
  - **Description**
    - Read data buffer.

- **pollData()**
  - **Syntax**

        - *pollData(addr, pollData, pollTimeOut)*
    - **Arguments**
        - *addr:* An *int* variable that specifies the start address
        - *pollData:* 8 bit vector queue that contains data buffer which should be compared with read data buffer
        - *pollTimeOut:* An *int* variable that specifies the poll time out clock cycles.
    - **Description**
        - Read data buffer starting from *addr* and compare it with *pollData* buffer. Repeat until buffers are equal or until time out occurred.
- **busIdle()**
    - **Syntax**
        - *busIdle(idleCycles)*
    - **Arguments**
        - *idleCycles:* A *int* variable which specifies wait clock cycles
    - **Description**
        - Holds the bus in the idle state for the specified clock cycles
- **waitCommandDone()**
    - **Syntax**
        - *waitCommandDone()*
    - **Description**
        - Waits until all commands in the command buffer are finished
- **printStatus()**
    - **Syntax**
        - *printStatus()*
    - **Description**
        - Prints all errors occurred during simulation time and returns error count. If there are no any errors returns zero.
- **startEnv()**
    - **Syntax**
        - *startEnv()*
    - **Description**
        - Starts Wishbone master environment. Don't use data transfer commands before the environment start.

Integration and Usage

The Wishbone Master Verification IP integration into your environment is very easy. Instantiate the *wshb_m_if* interface in you testbench and connect interface ports to your DUT. Then during compilation don't forget to compile *wshb_m.sv* and *wshb_m_if.sv* files located inside the *wshb_vip/verification_ip/master* folder.

For usage the following steps should be done:

1. Import *WSHB_M* package into your test.
   - **Syntax***: import WSHB_M::*;*
2. Create *WSHB_m_env* class object
   - **Syntax***: WSHB_m_env wshb= new(wshb_ifc_m, dataSize);*
   - **Description:** *wshb_ifc_m* is the reference to the Wishbone Master Interface instance name. *dataSize* is data block size in bytes. Use only 1, 2, 4 and 8.
3. Start Wishbone Master Environment.
   - **Syntax:** *wshb.startEnv();*

This is all you need for Wishbone master verification IP integration.

3. Wishbone Slave

The Wishbone Slave Verification IP models Wishbone slave device. It has an internal memory which is accessible by master devices as well as by corresponding commands.

Commands

Commands marked as a *queued* will be put in the one queue and executed sequentially.

Commands marked as a *blocking* will block other commands execution until finishing its own process.

- **Configuration Commands**
  - **setRndDelay(): -** *queued, non blocking*
  - **setRespMode():-** *queued, non blocking*
  - **setMemCleanMode():** - *queued, non blocking*
- **Data Processing Commands**
  - **putData(): -** *non queued, non blocking*
  - **getData(): -** *non queued, non blocking*
  - **pollData():** - *non queued, blocking*
- **Other Commands**
  - **startEnv(): -** *non blocking, should be called only once for current object*
  - **printStatus():** - *non queued, non blocking*

Commands Description

All commands are *WSHB_s_env* class methods.

- **setRndDelay()**
  - **Syntax**
    - *setRndDelay*(minAckDelay, maxAckDelay)
  - **Arguments**
    - *minAckDelay:* A *int* variable that specifies minimum value for acknowledge delay
    - *maxAckDelay:* A *int* variable that specifies maximum value for acknowledge delay
  - **Description**
    - Enables/Disables acknowledge random delays. To disable acknowledge delay set all arguments to zero.
- **setRespMode()**
  - **Syntax**
    - *setRespMode*(errAddr, rtyAddr, errEn, rtyNum)
  - **Arguments**
    - *errAddr:* A 32 *bit vector* which specifies slave error generation address
    - *rtyAddr:* A 32 *bit vector* which specifies slave retry generation address
    - *errEn*: An *int* variable which enables or disables slave error generation
    - *rtyNum*: An *int* variable which specifies the number of retry transfers
  - **Description**
    - Sets slave error and retry generation addresses.
- **setMemCleanMode()**
  - **Syntax**
    - *setMemCleanMode(memClean)*
  - **Arguments**
    - *memClean:* An *int* variable which specifies internal memory clean mode
  - **Description**
    - Sets internal memory clean mode. 0 – no clean. 1- Only master read will clean memory cell. 2 – Only *getData* command will clean memory cell. 3 – The both master read and *getData* command will clean memory cell.
- **putData()**
  - **Syntax**

        - *putData(startAddr, dataInBuff*)
    - **Arguments**
        - *startAddr:* An *int* variable that specifies the start byte address
        - *dataInBuff:* 8 bit vector queue that contains data buffer which will be written to the memory.
    - **Description**
        - Writes data buffer to the to the internal memory
- **getData()**
    - **Syntax**
        - *getData(startAddr, dataOutBuff, lenght)*
    - **Arguments**
        - *startAddr:* An *int* variable that specifies the start byte address
        - *dataOutBuff:* 8 bit vector queue that contains read data from memory.
        - *length:* An *int* variable that specifies the amount of bytes which will be read from the internal memory.
    - **Description**
        - Reads data from the internal memory.
- **pollData()**
    - **Syntax**
        - *pollData(address, pollData, pollTimeOut)*
    - **Arguments**
        - *address:* An *int* variable that specifies the start address
        - *pollData:* 8 bit vector queue that contains data buffer which should be compared with read data buffer
        - *pollTimeOut:* An *int* variable that specifies the poll time out clock cycles.
    - **Description**
        - Read data buffer from internal memory and compare it with *pollData* buffer. Repeat until buffers are equal or until time out occurred.
- **startEnv()**
    - **Syntax**
        - *startEnv()*
    - **Description**
        - Starts Wishbone slave environment.
- **printStatus()**
    - **Syntax**
        - *printStatus()*

- **Description**

  Prints all errors occurred during simulation time and returns error count. If there are no any errors returns zero.

Integration and Usage

The Wishbone Slave Verification IP integration into your environment is very easy. Instantiate the *wshb_s_if* interface in you testbench and connect interface ports to your DUT. Then during compilation don't forget to compile *wshb_s.sv* and *wshb_s_if.sv* files located inside the *wshb_vip/verification_ip/slave* folder.

For usage the following steps should be done:

1. Import *WSHB_S* package into your test.
   - **Syntax**: *import WSHB_S::*;*
2. Create *WSHB_s_env* class object
   - **Syntax:** *WSHB_s_env wshb = new(wshb_ifc_s, dataSize);*
   - **Description:** *wshb_ifc_s* is the reference to the Wishbone Slave interface instance name. *dataSize* is data block size in bytes. Use only 1, 2, 4 and 8.
3. Start WIshbone Slave Environment
   - **Syntax:** *wshb.startEnv();*

Now Wishbone slave verification IP is ready to respond transactions initiated by master device. Use data processing commands to put or get data from the internal memory.

4. Important Tips

In this section some important tips will be described to help you to avoid VIP wrong behavior.

Master Tips

1. Call *startEnv()* task as soon as you create *WSHB_m_env* object before any other commands. You should call it not more then once for current object.
2. Before using Data Transfer Commands be sure that external hardware reset is done. As current release does not support external reset detection feature, the best way is to wait before DUT reset is done.

Slave Tips

1. Call *startEnv()* task as soon as you create *WSHB_s_env* object before any other commands. It should be called before the first valid transaction initiated by Wishbone master.