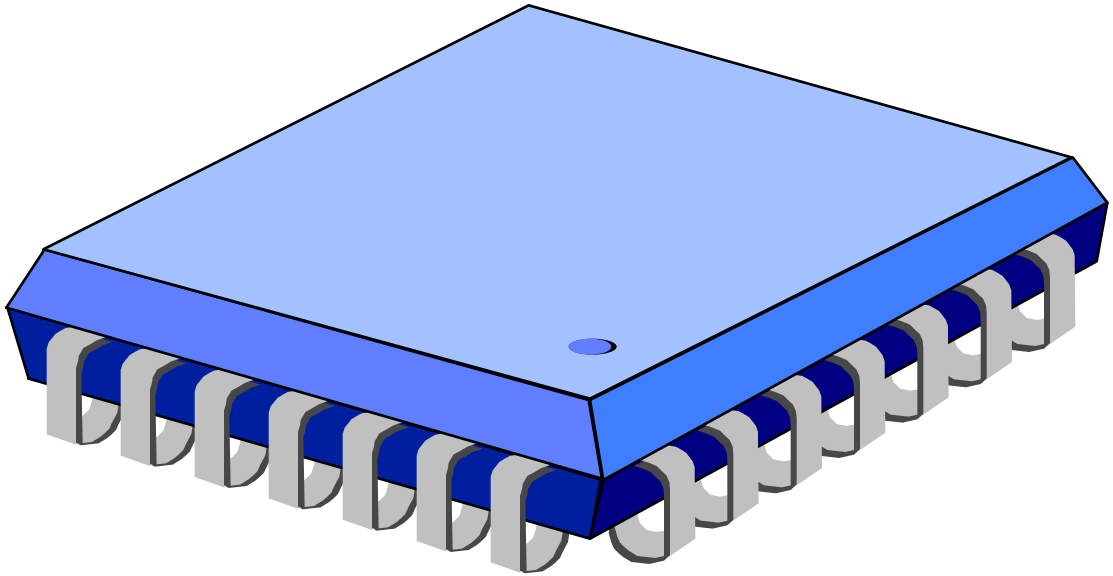
ARCHITECTUREs ARM



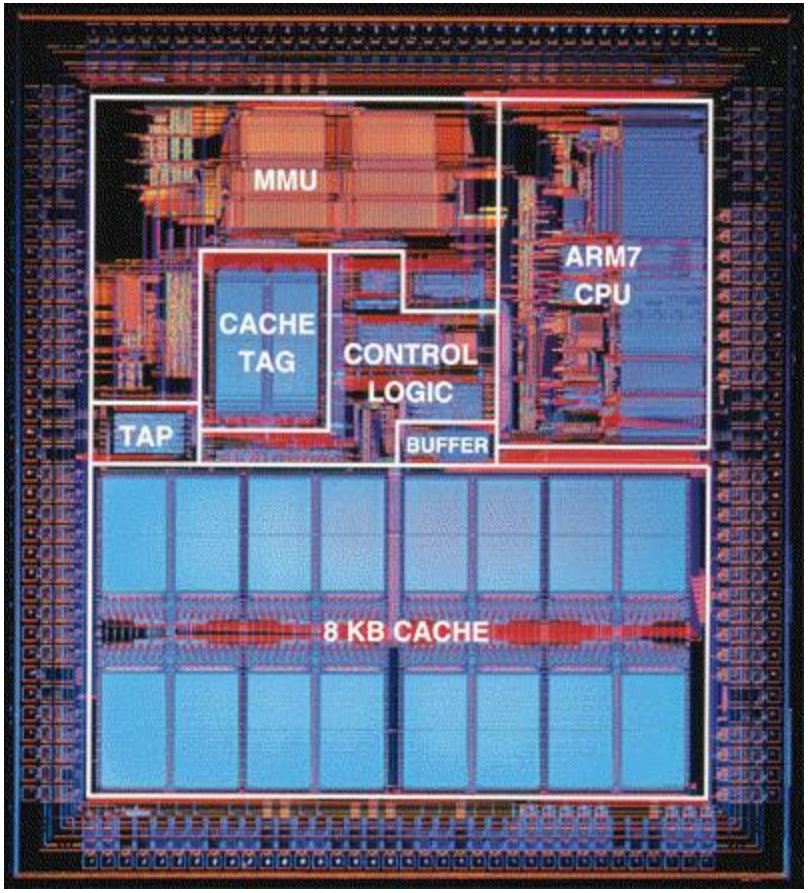
Plan

- **ARM7TDMI**
- **Cortex**
  - Cortex A family
  - Cortex M family
- **System bus**

Qu'est ce qu'un ARM7TDMI?

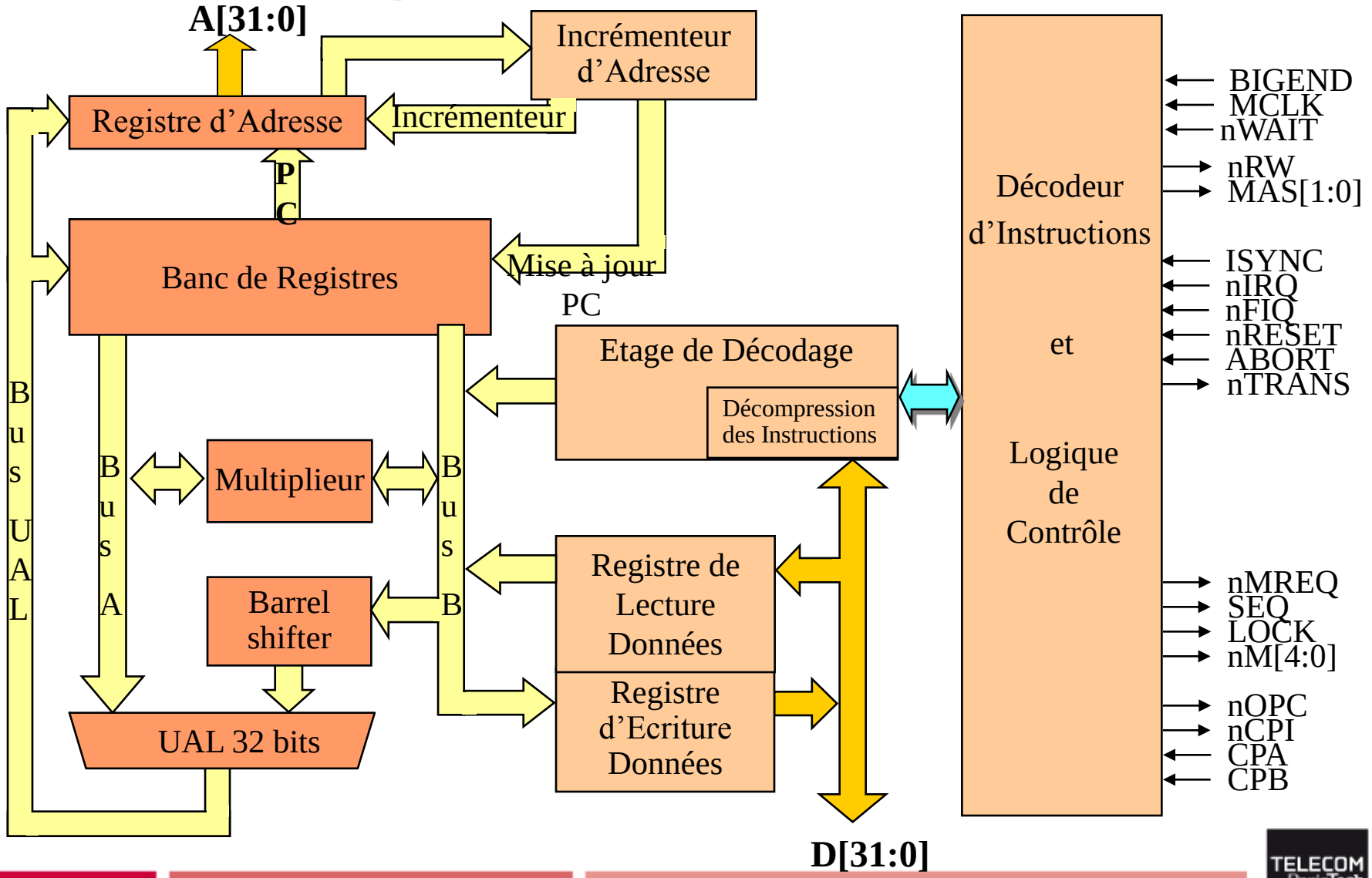
- **Processeur à Architecture « Von Neumann »**
  - Même bus mémoire pour instructions et données
- **3 étages de pipeline : Fetch, Decode, Execute**
- **Instructions sur 32 Bits**
- **2 instructions d'accès à la mémoire LOAD et STORE**
- **T : support du mode "Thumb" (instructions sur 16 bits)**
- **D : extensions pour la mise au point**
- **M : Multiplieur 32x8 et instructions pour résultats sur 64 bits.**
- **I : émulateur embarqué ("Embedded ICE")**

Vue d'une puce utilisant un ARM7

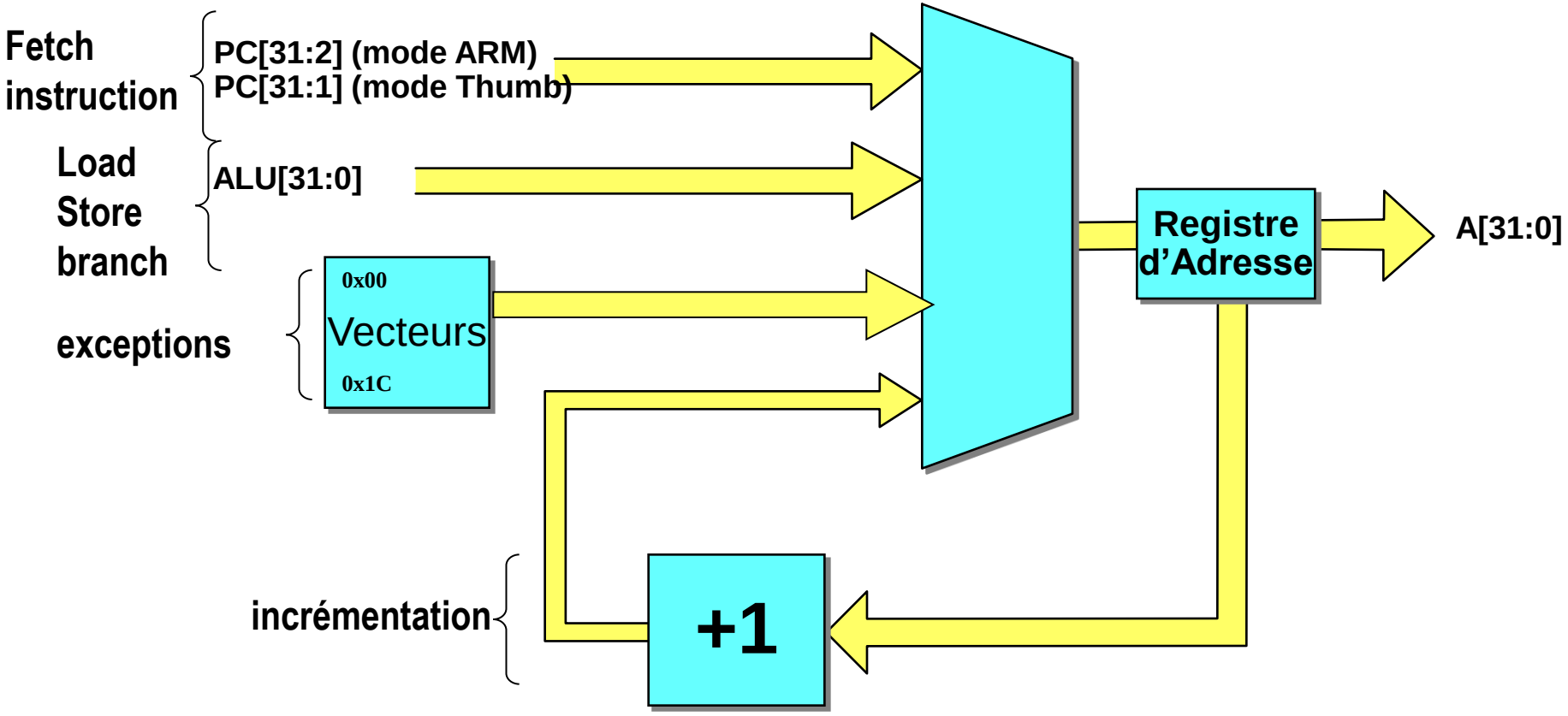


ARM710 (25mm² en 0.5µm (1995), 2.9 mm² en 0.18µm (2000))

Vue simplifiée du Cœur ARM7TDMI

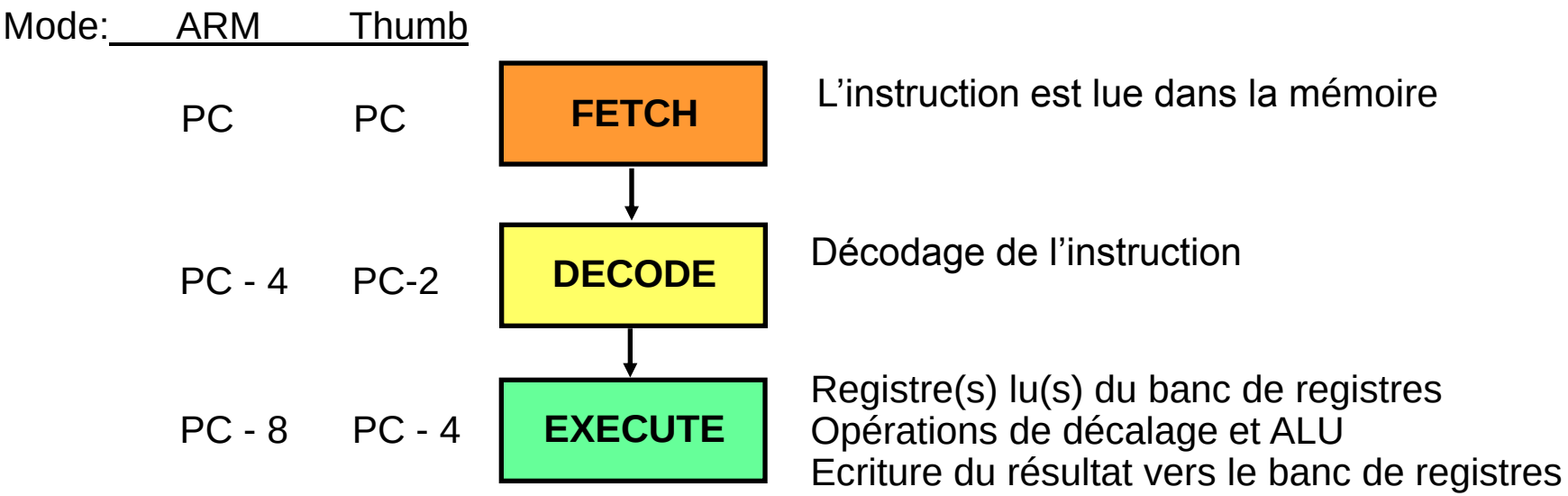


Génération des Adresses



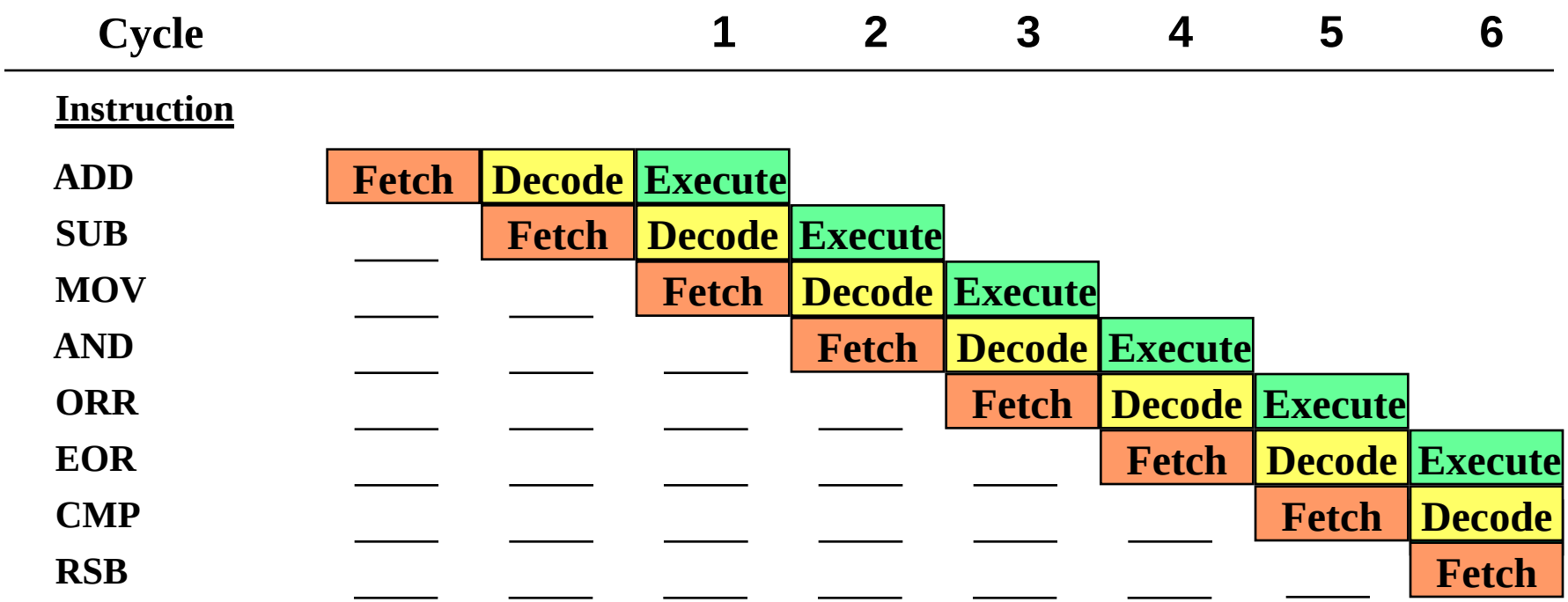
Le Pipeline d'Instructions

- La famille ARM7 utilise un pipeline à 3 étages pour augmenter la vitesse du flot d'instructions dans le microprocesseur.



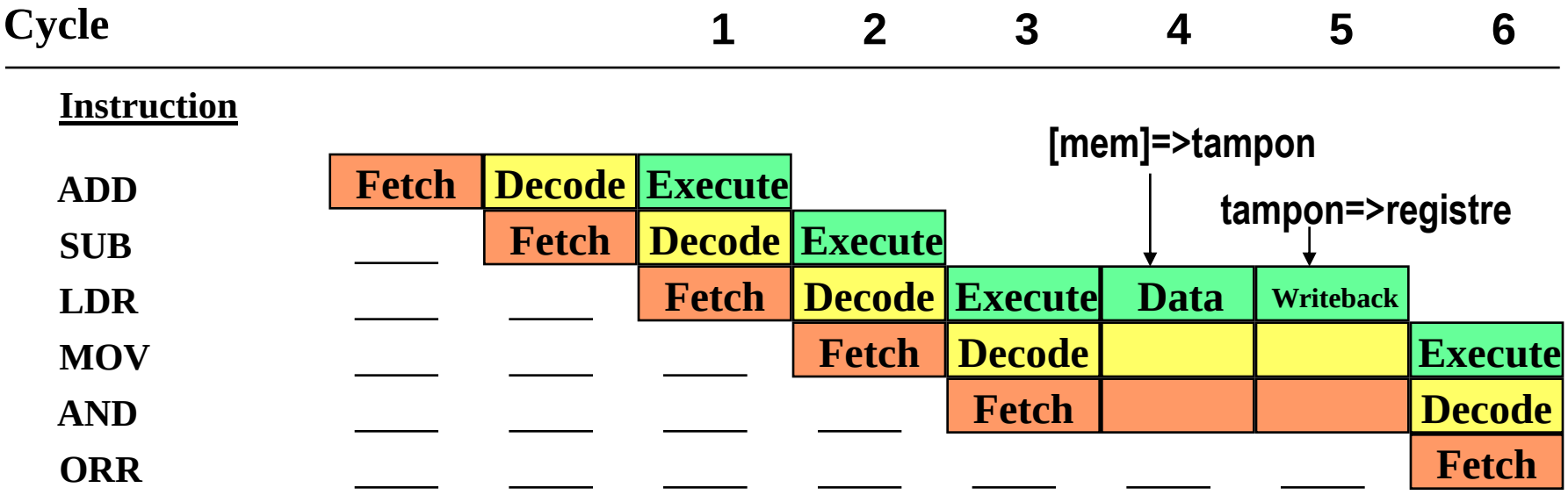
Le PC pointe sur l'instruction en cours de lecture (FETCHed), et non sur l'instruction en cours d'exécution.

Exemple: Pipeline Optimal



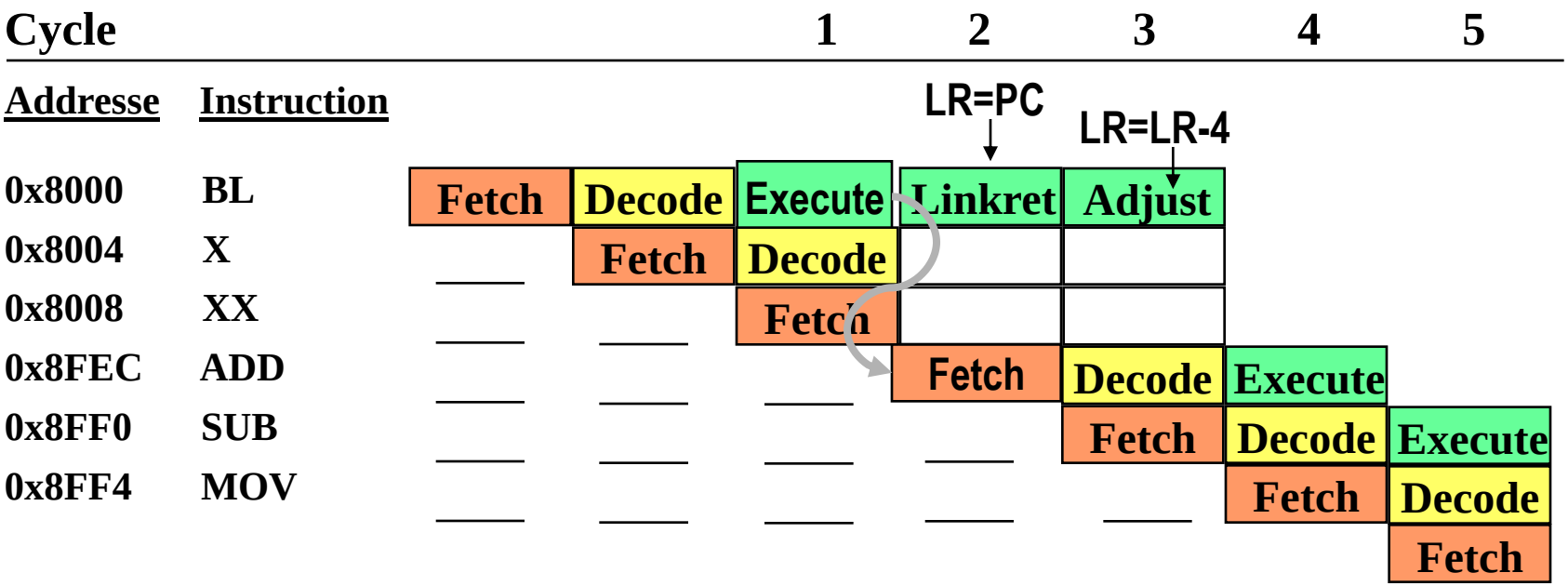
- il faut 6 cycles pour exécuter 6 instructions (CPI “Cycles Per Instruction”)=1
- Toutes les operations ne jouent que sur des registres (1 cycle)

Exemple: Pipeline avec LDR



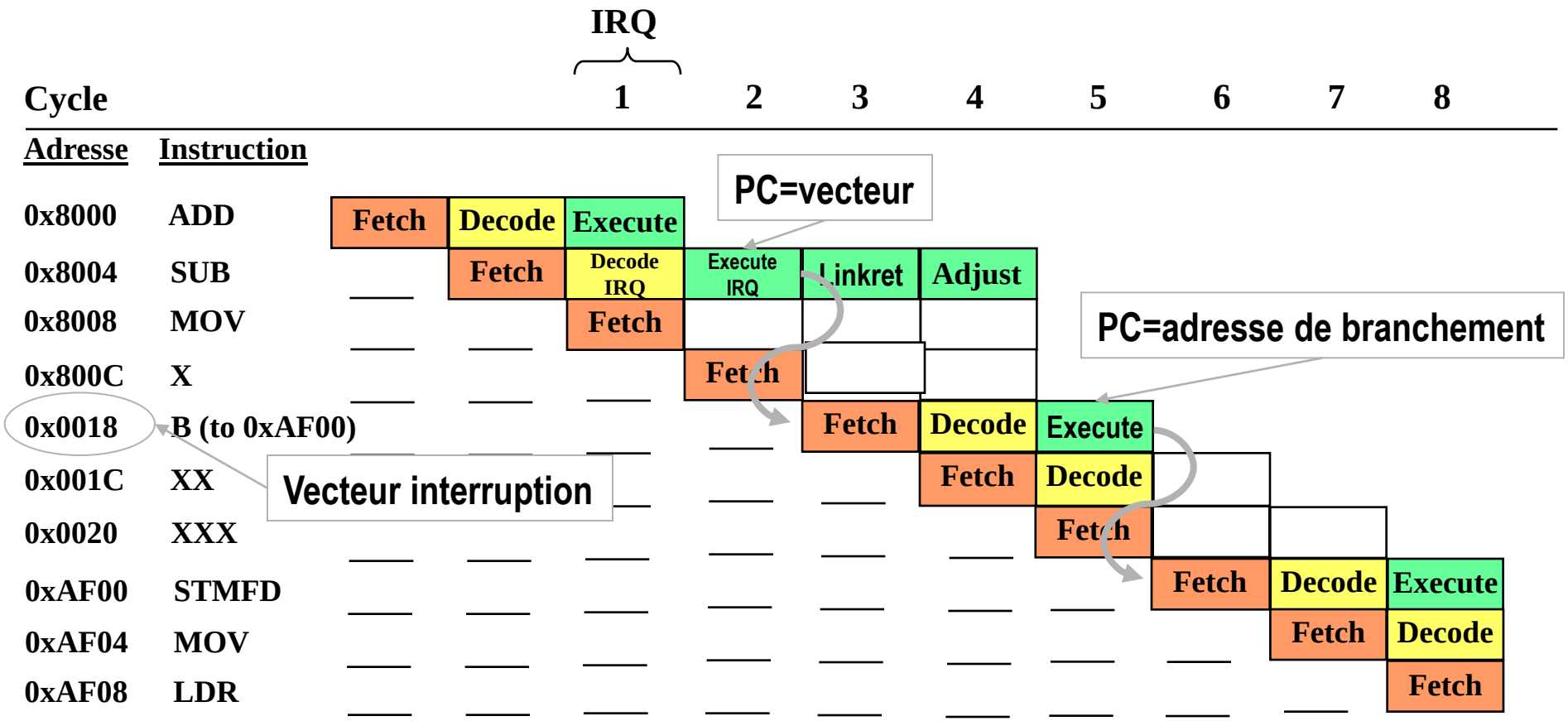
- L'instruction LDR lit une donnée en mémoire et la charge dans un registre
- il faut 6 cycles pour exécuter 4 instructions. CPI = 1,5

Exemple: Pipeline avec Saut



- L'instruction BL effectue un appel de sous-programme
- Le pipeline est cassé et 2 cycles sont perdus

Exemple: Pipeline avec Interruption



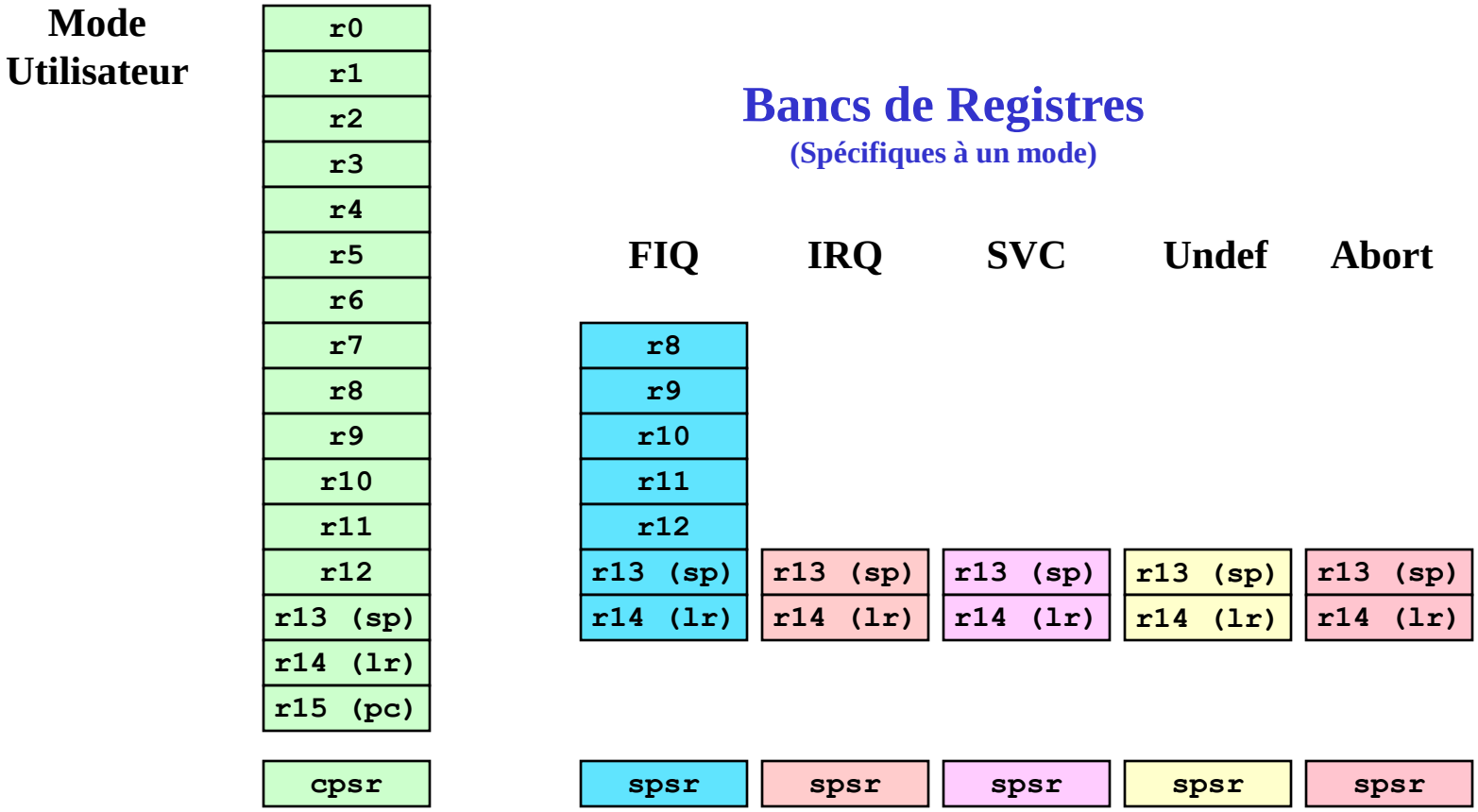
* Latence **minimum** pour le service de l'interruption IRQ = 7 cycles

Les Modes du Microprocesseur

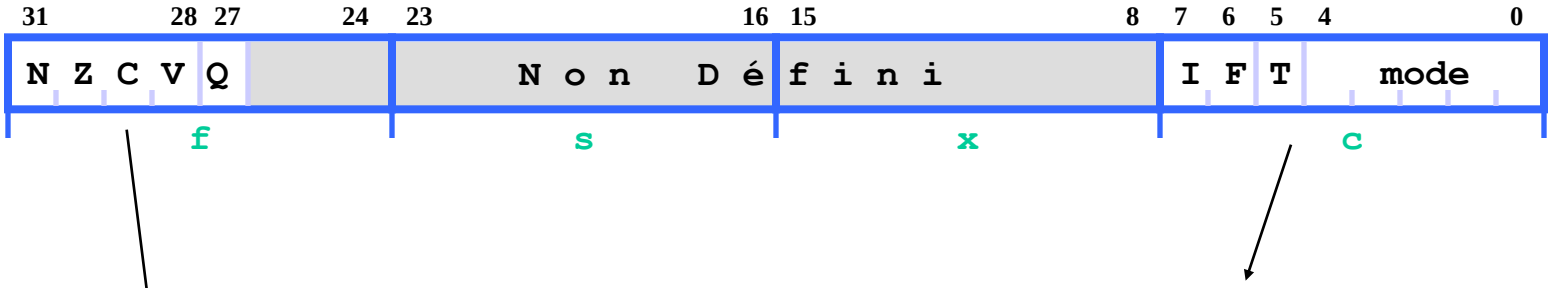
- **Un microprocesseur ARM a 7 modes opératoires de base :**
  - **User** : mode sans privilège où la plupart des tâches s'exécutent
  - **FIQ** : on y entre lors d'une interruption de priorité haute (rapide)
  - **IRQ** : on y entre lors d'une interruption de priorité basse (normale)
  - **Supervisor** : on y entre à la réinitialisation et lors d'une interruption logicielle (SWI "SoftWare Interrupt")
  - **Abort** : utilisé pour gérer les violations d'accès mémoire
  - **Undef** : utilisé pour gérer les instructions non définies ("undefined")
  - **System** : mode avec privilège utilisant les mêmes registres que le mode User

Les Registres

Registres actifs



Les Registres d'État CPSR et SPSR



- Indicateurs conditionnels
  - N = Résultat Négatif
  - Z = Résultat nul (Zéro)
  - C = Retenue (Carry)
  - V = Débordement (oVerflow)
  - Q = débordement avec mémoire

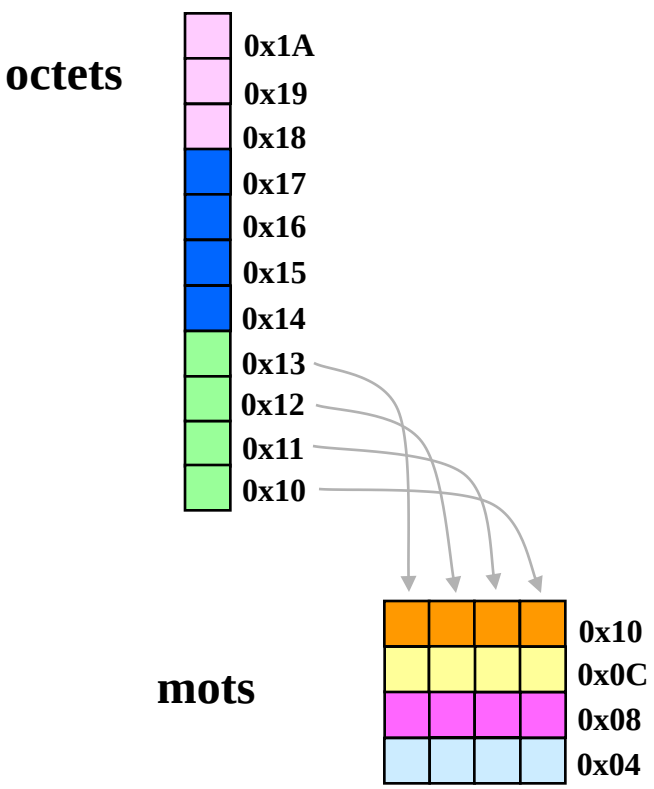
- Validation des interruptions
  - I = 1 dévalide IRQ.
  - F = 1 dévalide FIQ.
- Mode Thumb
  - T
- Indicateurs de mode
  - Indiquent le mode actif :

Accès à la Mémoire et aux E/S

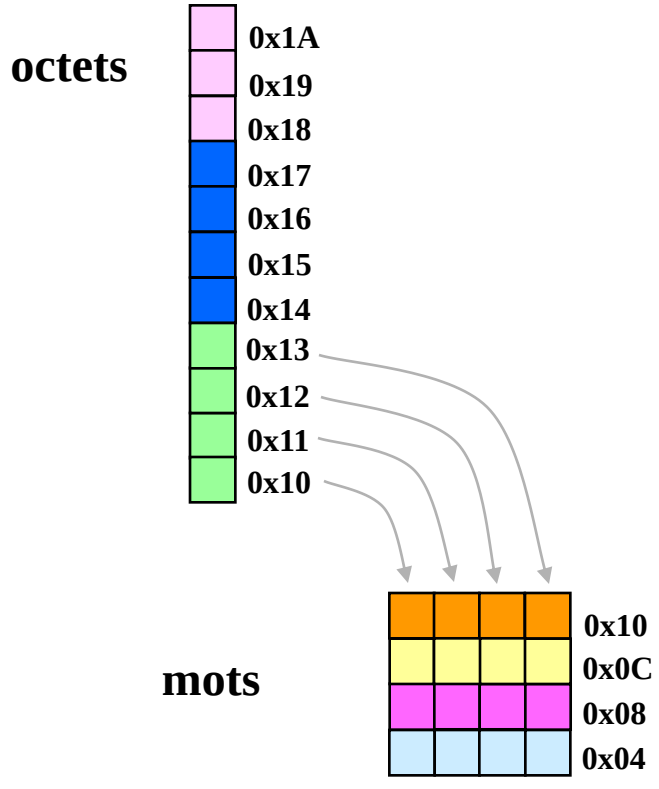
- **2 instructions d'accès :**
  - LOAD (LDR) et STORE (STR)
- **L'adressage mémoire se fait sur 32 bits**
  - => 4 Go.
- **Type des données :**
  - octets
  - demi-mots (16 bits)
  - mots (32 bits)
- **Les mots doivent être alignés sur des adresses multiples de 4 et les demi-mots, de 2.**
- **Les E/S sont dans la « mappe » mémoire**

Organisation de la mémoire

La mémoire peut être vue comme une ligne d'octets repliée en mots.
2 façon d'organiser 4 octets en mot :



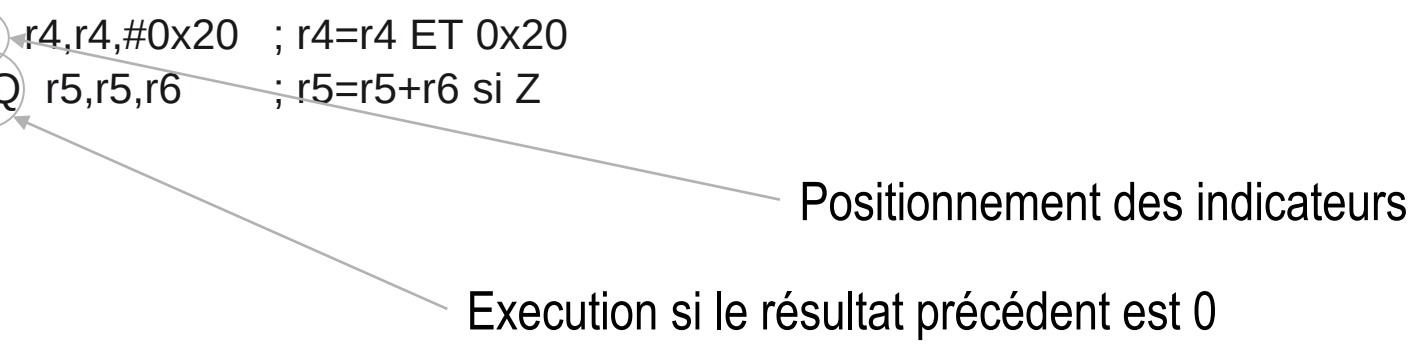
« Little Endian »



« Big Endian »

Jeu d'Instructions ARM(1)

- **Toutes les instructions ont 32 bits**
- **La plupart des instructions s'exécutent en un seul cycle**
- **Les instructions peuvent être exécutées conditionnellement**
- **Architecture Load/Store**
  - Instructions de traitement de données
    - SUB r0,r1,#5 ; r0= r1-5
    - ADD r2,r3,r3,LSL #2 ; r2=R3+4*r3=5*r3
    - ANDS r4,r4,#0x20 ; r4=r4 ET 0x20
    - ADDEQ r5,r5,r6 ; r5=r5+r6 si Z



Jeu d'Instructions ARM(2)

- Instructions spécifiques d'accès à la mémoire
  - `LDR r0,[r1],#4` *; r0=mem(r1), r1= r1+4*
  - `STRNEB r2,[r3,r4]` *; mem(r3+r4)=r2*
  - `LDRSH r5,[r6,#8]!` *; r5=mem(r6+8), r6=r6+8*
  - `STMFD sp!,{r0,r2-r4}` *; transferts multiples*

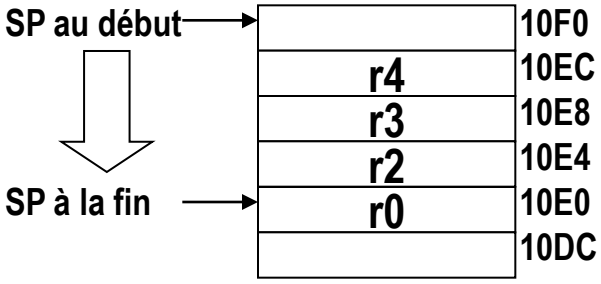
;empilage : mem(sp+i)=liste de registres

Si Z=0

En octet

En 16 bits
Avec extension
de signe sur les
16MSBs

r6=r6+8 en fin d'execution

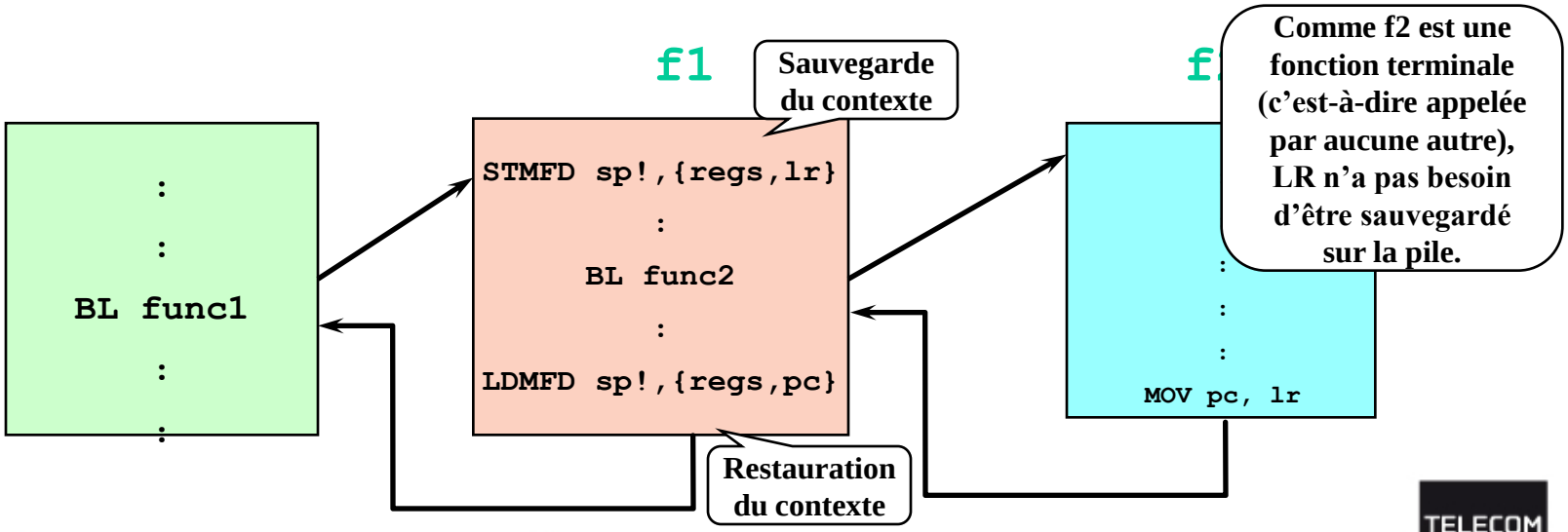


Opération réciproque de dépilage :

```
LDMFD  sp!,{r0,r2-r4}
; liste de registres=mem(sp+i
```

Jeu d'Instructions ARM(3)

- **Branchement et sous programmes :**
- **B <étiquette>**
  - Calculé par rapport au PC. Étendue du branchement: ±32 Mbyte.
- **BL <subroutine>**
  - Stocke l'adresse de retour dans le registre LR
  - Le retour se fait en rechargeant le registre LR dans le PC
  - Pour les fonctions non terminales, LR devra être sauvegardé



Executions conditionnelles

- La plupart des instructions peuvent être exécutées conditionnellemnt aux indicateurs Z,C,V,N
- `CMP  r0,#8  ; r0=8?`
- `BEQ  fin  ; si oui (Z=1) PC=fin`
- `ADD  r1,r1,#4 ;`

- `Équivalent à`
- `CMP  r0,#8  ; r0=8?`
- `ADDNE r1,r1,#4  ; si non (Z=0)`

+ petit et
+ rapide

Conditions courantes : EQ, NE,PL,MI,CS,VS

=0, ≠0,≥0,<0,carry set, debordement

Exemple: Séquence d'Instructions

```
LDR     R0, [R8, 0x10]
ADD     R1, R0, R4, LSL #2
STR     R1, [R8, 0x14]
```

charger le mot de l'adresse [R8+0x10] dans R0

R1 ← R0 + (R4 << 2)

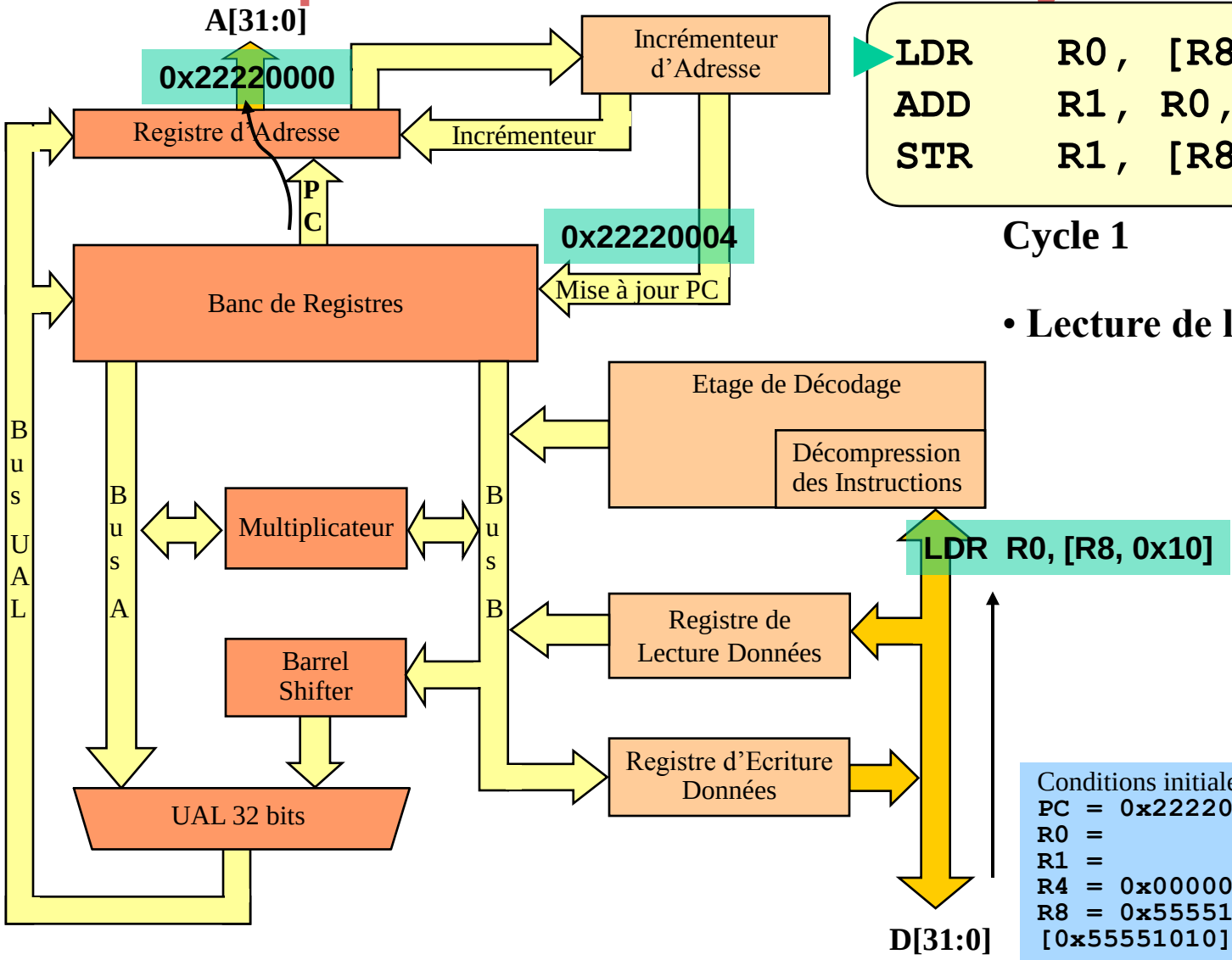
ranger le contenu de R1 à l'adresse [R8+0x14]

Conditions initiales :

```
PC = 0x22220000
R4 = 0x00000721
R8 = 0x55551000
[0x55551010] = 0x00000834
```

Les diagrammes suivants supposent que les instructions précédentes s'exécutent en un cycle mais ne montrent pas leur comportement.

Séquence d'Instructions : Cycle 1




```
LDR    R0, [R8, 0x10]
ADD    R1, R0, R4, LSL #2
STR    R1, [R8, 0x14]
```

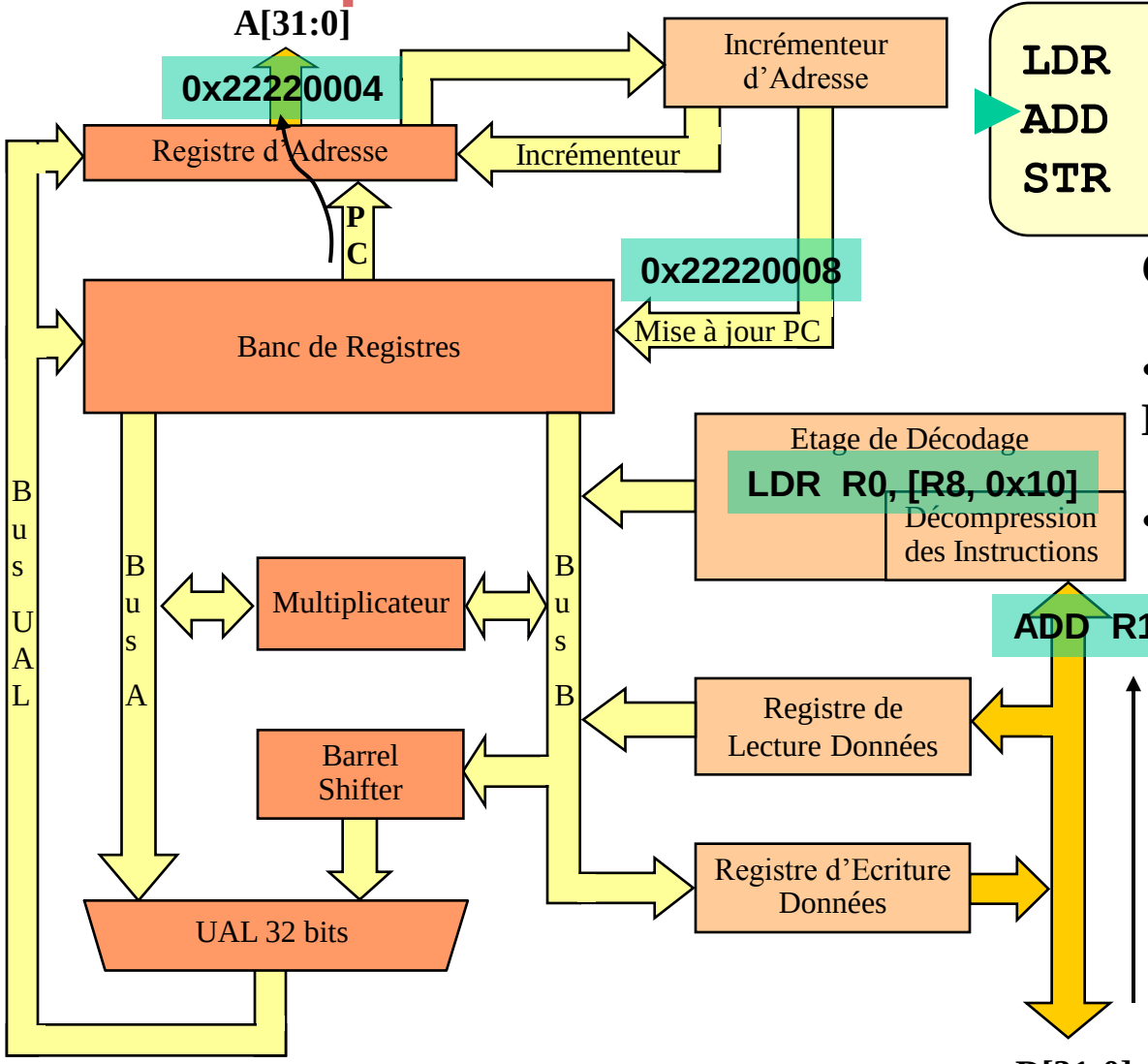
Cycle 1

- **Lecture de l'instruction LDR**

Conditions initiales :

```
PC = 0x22220000
R0 =
R1 =
R4 = 0x00000721
R8 = 0x55551000
[0x55551010] = 0x00000834
```

Séquence d'Instructions : Cycle 2



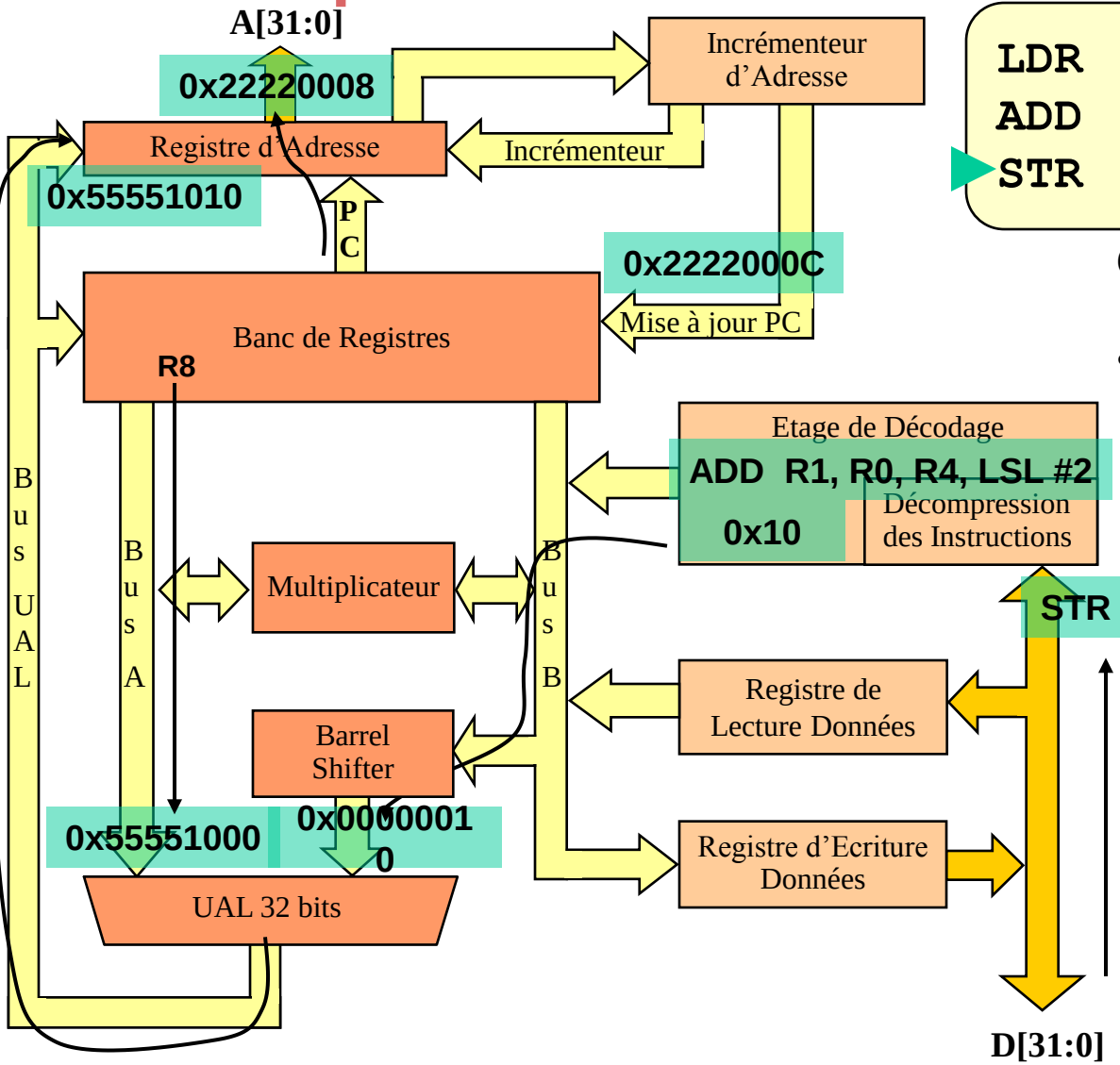

```
LDR    R0, [R8, 0x10]
ADD    R1, R0, R4, LSL #2
STR    R1, [R8, 0x14]
```

Cycle 2

- **Décodage de l'instruction LDR**
- **Lecture de l'instruction ADD**

```
PC = 0x22220004
R0 =
R1 =
R4 = 0x00000721
R8 = 0x55551000
[0x55551010] = 0x00000834
```

Séquence d'Instructions : Cycle 3



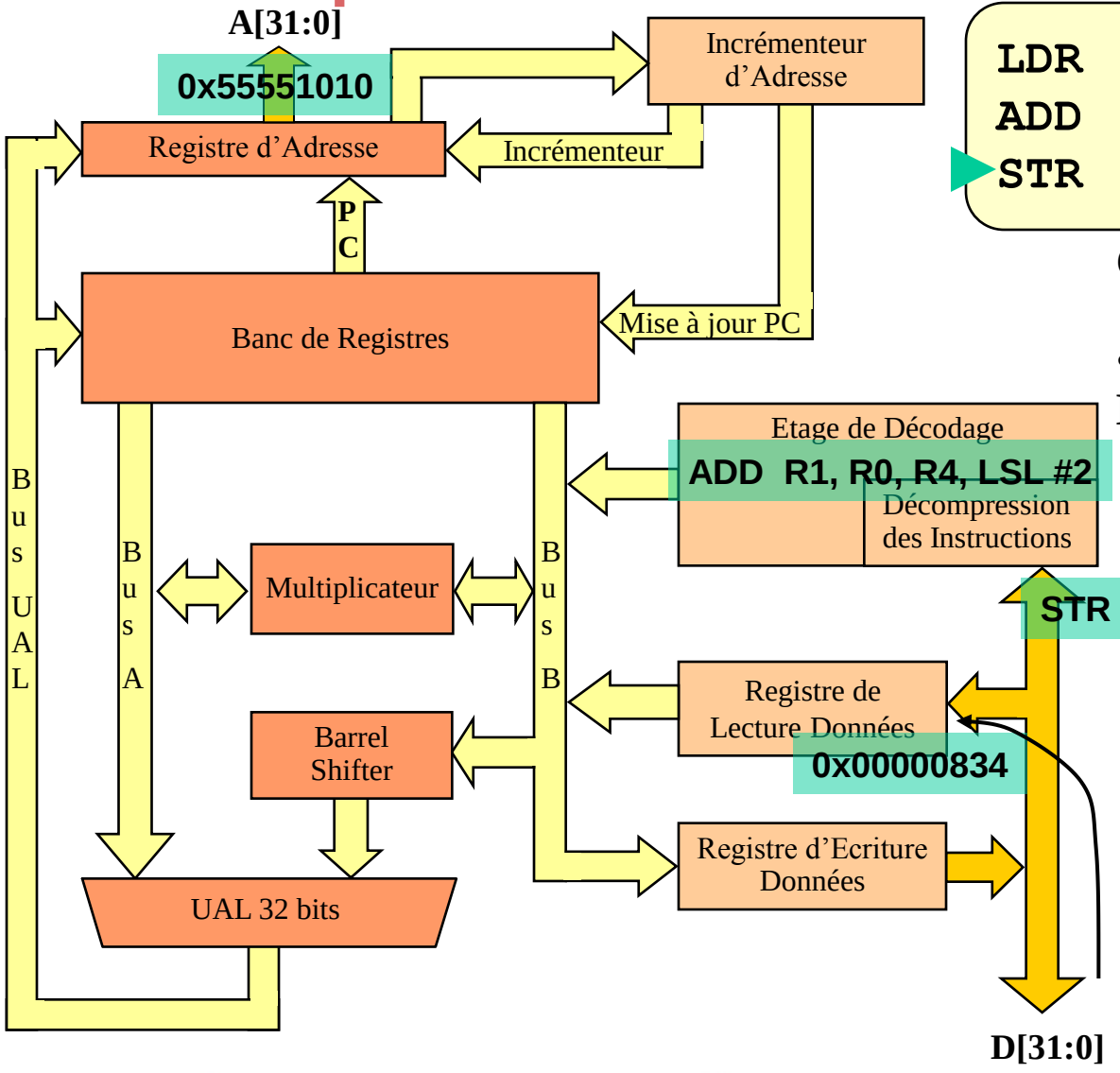

```
LDR    R0, [R8, 0x10]
ADD    R1, R0, R4, LSL #2
STR    R1, [R8, 0x14]
```

Cycle 3

- 1er cycle d'exécution de LDR
  - Calcul de l'adresse en mémoire de données
  - Décodage de l'instruction ADD
- Lecture de l'instruction STR

```
PC = 0x2222000C
R0 =
R1 =
R4 = 0x00000721
R8 = 0x55551000
[0x55551010] = 0x00000834
```

Séquence d'Instructions : Cycle 4



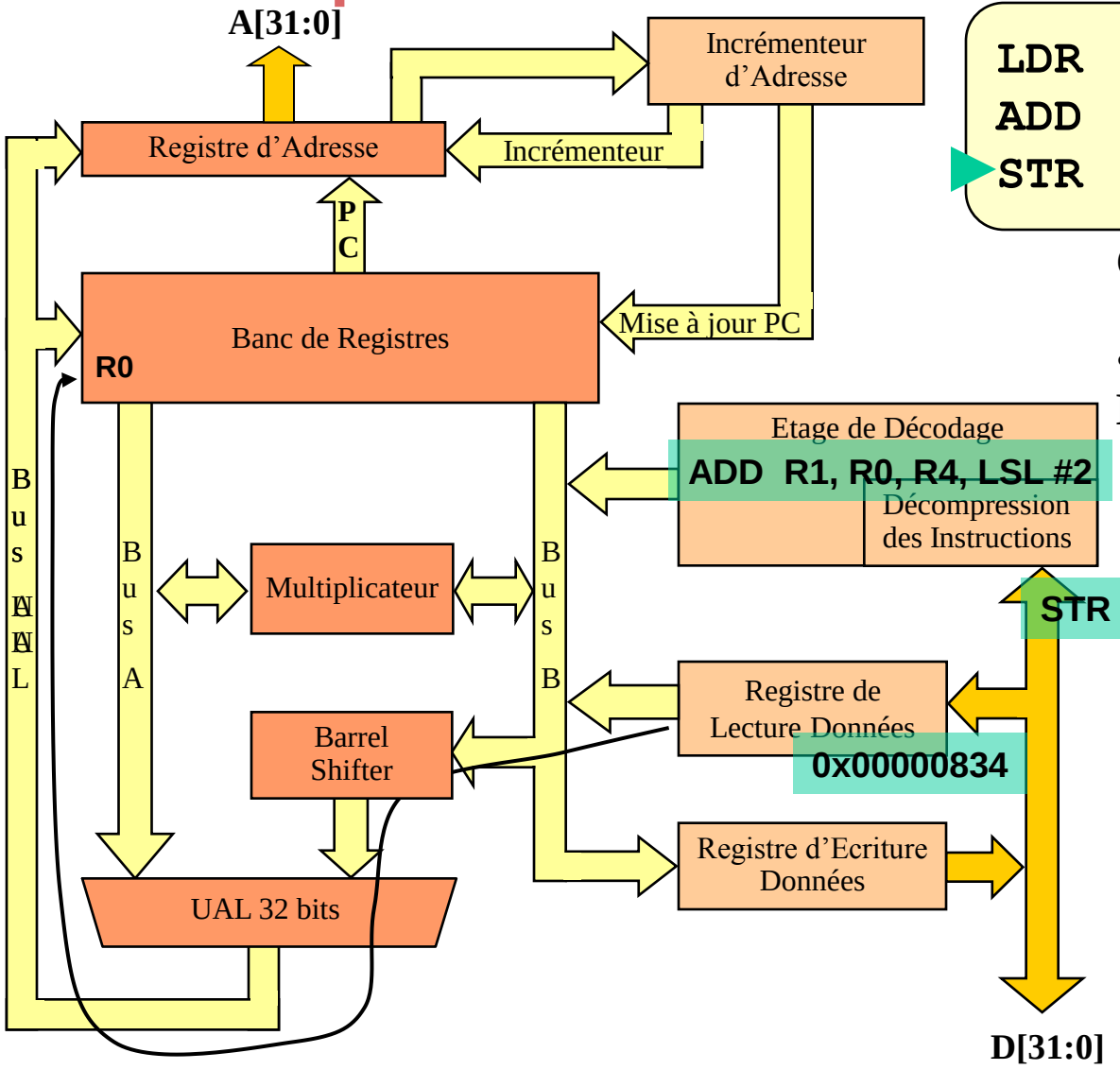

```
LDR    R0, [R8, 0x10]
ADD    R1, R0, R4, LSL #2
STR    R1, [R8, 0x14]
```

Cycle 4

- 2ème cycle d'exécution de LDR
  - Lecture de la mémoire de données

```
PC = 0x2222000C
R0 =
R1 =
R4 = 0x00000721
R8 = 0x55551000
[0x55551010] = 0x00000834
```

Séquence d'Instructions : Cycle 5



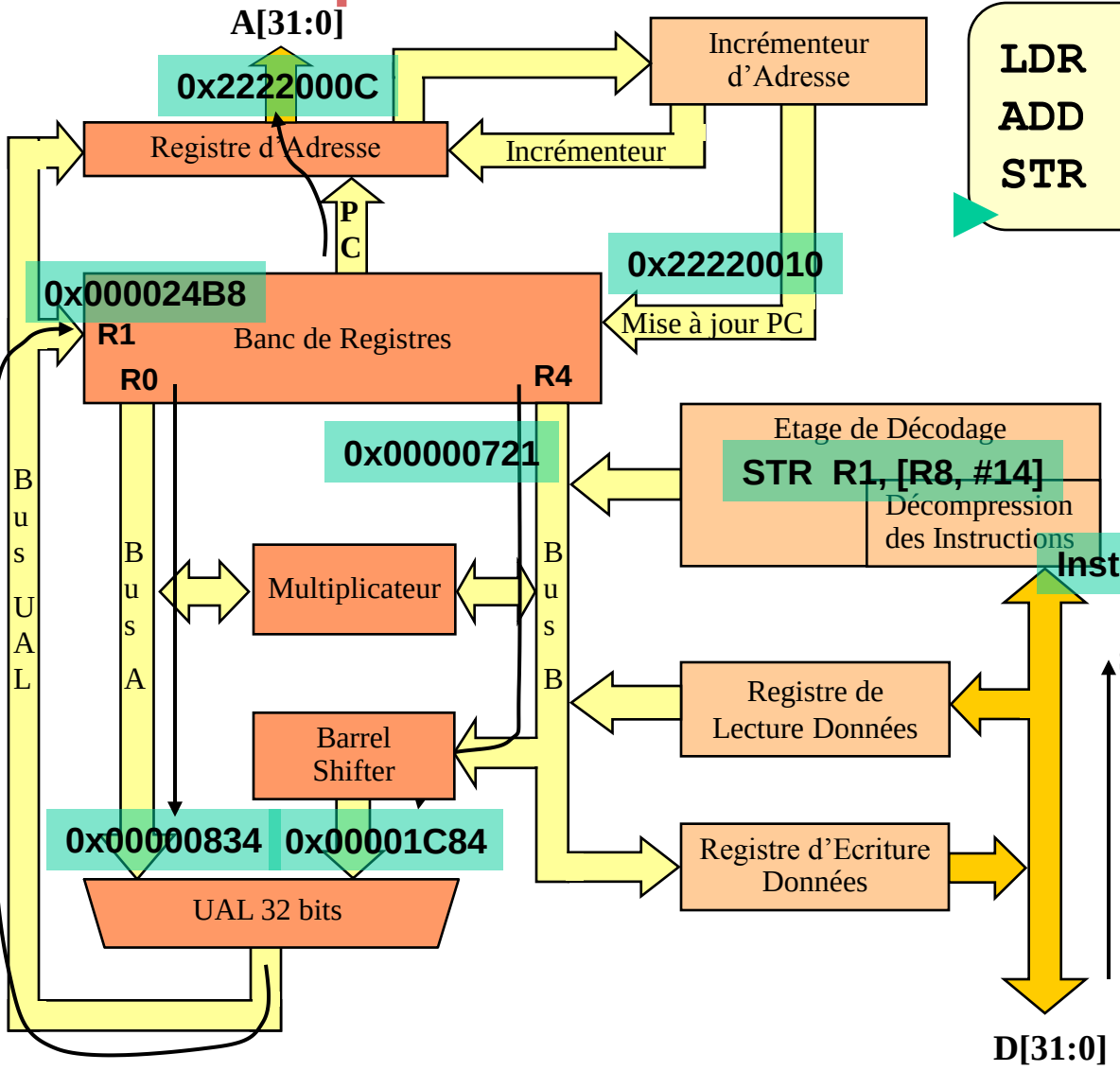

```
LDR    R0, [R8, 0x10]
ADD    R1, R0, R4, LSL #2
STR    R1, [R8, 0x14]
```

Cycle 5

- **3ème cycle d'exécution de LDR :**
  - **Transfert de la donnée dans le registre-destination**

```
PC = 0x2222000C
R0 = 0x00000834
R1 =
R4 = 0x00000721
R8 = 0x55551000
[0x55551010] = 0x00000834
```

Séquence d'Instructions : Cycle 6



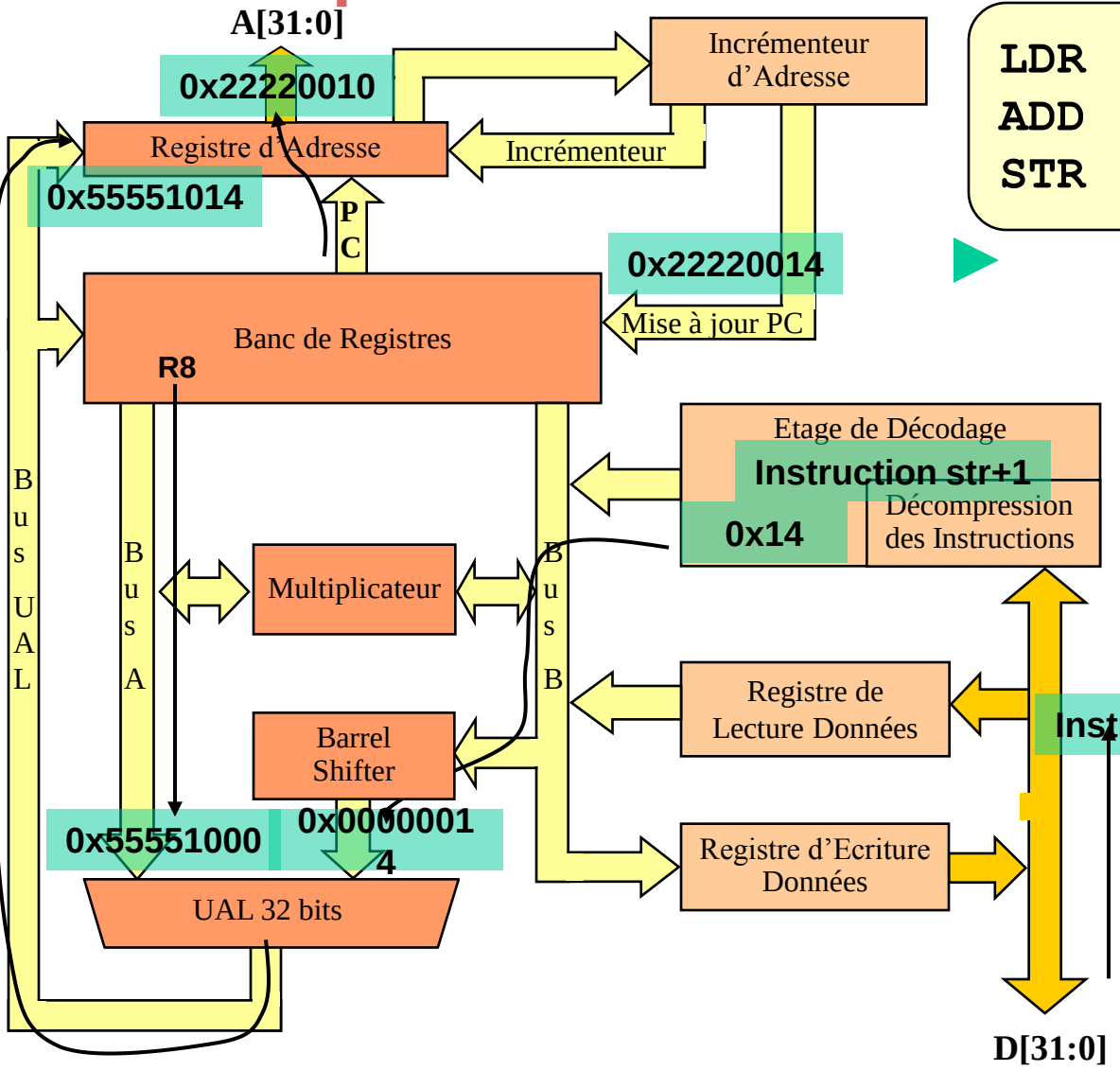

```
LDR    R0, [R8, 0x10]
ADD    R1, R0, R4, LSL #2
STR    R1, [R8, 0x14]
```

Cycle 6

- **Exécution de ADD**
- **Décodage de l'instruction STR**
- **Lecture de l'instruction suivant STR**

```
PC = 0x22220010
R0 = 0x00000834
R1 = 0x000024B8
R4 = 0x00000721
R8 = 0x55551000
[0x55551010] = 0x00000834
```

Séquence d'Instructions : Cycle 7



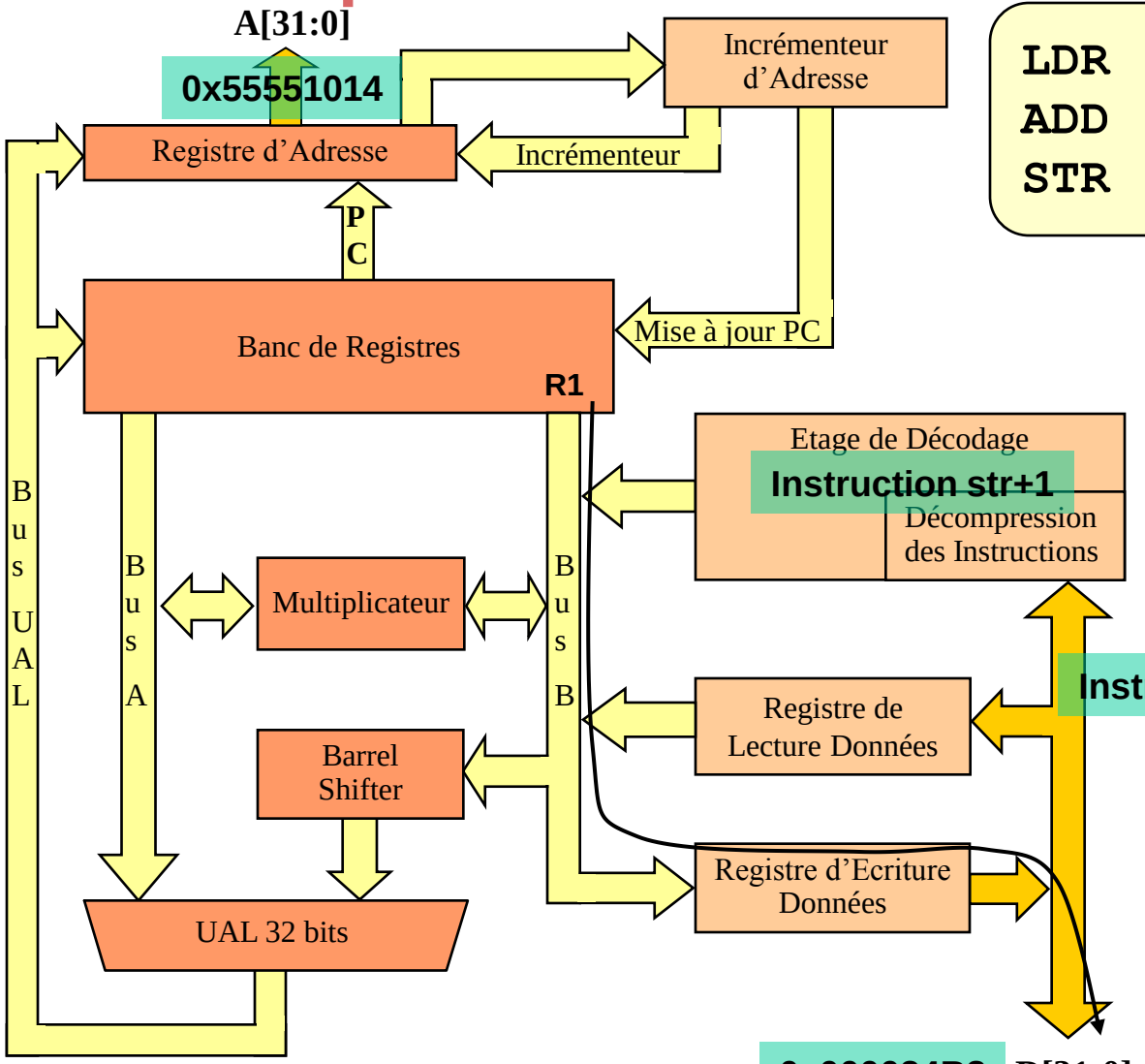

```
LDR    R0, [R8, 0x10]
ADD    R1, R0, R4, LSL #2
STR    R1, [R8, 0x14]
```

Cycle 7

- **1er cycle d'exécution de STR**
  - **Calcul de l'adresse en mémoire de données**
  - **Décodage de l'instruction suivant STR**
  - **Lecture de l'instruction STR+2**

```
PC = 0x22220014
R0 = 0x00000834
R1 = 0x000024B8
R4 = 0x00000721
R8 = 0x55551000
[0x55551010] = 0x00000834
```

Séquence d'Instructions : Cycle 8




```
LDR    R0, [R8, 0x10]
ADD    R1, R0, R4, LSL #2
STR    R1, [R8, 0x14]
```

Cycle 8

- 2ème cycle d'exécution de STR
  - Ecriture en mémoire de données

```
PC = 0x22220014
R0 = 0x00000834
R1 = 0x000024B8
R4 = 0x00000721
R8 = 0x55551000
[0x55551014] = 0x000024B8
```

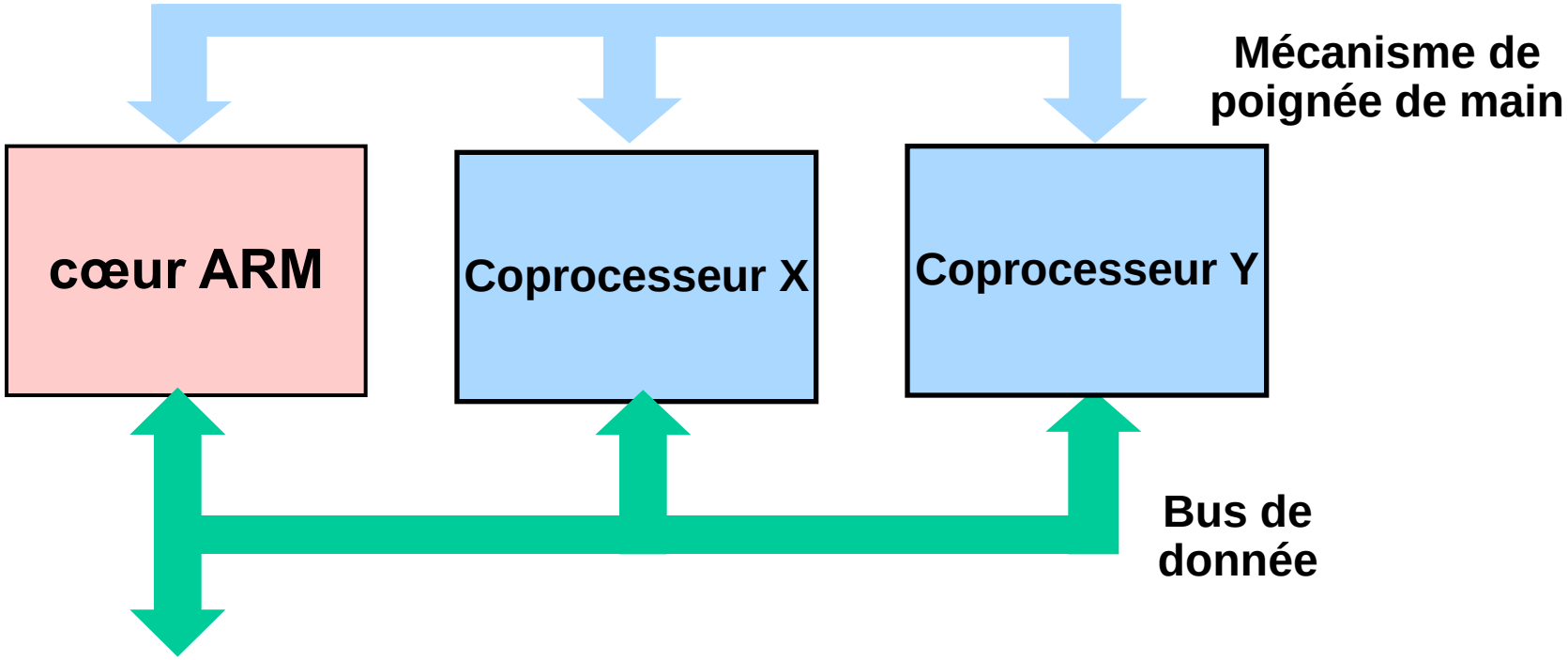
Exceptions

L'activation d'une exception donne lieu au passage dans une mode particulier et au branchement dans un programme par le biais d'une table de vecteurs
7 sortes :

TYPE	MODE	VECTEUR	Retour en USER
RESET	Supervisor	0x00000000	
Instruction indéfinie	Undef	0x00000004	MOVS PC,r14
Interruption logicielle SWI	Supervisor	0x00000008	MOVS PC,r14
Problème Fetch instruction	Abort	0x0000000C	SUBS PC,r14,#4
Problème Fetch donnée	Abort	0x00000010	SUBS PC,r14,#8
Interruption matérielle IRQ	IRQ	0x00000018	SUBS PC,r14,#4
Interruption matérielle FIRQ	FIQ	0x0000001C	SUBS PC,r14,#4

Le CSPR et le PC sont restaurés en même temps

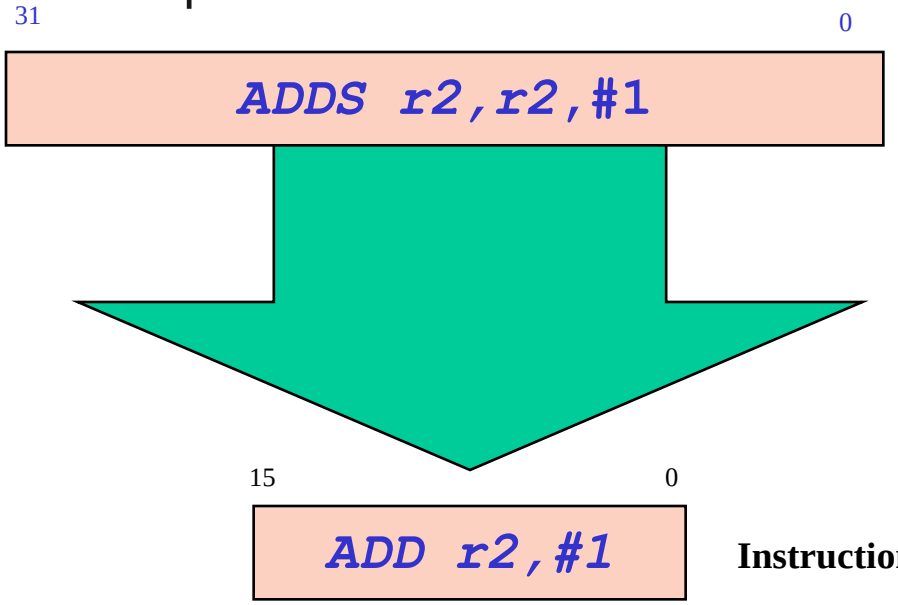
Coprocesseurs



- 16 coprocesseurs peuvent être définis pour étendre le jeu d'instruction ARM
- Les coprocesseurs accèdent à la mémoire et aux registres directement par le biais d'instructions spécifiques
- Le coprocesseur 15 est dédié au contrôleur cache et MMU

Mode Thumb

- **Jeu d'instructions codé sur 16 bits**
  - Optimisé pour la densité de code à partir de code écrit en langage C
  - Augmente les performances pour des espaces mémoires réduits
  - Sous ensemble des fonctionnalités du jeu d'instructions ARM
- **Le cœur a deux modes d'exécution : ARM et Thumb**
  - On passe d'un mode à l'autre en utilisant l'instruction *BX*



Pour la plupart des instructions générées par le compilateur:

- **L'exécution conditionnelle n'est pas utilisée**
- **Les registres Source et Destination sont identiques**
- **Seul les premiers registres sont utilisés**
- **Les constantes sont de taille limitée**
- **Le registre à décalage n'est pas utilisé au sein d'une même instruction**

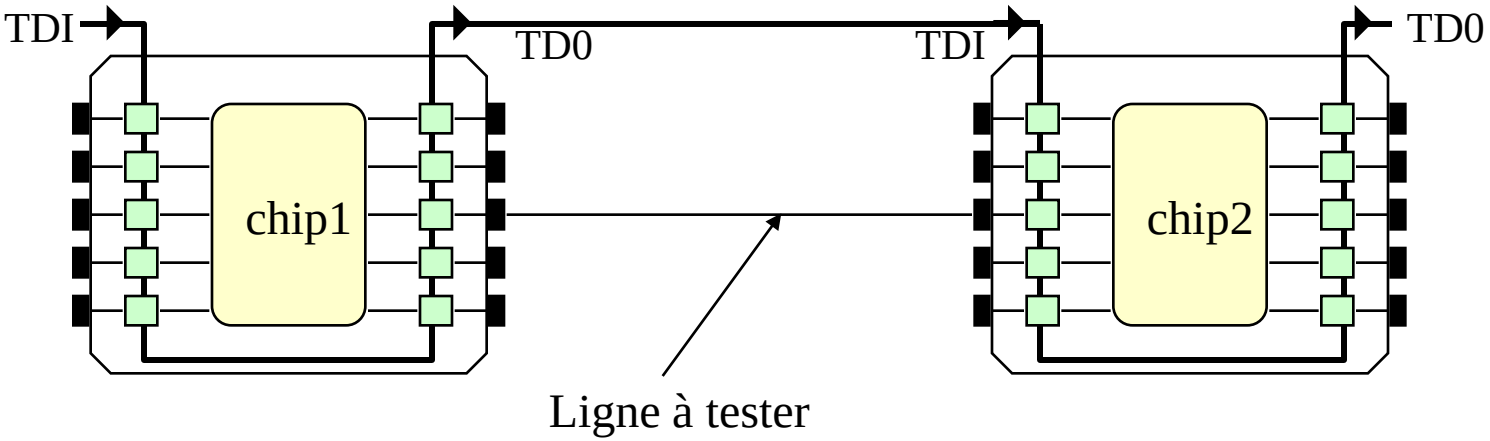
Instruction en mode Thumb (16 bits)

Port JTAG

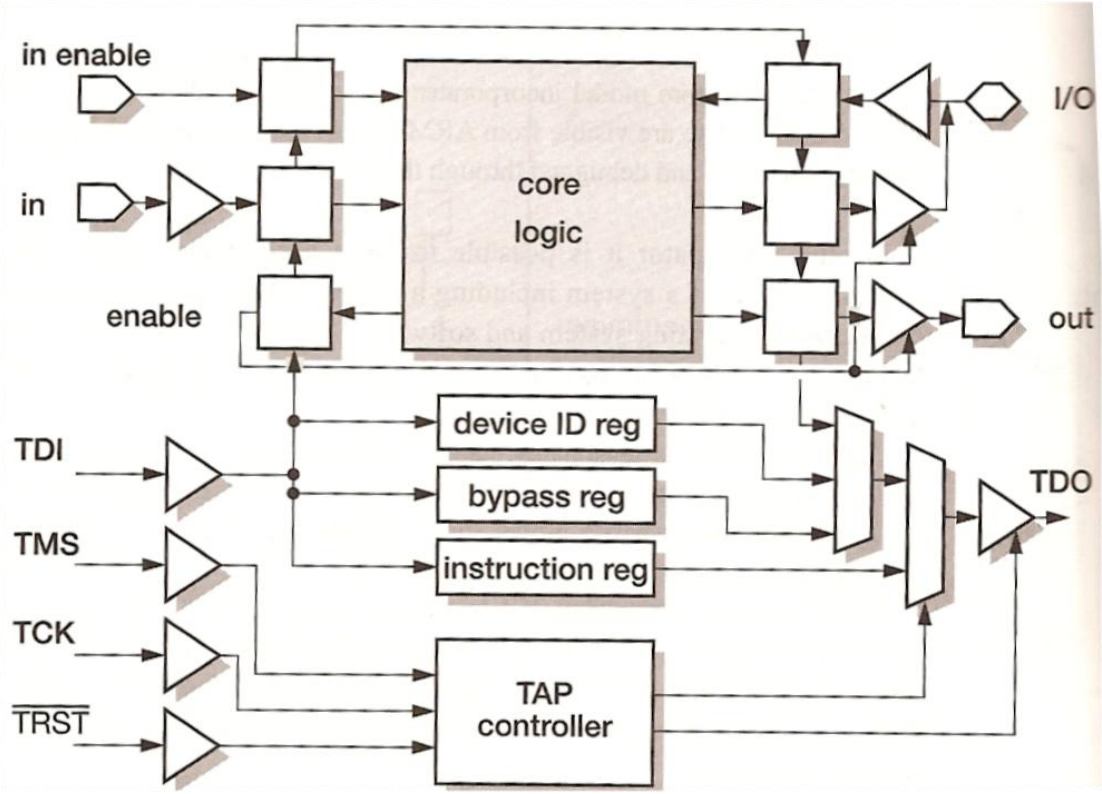
Le JTAG est un contrôleur interne ou "TAP controller" qui permet

- Test de la connectivité (par Boundary Scan testing)
- Accès aux resssources internes des processeurs pour le débogage
- Chargement de netlists pour les FPGAs
- Fonctions personnalisées

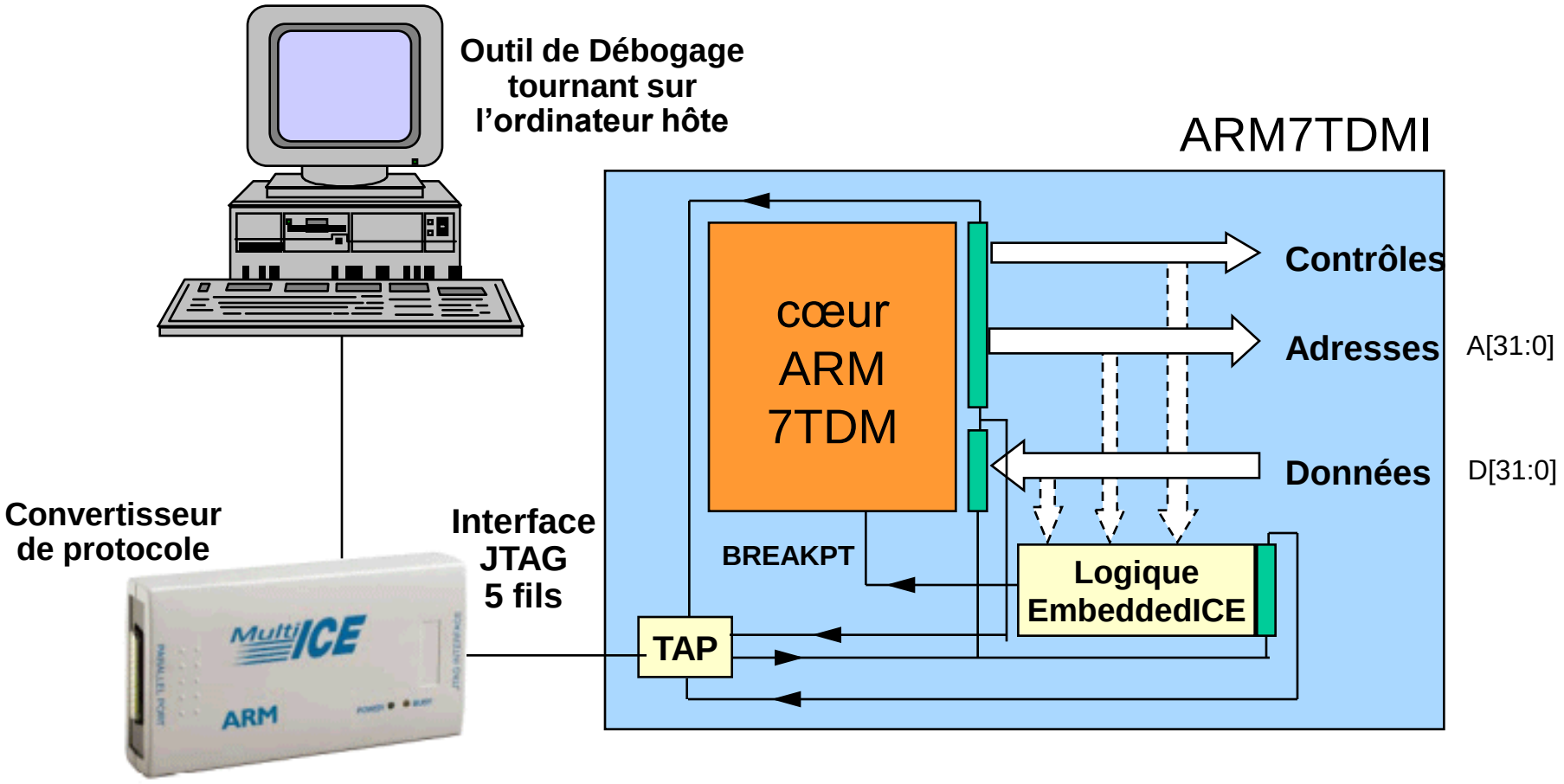
Boundary Scan Test :



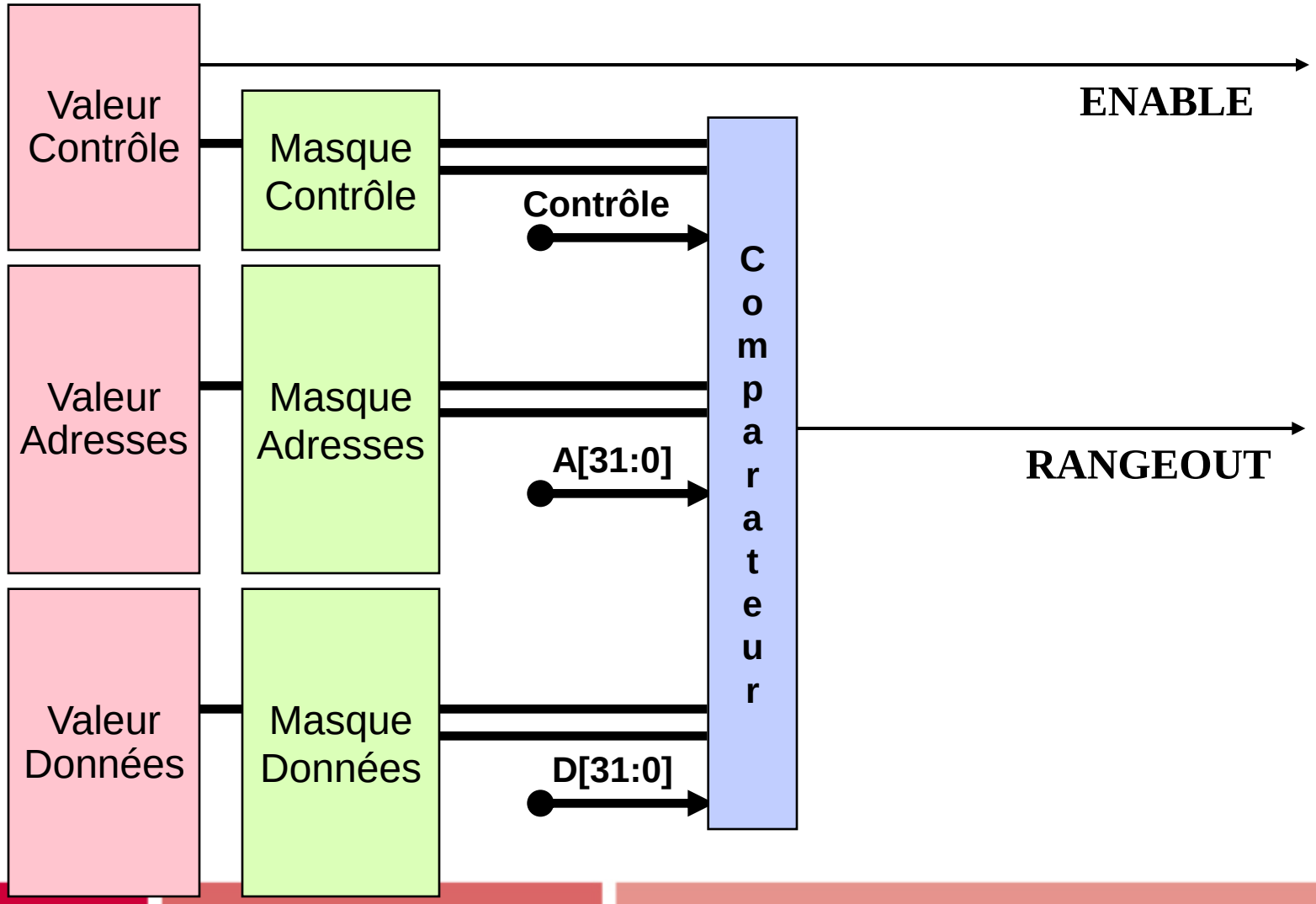
JTAG architecture



Débogage Embarqué



Unité de Gestion des Points d'Arrêt



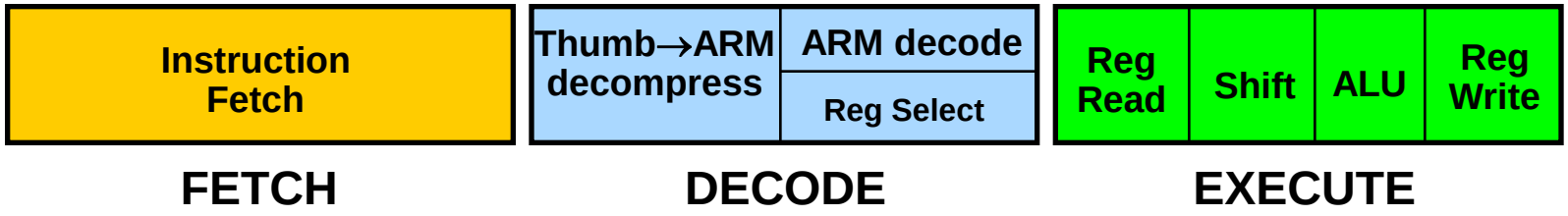
ARM9TDMI

- **Implémentation à double bus (architecture Harvard)**
  - Augmente la bande passante entre le microprocesseur et la mémoire
    - Interface mémoire Instructions
    - Interface mémoire Données
  - Permet l'accès simultané aux mémoires Instructions et Données
  - => Modifications pour améliorer le nombre de cycles par instruction (CPI "Cycles Per Instruction") jusqu'à ~1.5

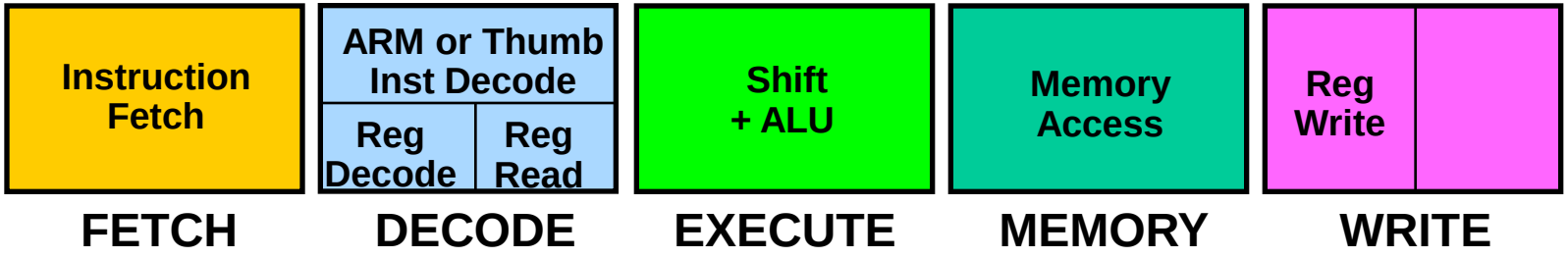
- **5 niveaux de pipeline**
  - => Modifications pour améliorer la fréquence maximum de l'horloge

Modifications du Pipeline pour le ARM9TDMI

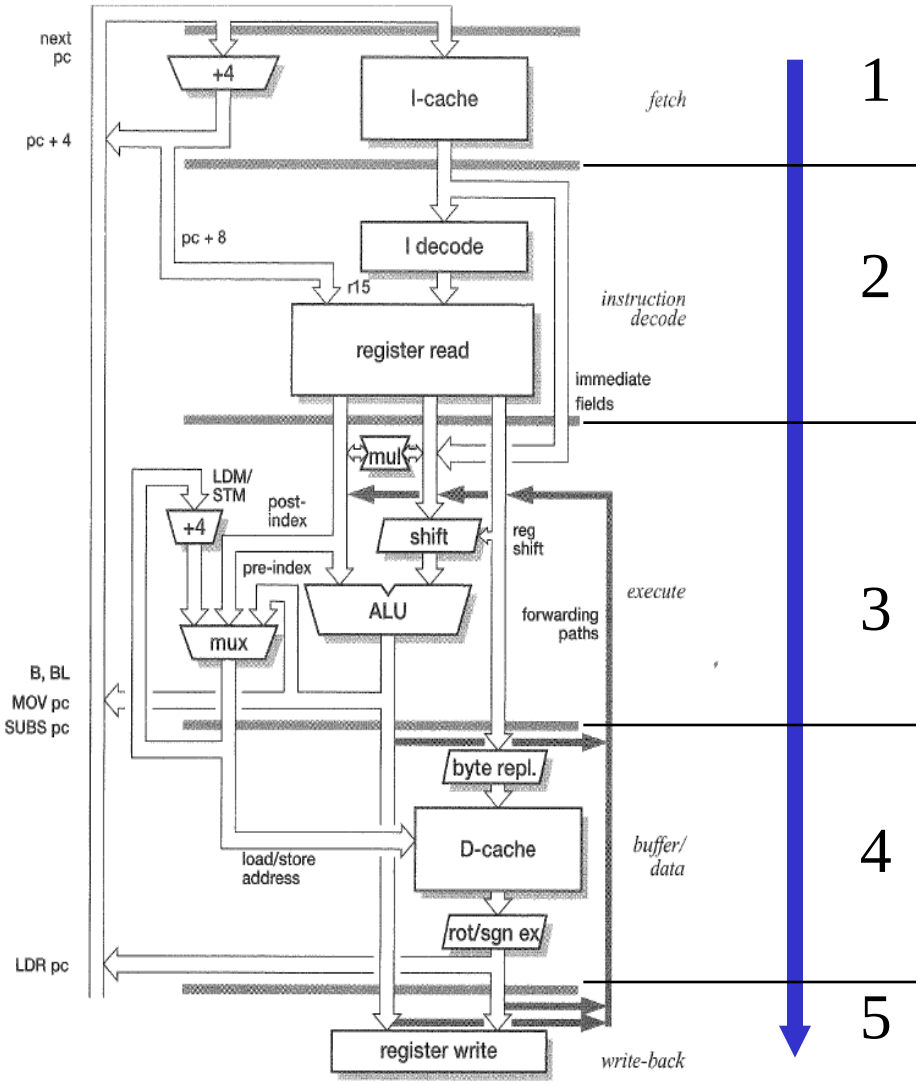
Pipeline du ARM7TDMI



Pipeline du ARM9TDMI



ARM9TDMI architecture

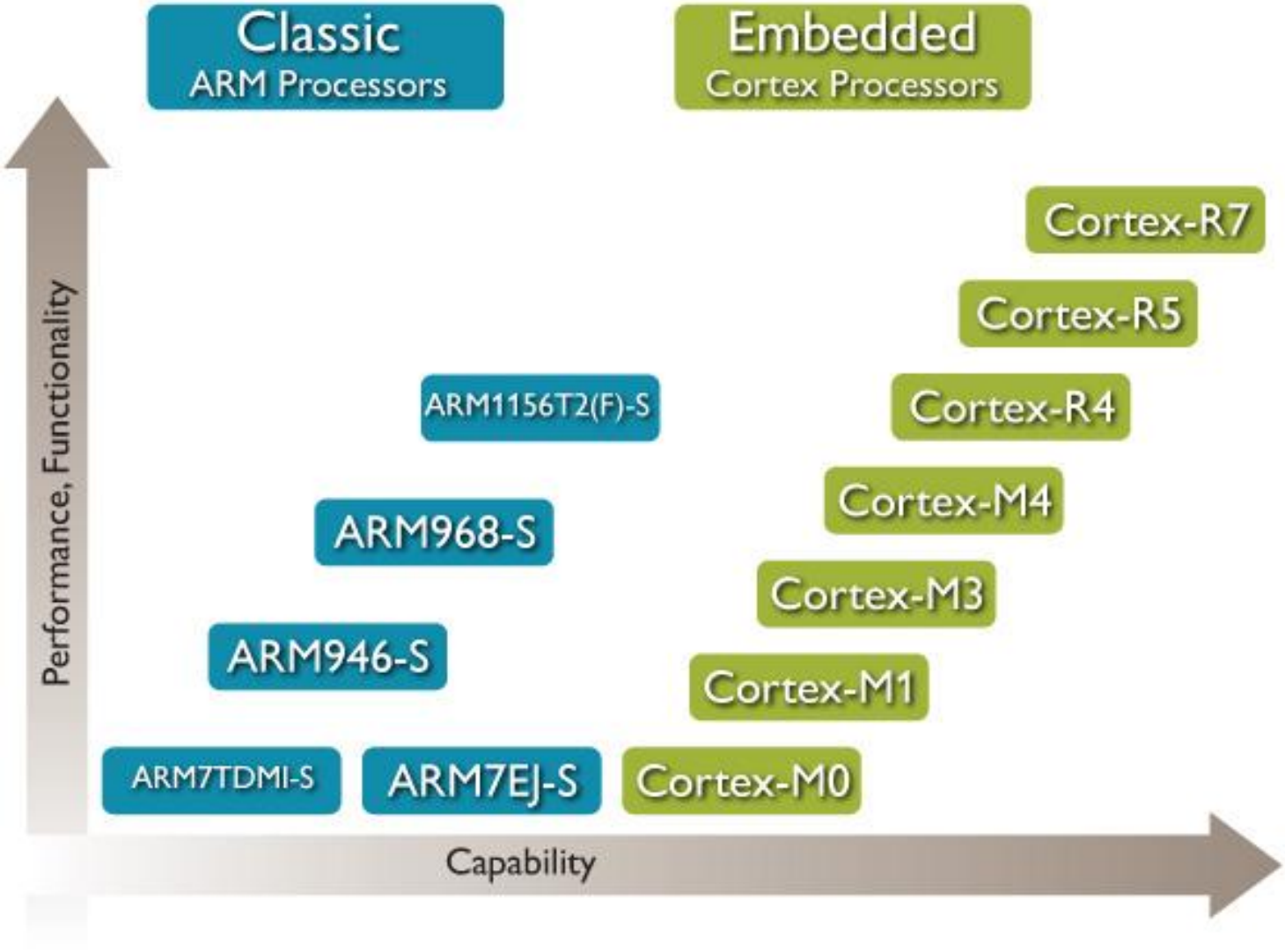


5 étages de pipeline

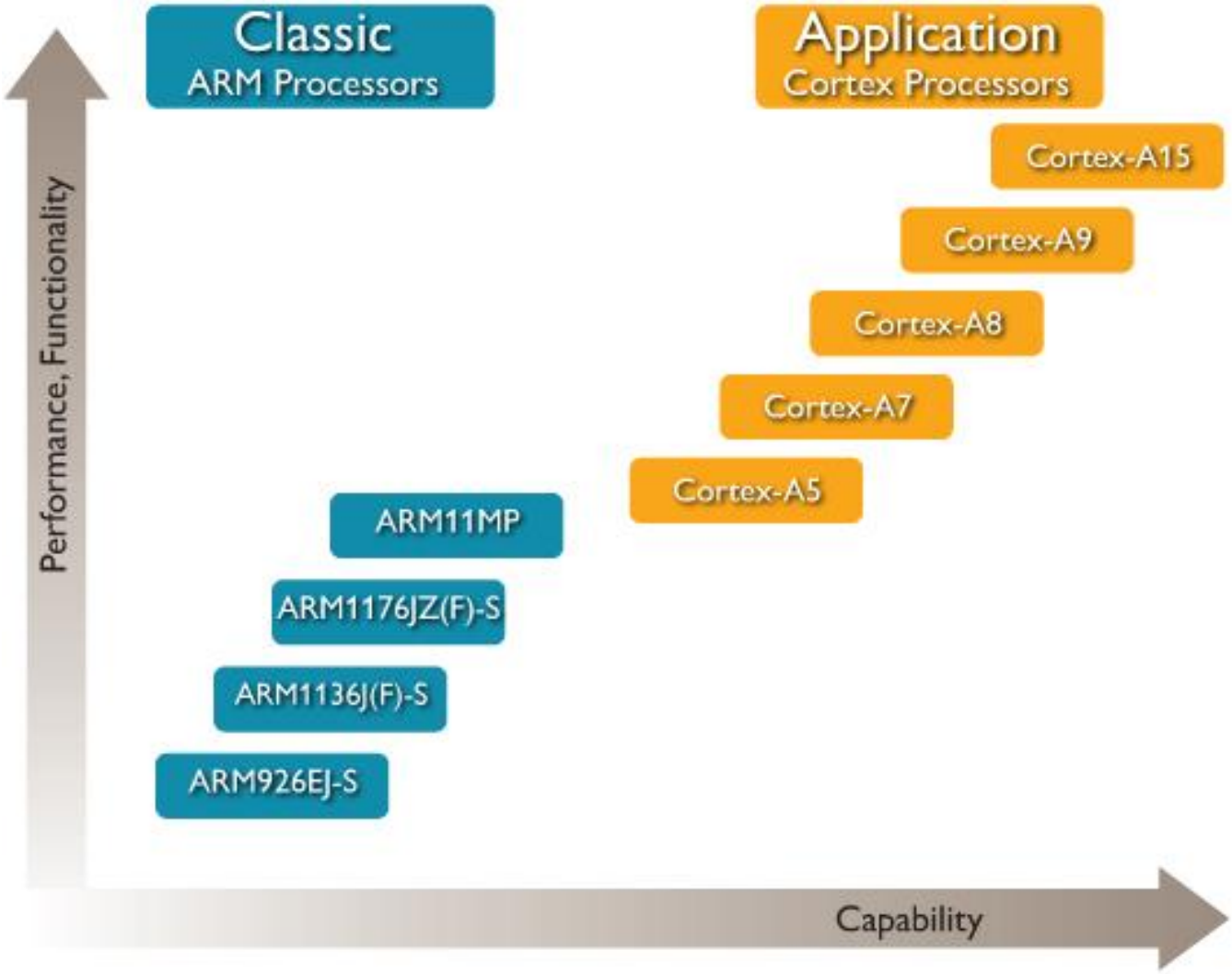
Plan

- **ARM7TDMI**
- **Cortex**
  - Cortex A family
  - Cortex M family
- **System bus**

Embedded Processors



Application Processors



Development of the ARM Architecture

Halfword and signed halfword / byte support

System mode

Thumb instruction set (v4T)

Improved interworking
CLZ
Saturated arithmetic
DSP MAC instructions

Extensions:
- Jazelle (5TEJ)

SIMD Instructions
Multi-processing
v6 Memory architecture
Unaligned data support

Extensions:
- Thumb-2 (6T2)
- TrustZone® (6Z)
- Multicore (6K)
- Thumb only (6-M)

Thumb-2

Architecture Profiles

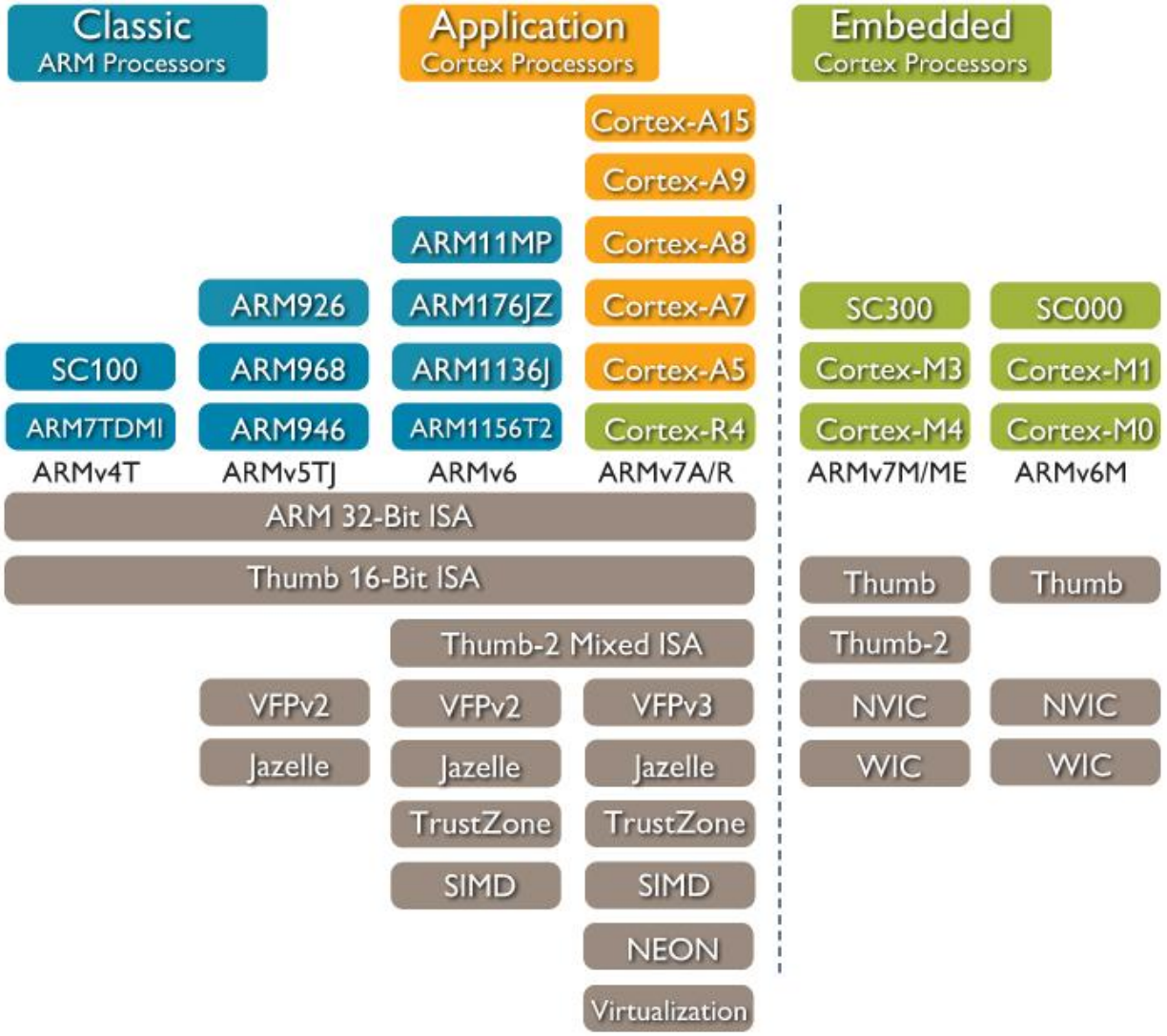
- 7-A - Applications
- 7-R - Real-time
- 7-M - Microcontroller

- **Note that implementations of the same architecture can be different**
  - Cortex-A8 - architecture v7-A, with a 13-stage pipeline
  - Cortex-A9 - architecture v7-A, with an 8-stage pipeline

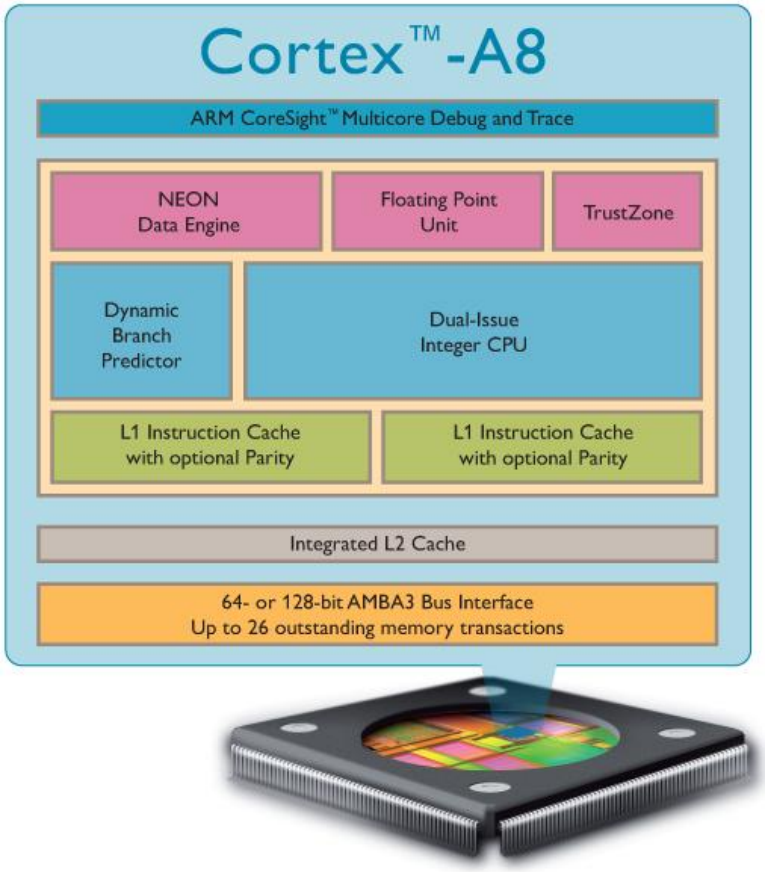
Architecture ARMv7 profiles

- **Application profile (ARMv7-A)**
  - Memory management support (MMU)
  - Highest performance at low power
    - Influenced by multi-tasking OS system requirements
  - TrustZone and Jazelle-RCT for a safe, extensible system
  - e.g. Cortex-A5, Cortex-A9
- **Real-time profile (ARMv7-R)**
  - Protected memory (MPU)
  - Low latency and predictability ‘real-time’ needs
  - Evolutionary path for traditional embedded business
  - e.g. Cortex-R4
- **Microcontroller profile (ARMv7-M, ARMv7E-M, ARMv6-M)**
  - Lowest gate count entry point
  - Deterministic and predictable behavior a key priority
  - Deeply embedded use
  - e.g. Cortex-M3

Which architecture is my processor?



Cortex-A8

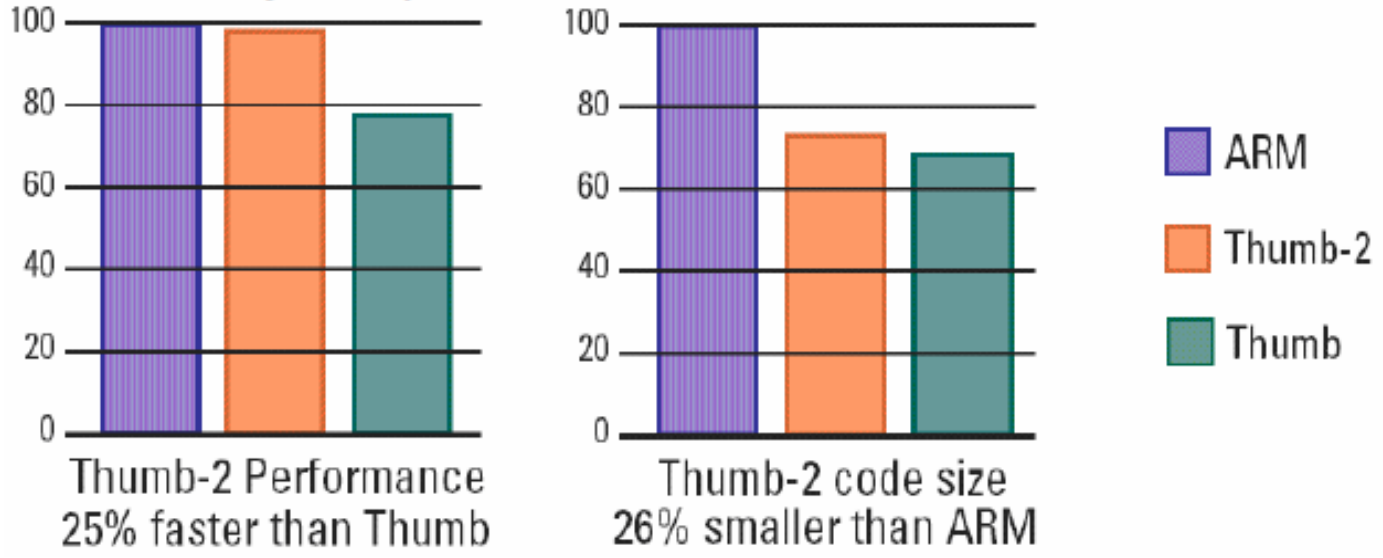


- **ARMv7-A Architecture**
  - Thumb-2
  - Thumb-2EE (Jazelle-RCT)
  - TrustZone extensions
- **Custom or synthesized design**
- **MMU**
- **64-bit or 128-bit AXI Interface**
- **L1 caches**
  - 16 or 32KB each
- **Unified L2 cache**
  - 0-2MB in size
  - 8-way set-associative
- **Optional features**
  - VFPv3 Vector Floating-Point
  - NEON media processing engine

- **Dual-issue, super-scalar 13-stage pipeline**
  - Branch Prediction & Return Stack
  - NEON and VFP implemented at end of pipeline

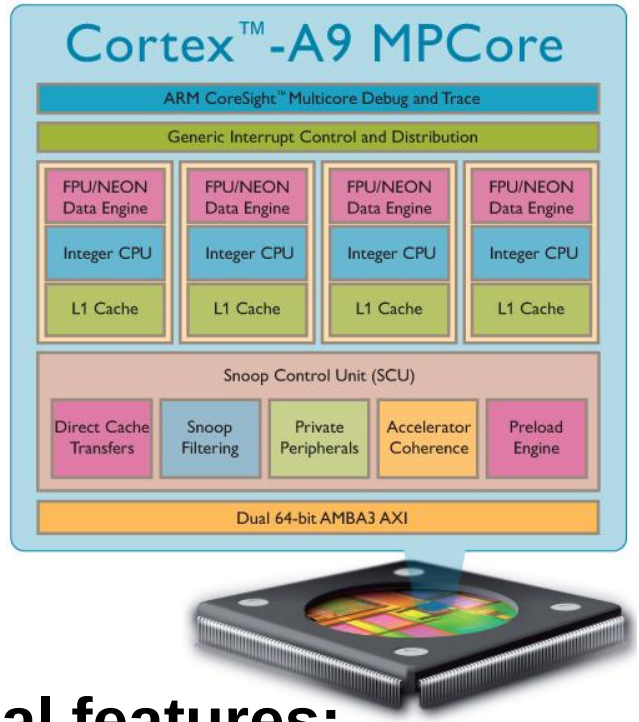
Thumb vs Thumb2

Figure 6. Relative Dhrystone performance and code size for ARM, Thumb and Thumb-2



Cortex-A9

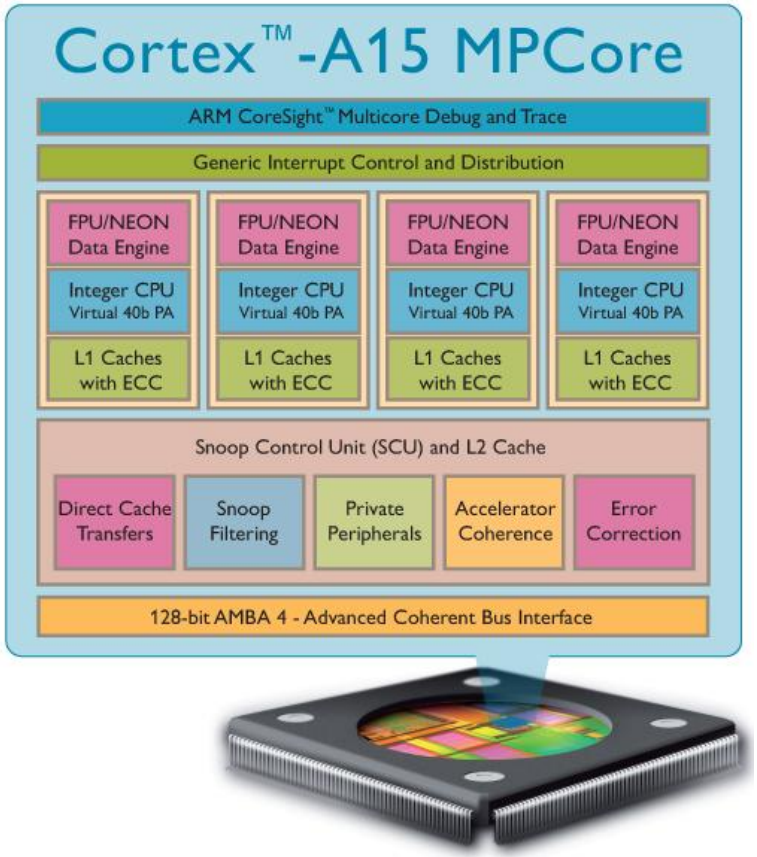
- **ARMv7-A Architecture**
  - Thumb-2, Thumb-2EE
  - TrustZone support
- **Variable-length Multi-issue pipeline**
  - Register renaming
  - Speculative data prefetching
  - Branch Prediction & Return Stack
- **64-bit AXI instruction and data interfaces**
- **TrustZone extensions**
- **L1 Data and Instruction caches**
  - 16-64KB each
  - 4-way set-associative



- **Optional features:**
  - PTM instruction trace interface
  - IEM power saving support
  - Full Jazelle DBX support
  - VFPv3-D16 Floating-Point Unit (FPU) or NEON™ media processing engine

Cortex-A15 MPCore

- 1-4 processors per cluster
- Fixed size L1 caches (32KB)
- Integrated L2 Cache
  - 512KB – 4MB
- System-wide coherency support with AMBA 4 ACE
- Backward-compatible with AXI3 interconnect
- Integrated Interrupt Controller
  - 0-224 external interrupts for entire cluster
- CoreSight debug
- Advanced Power Management

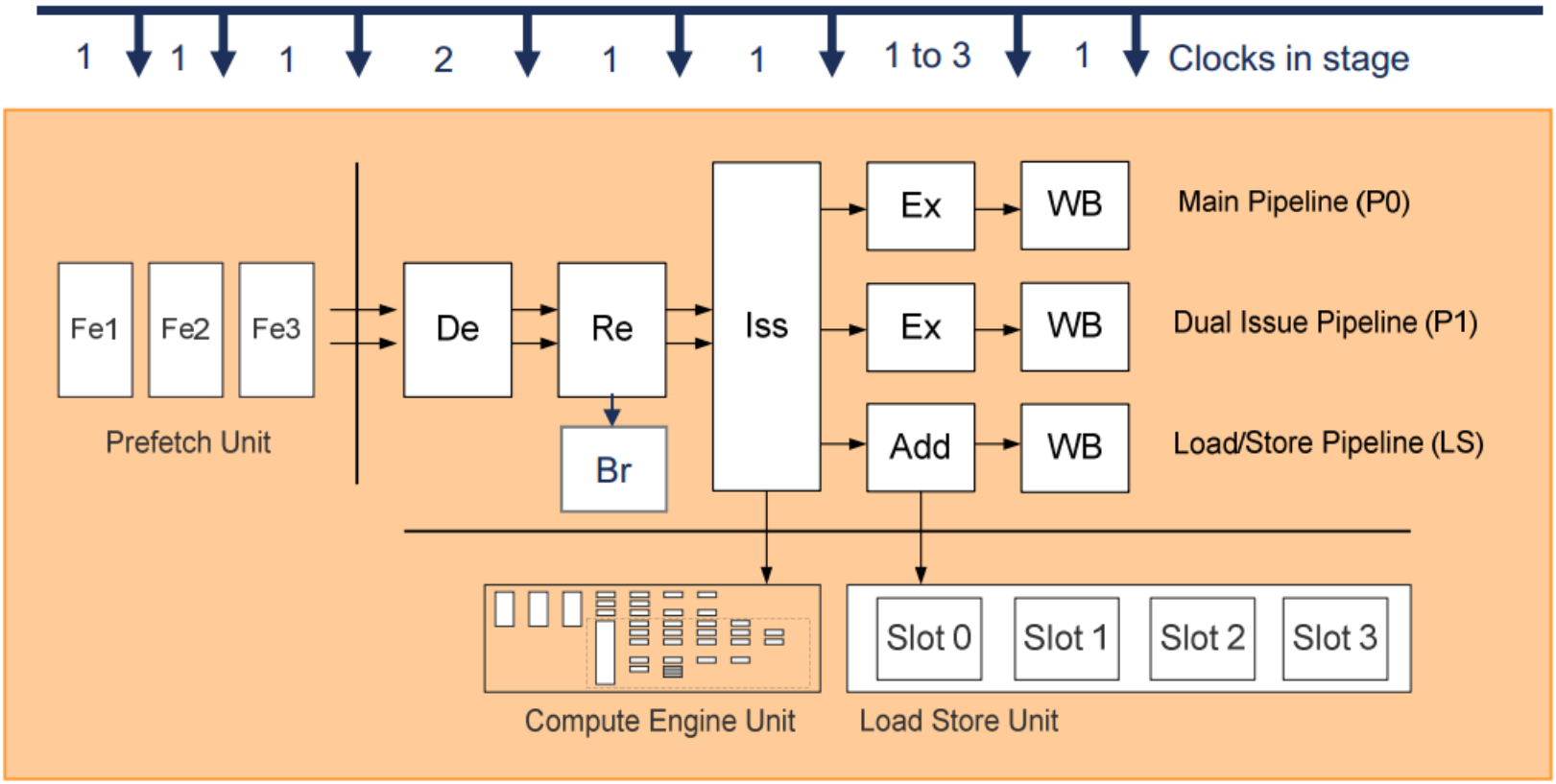


- Large Physical Address Extensions (LPAE) to ARMv7-A Architecture
- Virtualization Extensions to ARMv7-A Architecture

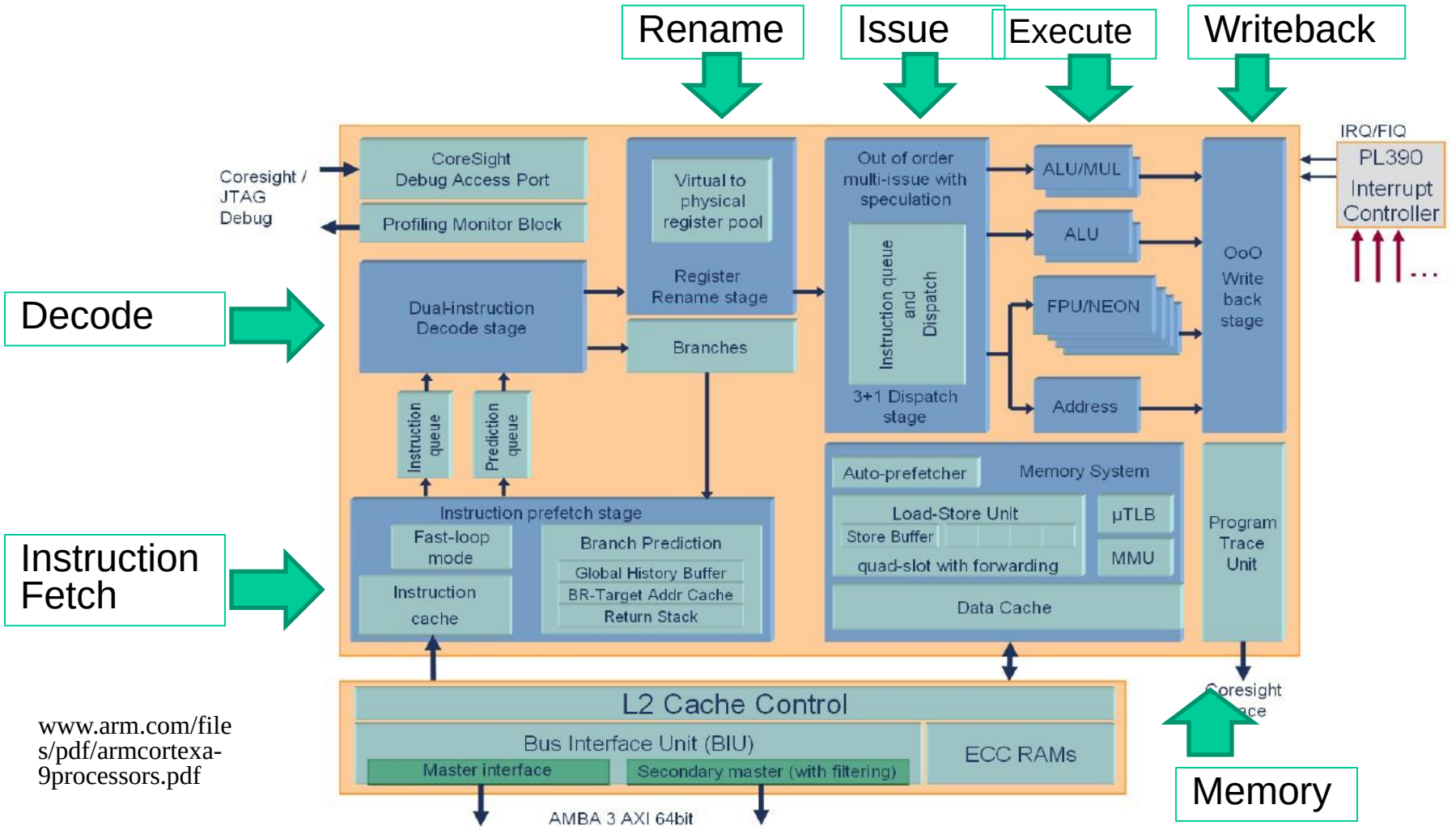
Cortex A9 Microarchitecture Overview

- **Variable length, out of order, superscalar pipeline**
  - Two instructions are fetched in one cycle
  - Issue up to 4 instructions per cycle into:
    - Primary data processing pipeline
    - Secondary data processing pipeline
    - Load-store pipeline
    - Compute engine (FPU/NEON) pipeline
- **Speculative execution**
  - Supporting virtual renaming of physical registers and removing pipelines stalls due to data dependencies

Cortex A9 Pipeline



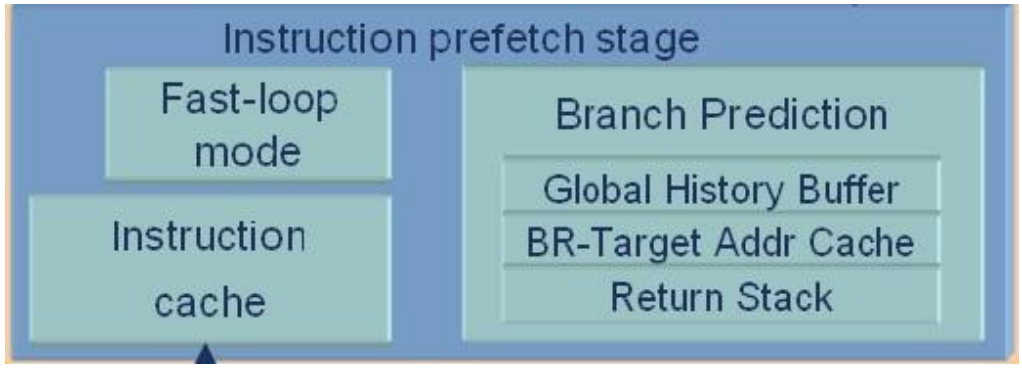
CortexA9 Microarchitecture



www.arm.com/files/pdf/armcortexa-9processors.pdf

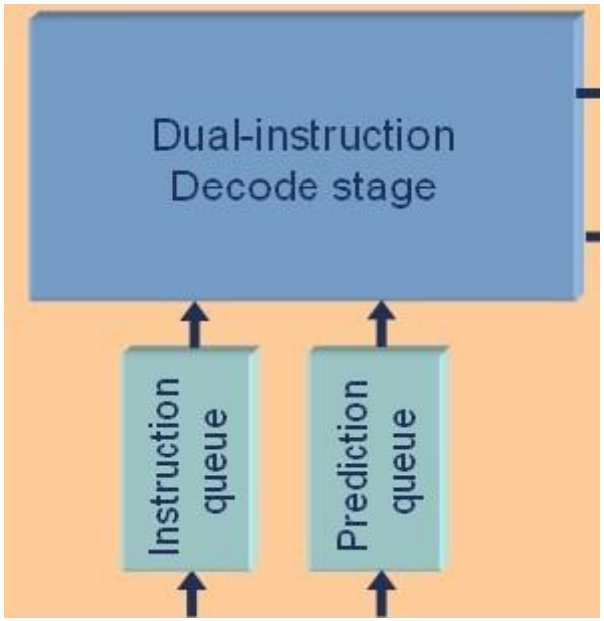
Fig. 1 Cortex-A9 microarchitecture structure and the single core interfaces.

Instruction Fetch



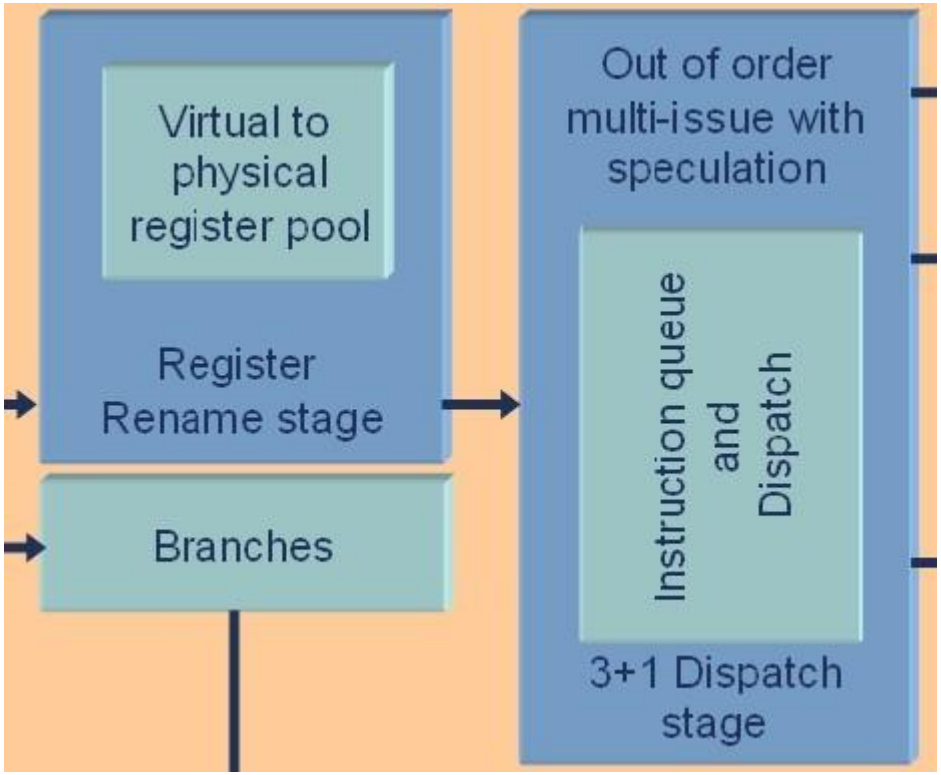
- **Instruction cache size: 16KB, 32KB, or 64KB**
- **Superscalar pipeline: fetching two instructions at once**
- **Branch Prediction:**
  - Global History Buffer: 1K ~ 16K entries
  - Branch-Target Address Cache: 512 ~ 4K entries
  - Return stack of 4 x 32 bits
- **Fast-loop mode: instruction loop that are smaller than 64 bytes often complete without additional instruction cache accesses**

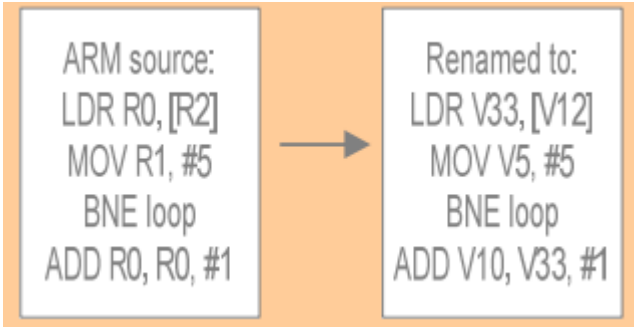
Instruction Decode



- Super Scalar Decoder
  - Capable of decoding two full instructions per cycle

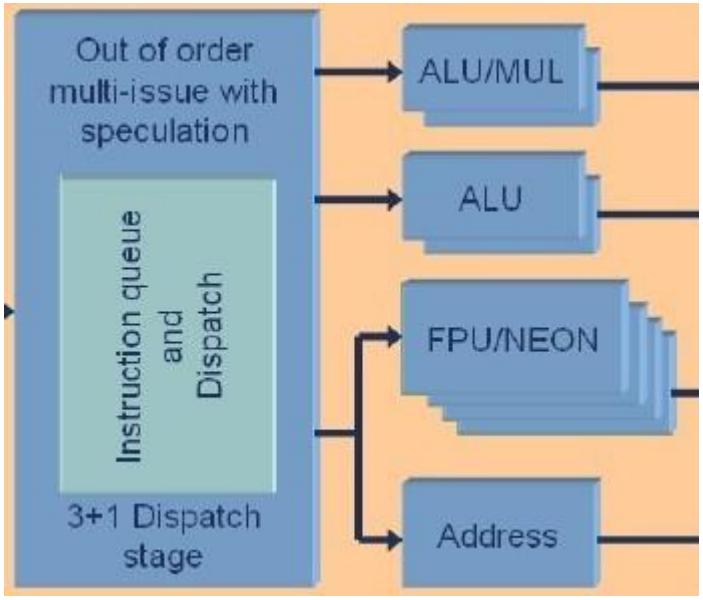
Rename





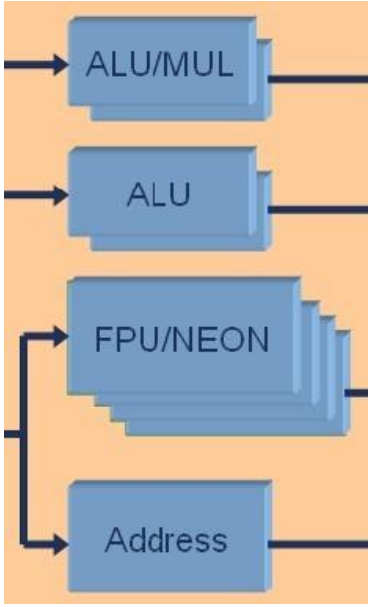
- Register Renaming
  - Resolving data dependencies and unroll small loops by hardware
  - Simplified stalling logic

Issue



- Issue can be fed maximum of 2 instructions per cycle
- Issue can dispatch up to 4 instructions per cycle
- Out of order selection of instructions from queue

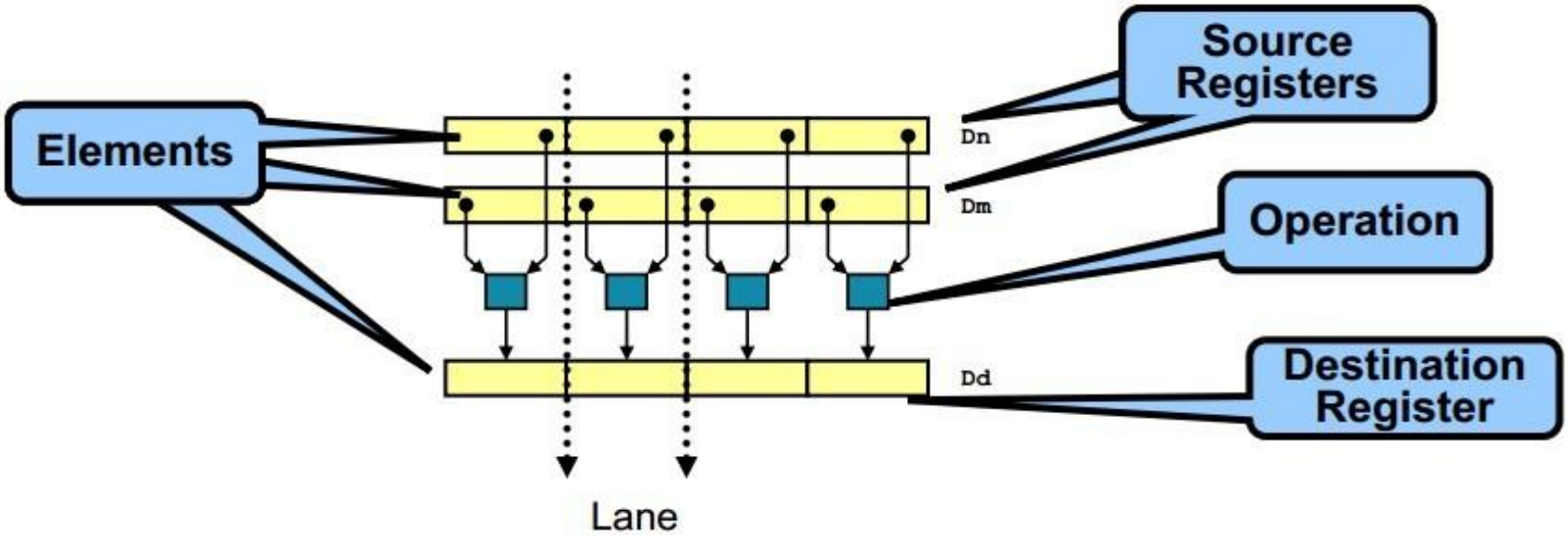
Execute



- Variable length Executing Stage (1 ~ 3 cycles)
  - Most Instructions finish within 1 cycle
  - Instruction which folds shifts and rotates can take 3 cycles
    - ADD r0, r1, r2 (1 cycle)
    - ADD r0, r1, r2 LSL #2 (2 cycle)
      - Corresponds to a = b + (c << 2);
    - ADD r0, r1, r2 LSL r3 (3 cycle)
      - Corresponds to a = b + (c << d);

NEON

- **wide SIMD data processing architecture**
  - 32 registers, 64 bit wide or 16 registers, 128 bit wide
- **NEON instructions perform "Packed SIMD" processing**
  - Registers can be considered as "vector" of same data type
  - Instructions perform the same operation in all lanes



http://www.arm.com/files/pdf/AT_-_NEON_for_Multimedia_Applications.pdf

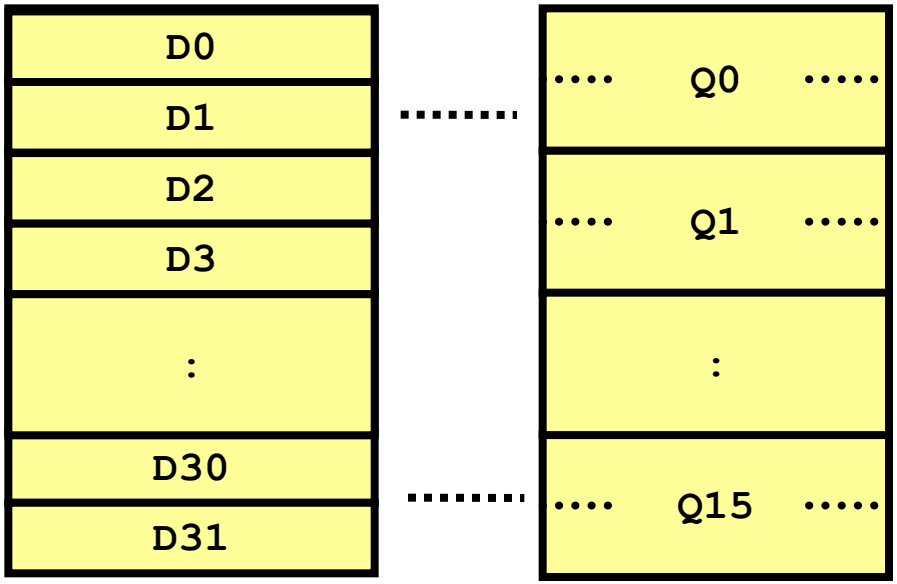
NEON Execute

- **NEON Media Processing Engine supports vector computations on:**
  - half-precision (16bit), single-precision (32bit), double-precision (64bit) floating-point numbers
  - 8, 16, 32 and 64 bit signed and unsigned integers

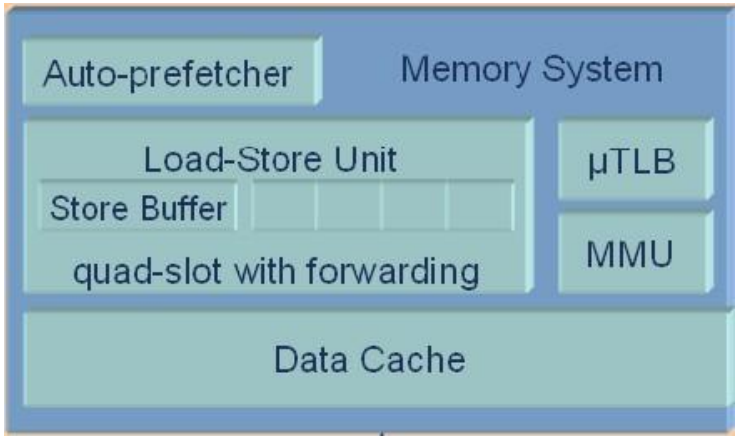
- **Supported Operations Include:**
  - addition, subtraction, multiplication
  - maximum or minimum of a vector of operands
  - Inverse square-root approximation (y = x^-(1/2))
  - many more

NEON Coprocessor registers

- **NEON has a 256-byte register file**
  - Separate from the core registers (r0-r15)
  - Extension to the VFPv2 register file (VFPv3)
- **Two different views of the NEON registers**
  - 32 x 64-bit registers (D0-D31)
  - 16 x 128-bit registers (Q0-Q15)
- **Enables register trade-offs**
  - Vector length can be variable
  - Different registers available



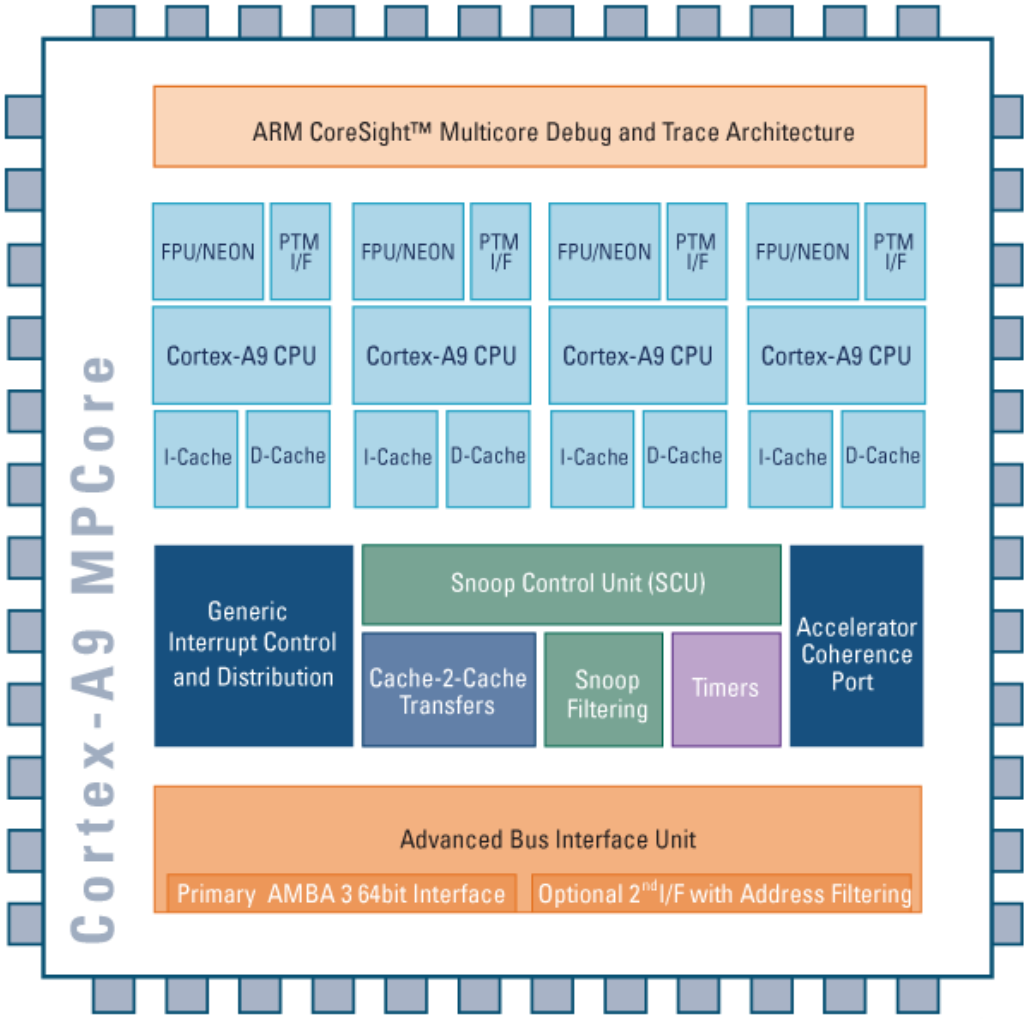
Memory



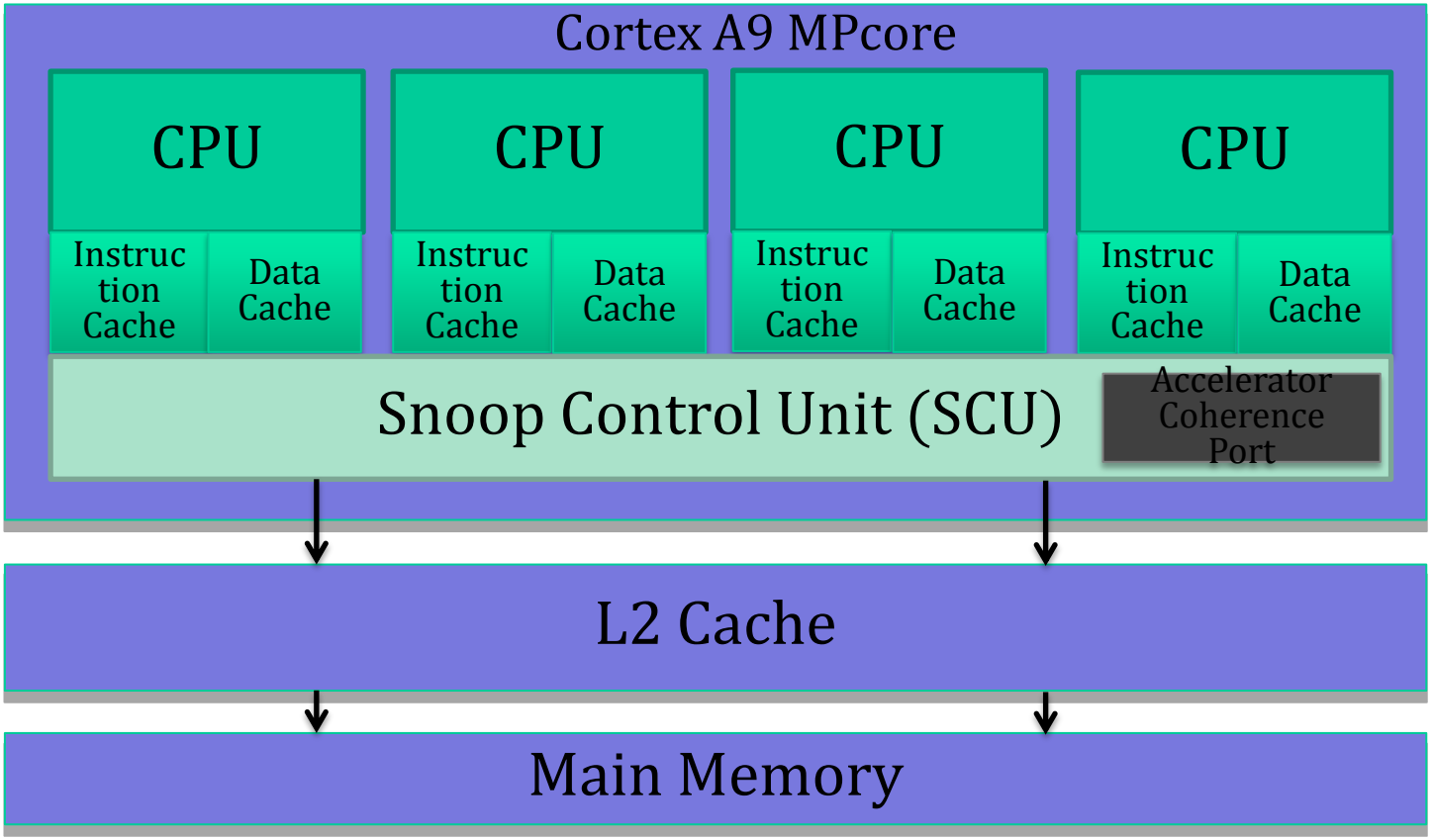
- **Data prefetcher**
  - monitor cache line requests by processor and cache misses to determine how much data to prefetch
  - can prefetch up to 8 independent data streams
- **load-store instructions forwarded for resolution within memory system**
- **2-level TLB structure**
  - micro TLB to reduce power consumed in translation and protection look-ups
  - main TLB

Cortex MPCore Processors

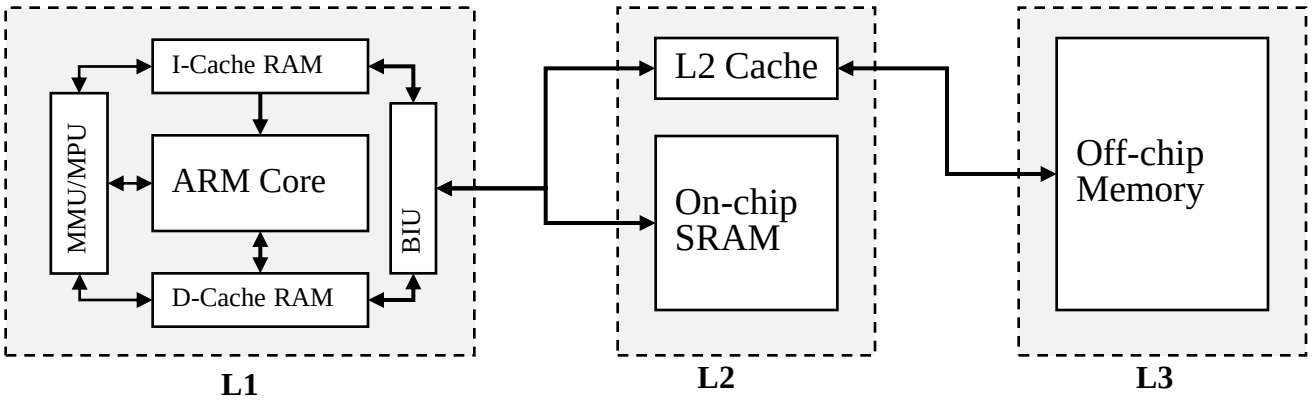
- **Standard Cortex cores, with additional logic to support MPCore**
  - Available as 1-4 CPU variants
- **Include integrated**
  - Interrupt controller
  - Snoop Control Unit (SCU)
  - Timers and Watchdogs



Memory Hierarchy

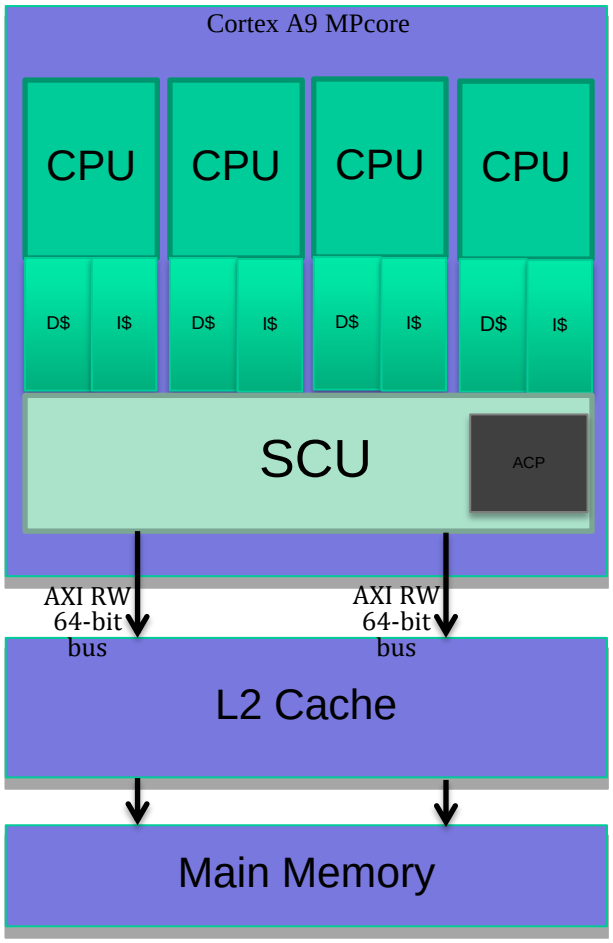


L1 and L2 Caches



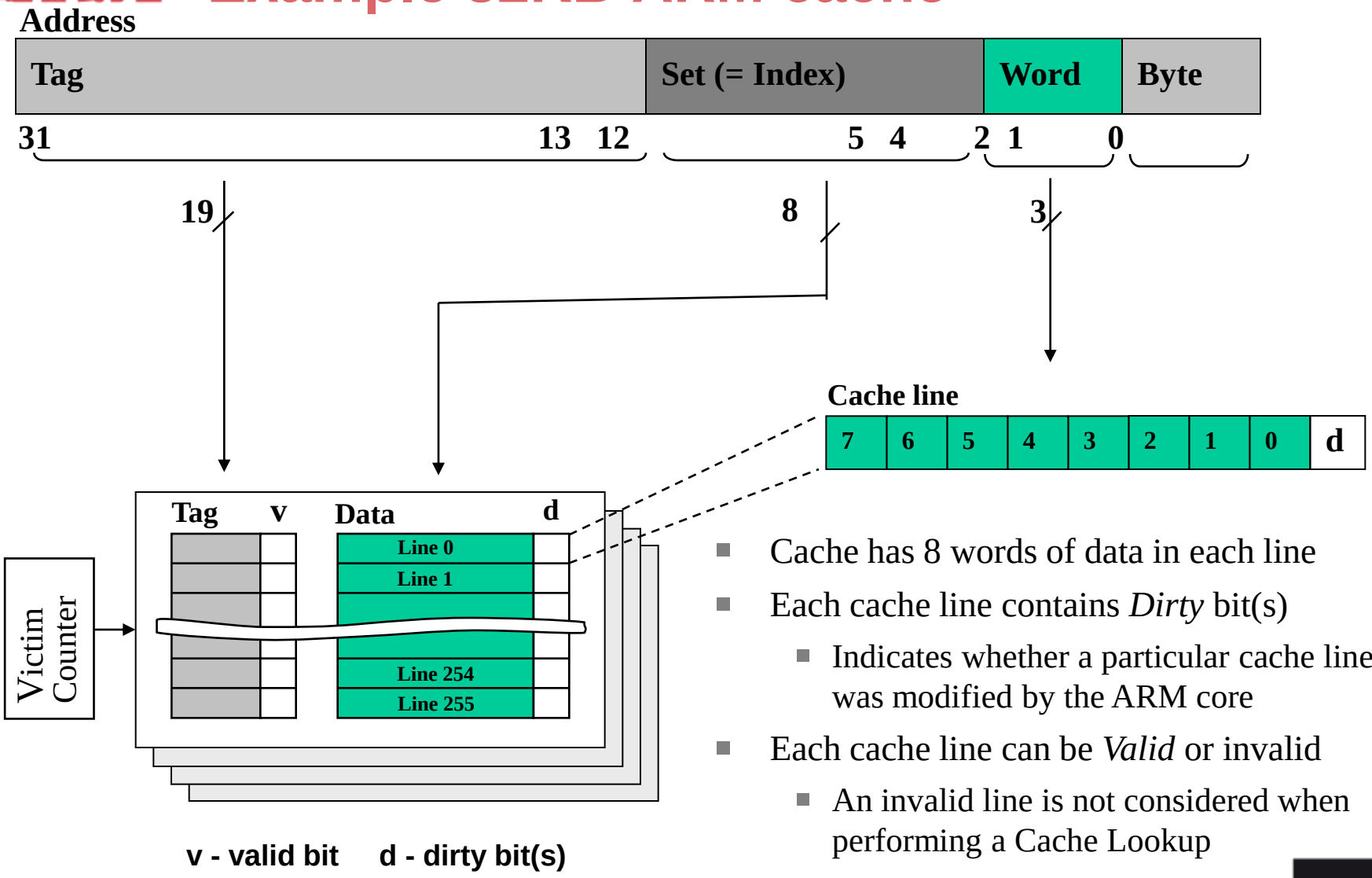
- **Typical memory system can have multiple levels of cache**
  - Level 1 memory system typically consists of L1-caches, MMU/MPU and TCMs
  - Level 2 memory system depends on the system design
- **Memory attributes determine cache behavior at different levels**
  - Controlled by the MMU/MPU
  - Inner Cacheable attributes define memory access behavior in the L1 memory system
  - Outer Cacheable attributes define memory access behavior in the L2 memory system (if external) and beyond (as signals on the bus)
- **Before caches can be used, software setup must be performed**

L1 caches



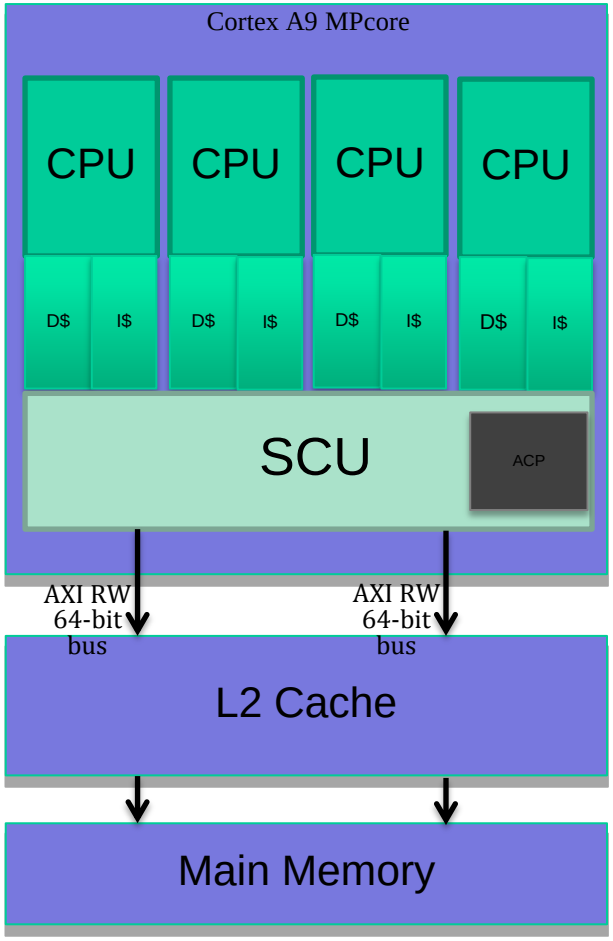
- Non-unified
  - 32 bytes line length
  - can be disabled independently
- 16, 32 or 64KB
- 4 - way associative
- support for Security Extensions
- I cache: VIP(virtual address)
- D cache: PIPT (physical address)
  - reduce number of caches flushes and refills and save energy

Example 32KB ARM cache



- Cache has 8 words of data in each line
- Each cache line contains *Dirty* bit(s)
  - Indicates whether a particular cache line was modified by the ARM core
- Each cache line can be *Valid* or invalid
  - An invalid line is not considered when performing a Cache Lookup

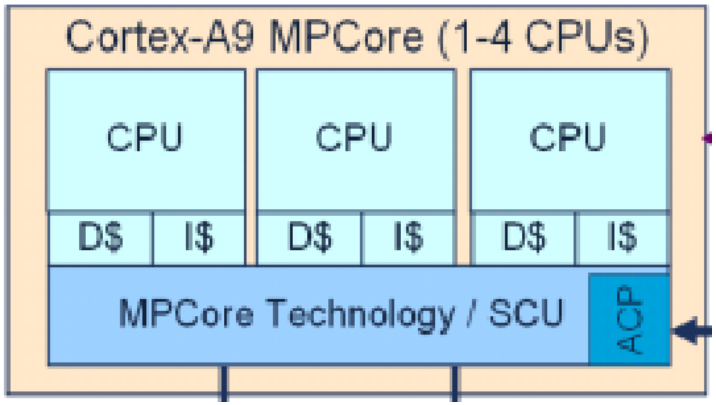
L2 cache



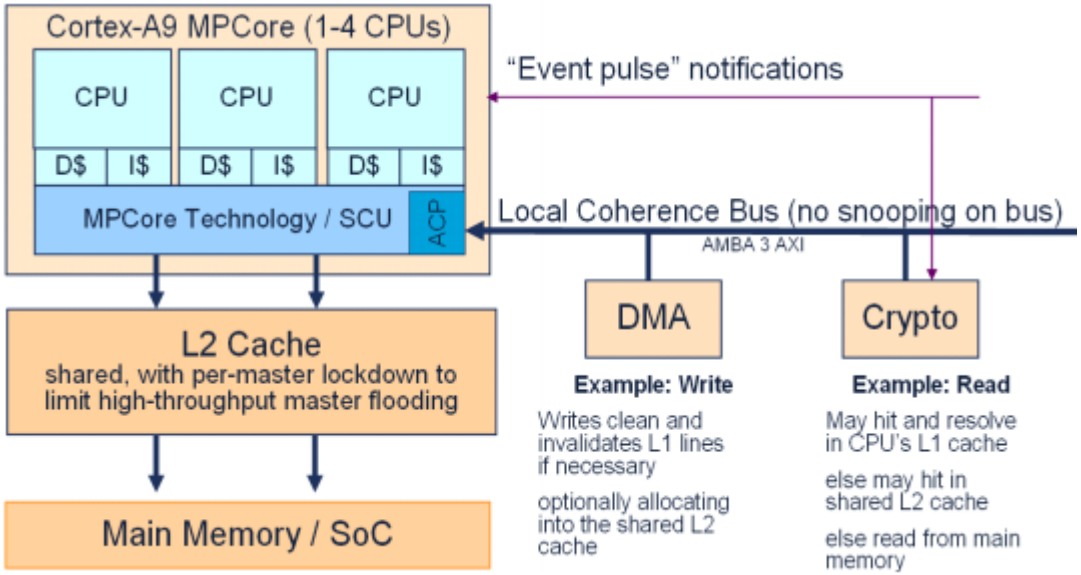
- shared, unified
- Off-chip
- 128KB to 8MB
- 4 to 16-way associative

Snoop Control Unit

- **Integral part of cache memory systems**
- **Connects processors to memory system through AXI interfaces**
- **SCU functions :**
  - maintain data cache coherency
  - initiate L2 memory accesses
  - arbitrate between processors' simultaneous request for L2 accesses
  - manages accesses from ACP
- **does not support instruction cache coherency**



Accelerator Coherence Port



- optional AXI 64-bit slave port
- allows to connect to non-cached system mastering peripherals and accelerators
  - —DMA engine or cryptographic accelerator
- SCU enforces memory coherency

Data Sizes and Instruction Sets

- **ARM is a 32-bit load / store RISC architecture**
  - The only memory accesses allowed are loads and stores
  - Most internal registers are 32 bits wide
  - Most instructions execute in a single cycle

- **When used in relation to ARM cores**
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
  - **Doubleword** means 64 bits (eight bytes)

- **ARM cores implement two basic instruction sets**
  - **ARM** instruction set – instructions are all 32 bits long
  - **Thumb** instruction set – instructions are a mix of 16 and 32 bits
    - Thumb-2 technology added many extra 32- and 16-bit instructions to the original 16-bit Thumb instruction set

- **Depending on the core, may also implement other instruction sets**
  - **VFP** instruction set – 32 bit (vector) floating point instructions
  - **NEON** instruction set – 32 bit SIMD instructions
  - **Jazelle-DBX** - provides acceleration for Java VMs (with additional software support)
  - **Jazelle-RCT** - provides support for interpreted languages

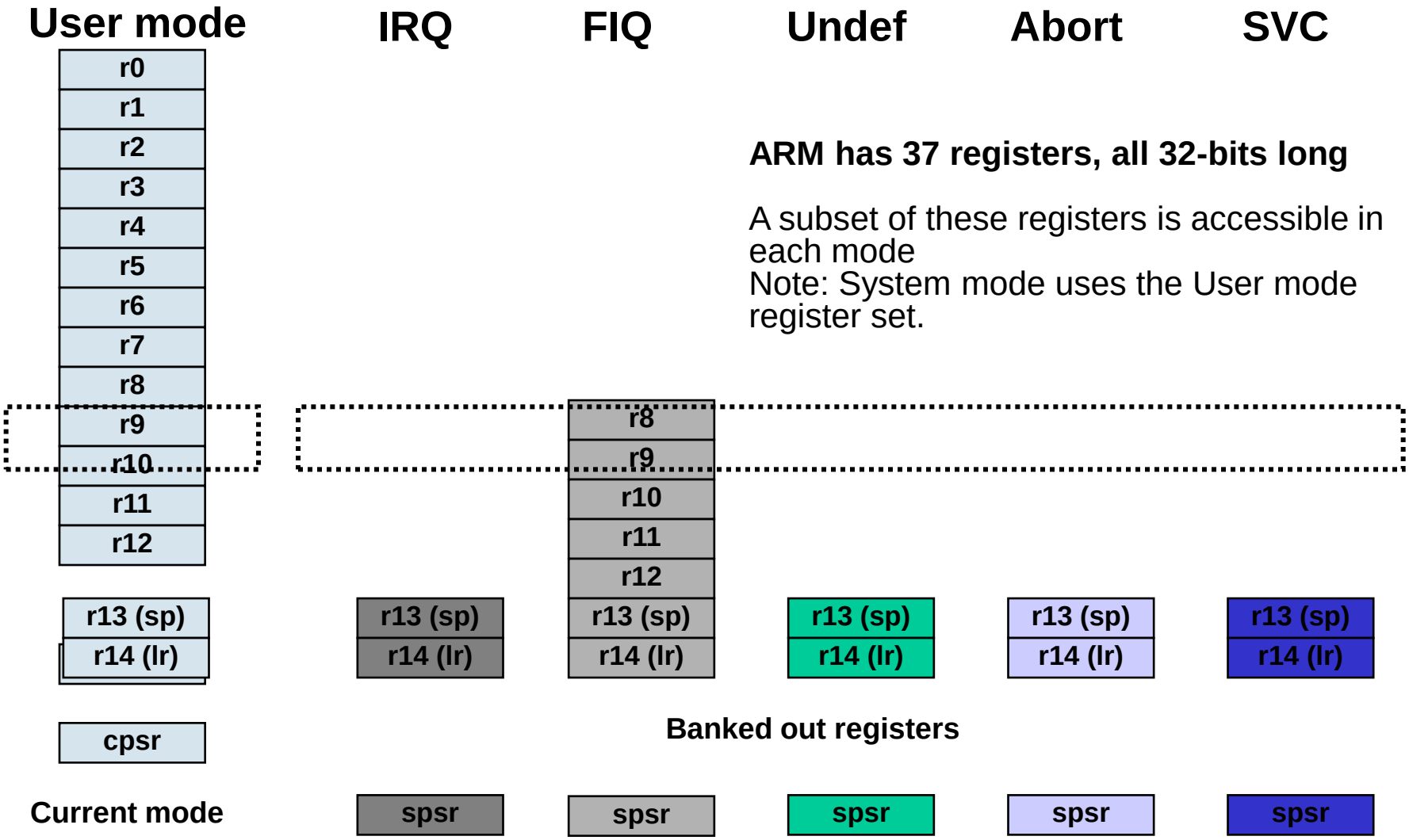
Processor Modes

- **ARM has seven basic operating modes**
  - Each mode has access to its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

Exception modes

Mode	Description	
Supervisor (SVC)	Entered on reset and when a Supervisor call instruction (SVC) is executed	**Privileged modes**
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a normal priority interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode	
User	Mode under which most Applications / OS tasks run	**Unprivileged mode**

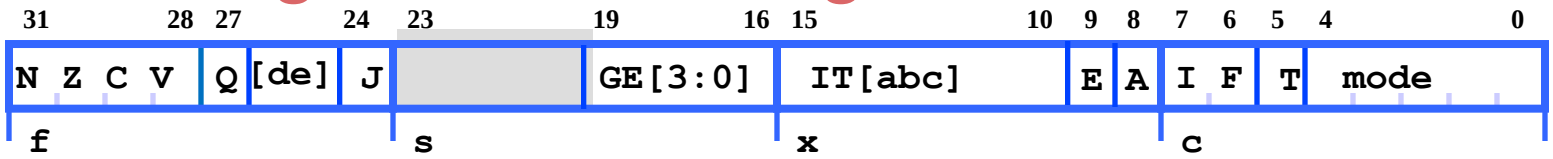
The ARM Register Set



ARM has 37 registers, all 32-bits long

A subset of these registers is accessible in each mode
Note: System mode uses the User mode register set.

Program Status Registers



- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- **Sticky Overflow flag - Q flag**
  - Indicates if saturation has occurred
- **SIMD Condition code bits – GE[3:0]**
  - Used by some SIMD instructions
- **IF THEN status bits – IT[abcde]**
  - Controls conditional execution of Thumb instructions
- **T bit**
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- **J bit**
  - J = 1: Processor in Jazelle state
- **Mode bits**
  - Specify the processor mode
- **Interrupt Disable bits**
  - I = 1: Disables IRQ
  - F = 1: Disables FIQ
- **E bit**
  - E = 0: Data load/store is little endian
  - E = 1: Data load/store is bigendian
- **A bit**
  - A = 1: Disable imprecise data aborts

Instruction Set basics

- **The ARM Architecture is a Load/Store architecture**
  - No direct manipulation of memory contents
  - Memory must be loaded into the CPU to be modified, then written back out
- **Cores are either in ARM state or Thumb state**
  - This determines which instruction set is being executed
  - An instruction must be executed to switch between states
- **The architecture allows programmers and compilation tools to reduce branching through the use of conditional execution**
  - Method differs between ARM and Thumb, but the principle is that most (ARM) or all (Thumb) instructions can be executed conditionally.

Data Processing Instructions

- **These instructions operate on the contents of registers**
  - They DO NOT affect memory

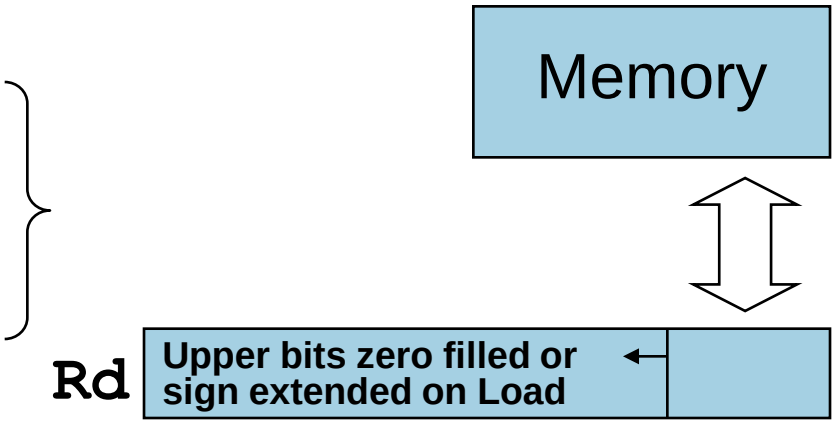
	arithmetic		logical		move
manipulation (has destination register)	ADD, ADC	SUB, SBC, RSB, RSC	AND, BIC	EOR, ORR, ORN	MOV, MVN
comparison (set flags only)	CMN (ADDS)	CMP (SUBS)	TST (ANDS)	TEQ (EORS)	

- **Syntax:**
  - <Operation>{<cond>}{S} {Rd,} Rn, Operand2
- **Examples:**
  - ADD r0, r1, r2 ; r0 = r1 + r2
  - TEQ r0, r1 ; if r0 = r1, Z flag will be set
  - MOV r0, r1 ; copy r1 to r0

Single Access Data Transfer

- **Use to move data between one or two registers and memory**

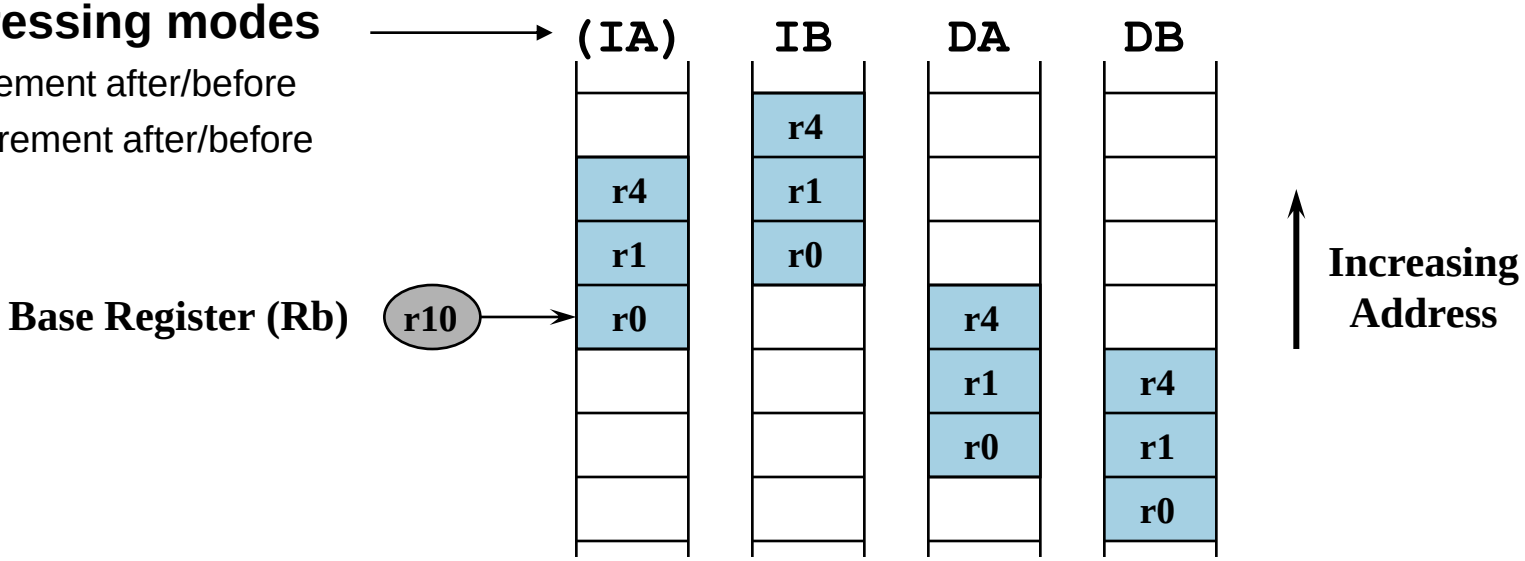
`LDRD STRD`	Doubleword
`LDR STR`	Word
`LDRB STRB`	Byte
`LDRH STRH`	Halfword
`LDRSB`	Signed byte load
`LDRSH`	Signed halfword load



- **Syntax:**
  - `LDR{<size>}{<cond>} Rd, <address>`
  - `STR{<size>}{<cond>} Rd, <address>`
- **Example:**
  -
```
LDRB r0, [r1]        ; load bottom byte of r0 from the
                     ; byte of memory at address in r1
```

Multiple Register Data Transfer

- **These instructions move data between multiple registers and memory**
- **Syntax**
  - **<LDM|STM>**{<addressing_mode>}{<cond>} Rb{!}, <register list>
- **4 addressing modes**
  - Increment after/before
  - Decrement after/before

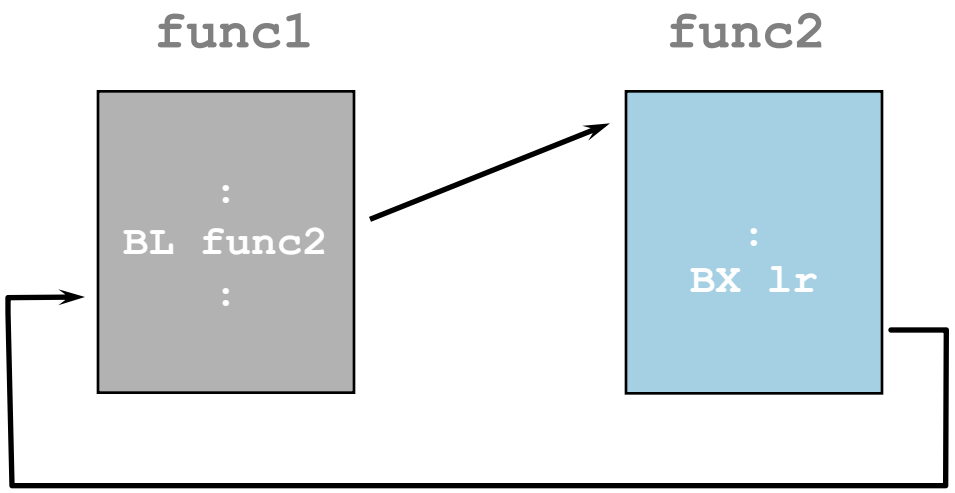


- **Also**
  - **PUSH/POP,** equivalent to **STMDB/LDMIA** with **SP!** as base register
- **Example**
  - **LDM r10, {r0,r1,r4} ; load registers, using r10 base**
  - **PUSH {r4-r6,pc} ; store registers, using SP base**

Subroutines

- **Implementing a conventional subroutine call requires two steps**
  - Store the return address
  - Branch to the address of the required subroutine
- **These steps are carried out in one instruction, `BL`**
  - The return address is stored in the link register (**`lr/r14`**)
  - Branch to an address (range dependent on instruction set and width)
- **Return is by branching to the address in `lr`**

```
void func1 (void)
{
        :
        func2();
        :
}
```



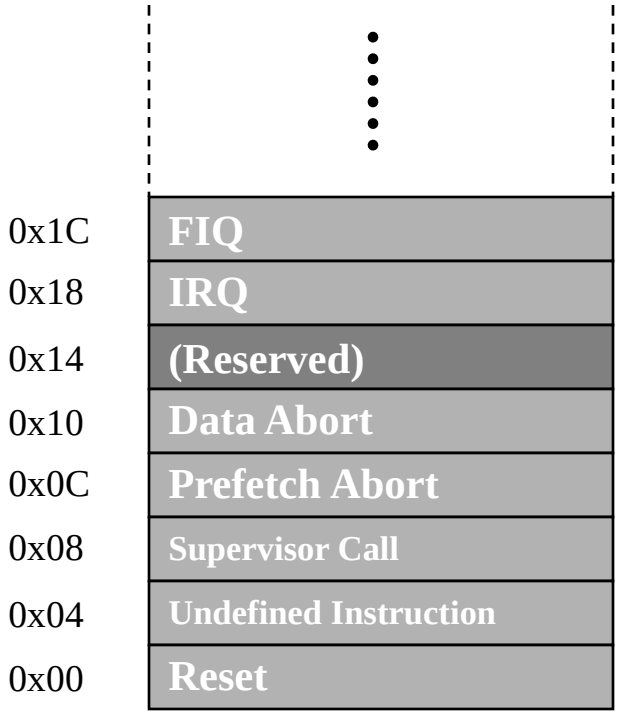
Supervisor Call (SVC)

SVC{<cond>} <SVC number>

- **Causes an SVC exception**
- **The SVC handler can examine the SVC number to decide what operation has been requested**
  - But the core ignores the SVC number
- **By using the SVC mechanism, an operating system can implement a set of privileged operations (system calls) which applications running in user mode can request**
- **Thumb version is unconditional**

Exception Handling

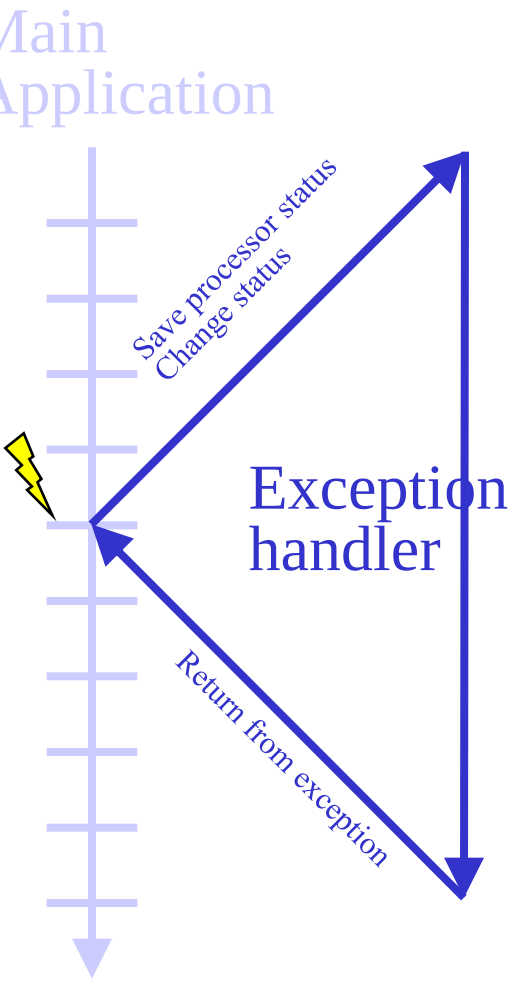
- **When an exception occurs, the core…**
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state (if appropriate)
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- **To return, exception handler needs to…**
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>
- **Cores can enter ARM state or Thumb state when taking an exception**
  - Controlled through settings in CP15
- **Note that v7-M and v6-M exception model is different**



Vector Table

Vector table can also be at **`0xFFFF0000`** on most cores

Exception handling process



1. **Save processor status**
   - Copies `CPSR` into `SPSR_<mode>`
   - Stores the return address in `LR_<mode>`
   - Adjusts LR based on exception type
2. **Change processor status for exception**
   - Mode field bits
   - ARM or Thumb state
   - Interrupt disable bits (if appropriate)
   - Sets PC to vector address
3. **Execute exception handler**
   - <users code>
4. **Return to main application**
   - Restore `CPSR` from `SPSR_<mode>`
   - Restore PC from `LR_<mode>`

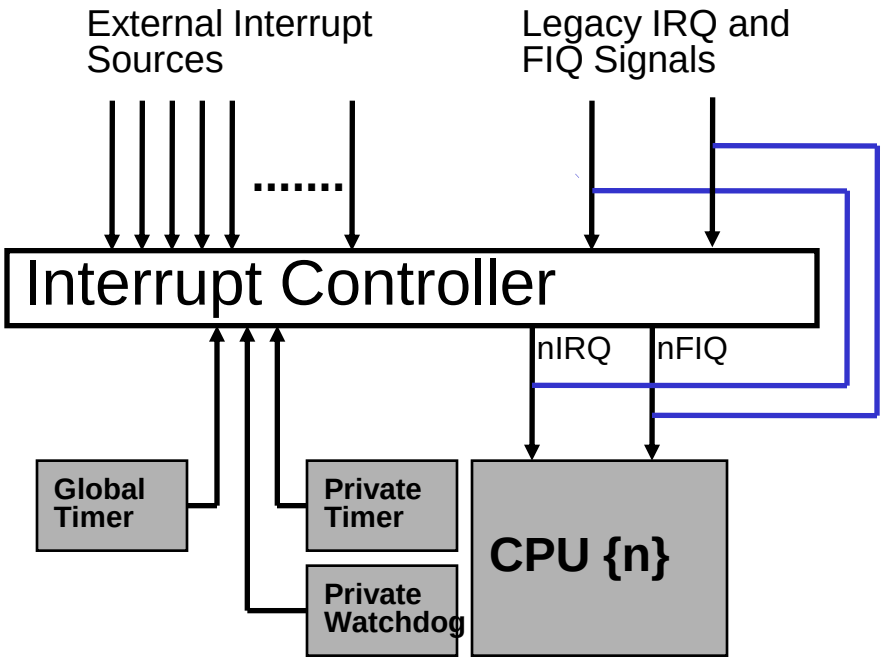
- **1 and 2 performed automatically by the core**
- **3 and 4 responsibility of software**

Memory Types

- **Each defined memory region will specify a memory type**
- **The memory type controls the following:**
  - Memory access ordering rules
  - Caching and buffering behaviour
- **There are 3 mutually exclusive memory types:**
  - Normal
  - Device
  - Strongly Ordered
- **Normal and Device memory allow additional attributes for specifying**
  - The cache policy
  - Whether the region is Shared
  - Normal memory allows you to separately configure Inner and Outer cache policies (discussed in the Caches and TCMs module)

Interrupt Controller

- **MPCore processors include an integrated Interrupt Controller (IC)**
  - Implementation of the Generic Interrupt Controller (GIC) architecture
- **The IC provides:**
  - Configurable number of external interrupts (max 224)
  - Interrupt prioritization and pre-emption
  - Interrupt routing to different cores
- **Enabled per CPU**
  - When not enabled, that CPU will use legacy nIRQ[n] and nFIQ[n] signals



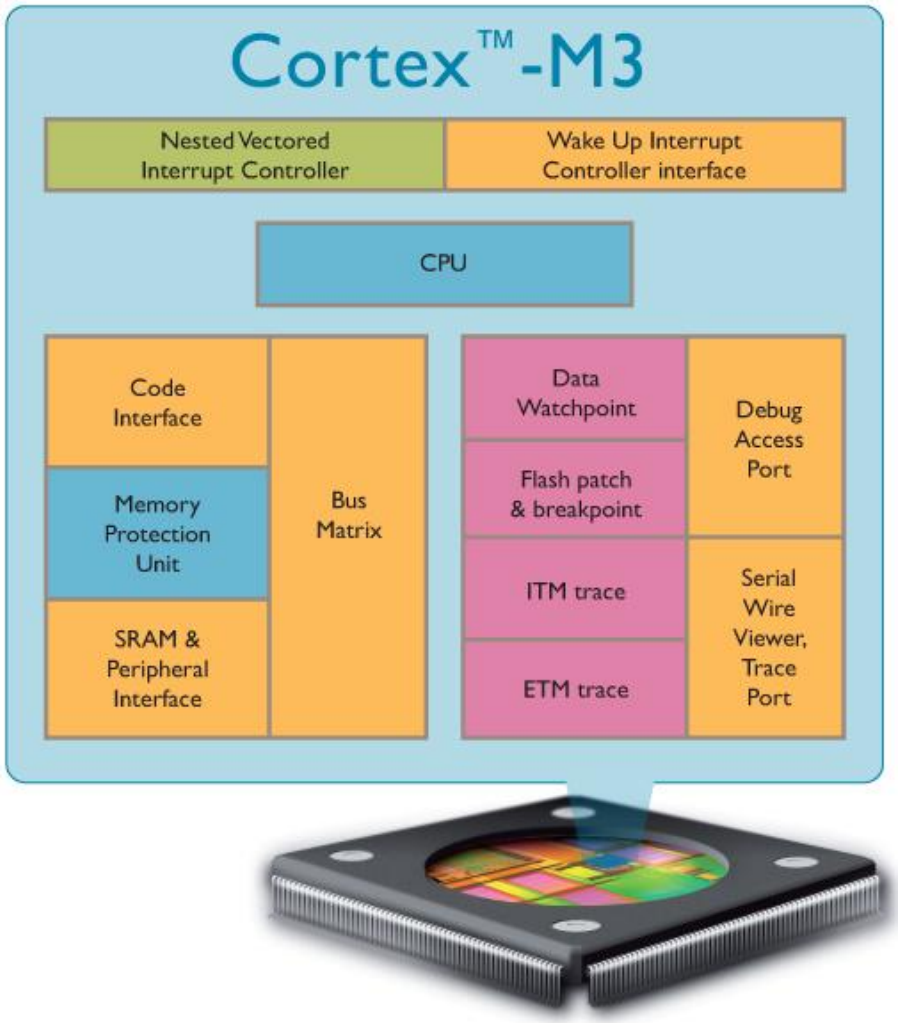
Plan

- **ARM7TDMI**
- **Cortex**
  - Cortex A family
  - Cortex M family
- **System bus**

ARMv7-M Profile Overview

- **v7-M Cores targeted to the microcontroller market**
  - Simpler to program – entire application can be programmed in C
  - Fewer features needed than in application processors
- **Register and ISA changes from other ARM cores**
  - No ARM instruction set support
  - Only one set of registers
  - xPSR has different bits than CPSR
- **Different modes and exception models**
  - Only two modes: Thread mode and Handler mode
  - Vector table is addresses, not instructions
  - Exceptions automatically save state (r0-r3, r12, lr, xPSR, pc) on the stack
- **Different system control/memory layout**
  - Cores have a fixed memory map
  - No coprocessor 15 – controlled through memory mapped control registers

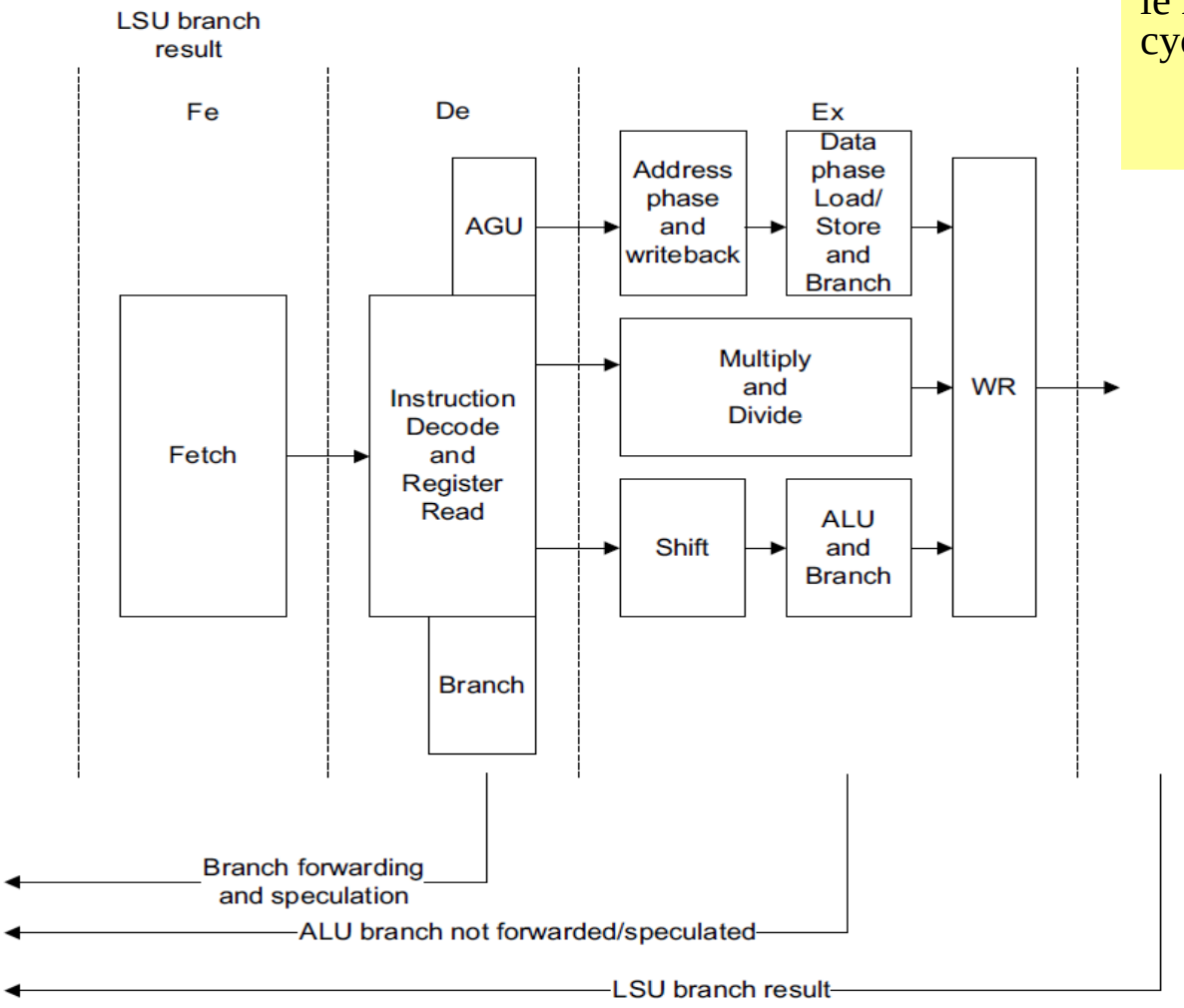
Cortex-M3



- **ARMv7-M Architecture**
  - Thumb-2 only
- **Fully programmable in C**
- **3-stage pipeline**
- **von Neumann architecture**
- **Optional MPU**
- **AHB-Lite bus interface**
- **Fixed memory map**
- **1-240 interrupts**
  - Configurable priority levels
  - Non-Maskable Interrupt support
  - Debug and Sleep control
- **Serial wire or JTAG debug**
- **Optional ETM**

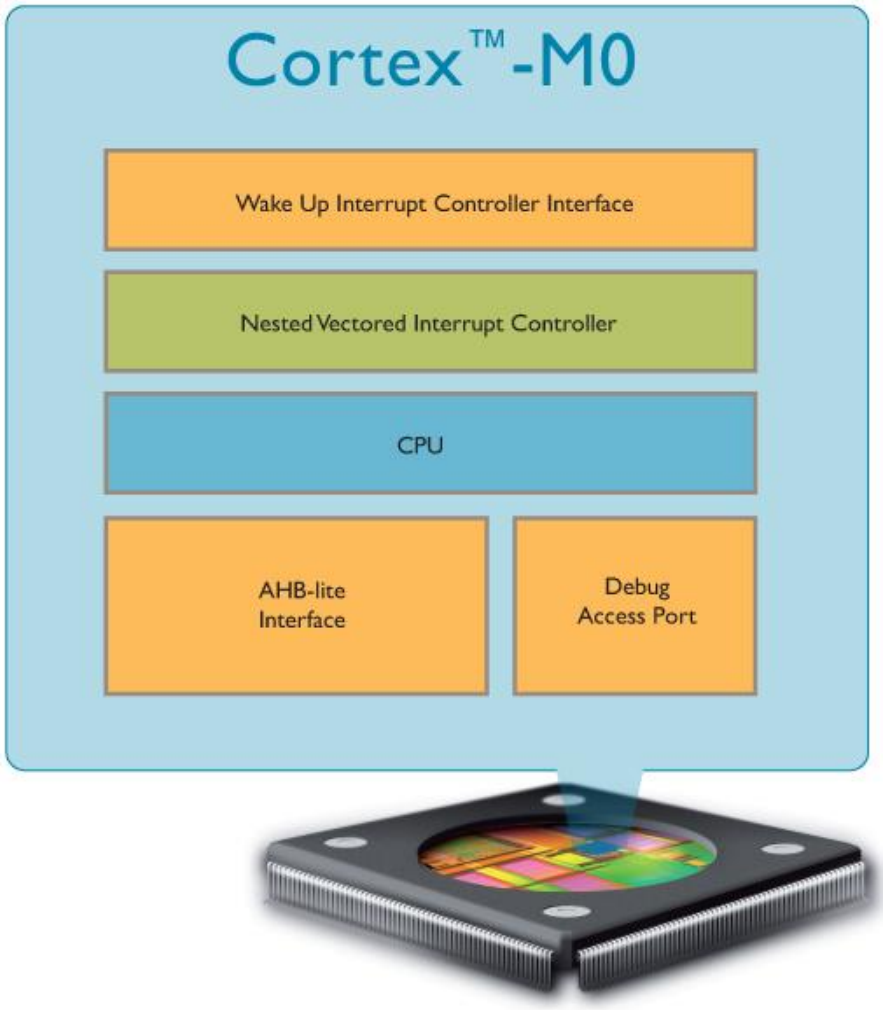
Cortex M3 pipeline

LD/ST dans le même cycle EXE



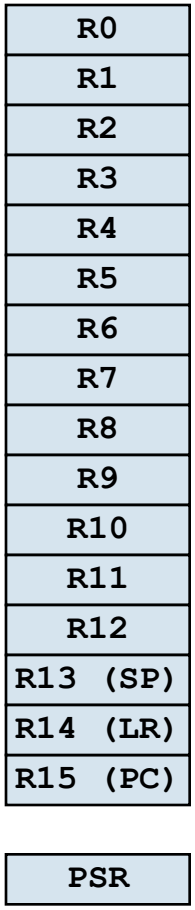
Branchement spéculatif => -1 cycle

Cortex-M0



- **ARMv6-M Architecture**
  - 16-bit Thumb-2 with system control instructions
- **Fully programmable in C**
- **3-stage pipeline**
- **von Neuman architecture**
- **AHB-Lite bus interface**
- **Fixed memory map**
- **1-32 interrupts**
  - Configurable priority levels
  - Non-Maskable Interrupt support
- **Low power support**
- **Core configured with or without debug**
  - Variable number of watchpoints and breakpoints

Processor Register Set

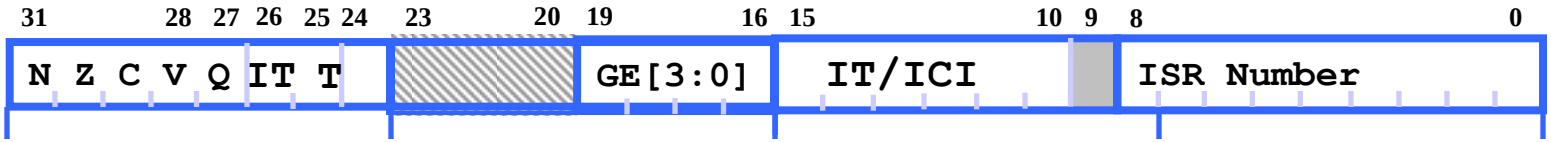


- **Registers R0-R12**
  - General-purpose registers
- **R13 is the stack pointer (SP) - 2 banked versions**
- **R14 is the link register (LR)**
- **R15 is the program counter (PC)**
- **PSR (Program Status Register)**
  - Not explicitly accessible
  - Saved to the stack on an exception
  - Subsets available as APSR, IPSR, and EPSR

Special Purpose Registers

- **Program Status Register**
- **Special Purpose Mask Registers : PRIMASK, FAULTMASK, BASEPRI**
  - Used to modify exception priorities by CPSx instructions
- **Special Purpose CONTROL Register**
  - 2 bits:
    - Bit 0 defines Thread mode privilege
    - Bit 1 defines Thread mode stack
- **The Special Purpose Registers are not memory-mapped**
  - Accessed via specific instructions
  - MRS – Move special purpose register to general-purpose register
  - MSR – Move general-purpose register to special purpose register

xPSR - Program Status Register

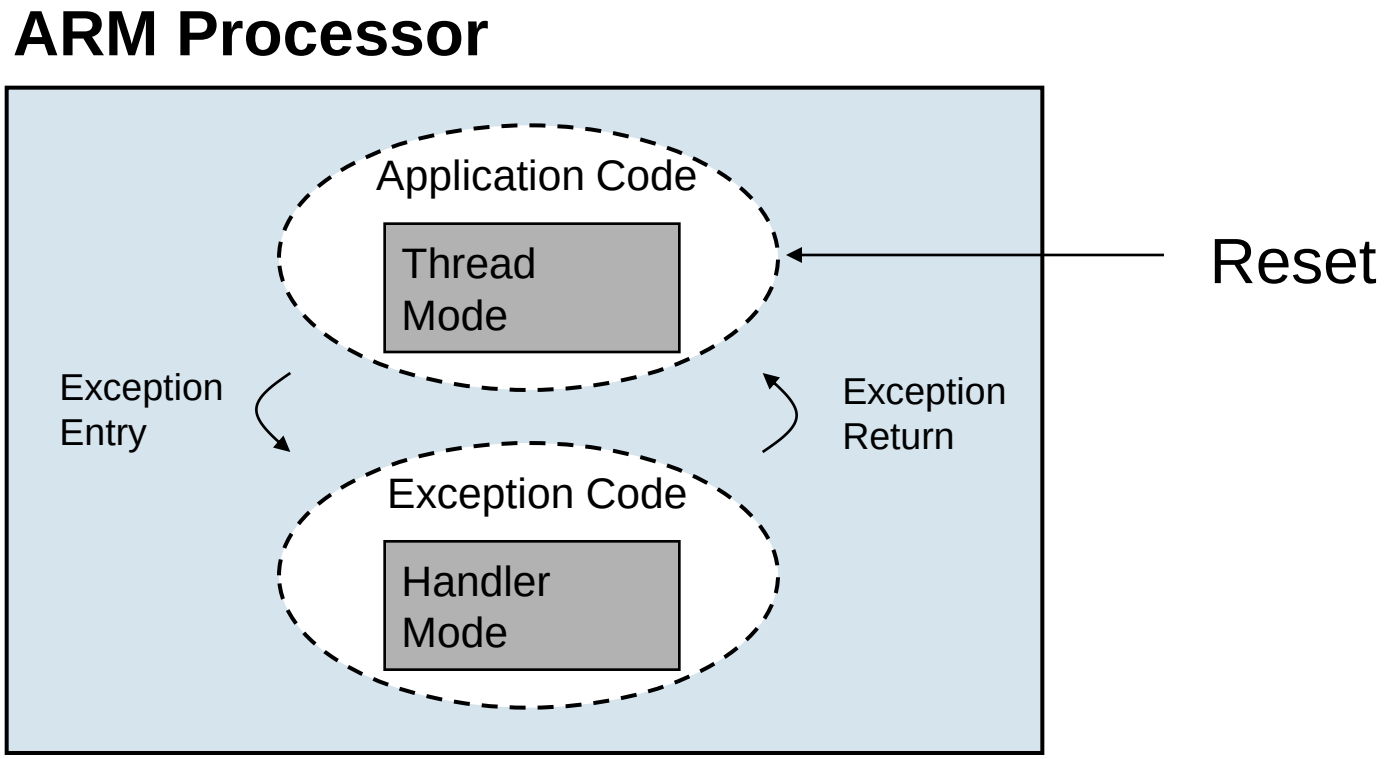


- **xPSR stored on stack during exceptions**
- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation carry out
  - V = ALU operation overflow
  - Q = Saturated math overflow
- **IT/ICI bits**
  - Contain IF-THEN base condition code or Interrupt Continue information
- **ISR Number**
  - Stacked xPSR shows which exception was pre-empted
- **T=1**

System Timer – SysTick

- **Flexible system timer**
  - 24-bit self-reloading down counter
    - Reload on count == 0
    - Optionally cause SysTick interrupt on count == 0
  - Reload register
  - Calibration value
- **Clock source is CPU clock or optional external timing reference**
  - Software selectable if provided
  - Reference pulse widths High/Low must exceed processor clock period
    - Counted by sampling on processor clock
- **Calibration Register provides value required for 10ms interval**
  - STCALIB inputs tied to appropriate value

Modes Overview



Not shown: Handler mode can also be re-entered on exception return

Instruction Set Examples:

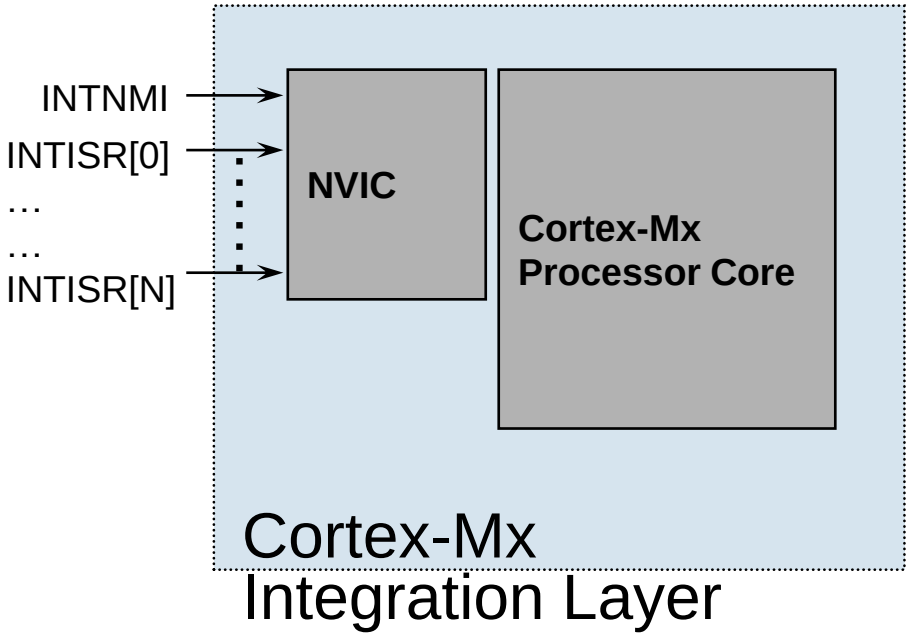
- **Data Processing:**
  - MOV r2, r5 ; r2 = r5
  - ADD r5, #0x24 ; r5 = r5 + 36
  - ADD r2, r3, r4, LSL #2 ; r2 = r3 + (r4 * 4)
  - LSL r2, #3 ; r2 = r2 * 8
  - MOVT r9, #0x1234 ; upper halfword of r9 = #0x1234
  - MLA r0, r1, r2, r3 ; r0 = (r1 * r2) + r3
- **Memory Access:**
  - STRB r2, [r10, r1] ; store lower byte in r2 at {r10 + r1}
  - LDR r0, [r1, r2, LSL #2] ; load r0 with data at address {r1 + r2 * 4}
- **Program Flow:**
  - BL <label> ; PC relative branch to <label> location, and return address stored in LR (r14)

Exception Handling

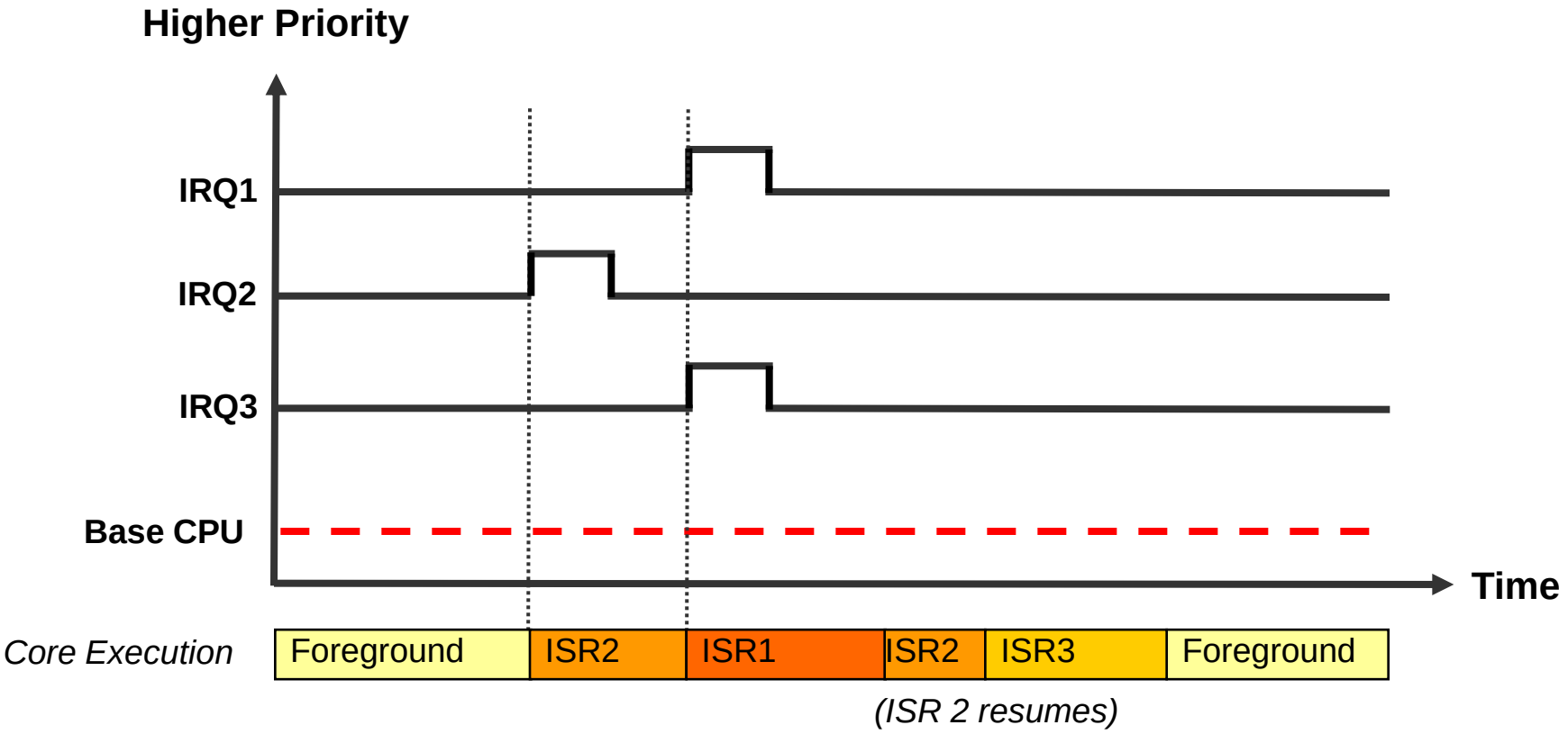
- **Exception types:**
  - Reset
  - Non-maskable Interrupts (NMI)
  - Faults
  - PendSV
  - SVCall
  - External Interrupt
  - SysTick Interrupt
- **Exceptions processed in Handler mode (except Reset)**
  - Exceptions always run privileged
- **Interrupt handling**
  - Interrupts are a sub-class of exception
  - Automatic save and restore of processor registers (xPSR, PC, LR, R12, R3-R0)
  - Allows handler to be written entirely in ‘C’

External Interrupts

- **External Interrupts handled by Nested Vectored Interrupt Controller (NVIC)**
  - Tightly coupled with processor core
- **One Non-Maskable Interrupt (NMI) supported**
- **ARMv7-M supports up to 496 interrupts**



Exception Handling Example

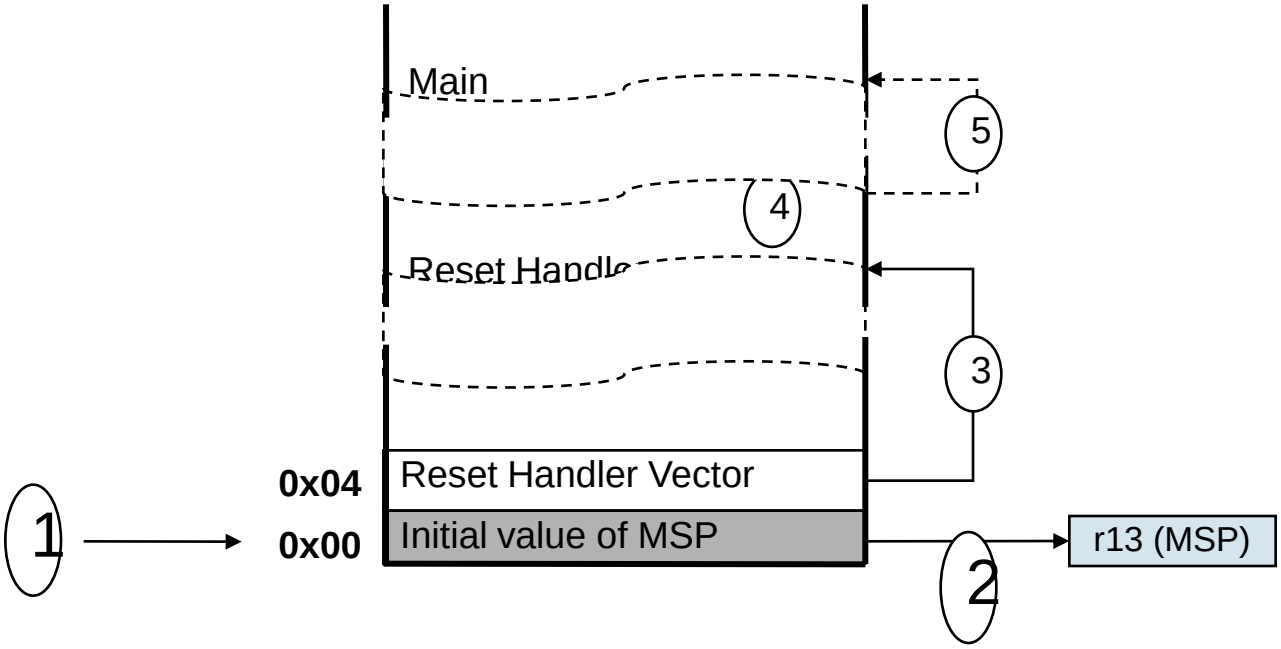


Vector Table for ARMv7-M

- **First entry contains initial Main SP**
- **All other entries are addresses for exception handlers**
  - Must always have LSBit = 1 (for Thumb)
- **Table has up to 496 external interrupts**
  - Implementation-defined
  - Maximum table size is 2048 bytes
- **Table may be relocated**
  - Use Vector Table Offset Register
  - Still require minimal table entries at 0x0 for booting the core
- **Each exception has a vector number**
  - Used in Interrupt Control and State Register to indicate the active or pending exception type
- **Table can be generated using C code**
  - Example provided later

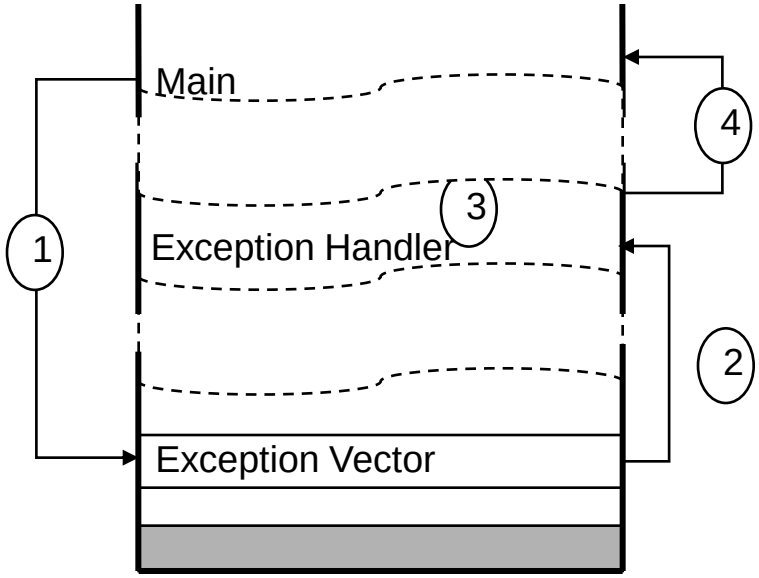
Address		Vector #
0x40 + 4*N	External N	16 + N
…	…	…
0x40	External 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVC	11
0x1C to 0x28	Reserved (x4)	7-10
0x18	Usage Fault	6
0x14	Bus Fault	5
0x10	Mem Manage Fault	4
0x0C	Hard Fault	3
0x08	NMI	2
0x04	Reset	1
0x00	Initial Main SP	N/A

Reset Behavior



1. **A reset occurs (Reset input was asserted)**
2. **Load MSP (Main Stack Pointer) register initial value from address 0x00**
3. **Load reset handler vector address from address 0x04**
4. **Reset handler executes in Thread Mode**
5. **Optional: Reset handler branches to the main program**

Exception Behaviour

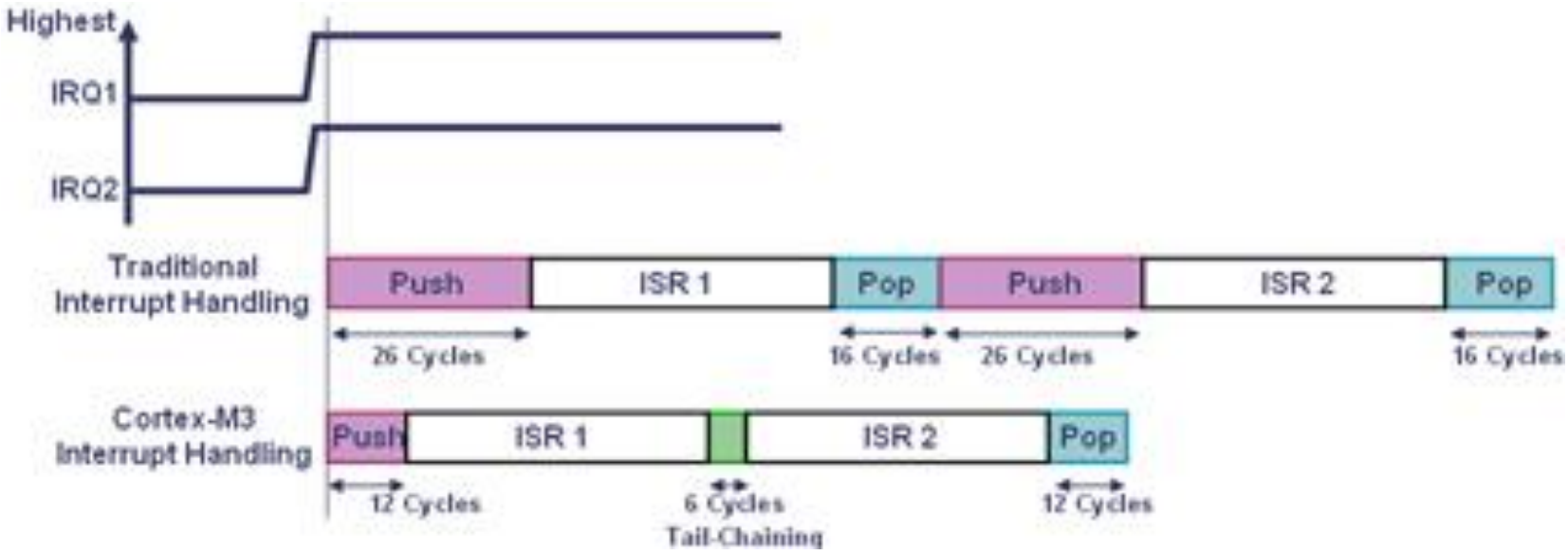


1. **Exception occurs**
   - Current instruction stream stops
   - Processor accesses vector table
2. **Vector address for the exception loaded from the vector table**
3. **Exception handler executes in Handler Mode**
4. **Exception handler returns to main**

Interrupt Service Routine Entry

- **When receiving an interrupt**
  - the processor will finish the current instruction for most instructions
  - To minimize interrupt latency, the processor can take an interrupt during the execution of a multi-cycle instruction - see next slide
- **Processor state automatically saved to the current stack**
  - 8 registers are pushed: PC, R0-R3, R12, LR, xPSR
  - Follows ARM Architecture Procedure Calling Standard (AAPCS)
- **During (or after) state saving the address of the ISR is read from the Vector Table**
- **Link Register is modified for interrupt return**
- **First instruction of ISR executed**
  - For Cortex-M3 or Cortex-M4 the total latency is normally 12 cycles, however, interrupt late-arrival and interrupt tail-chaining can improve IRQ latency
- **ISR executes from Handler mode with Main stack**

Nested Vectored Interrupt Controller



Returning From Interrupt

- **Return from interrupt with the following instructions**
  - The PC is loaded with “magic” value of 0xFFFF_FFFX (same format as EXC_RETURN)
    - LDR PC, 0xFF..FX or LDM/POP or BX LR (most common)
- **If no interrupts are pending, foreground state is restored**
  - Stack and state specified by EXC_RETURN is used
  - Context restore on Cortex-M3 and Cortex-M4 requires 10 cycles
- **If other interrupts are pending, the highest priority may be serviced**
  - Serviced if interrupt priority is higher than the foreground’s base priority
  - Process is called Tail-Chaining as foreground state is not yet restored
  - Latency for servicing new interrupt is only 6 cycles on M3/M4 (state already saved)
- **If state restore is interrupted, it is abandoned**
  - New ISR executed without state saving (original state still intact and valid)
  - Must still fetch new vector and refill pipeline (6-cycle latency on M3/M4

Vector Table in C

```
typedef void(* const ExecFuncPtr)(void) __irq;
#pragma arm section rodata="exceptions_area"
ExecFuncPtr exception_table[] = {
  (ExecFuncPtr)&Image$$ARM_LIB_STACK$$ZI$$Limit,    /* Initial SP */
  ExecFuncPtr)__main,                                /* Initial PC */
  NMIException,
  HardFaultException,
  MemManageException,
  BusFaultException,
  UsageFaultException,
  0, 0, 0, 0,                   /* Reserved */
  SVCHandler,
  DebugMonitor,
  0,                  /* Reserved */
  PendSVC,
  SysTickHandler
  /* Configurable interrupts start here...*/
};
#pragma arm section
```

The vector table at address 0x0 is minimally required to have 4 values: stack top, reset routine location, NMI ISR location, HardFault ISR location

The SVCall ISR location must be populated if the SVC instruction will be used

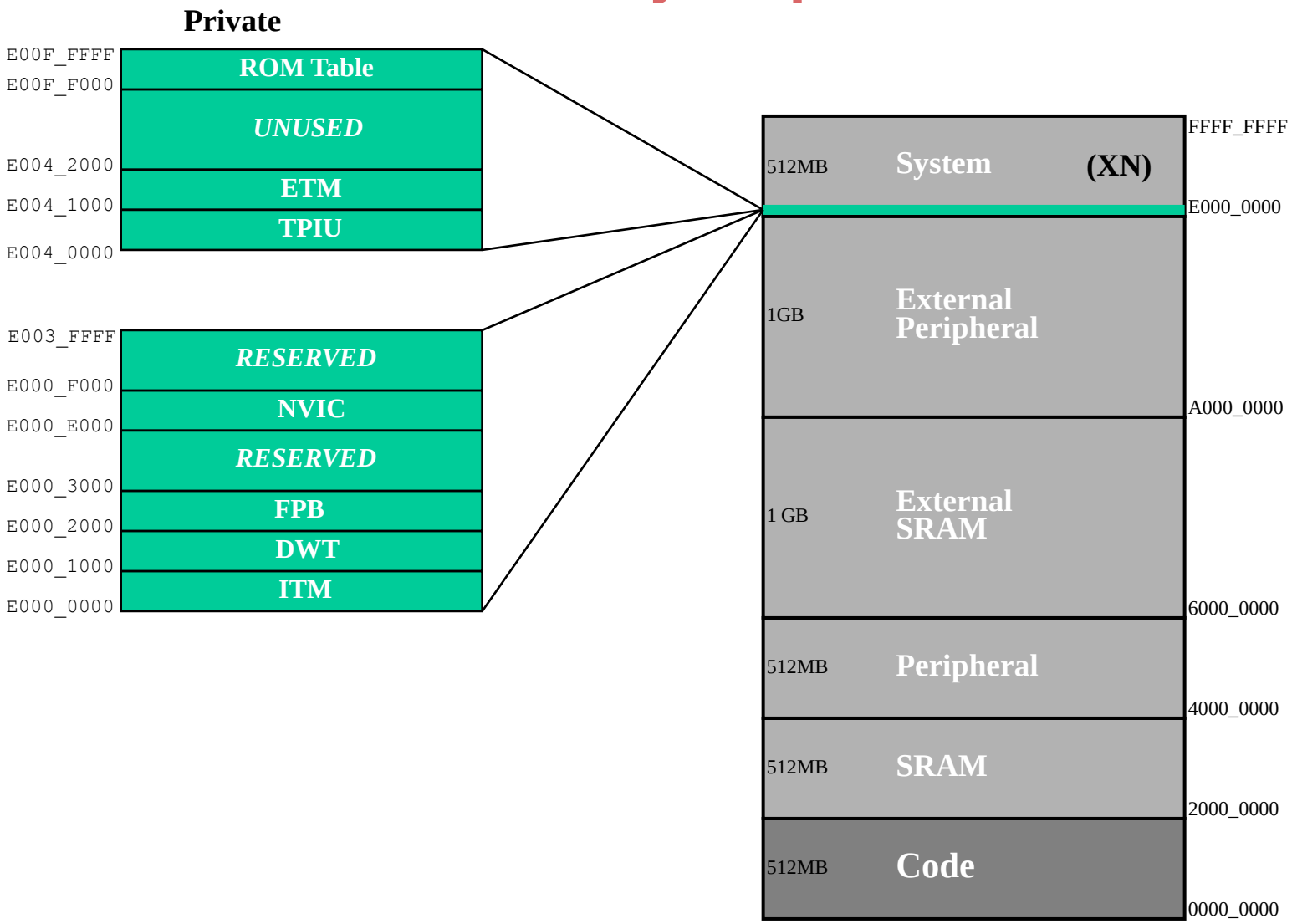
Once interrupts are enabled, the vector table (whether at 0 or in SRAM) must then have pointers to all enabled (by mask) exceptions

Vector Table in Assembly

```
                PRESERVE8
                THUMB
                IMPORT ||Image$$ARM_LIB_STACK$$ZI$$Limit||
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors

__Vectors       DCD     ||Image$$ARM_LIB_STACK$$ZI$$Limit||  ; Top of Stack
                DCD     Reset_Handler                ; Reset Handler
                DCD     NMI_Handler                  ; NMI Handler
                DCD     HardFault_Handler            ; Hard Fault Handler
                DCD     MemManage_Handler            ; MemManage Fault Handler
                DCD     BusFault_Handler             ; Bus Fault Handler
                DCD     UsageFault_Handler           ; Usage Fault Handler
                DCD     0, 0, 0, 0,                  ; Reserved x4
                DCD     SVC_Handler,                 ; SVCall Handler
                DCD     Debug_Monitor                ; Debug Monitor Handler
                DCD     0                            ; Reserved
                DCD     PendSV_Handler               ; PendSV Handler
                DCD     SysTick_Handler              ; SysTick Handler
                ; External vectors start here
```

Processor Memory Map



Memory Types and Properties

- **There are 3 different memory types:**
  - Normal, Device and Strongly Ordered
- **Normal memory** is the most flexible memory type:
  - Suitable for different types of memory, for example, ROM, RAM, Flash and SDRAM
  - Accesses may be restarted
  - Caches and Write Buffers are permitted to work alongside Normal memory
- **Device memory** is suitable for peripherals and I/O devices
  - Caches are not permitted, but write buffers are still supported
  - Unaligned accesses are unpredictable
  - Accesses must not be restarted
    - Load/store multiple instructions should not be used to access Device memory
- **Strongly ordered memory** is similar to Device memory
  - Buffers are not supported and the PPB is marked Strongly Ordered

System Control Block

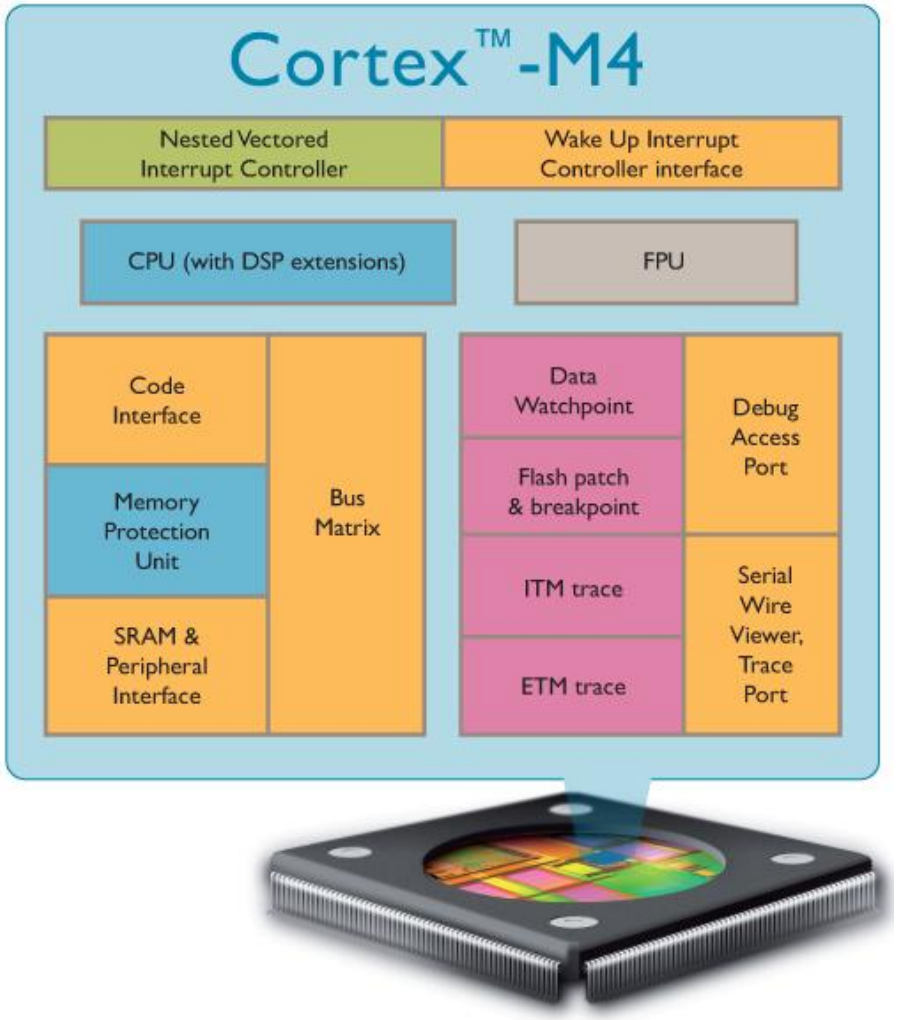
- **Memory mapped space containing registers to configure, control, and deal with interrupts, exceptions, and debug**
  - Replaces co-processor #15 in older ARM cores

Address	Type	Reset Value	Function
0xE000E000	Read/Write	0x00000000	Master Control register - RESERVED
0xE000E004	Read Only	IMP DEFINED	Interrupt Controller Type Register
0xE000ED00	Read Only	IMP DEFINED	CPUID Base Register
0xE000ED04	Read/Write	0x00000000	Interrupt Control State Register
0xE000ED08	Read/Write	0x00000000	Vector Table Offset Register
0xE000ED0C	Read/Write	Bits[10:8] = 000	Application Interrupt/Reset Control Register

More SCB Registers

Address	Type	Reset Value	Function
0xE000ED10	Read/Write	0x00000000	System Control Register
0xE000ED14	Read/Write	0x00000000	Configuration Control Register
0xE000ED18	Read/Write	0x00000000	System Handlers 4-7 Priority Register
0xE000ED1C	Read/Write	0x00000000	System Handlers 8-11 Priority Register
0xE000ED20	Read/Write	0x00000000	System Handlers 12-15 Priority Register
0xE000ED24	Read/Write	0x00000000	System Handler Control and State Register
0xE000ED28	Read/Write	n/a - status	Configurable Fault Status Registers (3)
0xE000ED2C	Read/Write	n/a - status	HardFault Status Register
0xE000ED30	Read/Write	n/a - status	DebugFault Status Register
0xE000ED34	Read/Write	Unpredictable	MemManage Address Register
0xE000ED38	Read/Write	Unpredictable	BusFault Address Register
0xE000ED3C	Read/Write	Unpredictable	Auxiliary Fault Status Register (vendor specific)
0xE000EF00	Write Only		Software Trigger Interrupt Register

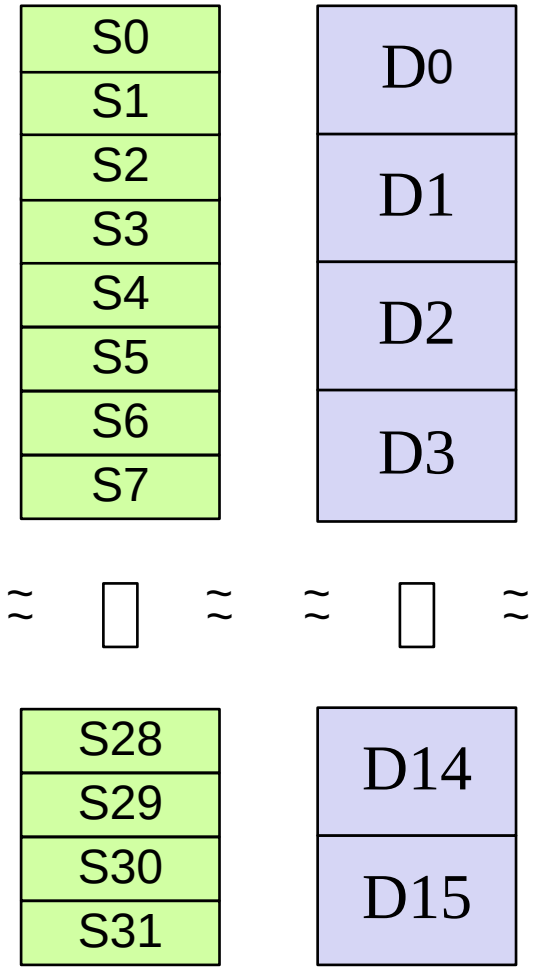
Cortex-M4



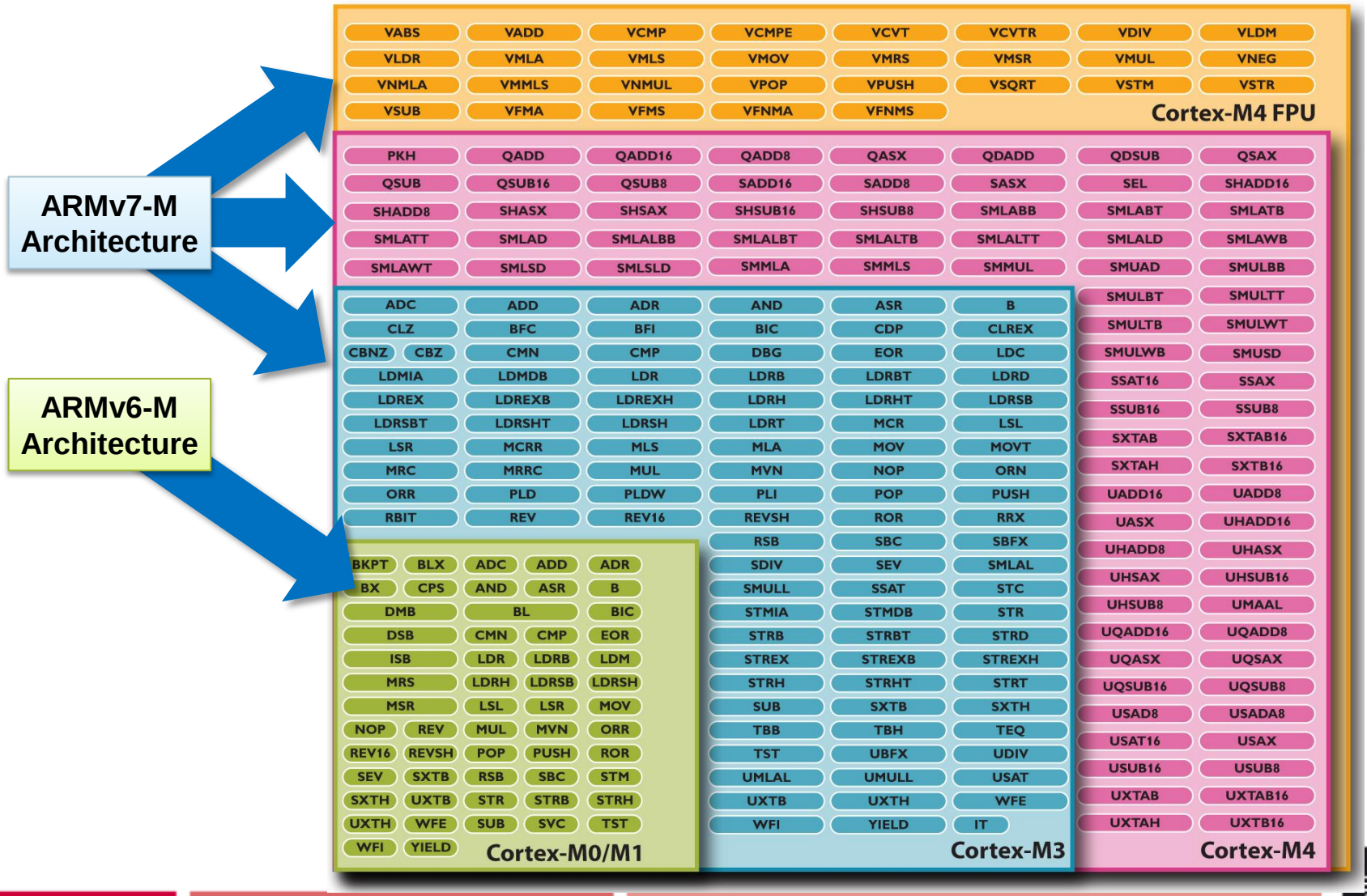
- **ARMv7E-M Architecture**
  - Thumb-2 only
  - DSP extensions
- **Optional FPU (Cortex-M4F)**
- **Otherwise, same as Cortex-M3**
- **Implements full Thumb-2 instruction set**
  - Saturated math (e.g. QADD)
  - Packing and unpacking (e.g. UXTB)
  - Signed multiply (e.g. SMULTB)
  - SIMD (e.g. ADD8)

Cortex-M4F Floating Point Registers

- **FPU provides a further 32 single-precision registers**
- **Can be viewed as either**
  - 32 x 32-bit registers
  - 16 x 64-bit doubleword registers
  - Any combination of the above



Binary Upwards Compatibility

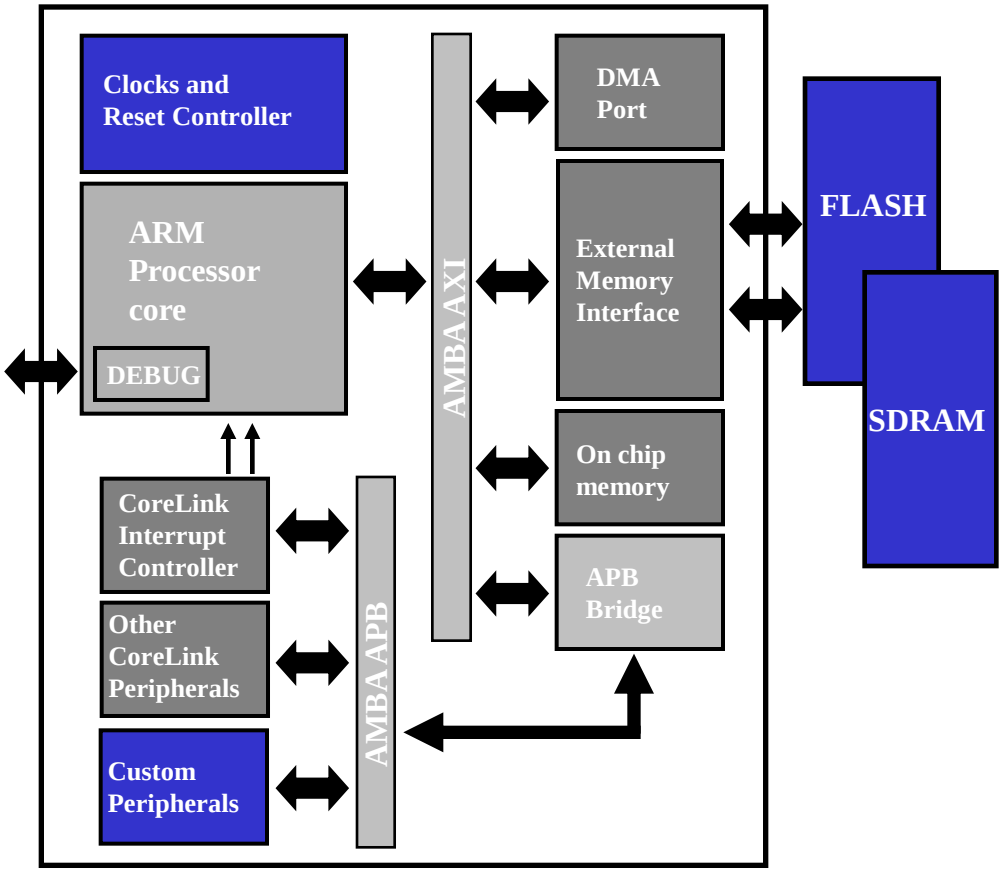


Plan

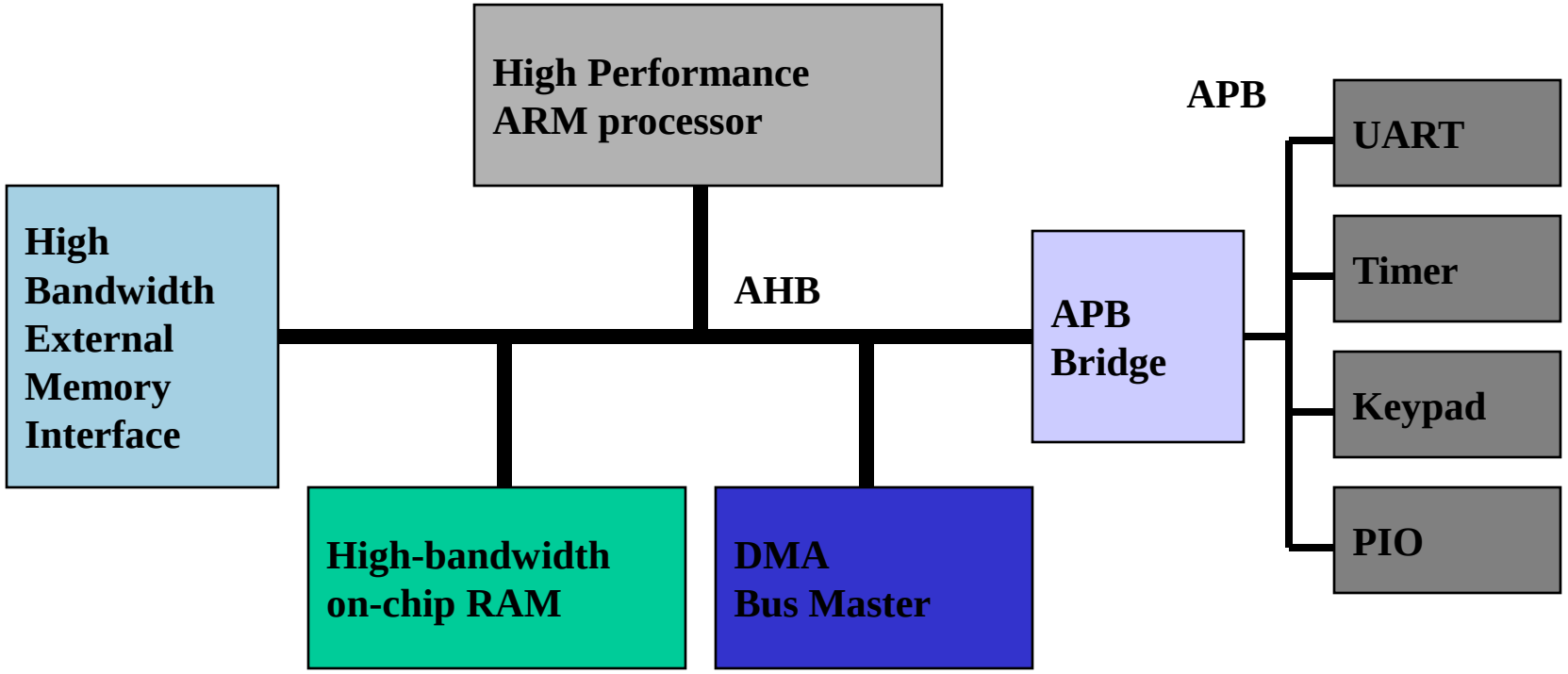
- **ARM7TDMI**
- **Cortex**
  - Cortex A family
  - Cortex M family
- **System bus**

ARM-based system

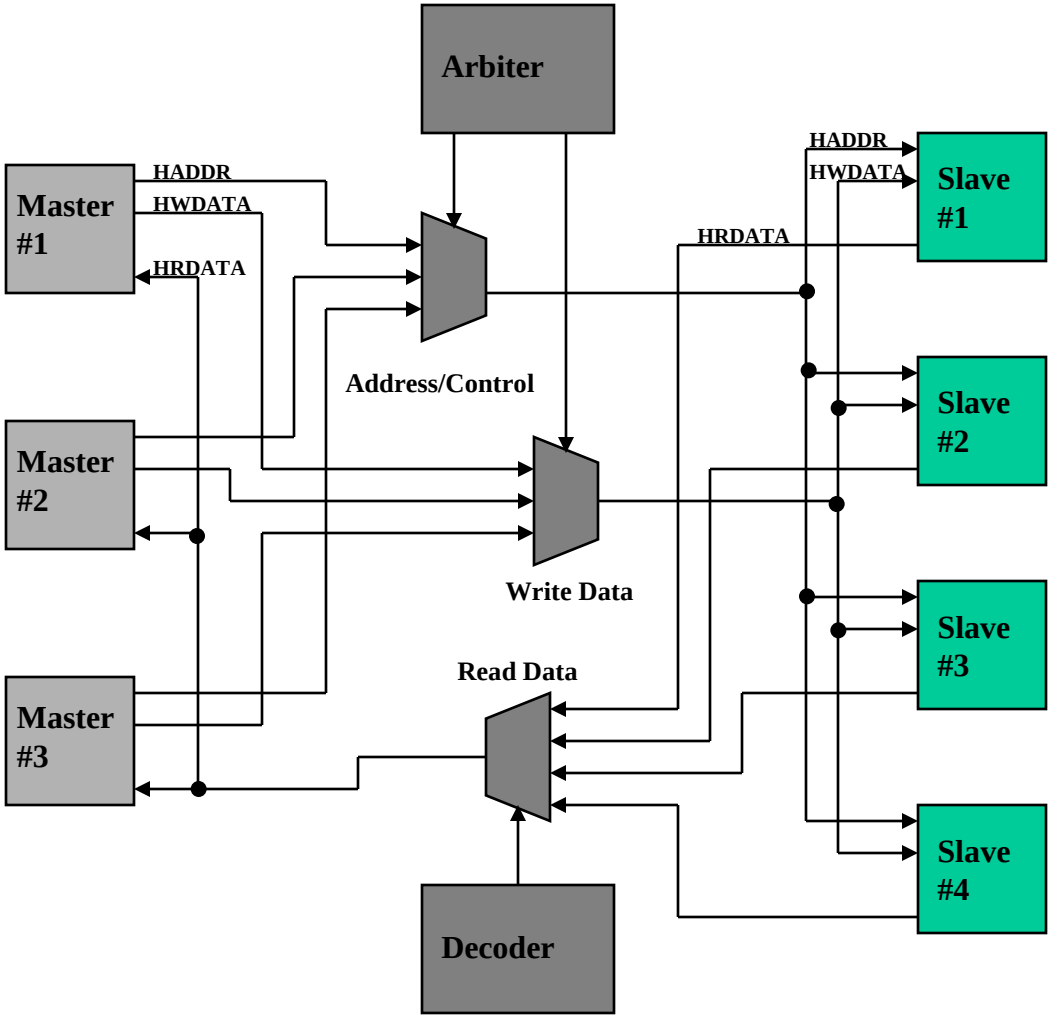
- **ARM core deeply embedded within an SoC**
  - External debug and trace via JTAG or CoreSight interface
- **Design can have both external and internal memories**
  - Varying width, speed and size – depending on system requirements
- **Can include ARM licensed CoreLink peripherals**
  - Interrupt controller, since core only has two interrupt sources
  - Other peripherals and interfaces
- **Can include on-chip memory from ARM Artisan Physical IP Libraries**
- **Elements connected using AMBA (Advanced Microcontroller Bus Architecture)**



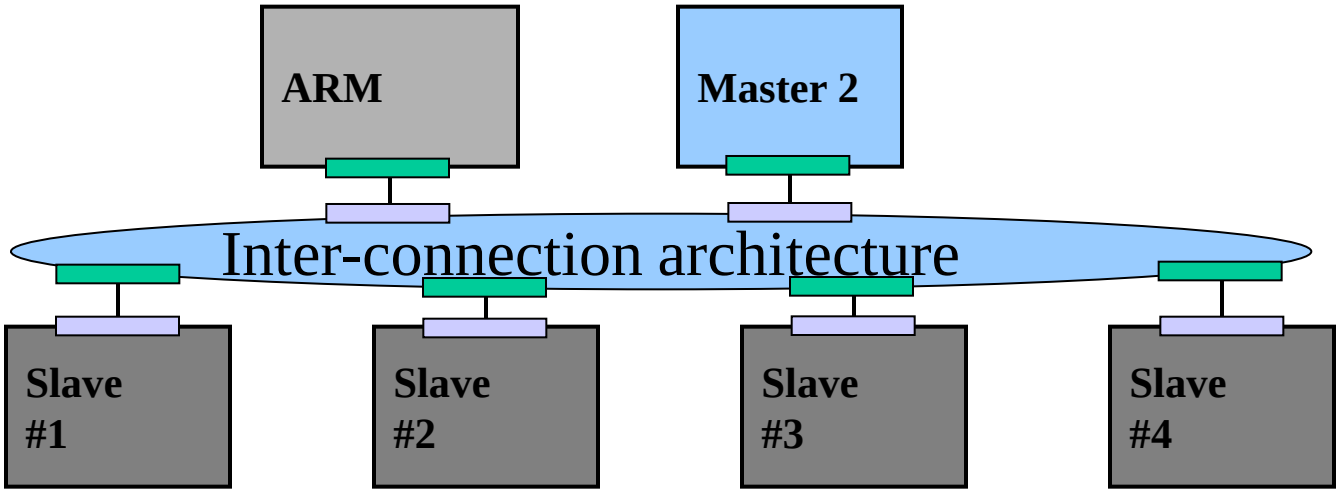
An Example AMBA System



AHB Structure



AXI Multi-Master System Design



Master interface

Slave interface