TELECOM Paris

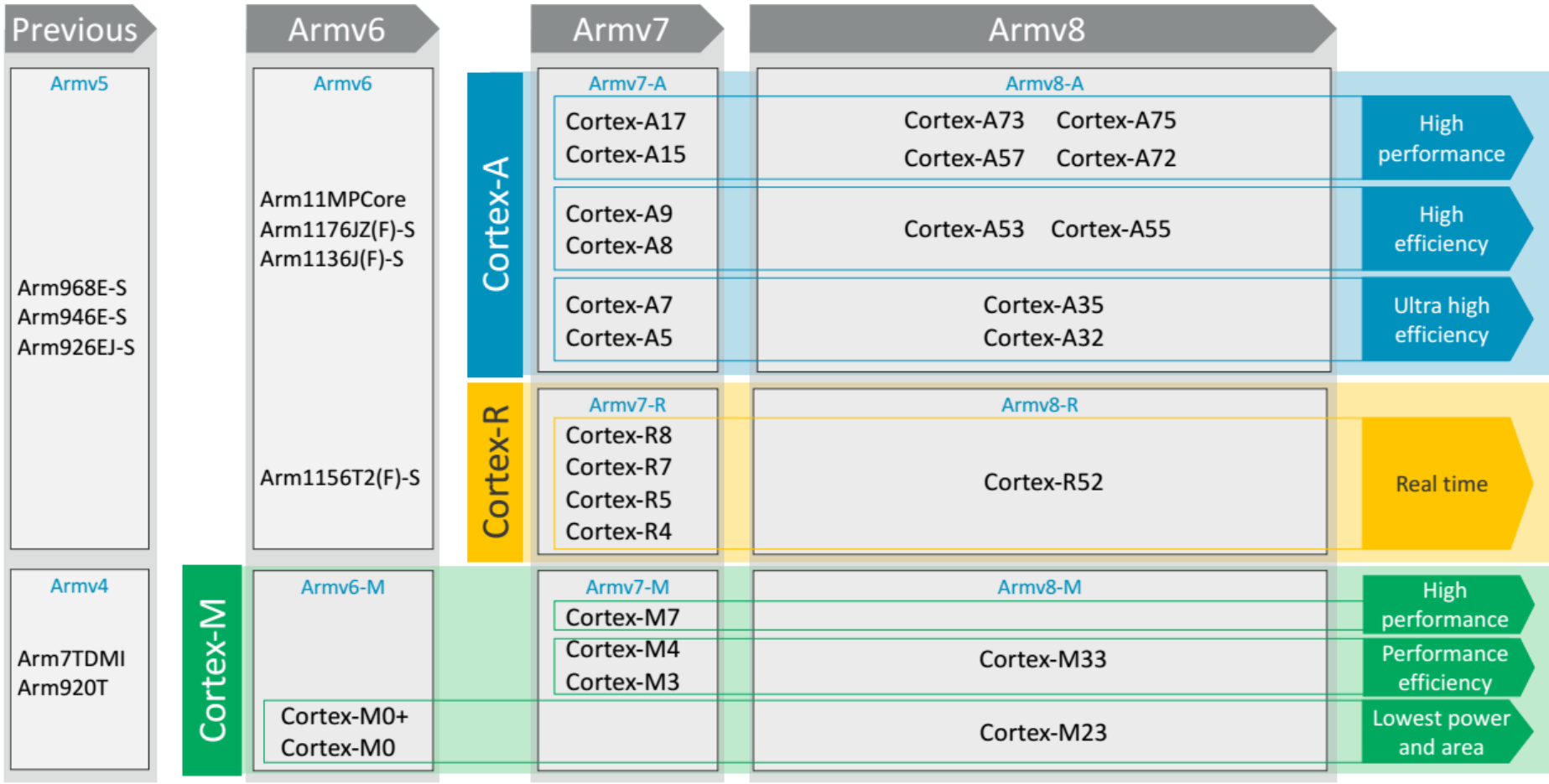
IP PARIS

# Architectures ARM

6 novembre 2019

**Jean-Luc DANGER**
**Tarik GRABA**

# La gamme des processeurs ARM



# ARM licenses (public)

- ARMv8-A
  - NVIDIA, Applied Micro, Cavium, AMD, Broadcom, Calxeda, HiSilicon, Samsung and STMicroelectronics
- Cortex-A15 4 ST-Ericson, TI, Samsung, nVIDIA
- Cortex-A9 9 NEC, nVIDIA, STMicroelectronics, TI, Toshiba …
- Cortex-A8 9 Broadcom, Freescale, Matsushita, Samsung, STMicroelectronics, Texas Instruments, PMC-Sierra
- Cortex-A5 3 AMD ---
- Cortex-R4 14 Broadcom, Texas Instruments, Toshiba, Inf
- Cortex-M4 5 Freescale, NXP, Atmel, ST
- Cortex-M3 29 Actel, Broadcom, Energy Micro, Luminary, Micro, NXP, STMicroelectronics, TI, Toshiba, Zilog, …
- Cortex-M0 14 Austria-microsystems, Chungbuk Technopark,NXP, Triad Semiconductor, Melfas, ST
- Cortex-M0+ Freescale, NXP
- ARM7 172, ARM9 271, ARM11 82

Source : G.N. Khan

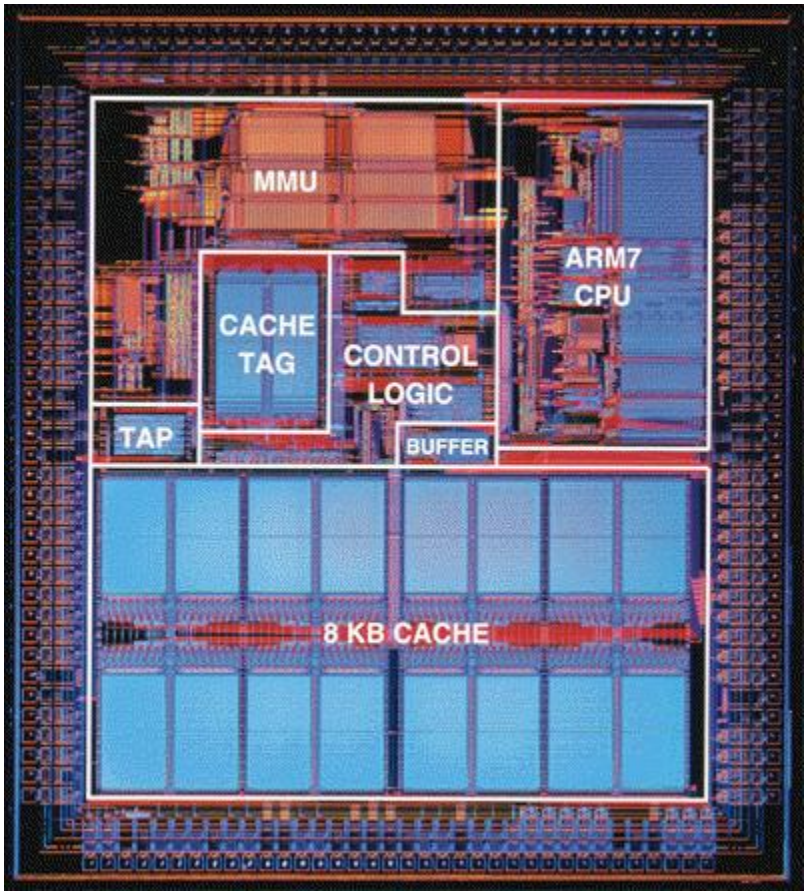
# Plan

- **ARM historique : l'ARM7TDMI**
- **Les Cortex**
  - Cortex M family
  - Cortex A family
- **System bus**

# Qu’est ce qu’un ARM7TDMI?

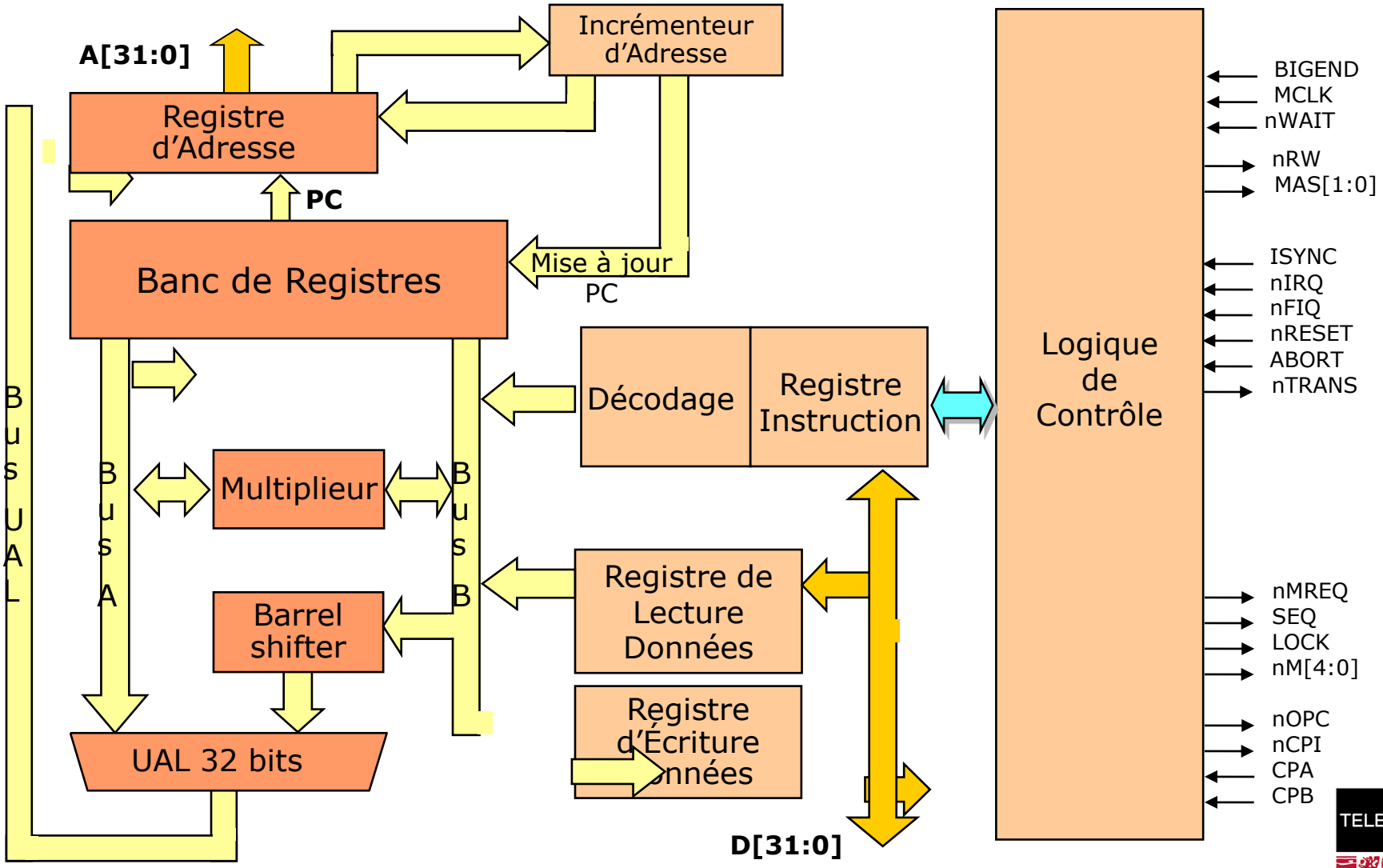
- **Processeur à Architecture « Von Neumann »**
  - Même bus mémoire pour instructions et données
- **3 étages de pipeline : Fetch, Decode, Execute**
- **Instructions sur 32 Bits**
- **2 instructions d'accès à la mémoire LOAD et STORE**
- **T : support du mode "Thumb" (instructions sur 16 bits)**
- **D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

# Vue d'une puce utilisant un ARM7

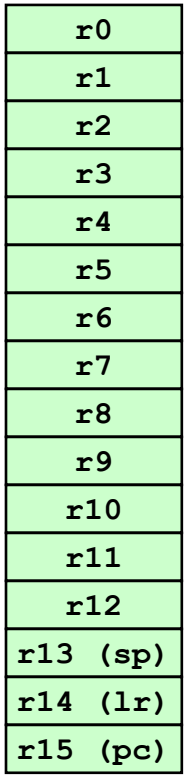



ARM710 (25mm² en 0.5µm (1995), 2.9 mm² en 0.18µm (2000))

# Vue simplifiée du Cœur ARM7TDMI



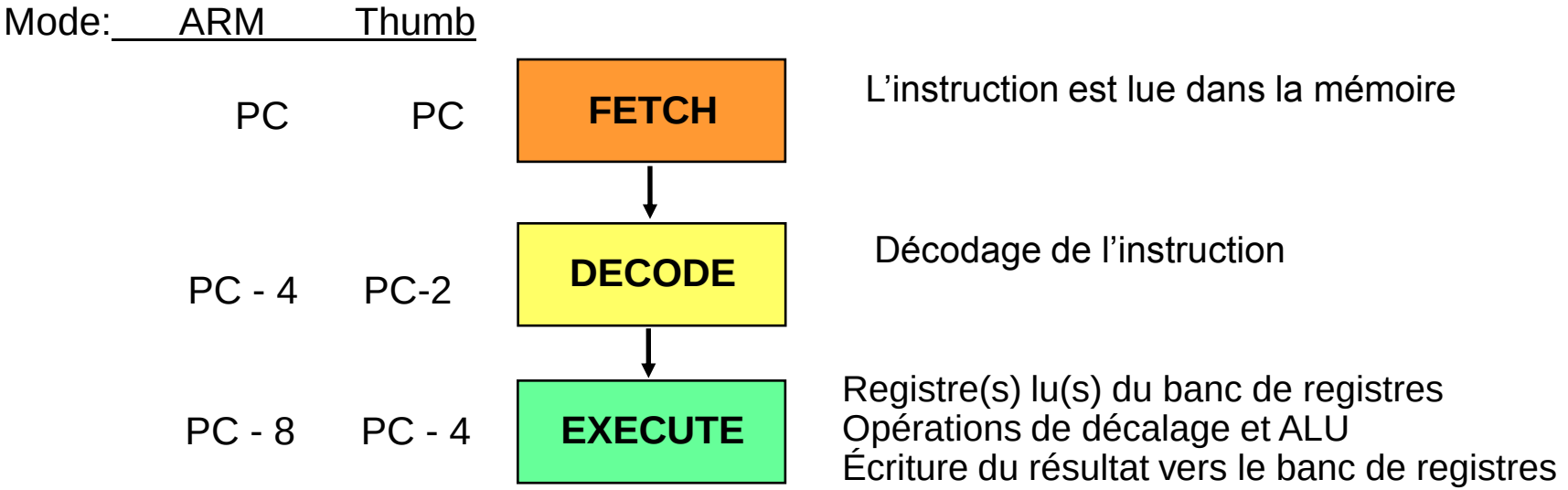
# Banc de registres




- **16** Registres génériques
  - De 0 à 15
  - Le PC en fait partie
- Les instructions utilisent forcement un registre :
  - Lire une donnée en mémoire dans un registre (LOAD)
  - Calcul entre registres
  - Écrire le contenu d'un registre en mémoire (STORE)

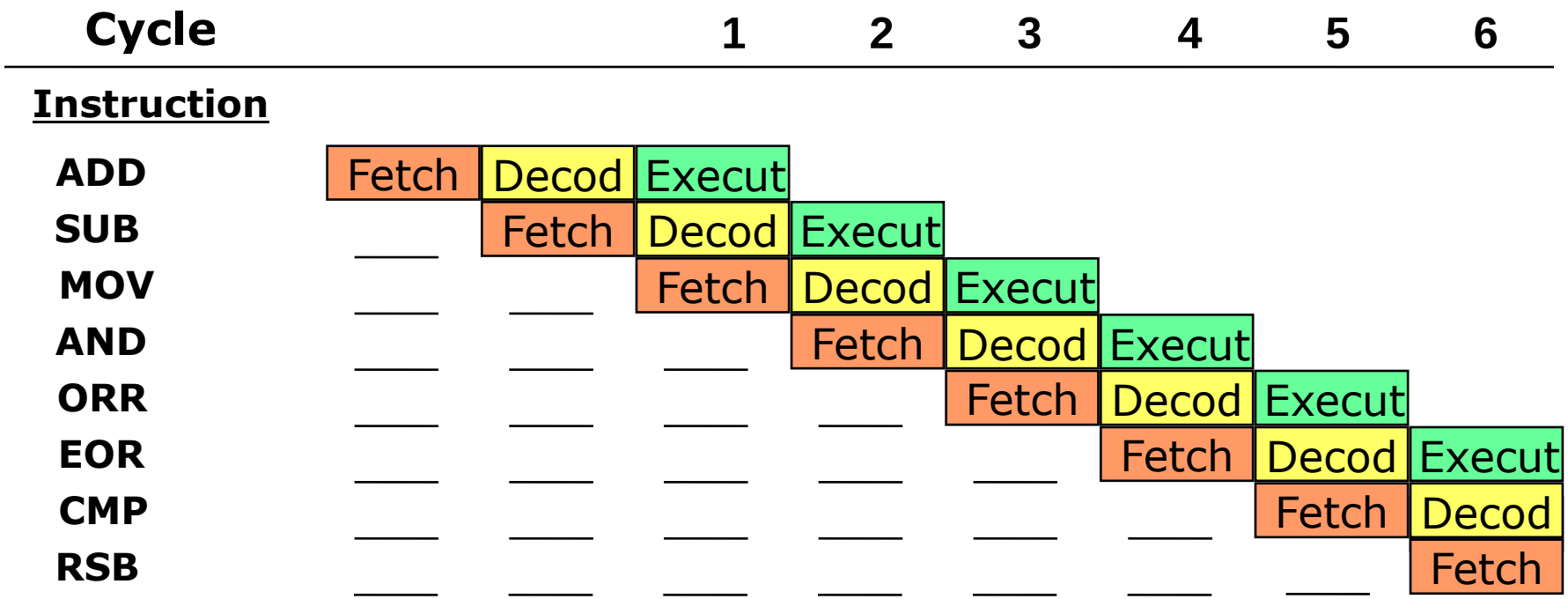
# Le Pipeline d'Instructions

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



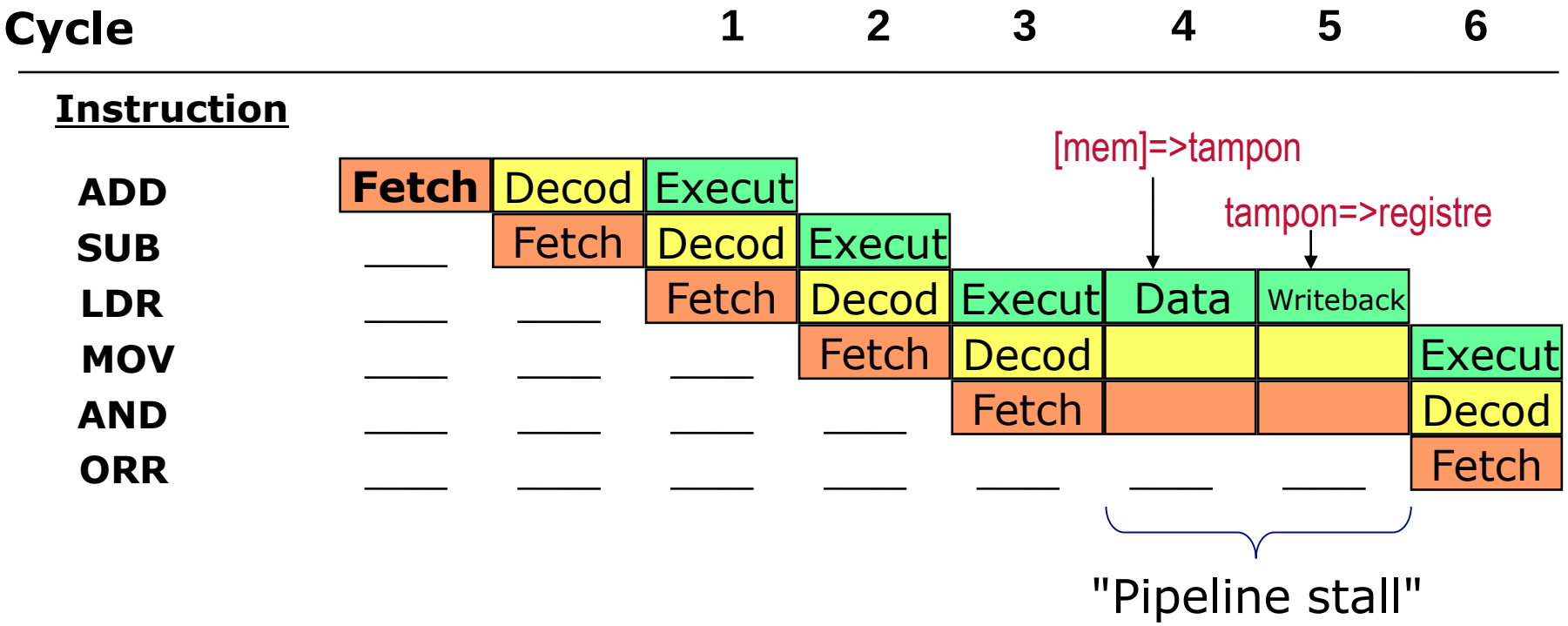

- Le PC pointe sur l'instruction en cours de lecture (FETCHed), et non sur l'instruction en cours d'exécution.

# Exemple: Pipeline Optimal




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

# Exemple: Pipeline avec LOAD




- 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: 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)

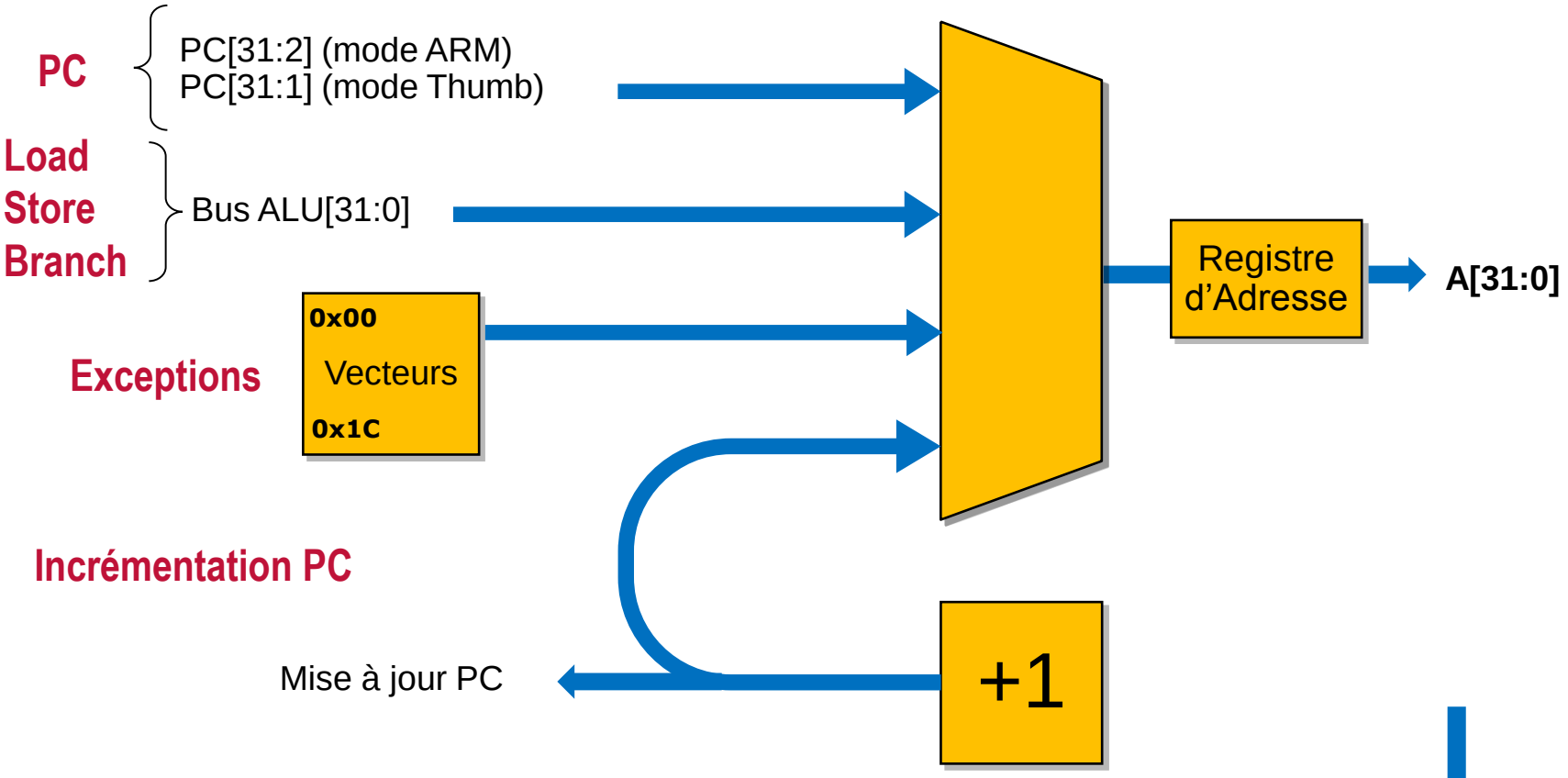
ranger le contenu de R1 à l'adresse [R8+0x14]

Conditions initiales :

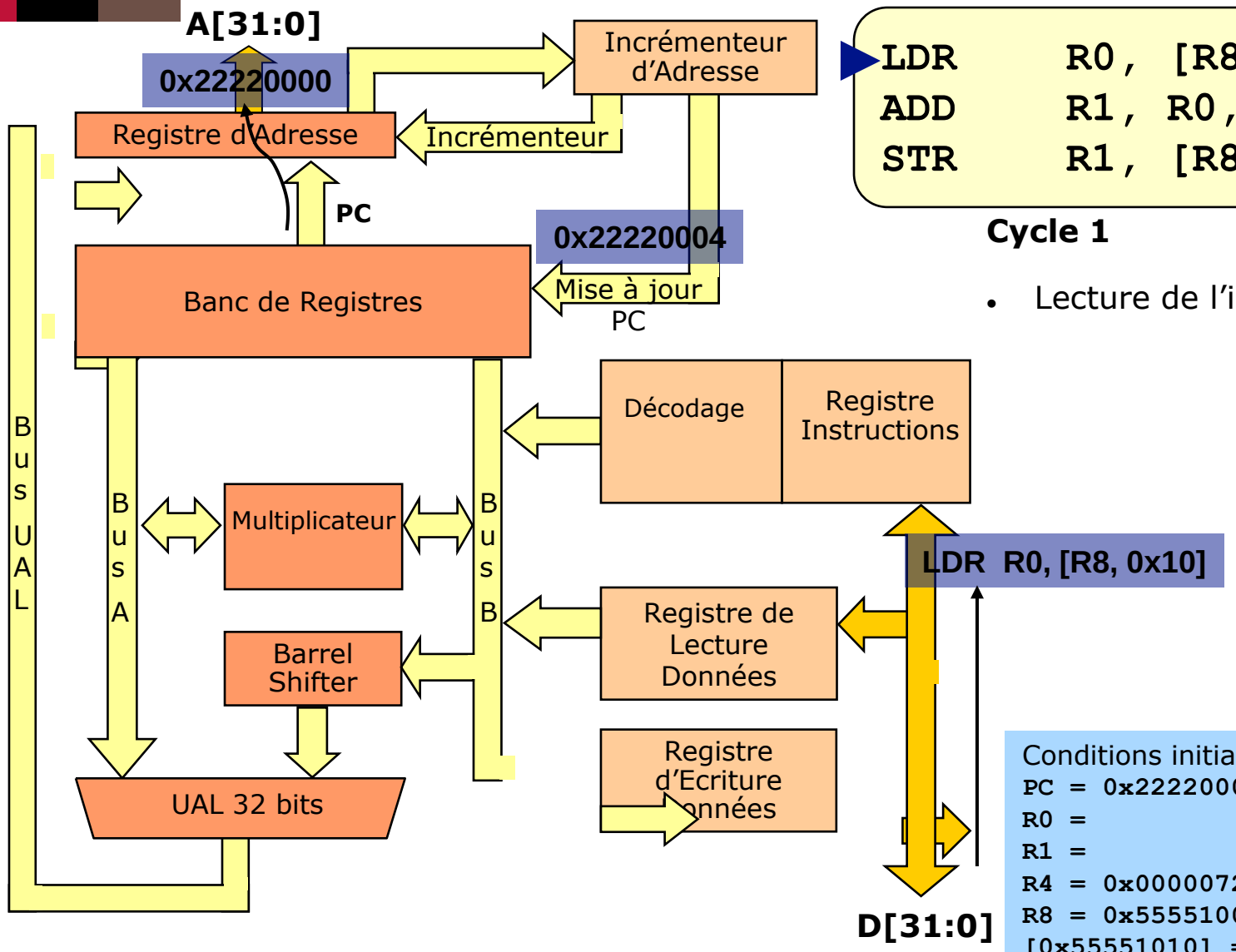
```
PC = 0x22220000
R4 = 0x00000721
R8 = 0x55551000
[0x55551010] = 0x00000834
```

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

# Génération des Adresses



# Séquence d'Instructions : Cycle 1




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

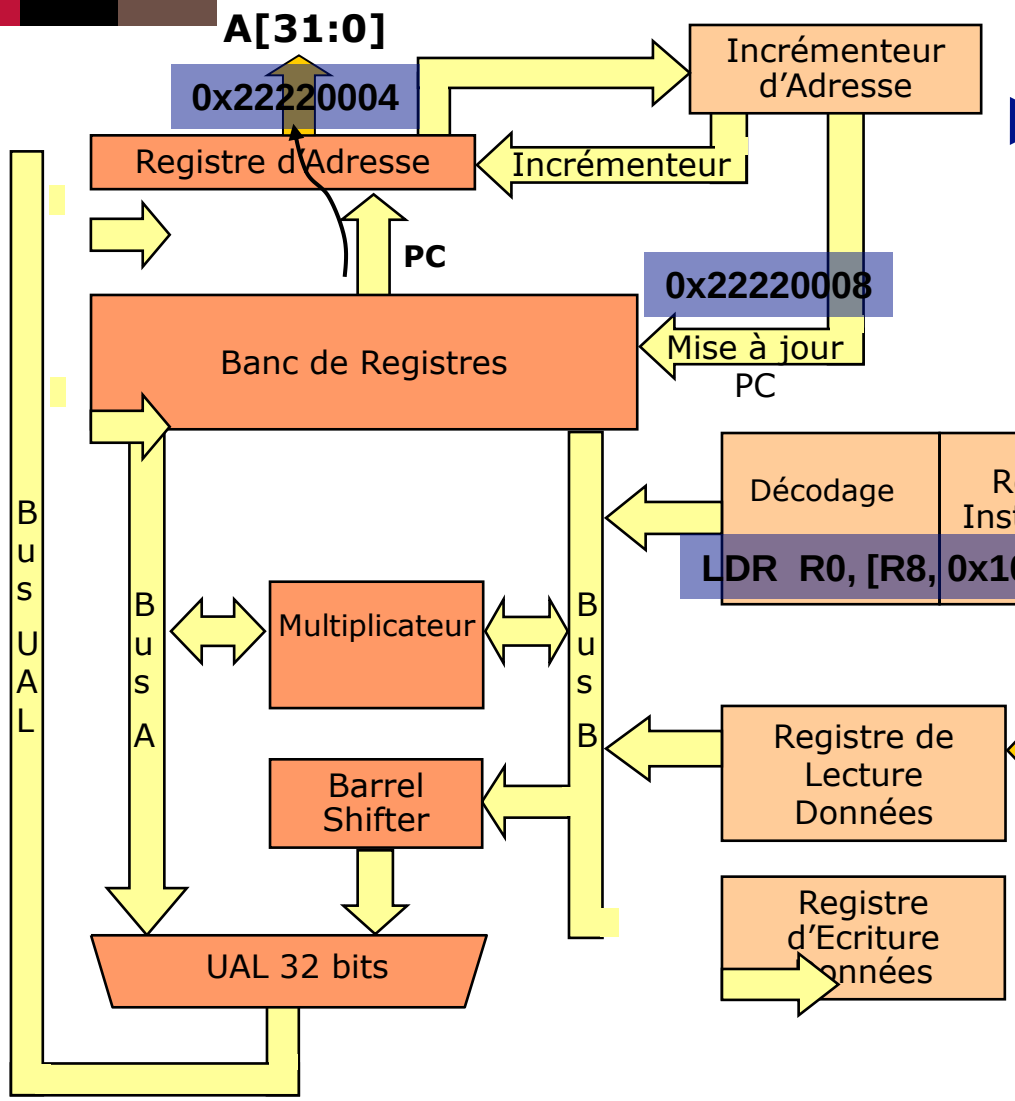
**Cycle 1**

- Lecture de l'instruction LDR

Conditions initiales :

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

# Séquence d'Instructions : Cycle 2



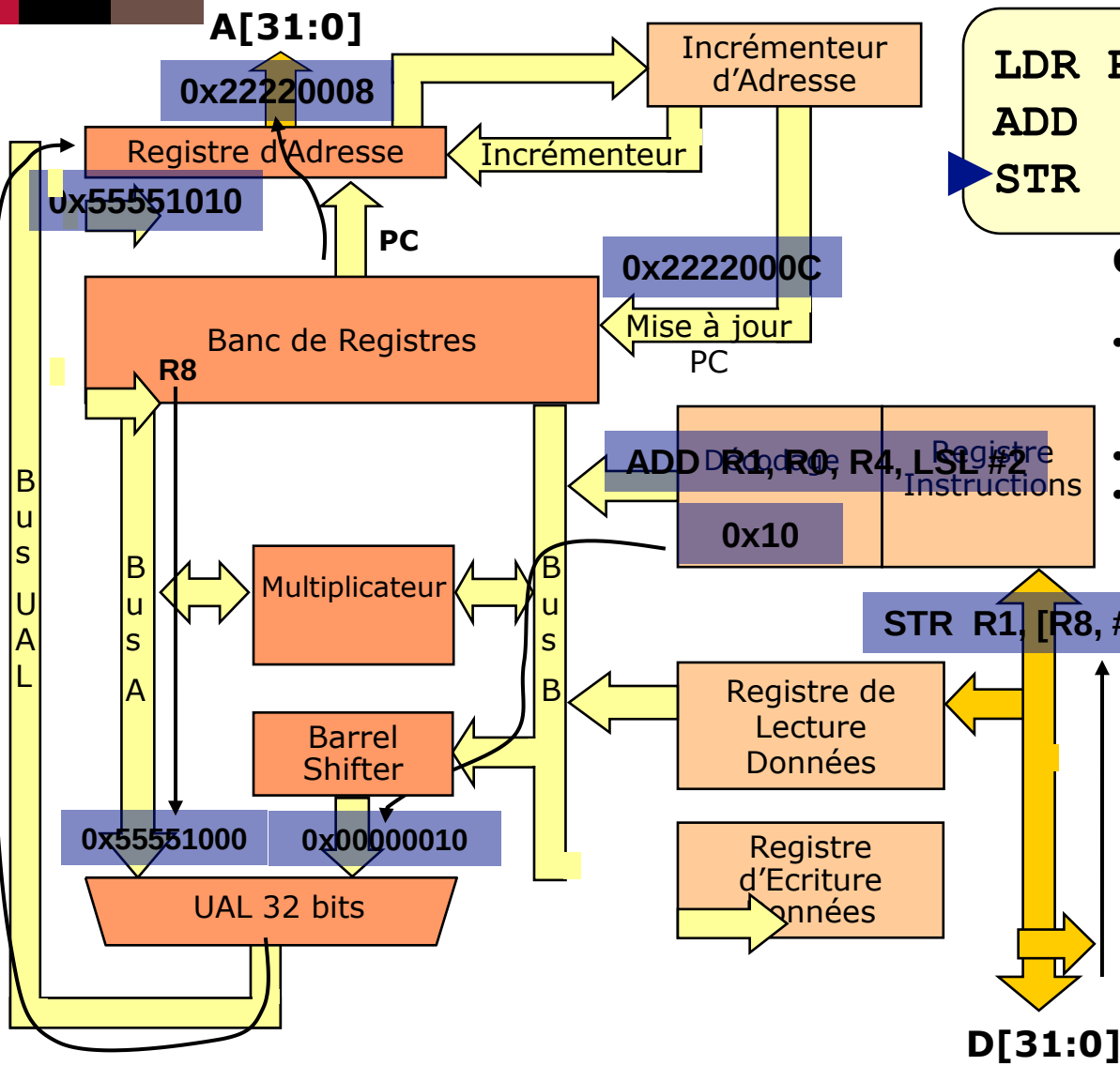

```
LDR     R0, [R8, 0x10]
ADD     R1, R0, R4, LSL #2
STR     R1, [R8, 0x14]
```

**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



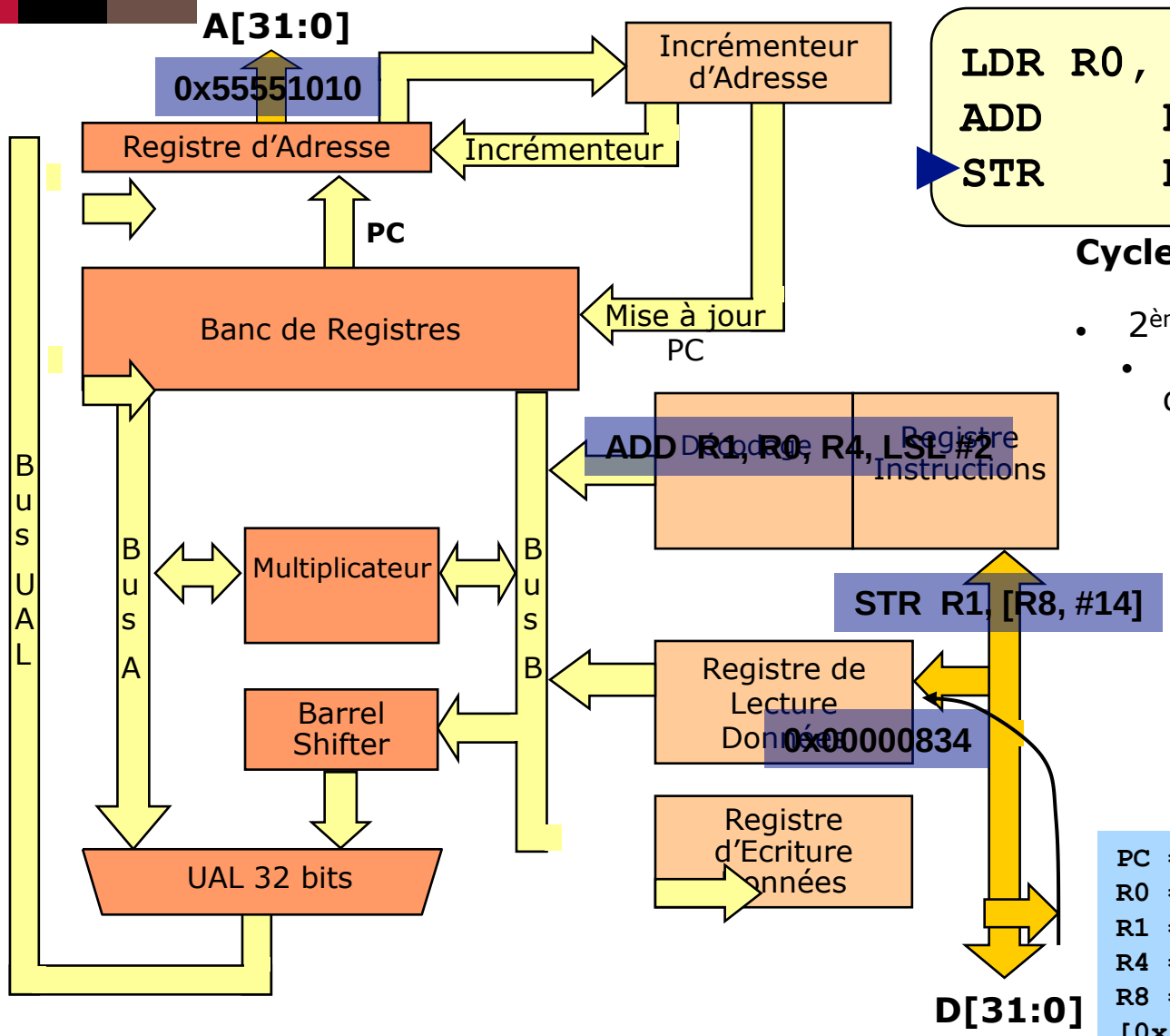

```
LDR R0, [R8, 0x10]
ADD     R1, R0, R4, LSL #2
STR     R1, [R8, 0x14]
```

**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



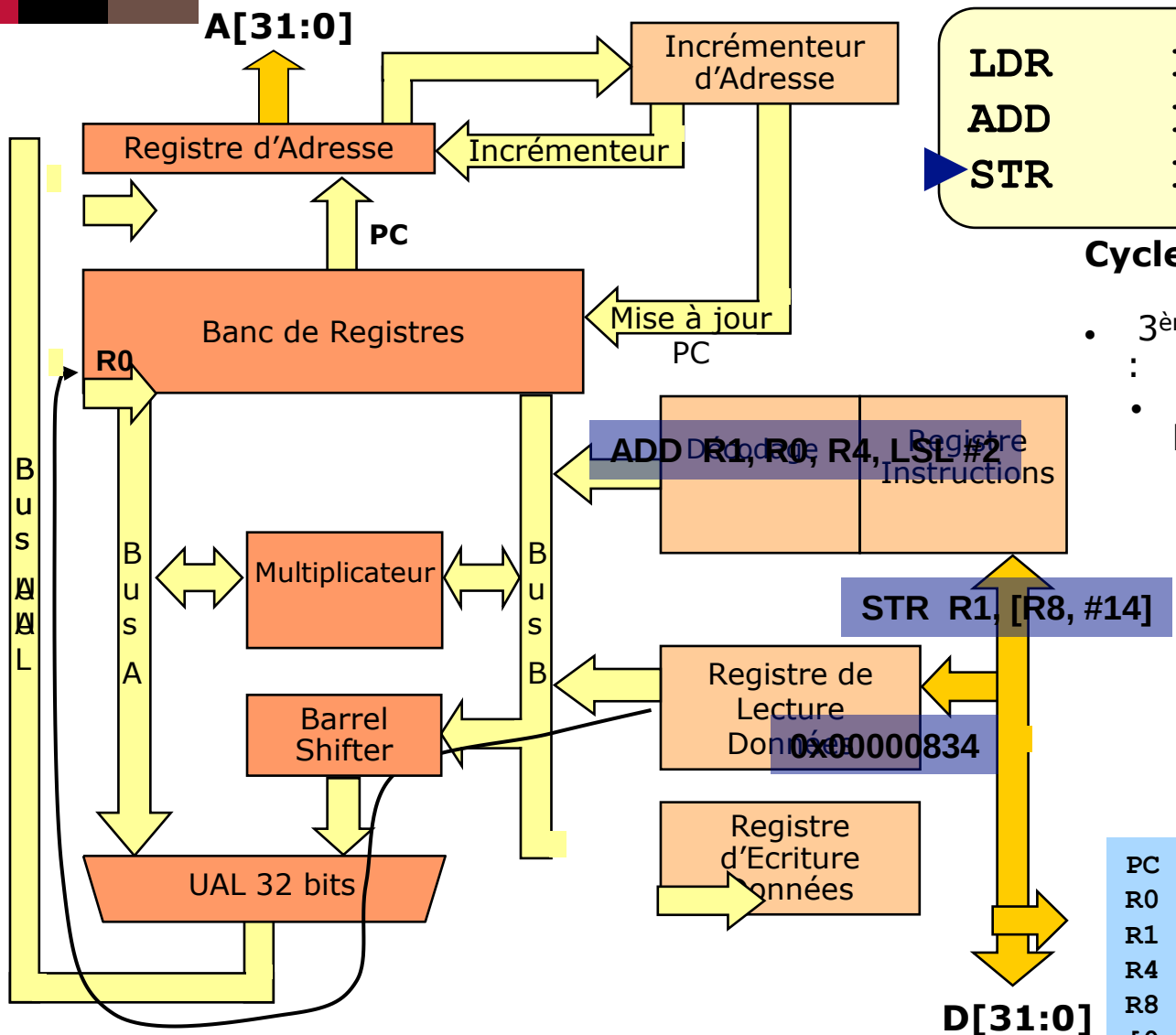

```
LDR R0, [R8, 0x10]
ADD     R1, R0, R4, LSL #2
STR     R1, [R8, 0x14]
```

**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



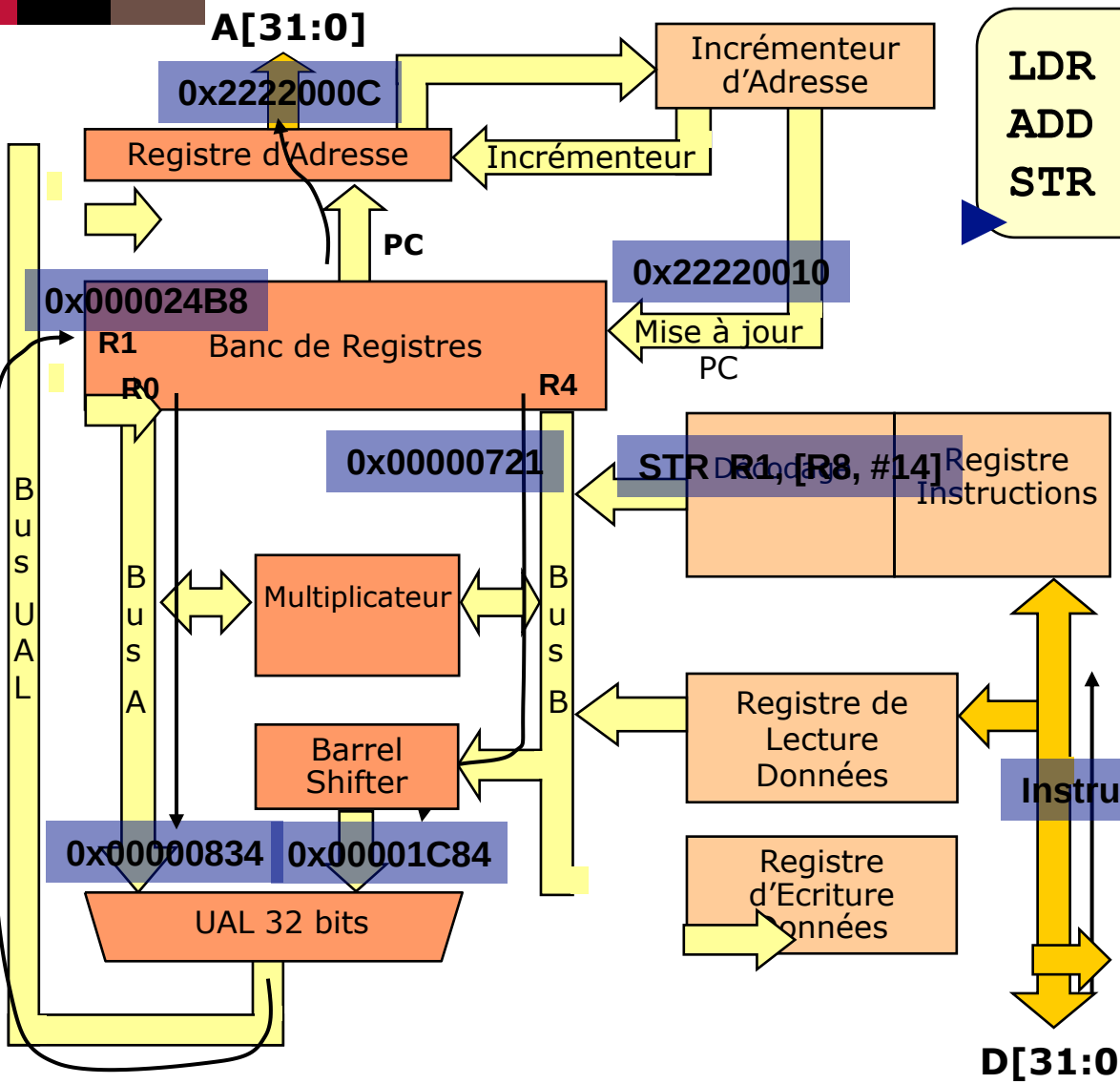

```
LDR     R0, [R8, 0x10]
ADD     R1, R0, R4, LSL #2
STR     R1, [R8, 0x14]
```

**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



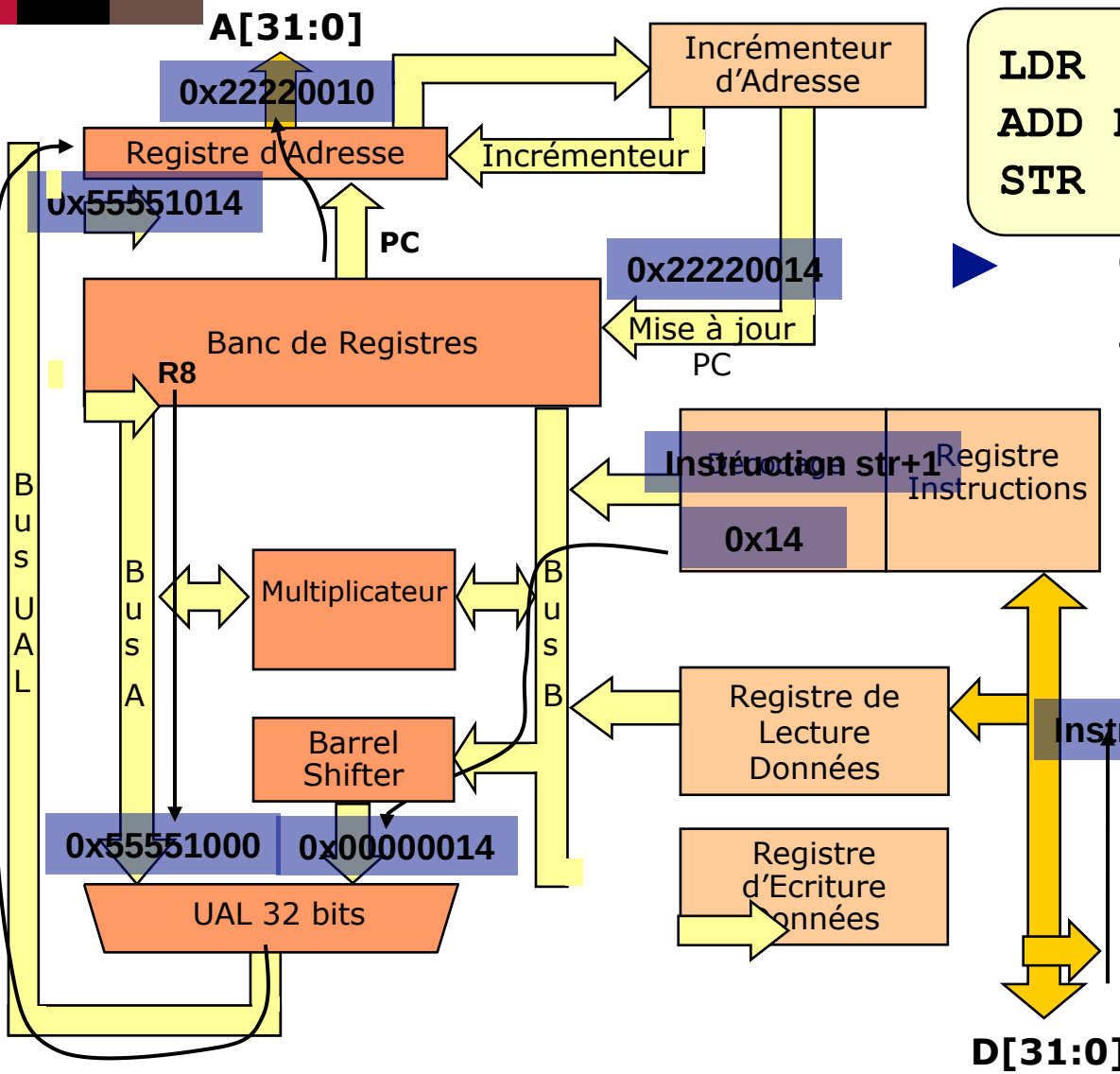

```
LDR     R0, [R8, 0x10]
ADD     R1, R0, R4, LSL #2
STR     R1, [R8, 0x14]
```

**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



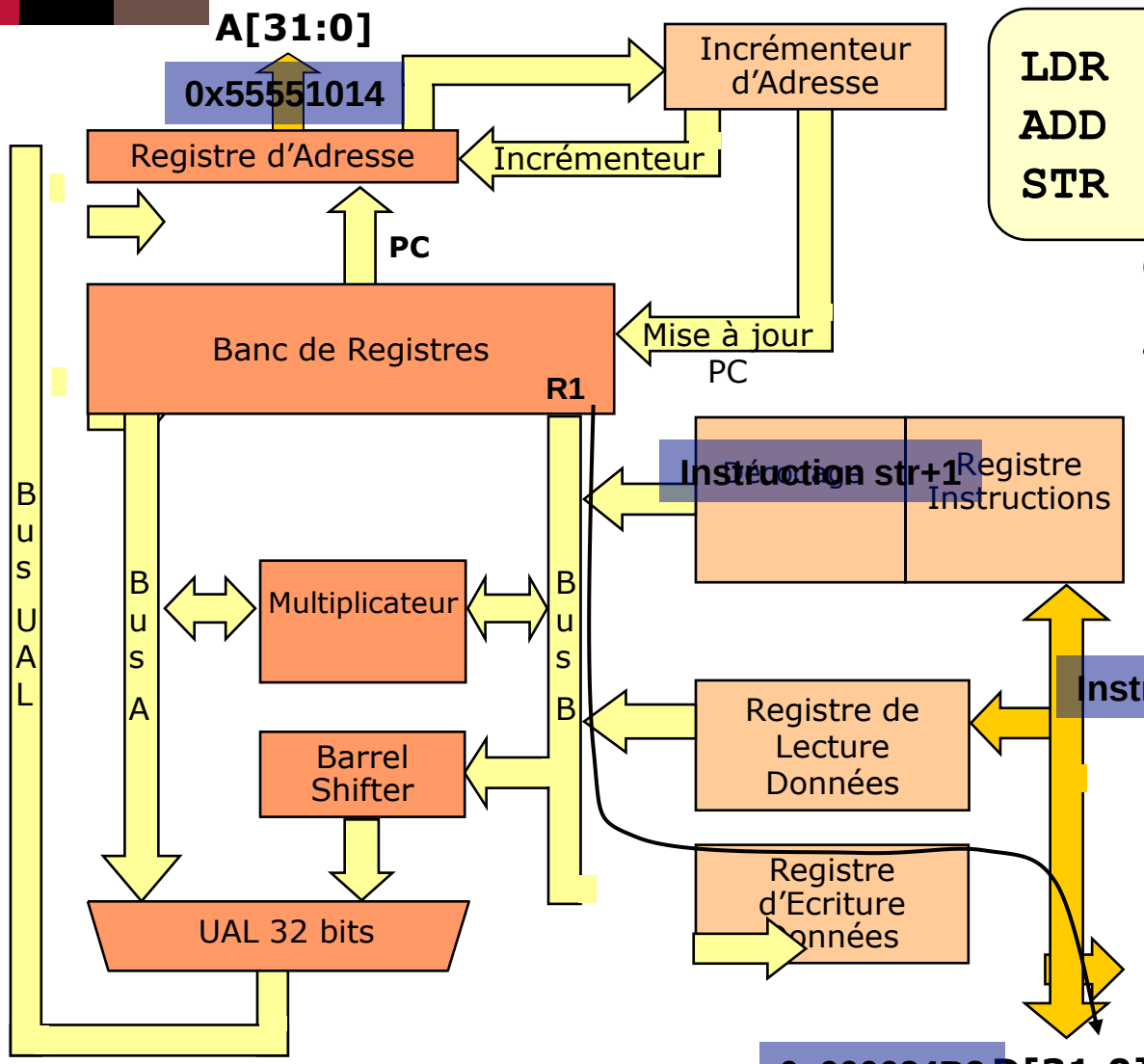

```
LDR     R0, [R8, 0x10]
ADD R1, R0, R4, LSL #2
STR     R1, [R8, 0x14]
```

**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

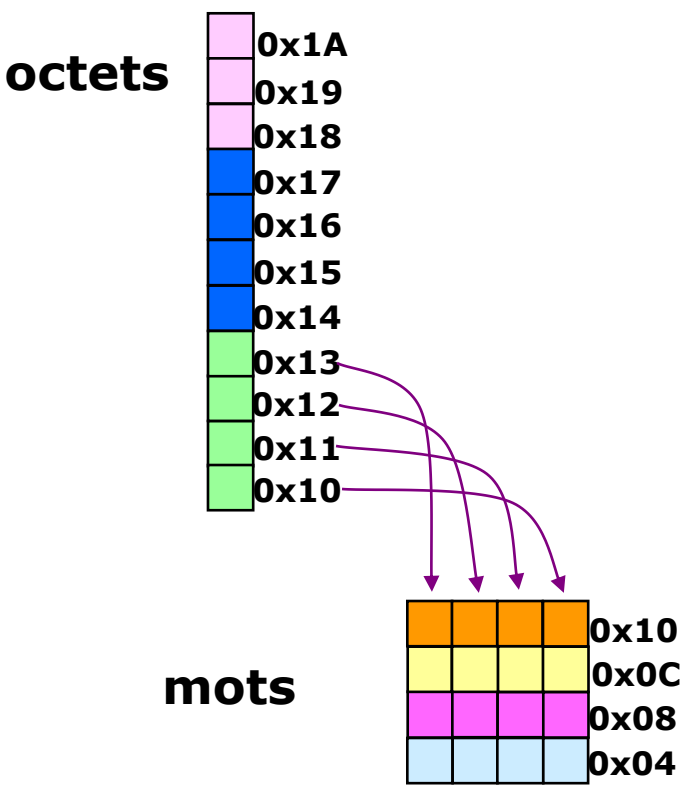
```
PC = 0x22220014
R0 = 0x00000834
R1 = 0x000024B8
R4 = 0x00000721
R8 = 0x55551000
[0x55551014] = 0x000024B8
```

# Accès à la Mémoire et aux E/S

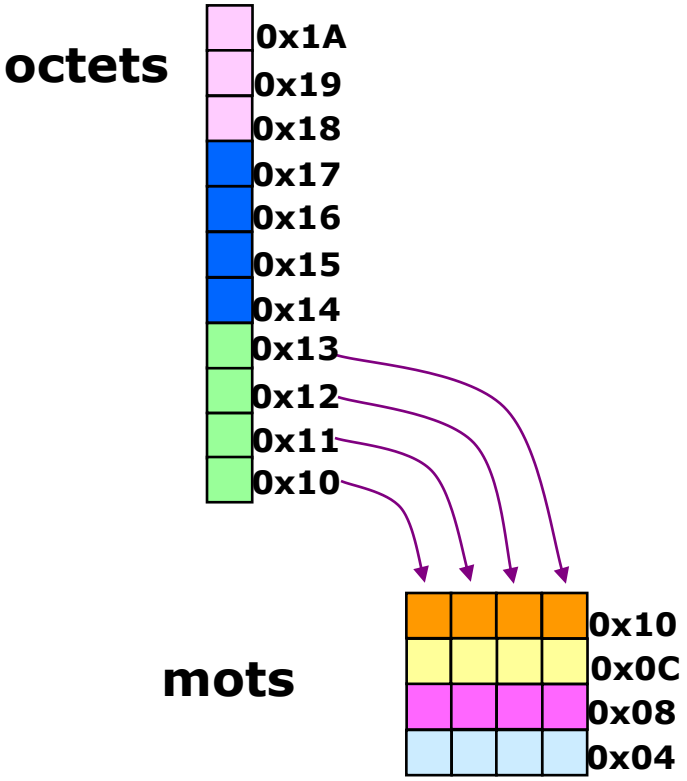
- **2 instructions d'accès :**
  - LOAD (LDR) et STORE (STR)
- **L'adressage mémoire se fait sur 32 bits**
  - => 4 Go.
- **Type des données :**
  - octets
  - demi-mots (16 bits)
  - mots (32 bits)
- **Les mots doivent être alignés sur des adresses multiples de 4 et les demi-mots, de 2.**
- **Les E/S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



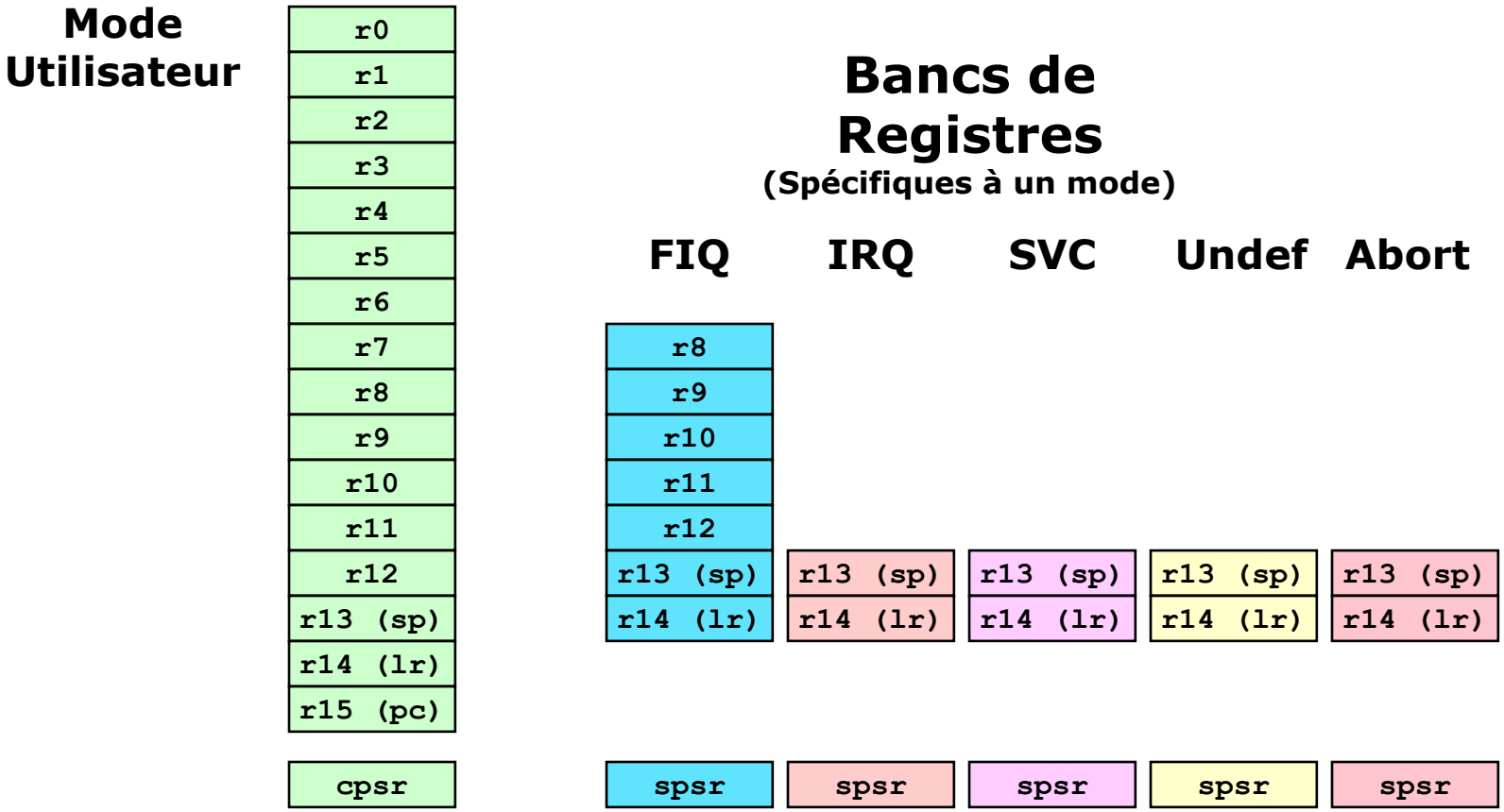

« Big Endian »

# Les Modes du Microprocesseur

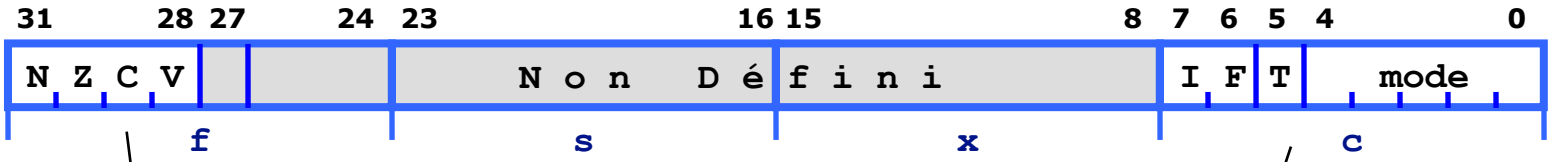
- **Un microprocesseur ARM a 7 modes opératoires de base :**
  - **User** : mode sans privilège où la plupart des tâches s'exécutent
  - **FIQ** : on y entre lors d'une interruption de priorité haute (rapide)
  - **IRQ** : on y entre lors d'une interruption de priorité basse (normale)
  - **Supervisor** : 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

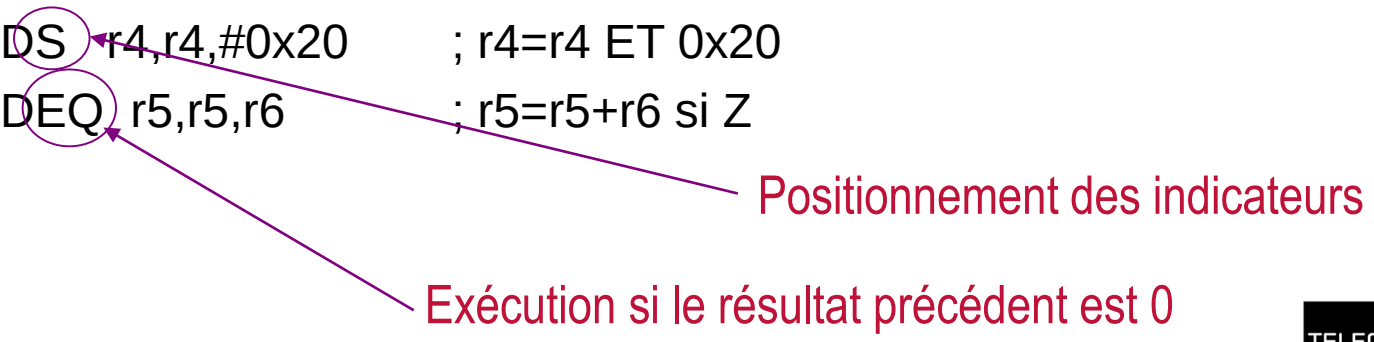



- Indicateurs conditionnels
  - N = Résultat Négatif
  - Z = Résultat nul (Zéro)
  - C = Retenue (Carry)
  - V = Débordement (oVerflow)

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

# Jeu d'Instructions ARM(1)

- **Les instructions sont sur 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)

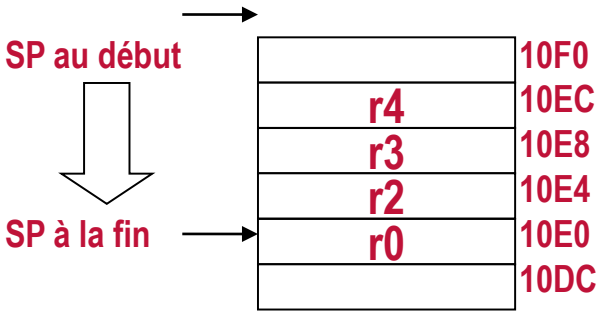
- Instructions spécifiques d'accès à la mémoire
  - `LDR r0,[r1],#4` *; r0=mem(r1), r1= r1+4*
  - `STRNEB r2,[r3,r4]` *; mem(r3+r4)=r2*
  - `LDRSH r5,[r6,#8]!` *; r5=mem(r6+8), r6=r6+8*
  - `STMFD sp!,{r0,r2-r4}` *; transferts multiples*
  - *;empilage : mem(sp+i)=liste de registres*

Si Z=0

En octet

En 16 bits
Avec extension
de signe sur les
16MSBs

r6=r6+8 en fin d'exécution

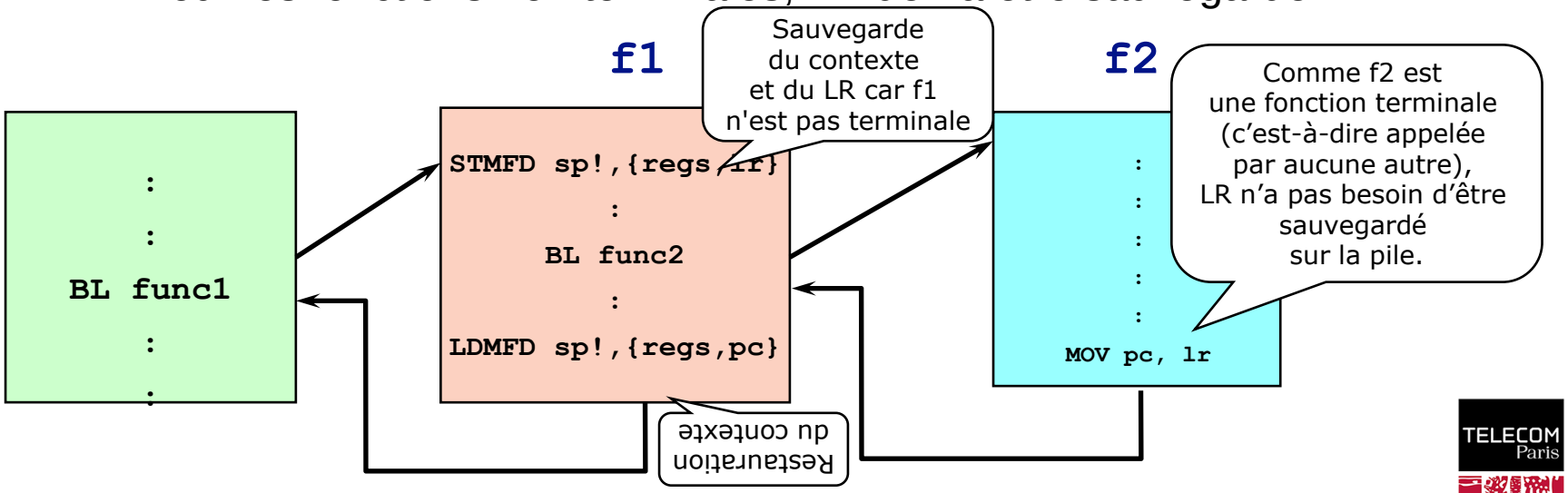



Opération réciproque de dépilage :

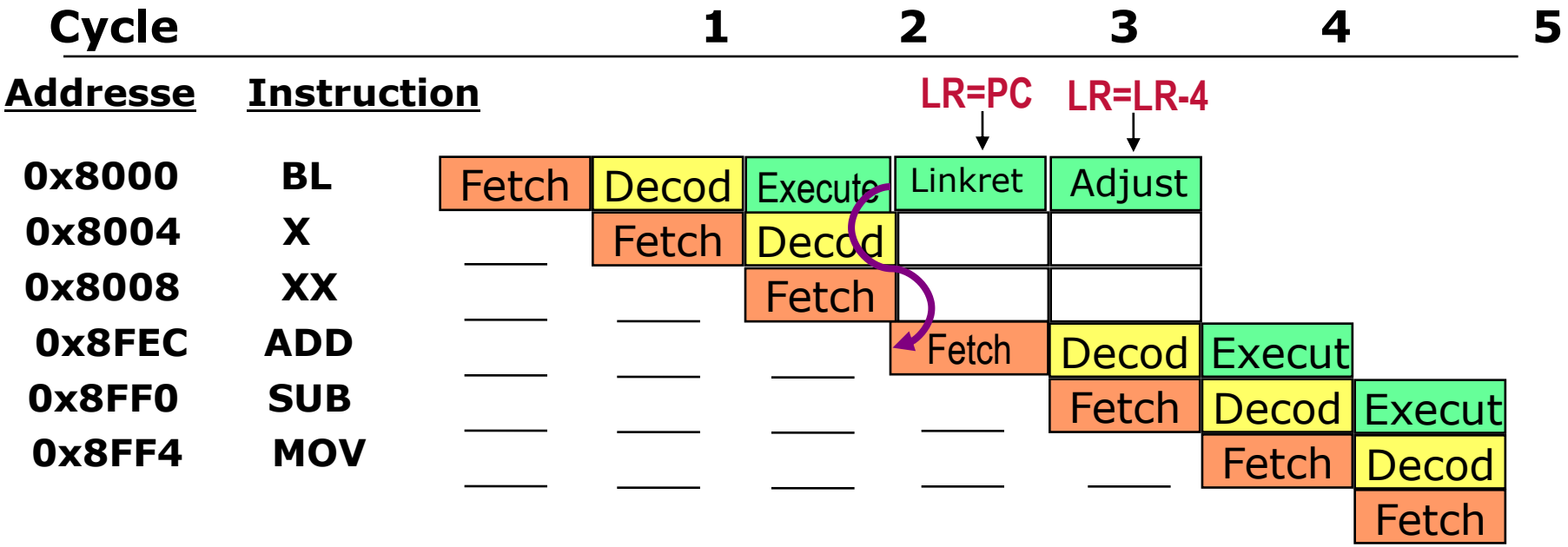
```
LDMFD   sp!,{r0,r2-r4}
; liste de registres=mem(sp+i
```

# 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



# Exemple: Pipeline avec Saut




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

# Exécutions conditionnelles

- La plupart des instructions peuvent être exécutées conditionnellement 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**

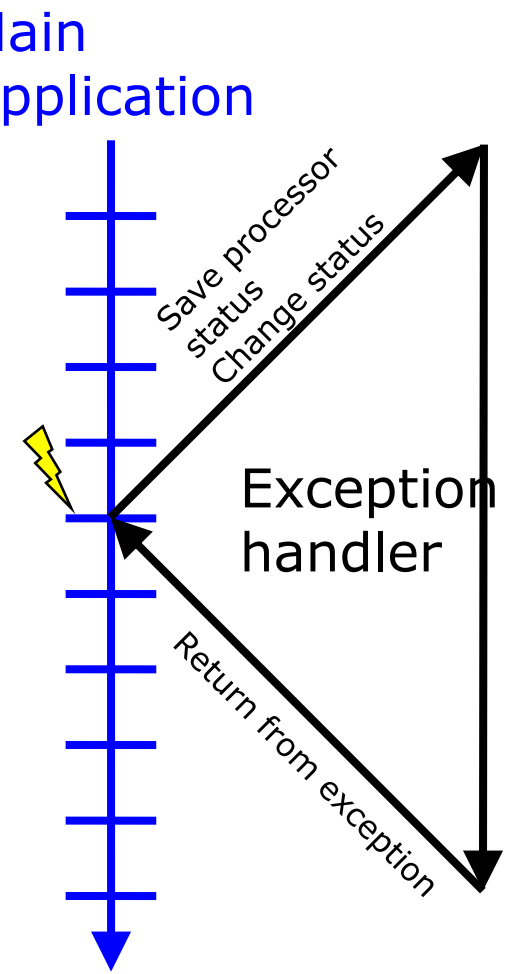
**=0, ≠0,≥0,<0,carry set, debordement**

# 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 | Le CSPR et le PC sont restaurés en même temps |
| 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 |

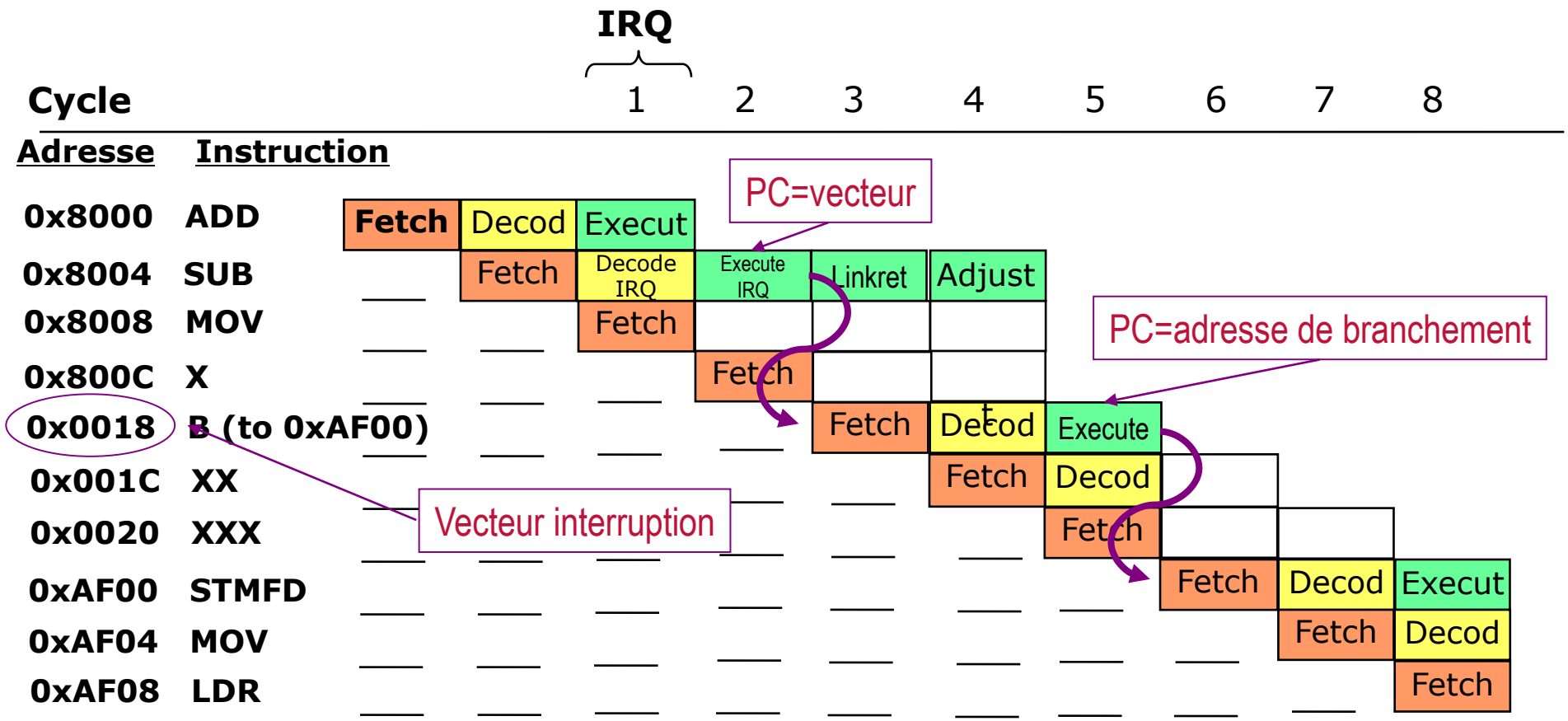
# Processus de gestion des Exceptions




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>`

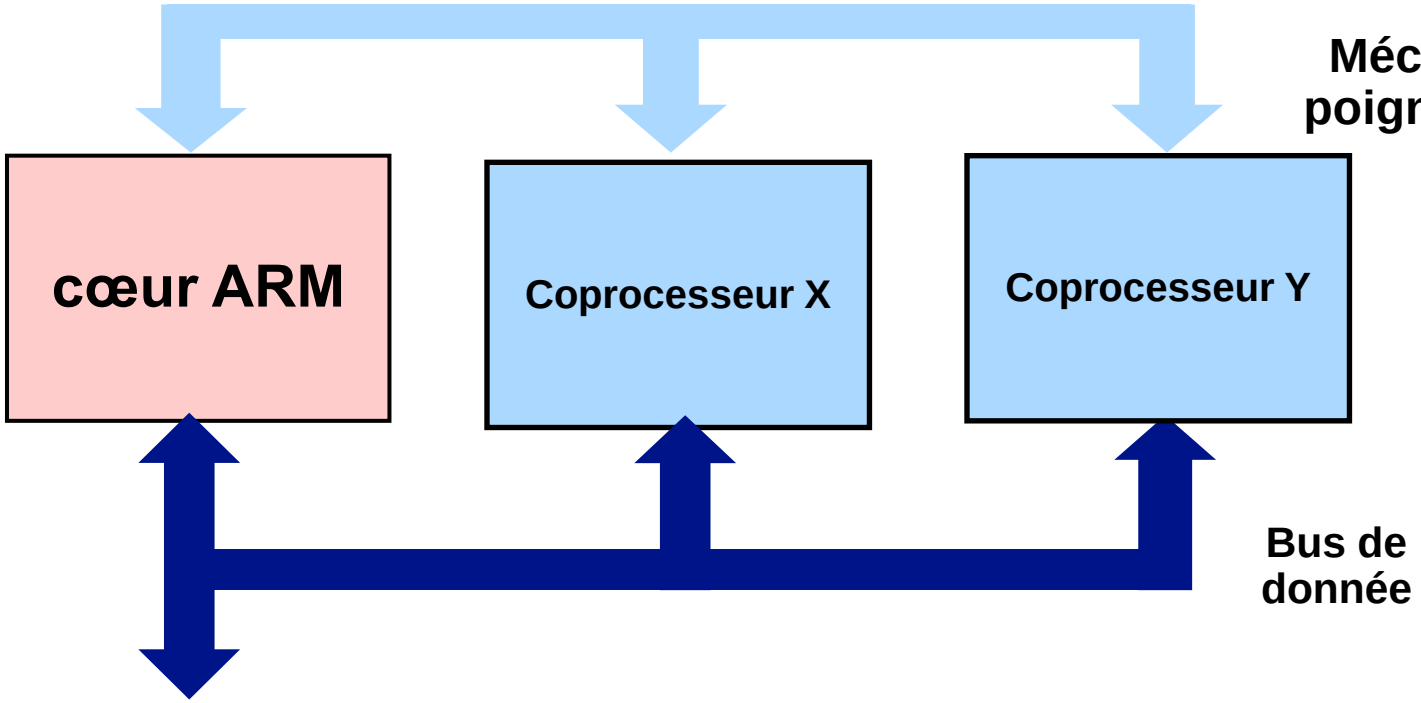
- ❑ **1 and 2 performed automatically by the core**
- ❑ **3 and 4 responsibility of software**

# Exemple: Pipeline avec Interruption




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

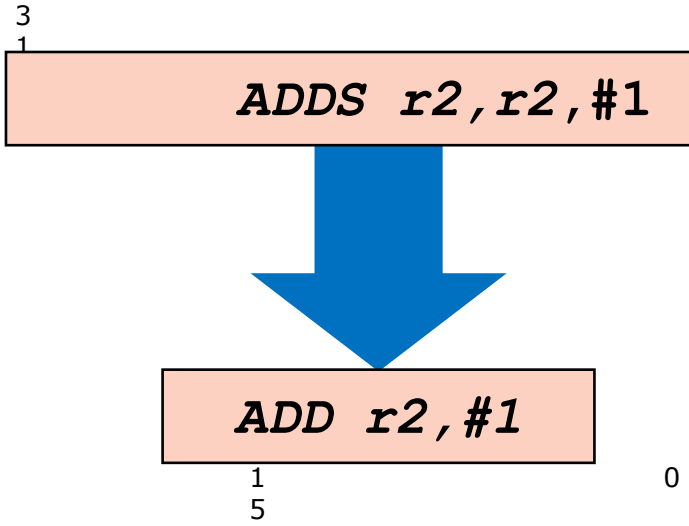
# Coprocesseurs




- 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 C (gain 30%)
  - Augmente les performances pour des espaces mémoires réduits
  - Sous ensemble des fonctionnalités du jeu d'instructions ARM
- **Passage du mode ARM à Thumb**
  - l'instruction ***BX***

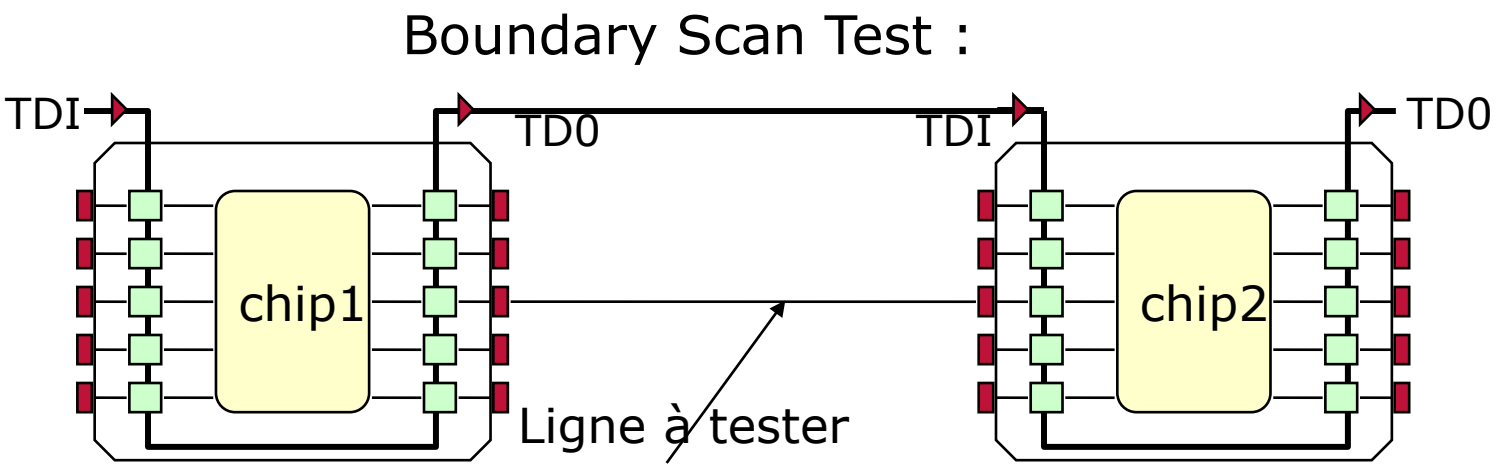



## Limitations

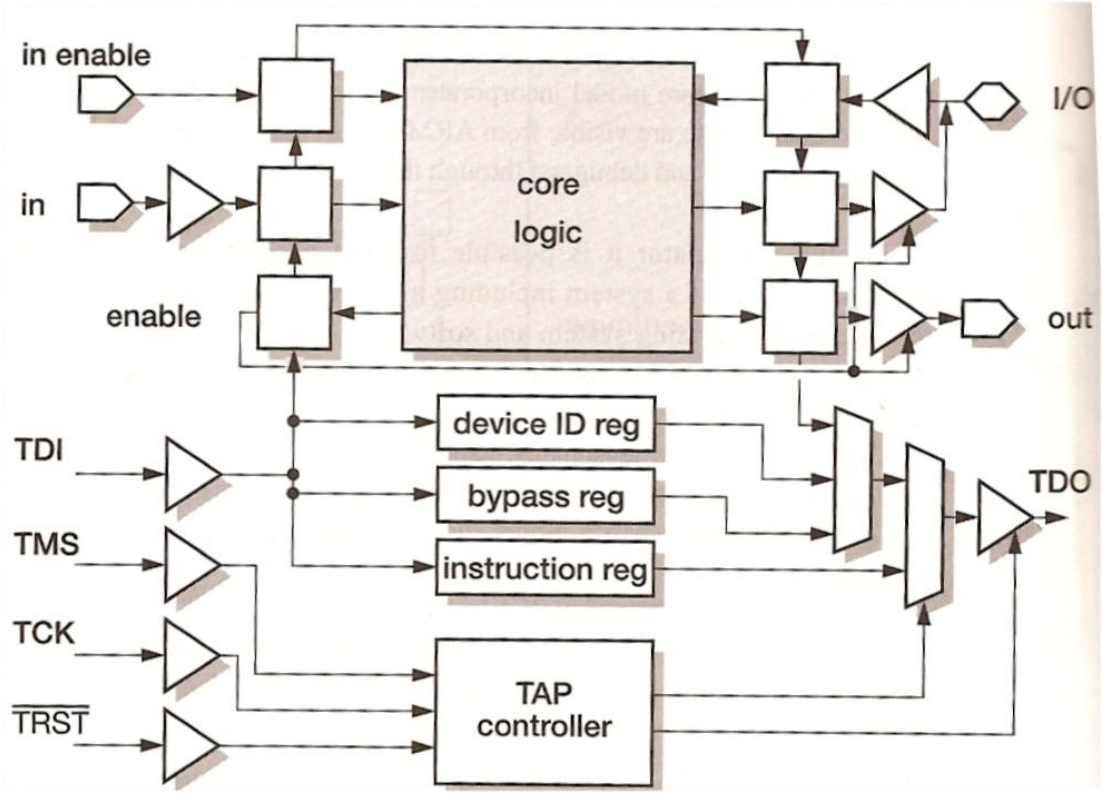
- Exécution conditionnelle pas utilisée
- Registres Source et Destination identiques (accumulateur)
- Seul les premiers registres sont utilisés
- Les constantes sont de taille limitée
- Pas de Barrel shifter dans une même instruction

# Port JTAG

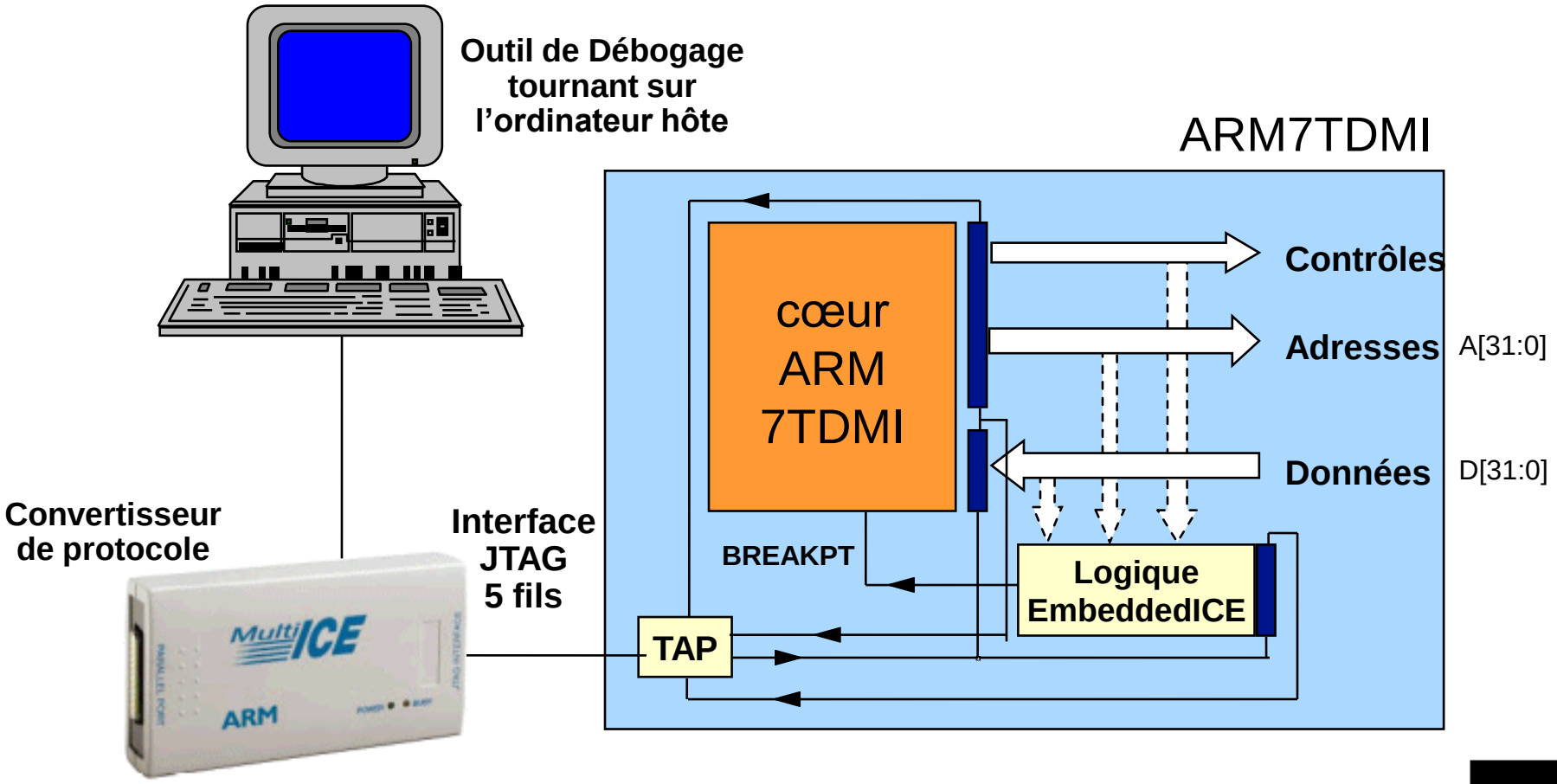
- **Contrôleur interne ou "TAP controller" pour**
  - Test de  la connectivité (par Boundary Scan testing)
  - Accès aux ressources internes des processeurs pour le débogage
  - Fonctions personnalisées



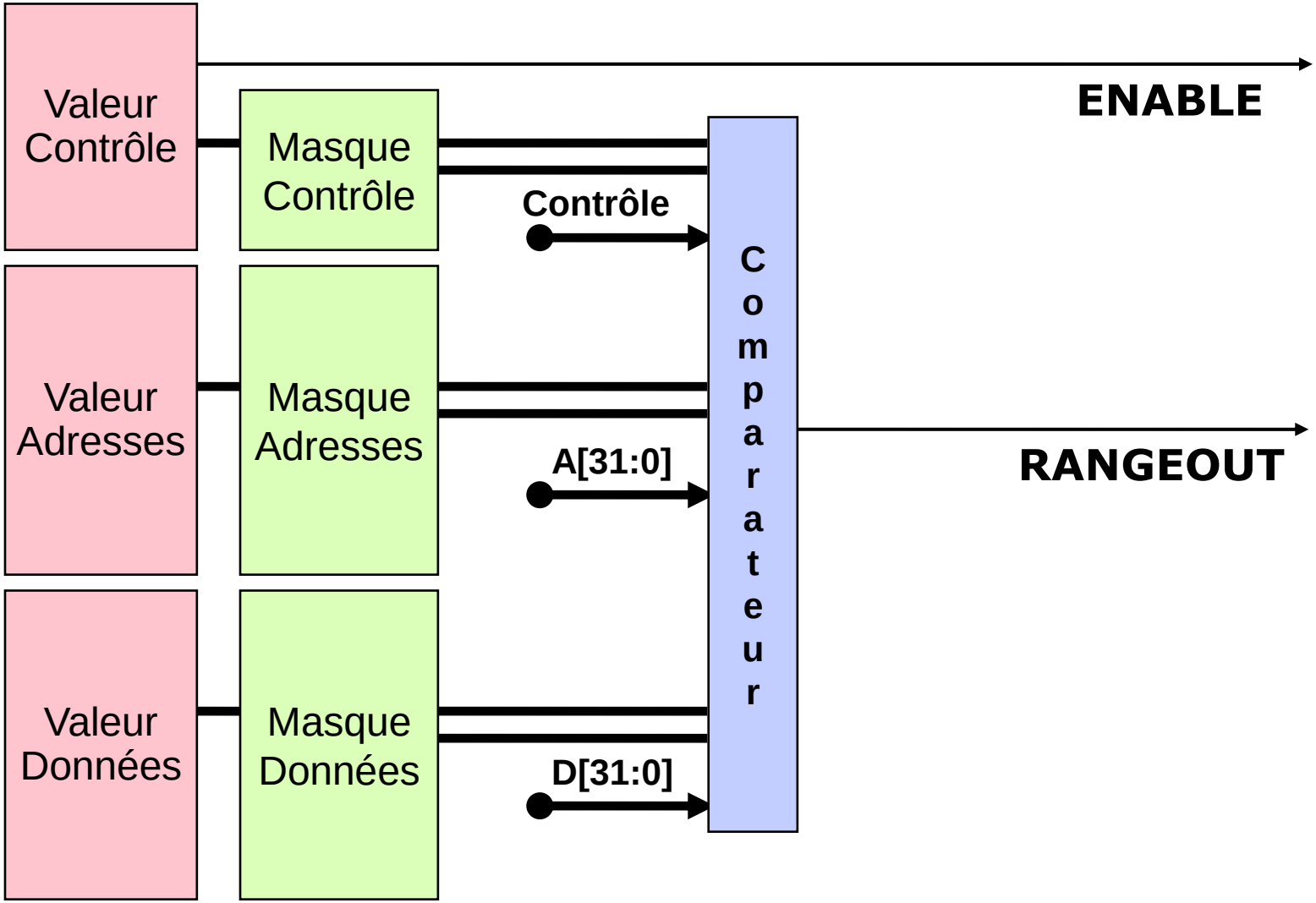
# JTAG architecture



# Débogage Embarqué



# Unité de Gestion des Points d'Arrêt



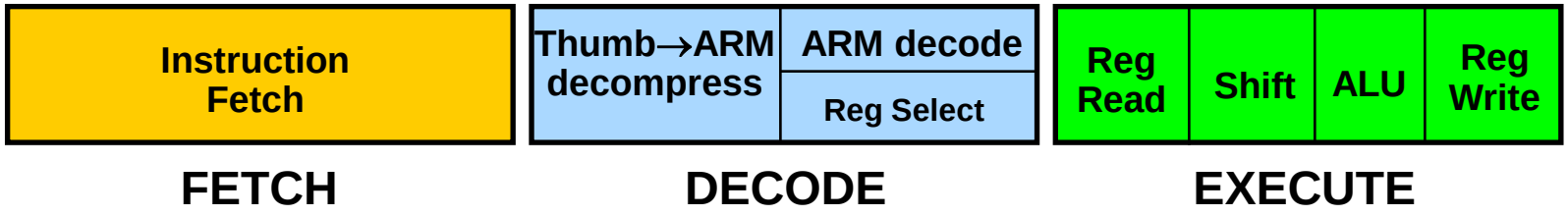
# ARM9TDMI

- **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

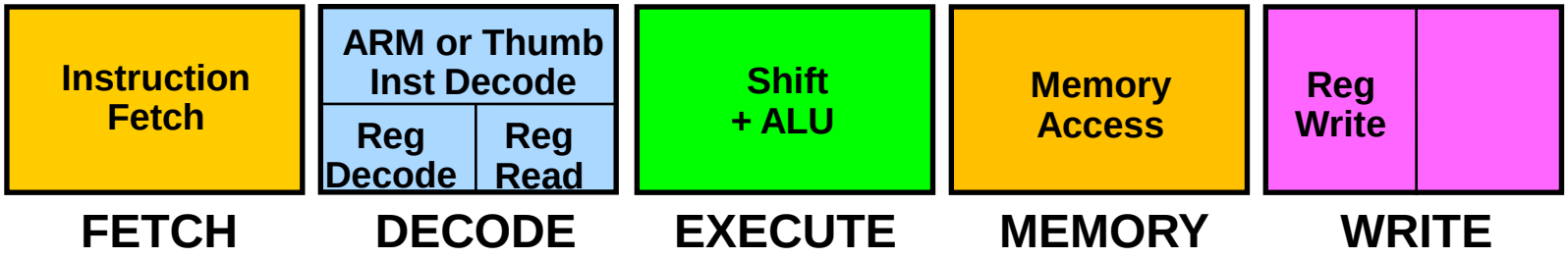
- **5 niveaux de pipeline**
  - => Modifications pour améliorer la fréquence maximum de l'horloge

# Modifications du Pipeline pour le ARM9TDMI

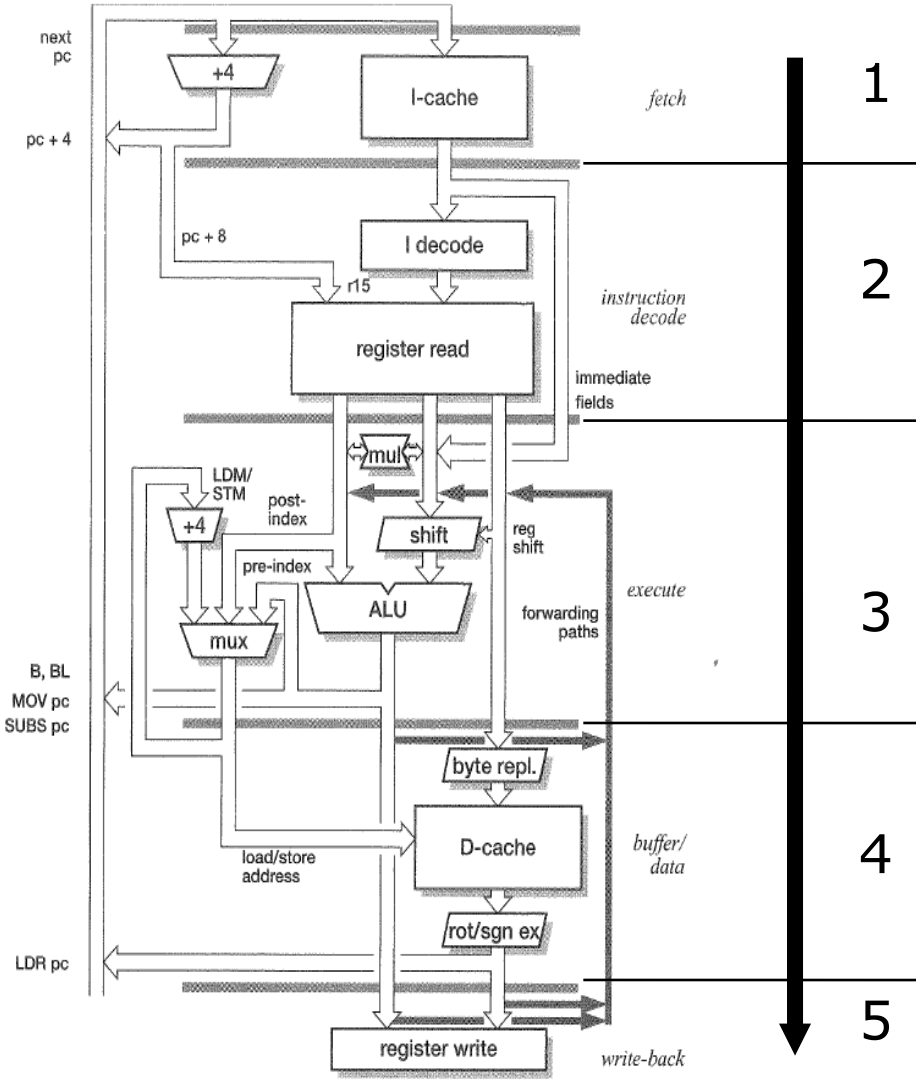
## Pipeline du ARM7TDMI




## Pipeline du ARM9TDMI



# ARM9TDMI architecture

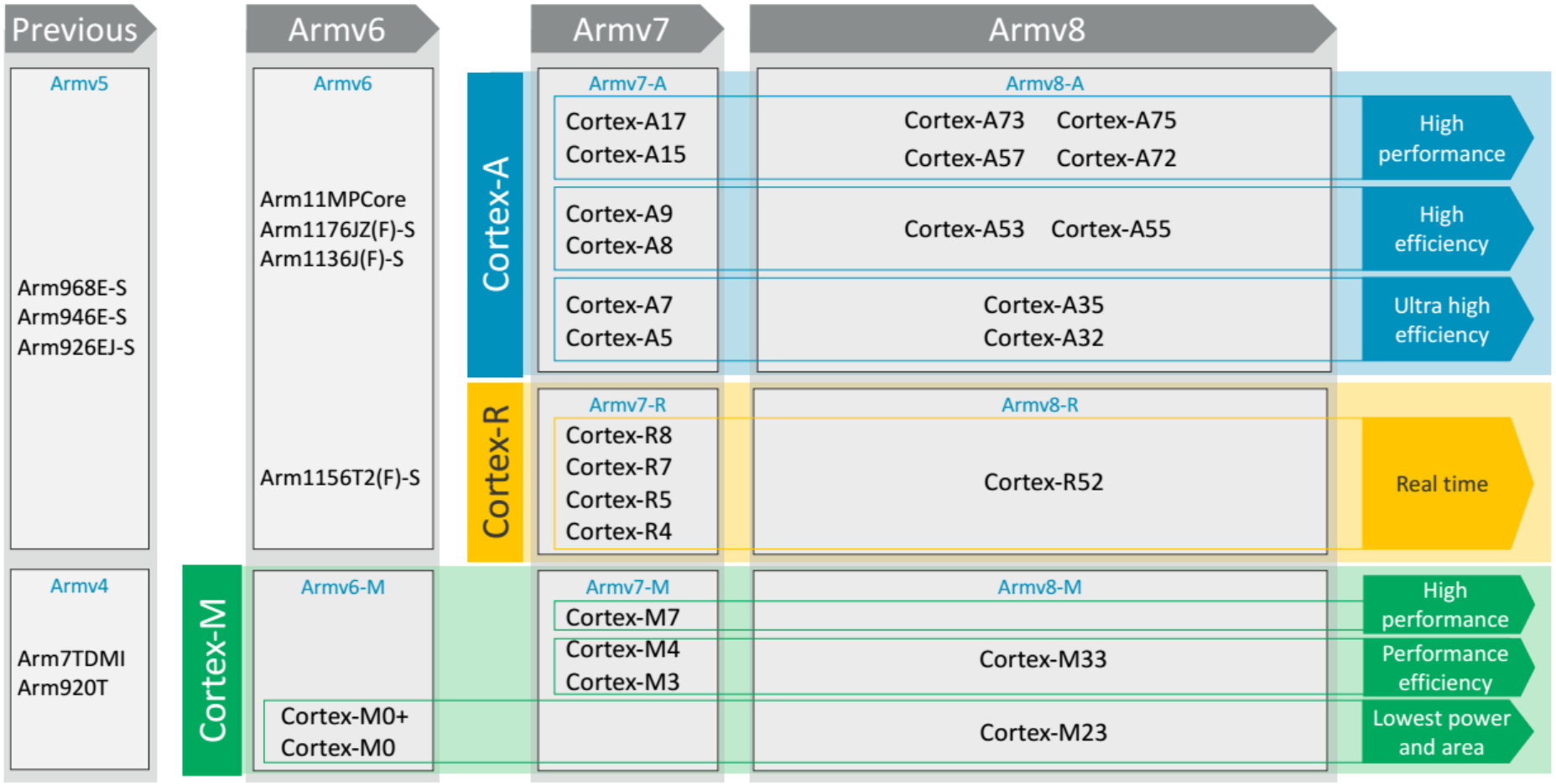



5 étages de pipeline

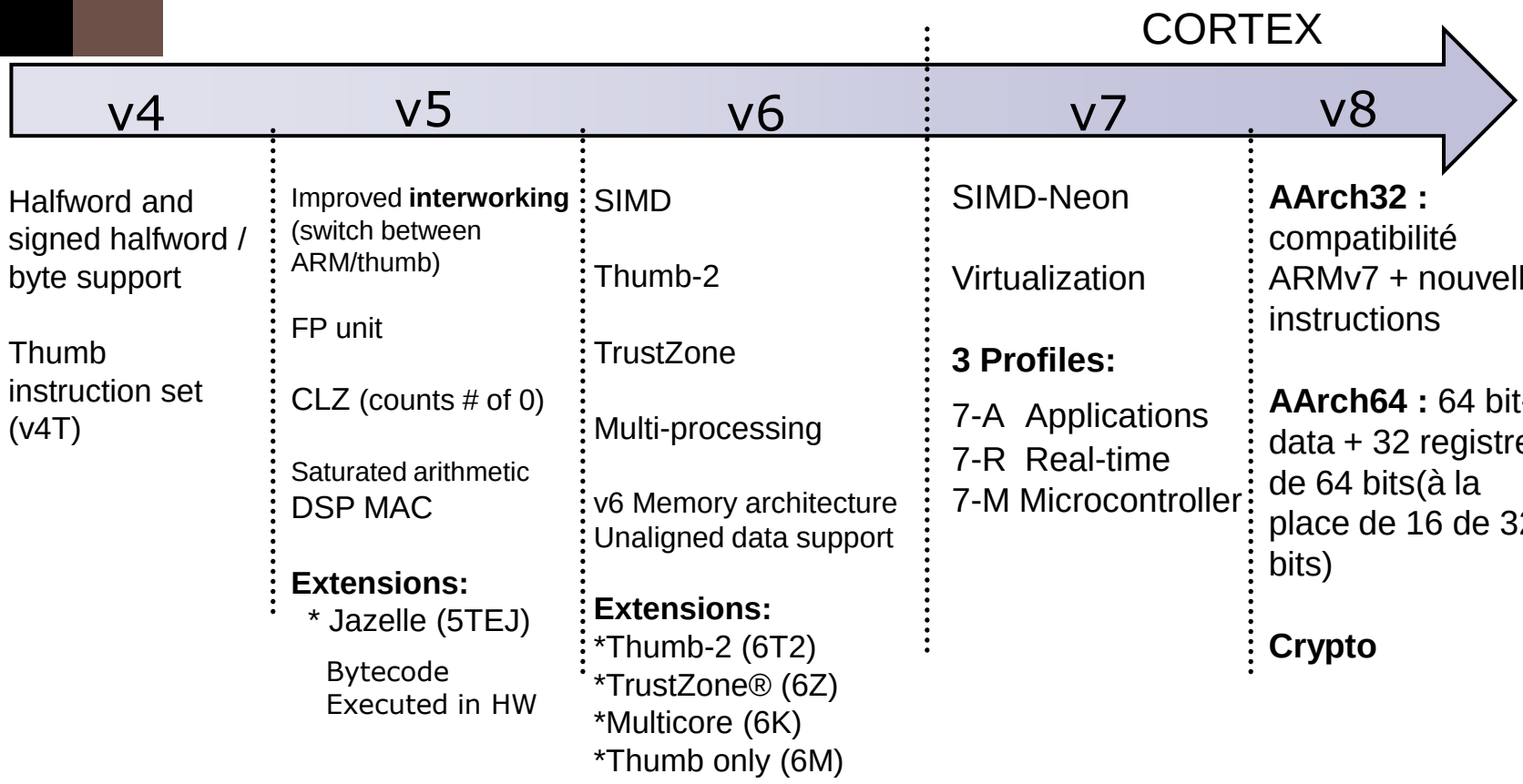
# Plan

- **ARM historique : l'ARM7TDMI**
- **Les Cortex**
  - Cortex M family
  - Cortex A family
- **System bus**

# La gamme des processeurs ARM

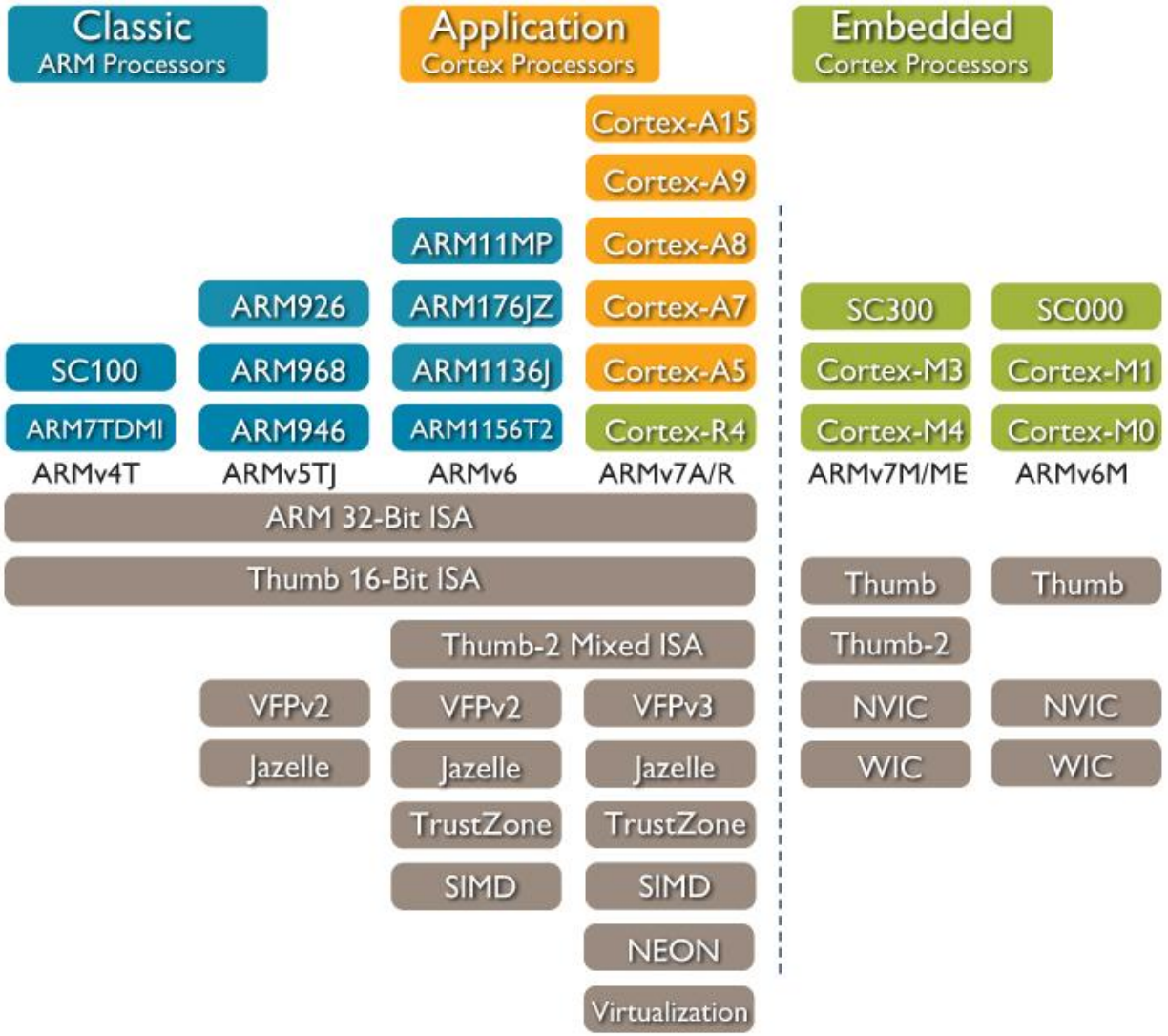


# Development of the ARM Architecture




- **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

# Which architecture is my processor?

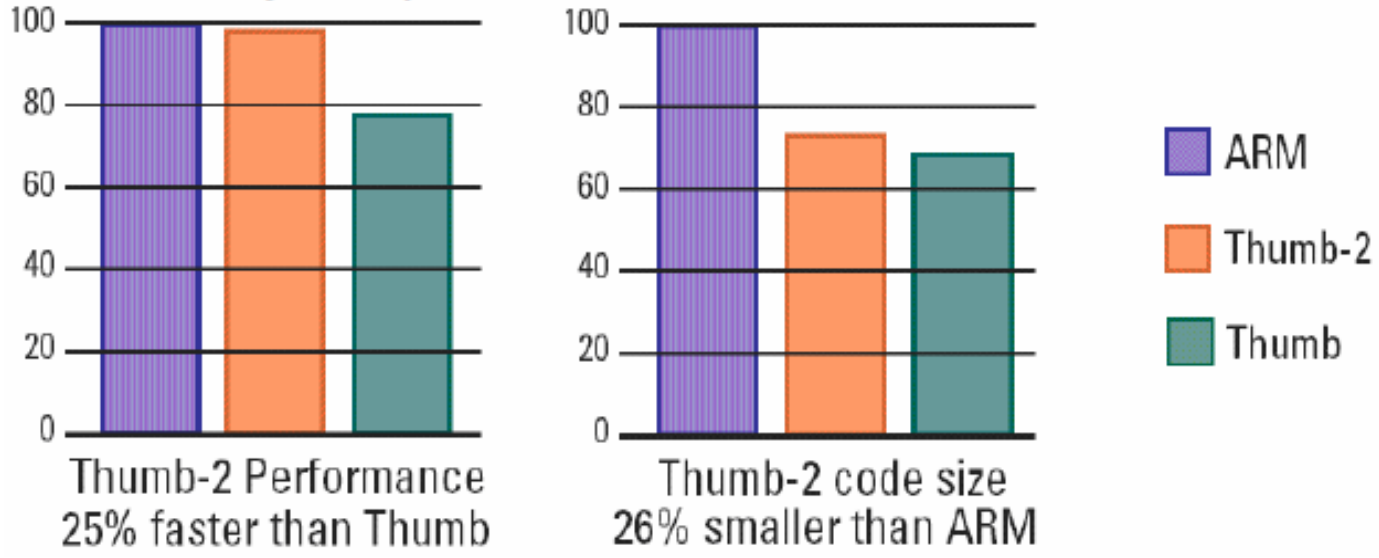


# Architecture ARMv7 profiles

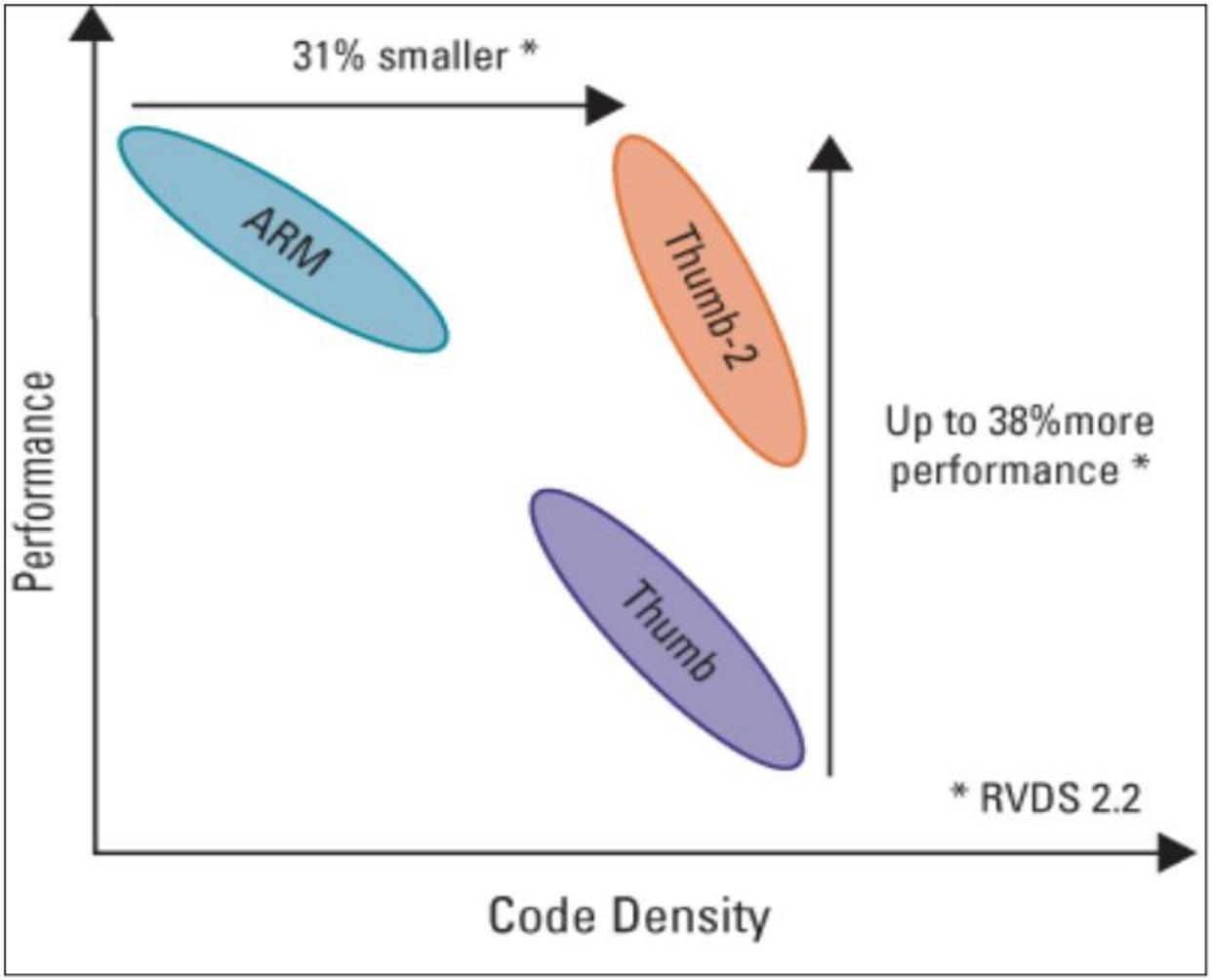
- **Application profile (ARMv7-A)**
  - Memory management support (MMU)
  - Highest performance
    - Influenced by multi-tasking OS system requirements
  - TrustZone: Insulation zones for security
  - Jazelle-RCT: supports for java JIT compilation
- **Real-time profile (ARMv7-R)**
  - Protected memory (MPU)
  - Low latency and predictability ‘real-time’ needs
- **Microcontroller profile (ARMv7-M, ARMv7E-M, ARMv6-M)**
  - Lowest gate count
  - Low energy

# Thumb vs Thumb2

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



# Perfomance/Code for instruction sets



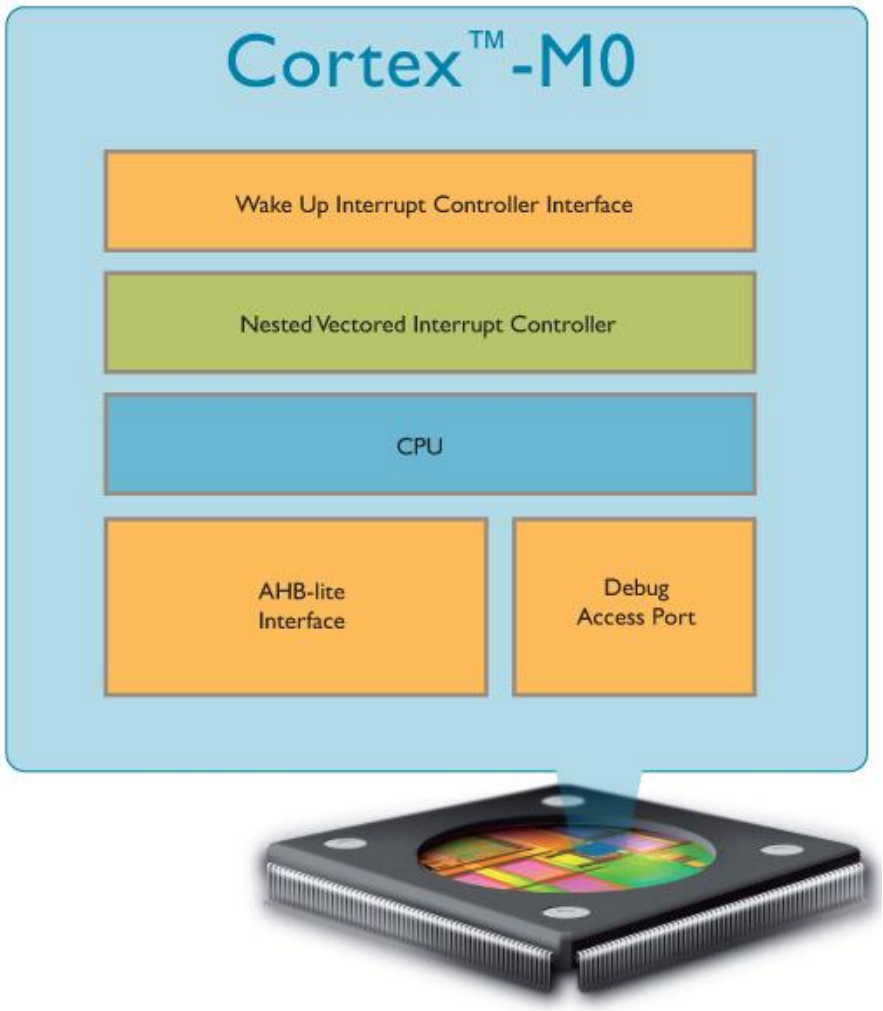
# Plan

- **ARM historique : l'ARM7TDMI**
- **Les Cortex**
  - Cortex M family
  - Cortex A family
- **System bus**

# ARMv7-M Profile Overview

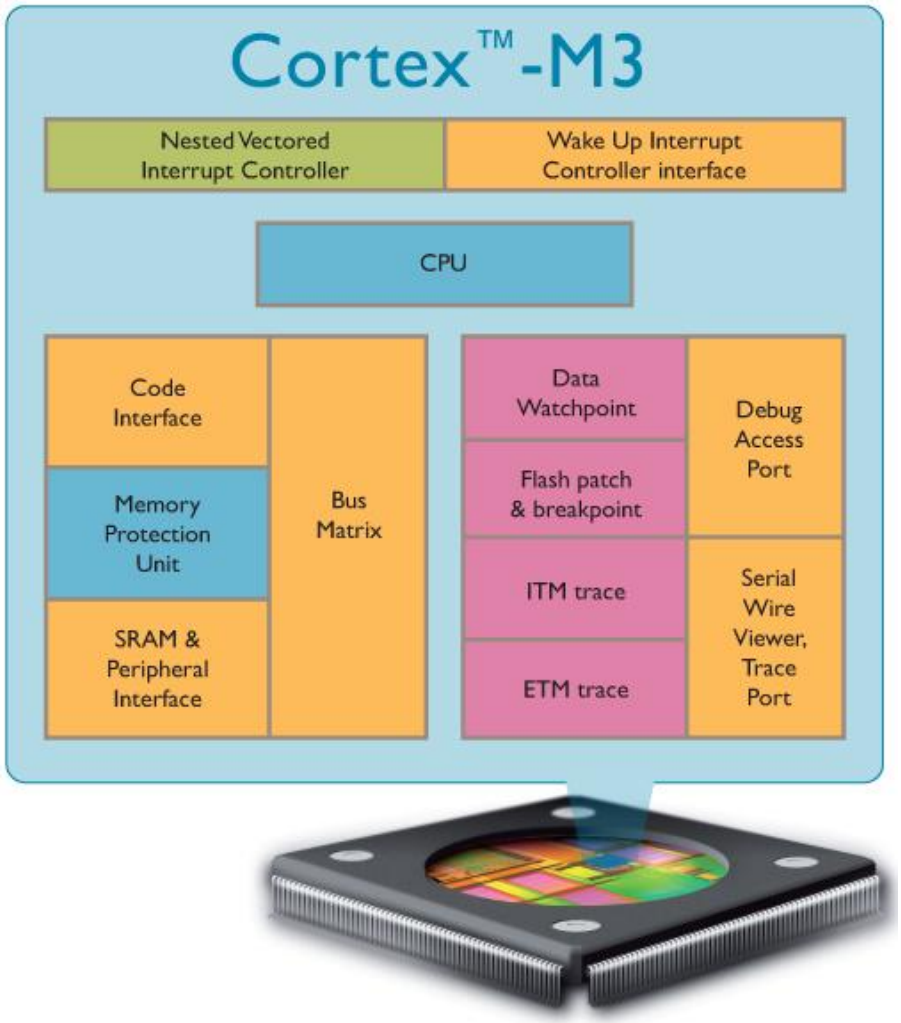
- **v7-M Cores targeted to the microcontroller market**
  - Simpler to program – entire application can be programmed in C
  - Fewer features needed than in application processors
- **Register and ISA changes from other ARM cores**
  - Only one set of registers
  - xPSR has different bits than CPSR
- **Different modes and exception models**
  - Only two modes: Thread mode and Handler mode
  - Vector table is a set of 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
  - Memory mapped control registers

# 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

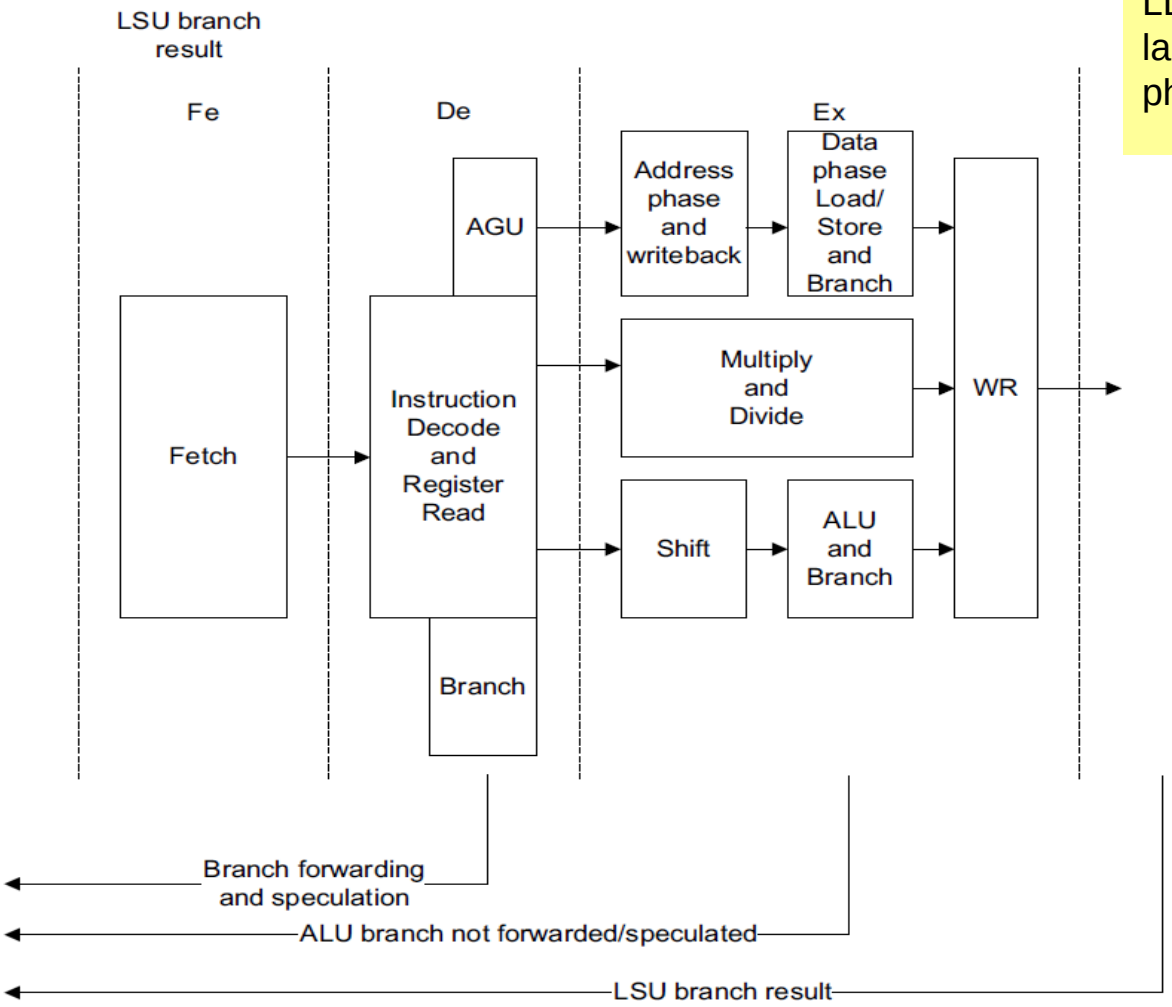
# Cortex-M3




- **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** (traces for debug)

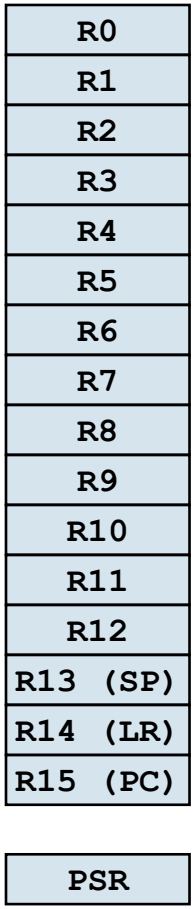
# Cortex M3 pipeline

LD/ST dans la même phase EXE




Branchement spéculatif => -1 cycle

# 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

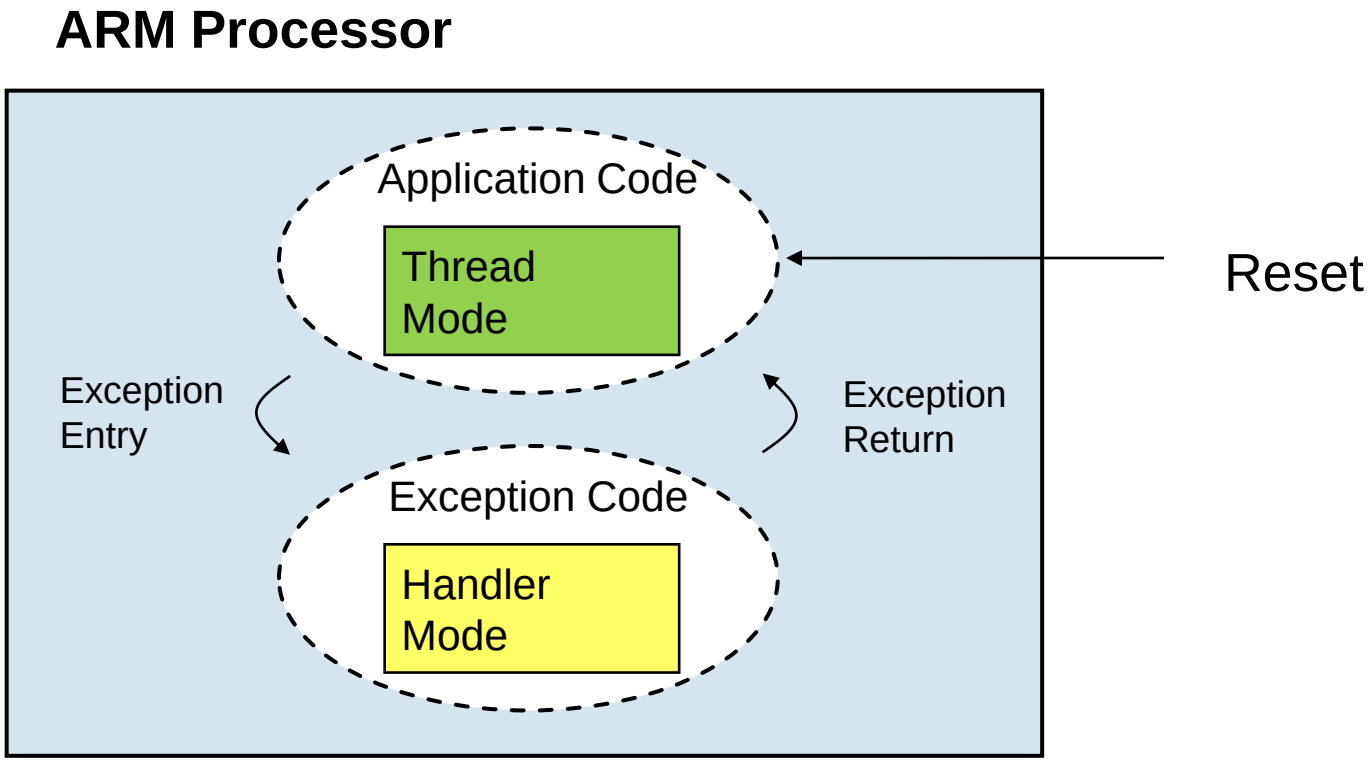
# Special Purpose Registers

- **Special Purpose Interrupt Mask Registers: PRIMASK, FAULTMASK, BASEPRI**
  - Used to modify exception priorities by special **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

# 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:

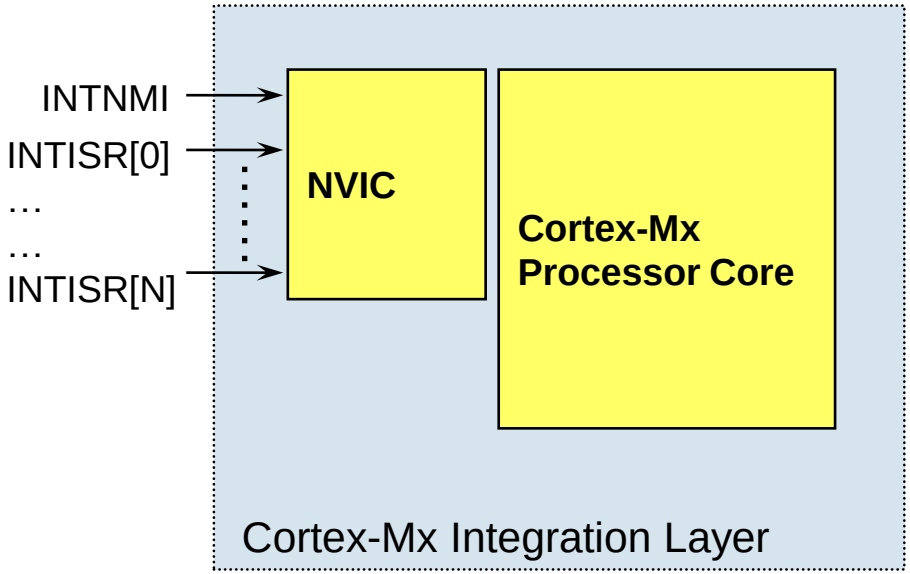
- **Data Processing:**
  - MOV r2, r5 ; r2 = r5
  - ADD r5, #0x24 ; r5 = r5 + 36
  - ADD r2, r3, r4, LSL #2 ; r2 = r3 + (r4 * 4)
  - LSL r2, #3 ; r2 = r2 * 8
  - MOVT r9, #0x1234 ; upper halfword of r9 = #0x1234
  - MLA r0, r1, r2, r3 ; r0 = (r1 * r2) + r3
- **Memory Access:**
  - STRB r2, [r10, r1] ; 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

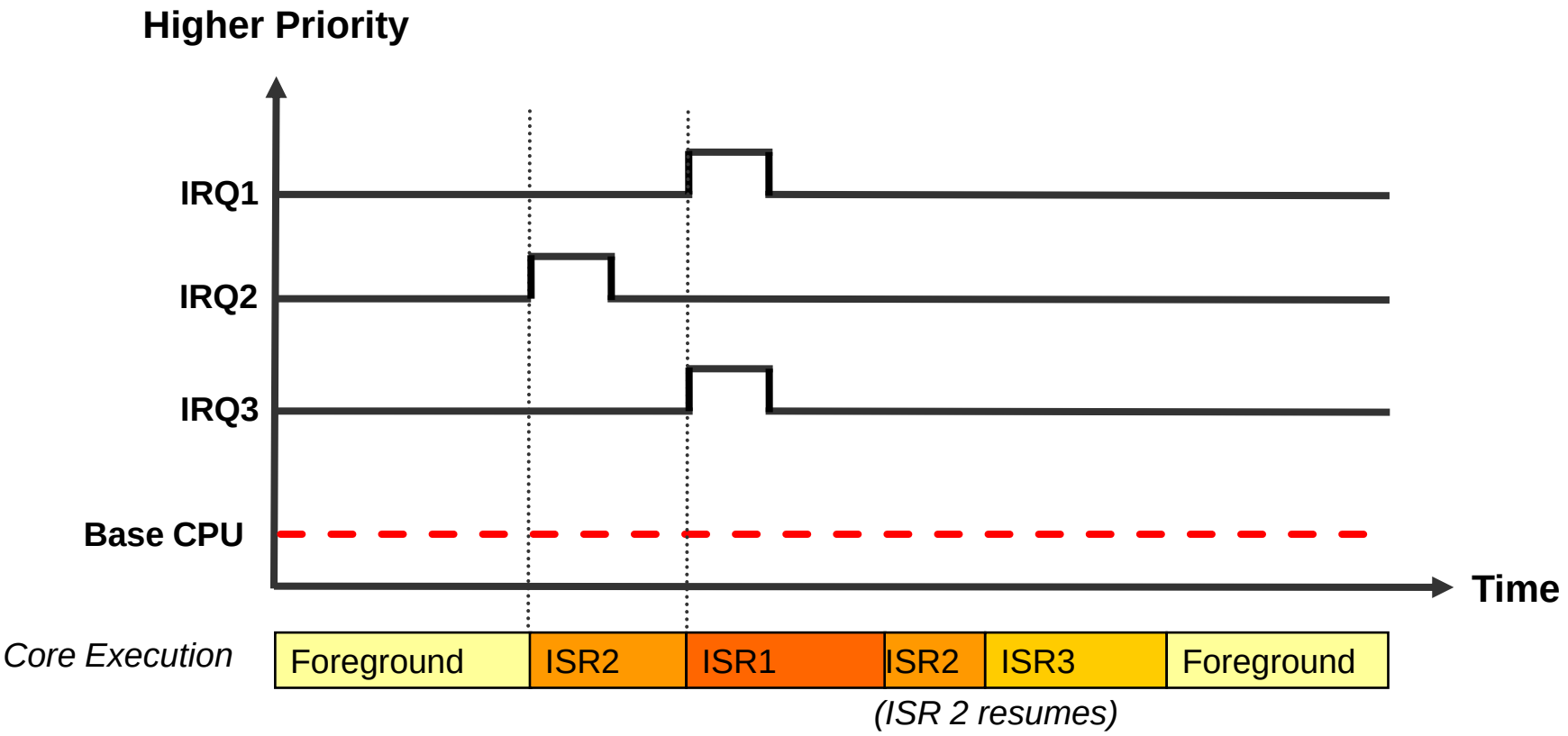
- **Exception main types:**
  - Reset
  - Non-maskable Interrupts (NMI)
  - Faults
  - SVCall
  - External Interrupt
  - SysTick Interrupt
- **Exceptions processed in Handler mode (except Reset)**
  - Exceptions always run privileged
- **Interrupt handling**
  - Interrupts are a sub-class of exception
  - **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

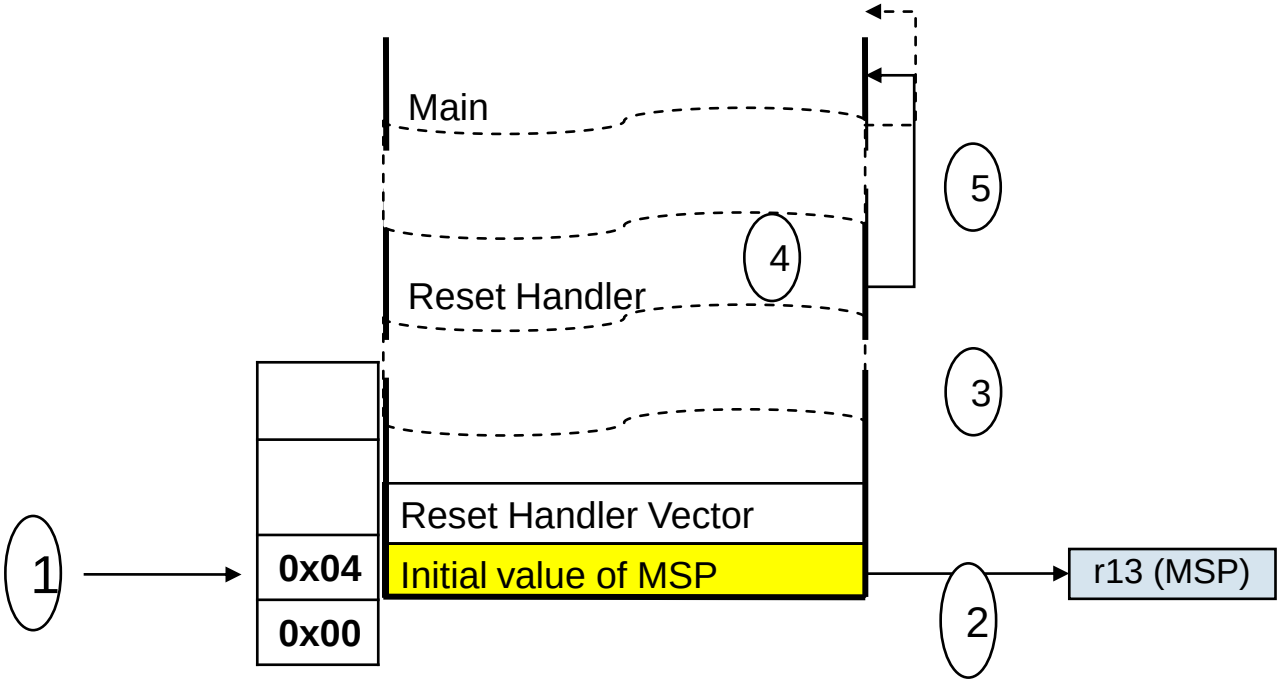


# Vector Table for ARMv7-M

- **First entry contains initial Main SP**
- **All other entries are addresses for exception handlers**
- **Table has up to 496 external interrupts**
- **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**
  - The compiler knows the SP at boot time
  - Example provided later

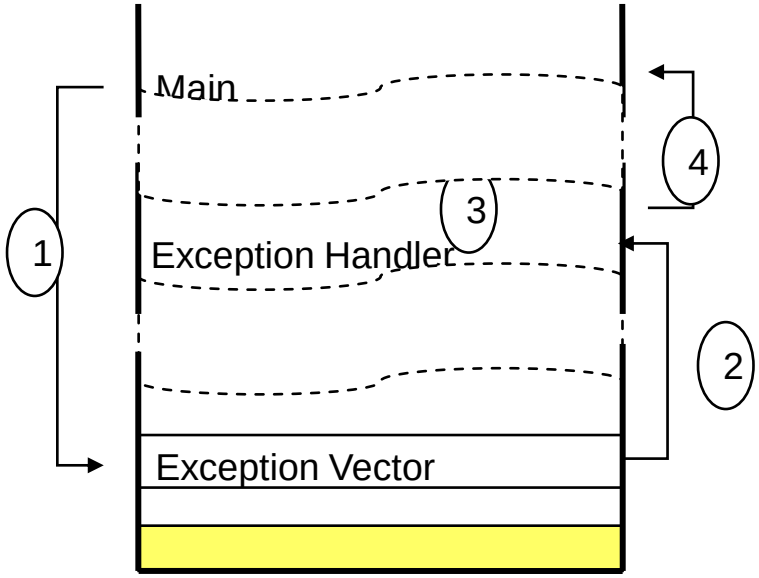
| Address | | Vector # |
|---|---|---|
| 0x40 + 4*N | External N | 16 + N |
| … | … | … |
| 0x40 | External 0 | 16 |
| 0x3C | SysTick | 15 |
| 0x38 | PendSV | 14 |
| 0x34 | Reserved | 13 |
| 0x30 | Debug Monitor | 12 |
| 0x2C | SVC | 11 |
| 0x1C to 0x28 | Reserved (x4) | 7-10 |
| 0x18 | Usage Fault |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

# 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

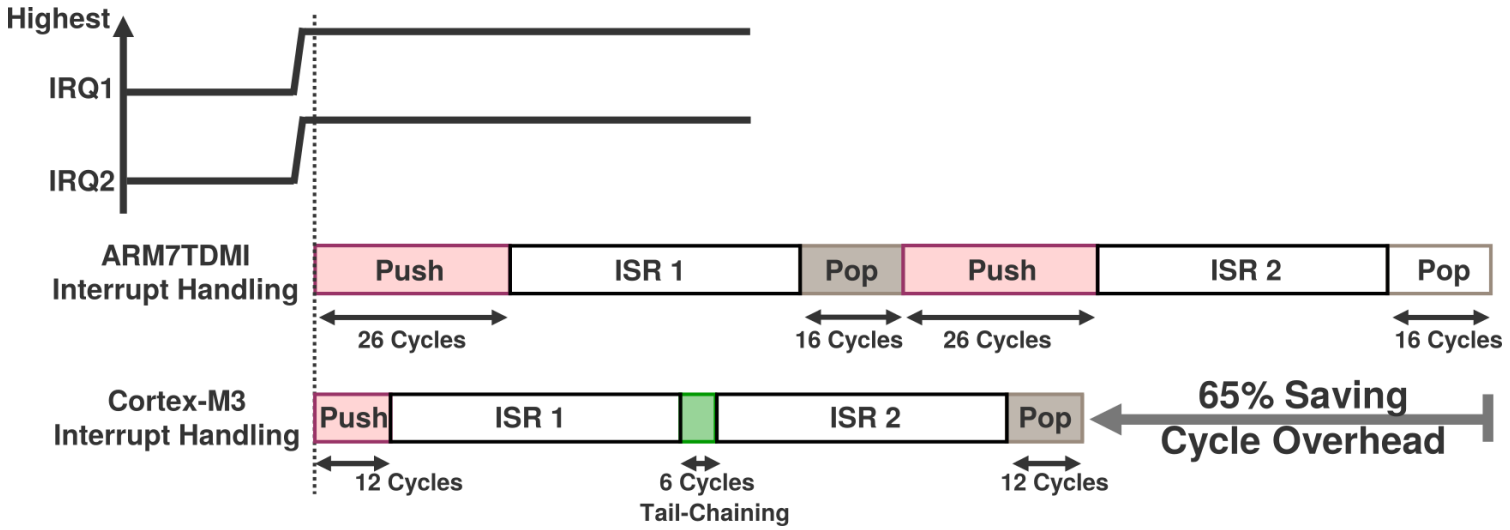



1. **Exception occurs**
   - Current instruction stream stops
   - Processor accesses vector table
2. **Vector address for the exception loaded from the vector table**
3. **Exception handler executes in Handler Mode**
4. **Exception handler returns to main**

# 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,
- **ISR executes from Handler mode with Main stack**

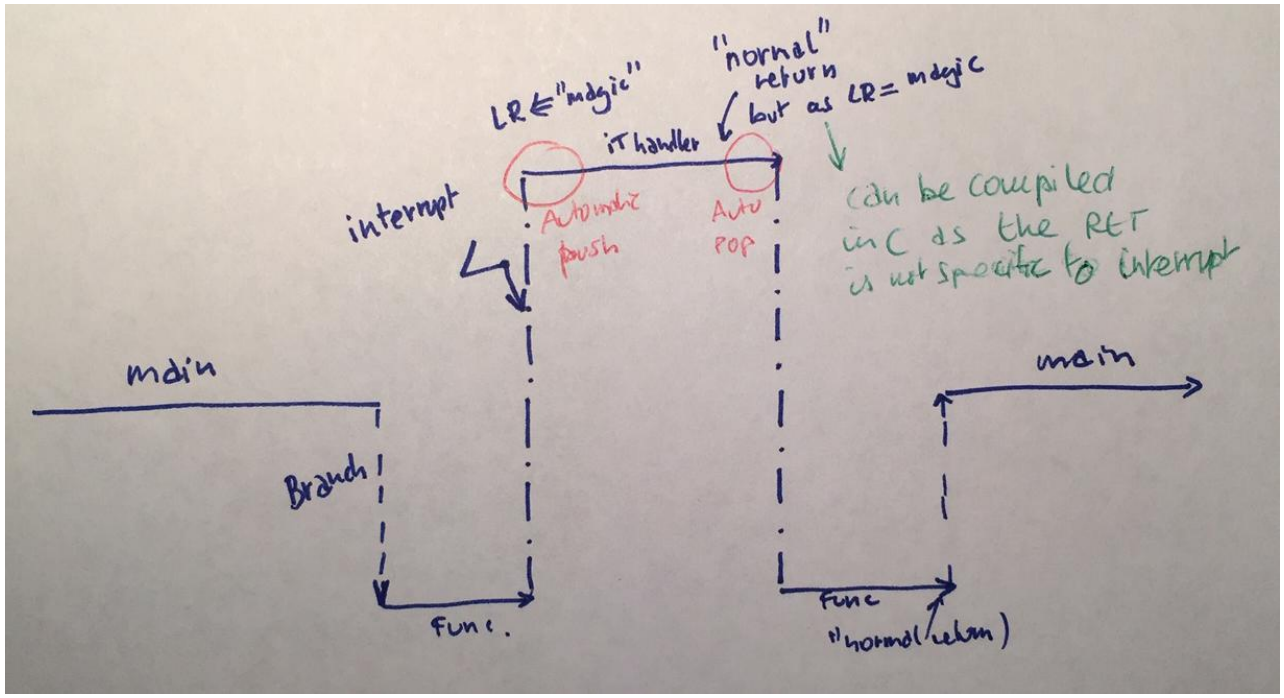
# Nested Vectored Interrupt Controller




| ARM7TDMI | Cortex-M3 |
|---|---|
| 26 cycles from IRQ1 to ISR1 (up to 42 cycles if in LSM)<br>42 cycles from ISR1 exit to ISR2 entry<br>16 cycles to return from ISR2 | 12 cycles from IRQ1 to ISR1 (Interruptible/Continual LSM)<br>6 cycles from ISR1 exit to ISR2 entry<br>12 cycles to return from ISR2 |

# 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)



# 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

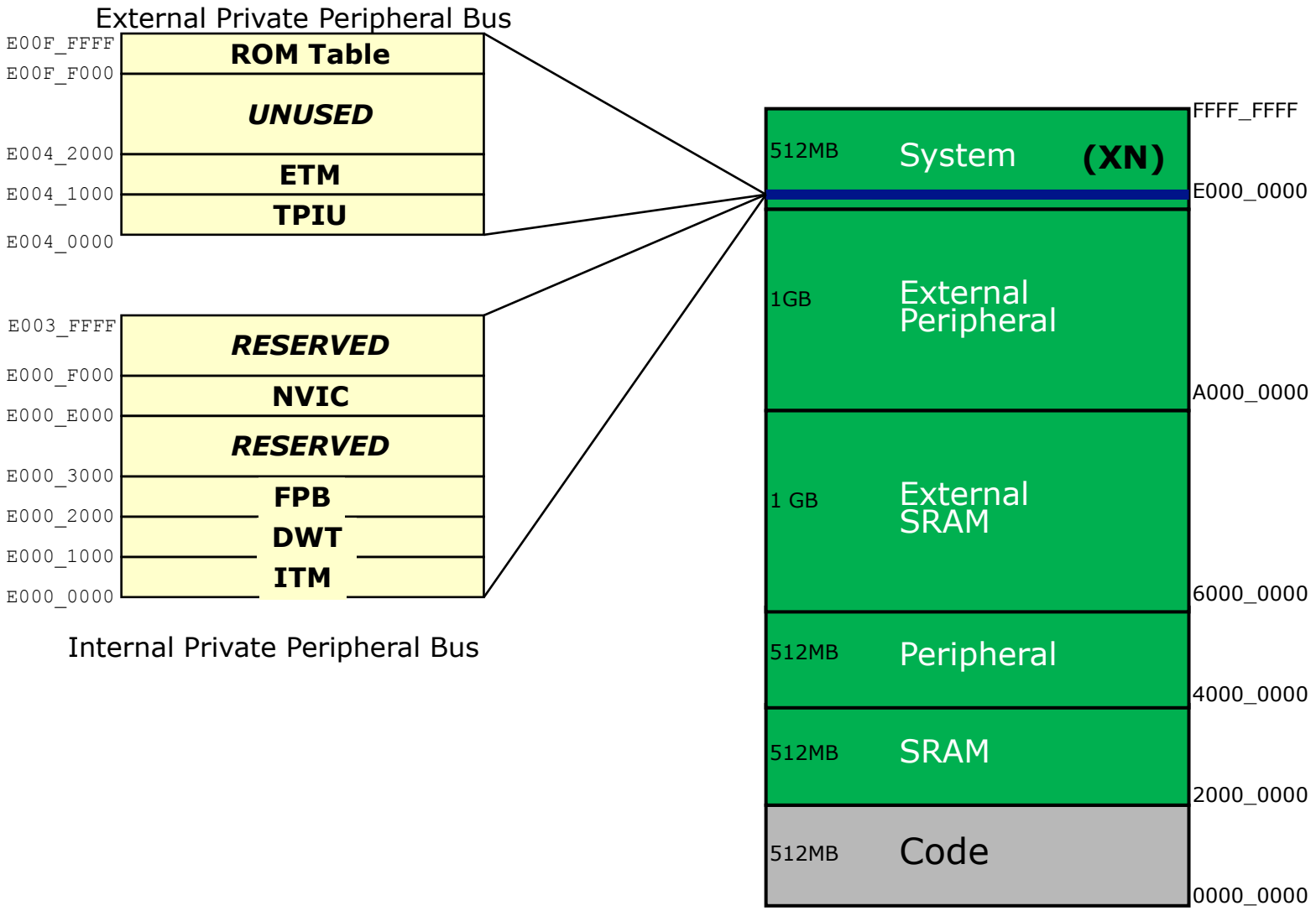
```
typedef void(* const ExecFuncPtr)(void) __irq;
#pragma arm section rodata="exceptions_area"
ExecFuncPtr exception_table[] = {
  (ExecFuncPtr)&Image$$ARM_LIB_STACK$$ZI$$Limit,    /* Initial SP */
  ExecFuncPtr)__main,                               /* Initial PC */
  NMIException,
  HardFaultException,
  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

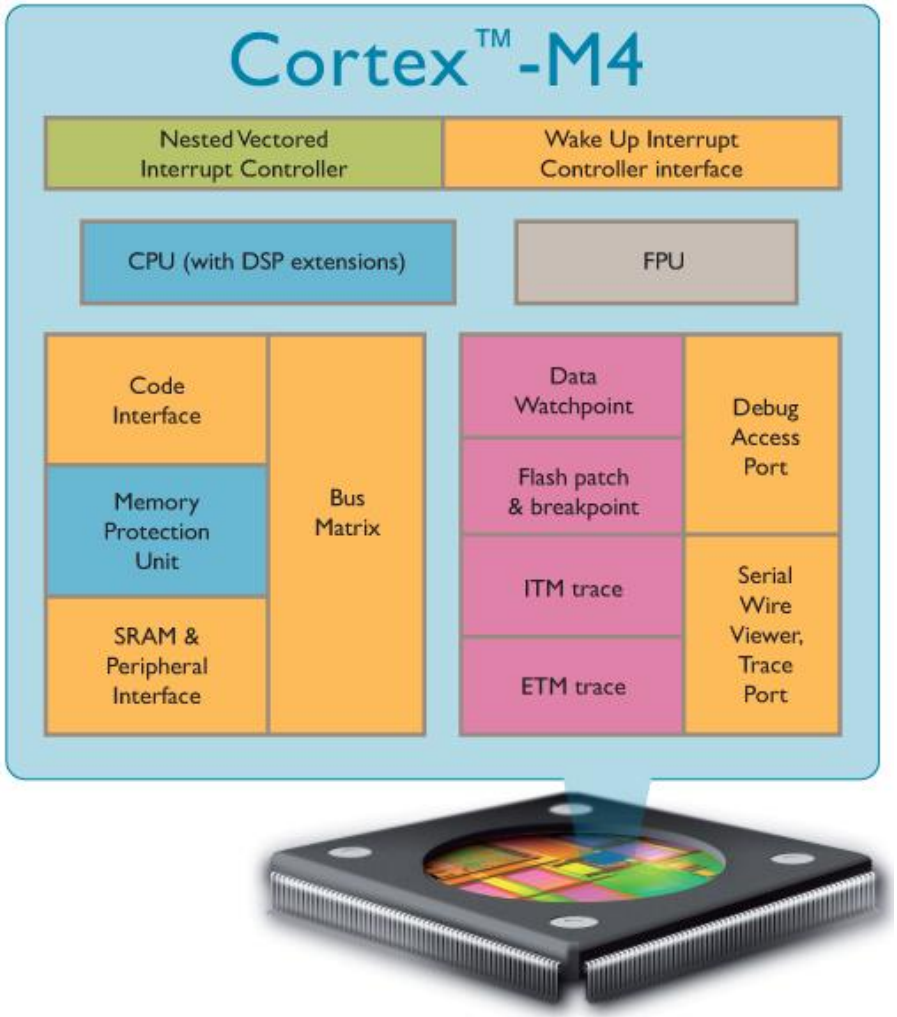
The SVCall ISR location must be populated if the SVC instruction will be used

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

# Processor Memory Map



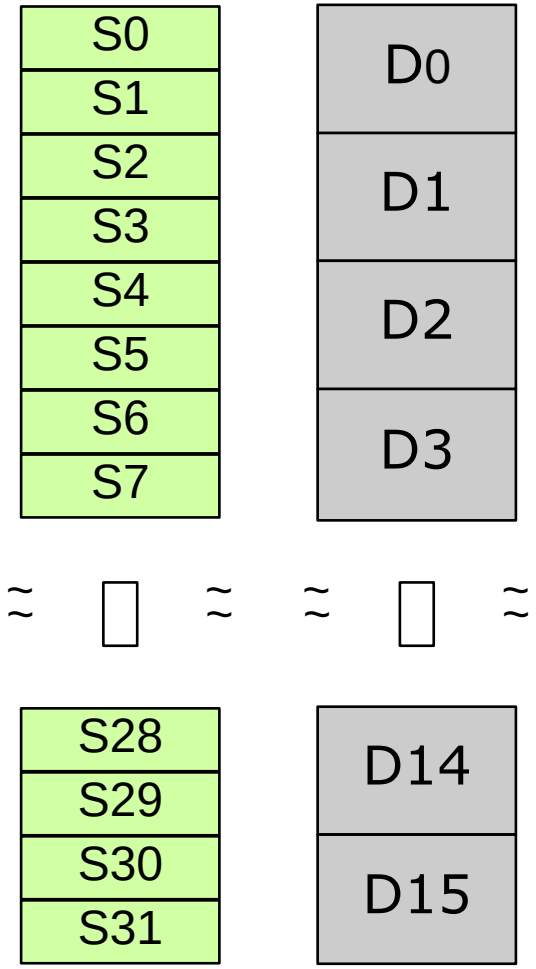
# Cortex-M4



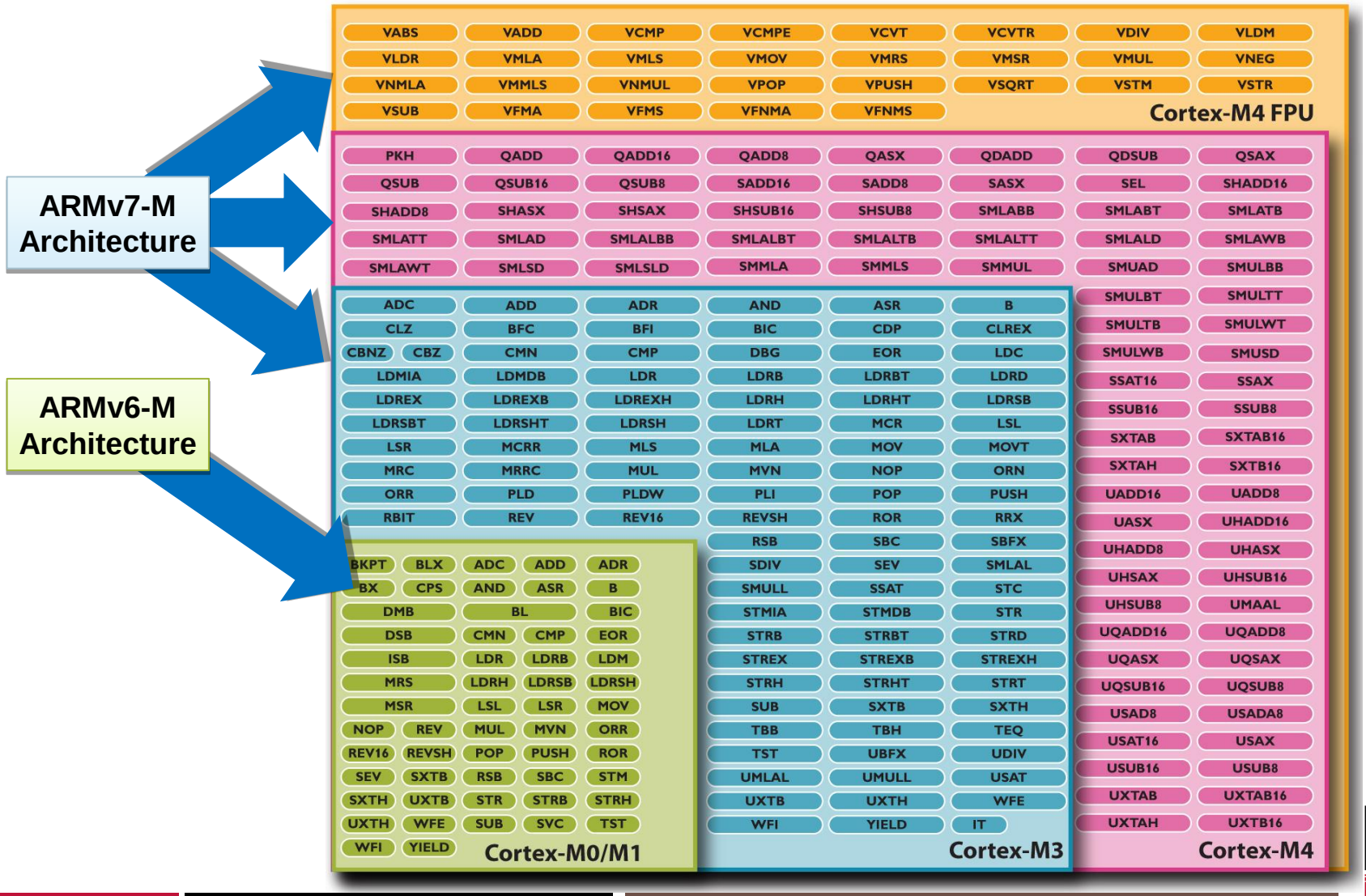

- **ARMv7E-M Architecture**
  - Thumb-2 only
  - DSP extensions
- **Optional FPU (Cortex-M4F)**
- **Otherwise, same as Cortex-M3**
- **Implements full Thumb-2 instruction set**
  - Saturated math (e.g. QADD)
  - Packing and unpacking (e.g. UXTB)
  - Signed multiply (e.g. SMULTB)
  - SIMD (e.g. ADD8)

# 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



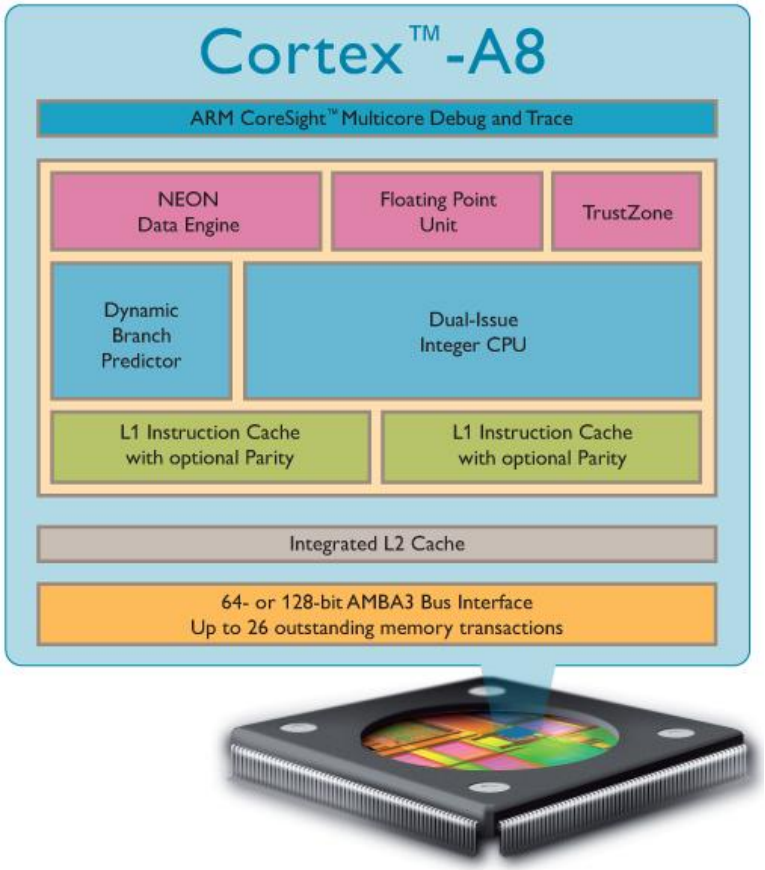
# Binary Upwards Compatibility



# Plan

- **ARM historique : l'ARM7TDMI**
- **Les Cortex**
  - Cortex M family
  - Cortex A family
- **System bus**

# Cortex-A8

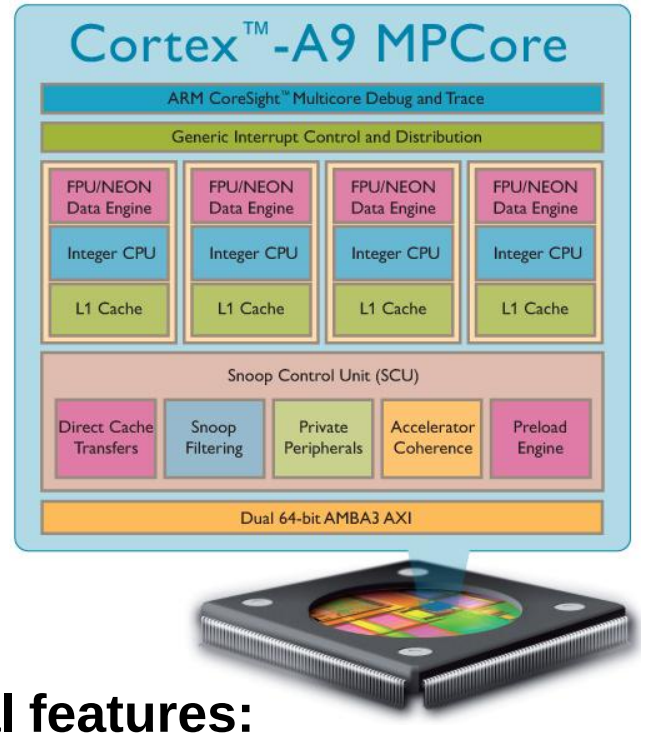



- **ARMv7-A Architecture**
  - Thumb-2
  - Thumb-2EE (Jazelle-RCT)
  - TrustZone extensions
- **MMU**
- **64-bit or 128-bit AXI Interface**
- **L1 caches**
  - 16 or 32KB each
- **Unified L2 cache**
  - 0-2MB in size
  - 8-way set-associative

- **Dual-issue, super-scalar 13-stage pipeline**
  - Branch Prediction & Return Stack
  - NEON and VFP implemented at end of pipeline
- **Optional features**
  - VFPv3 Vector Floating-Point
  - NEON media processing engin

# Cortex-A9

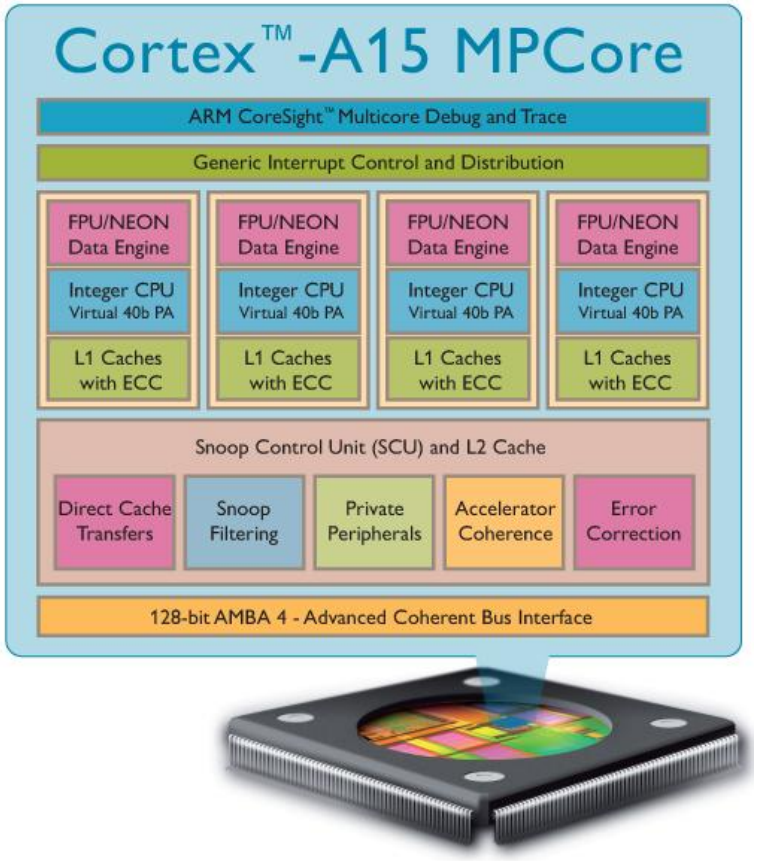
- **ARMv7-A Architecture**
  - Thumb-2, Thumb-2EE
  - TrustZone support
- **Variable-length Multi-issue pipeline**
  - Register renaming
  - Speculative data prefetching
  - Branch Prediction & Return Stack
- **64-bit AXI instruction and data interfaces**
- **TrustZone extensions**
- **L1 Data and Instruction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




- **Optional features:**
  - PTM instruction to Trace execution
  - 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

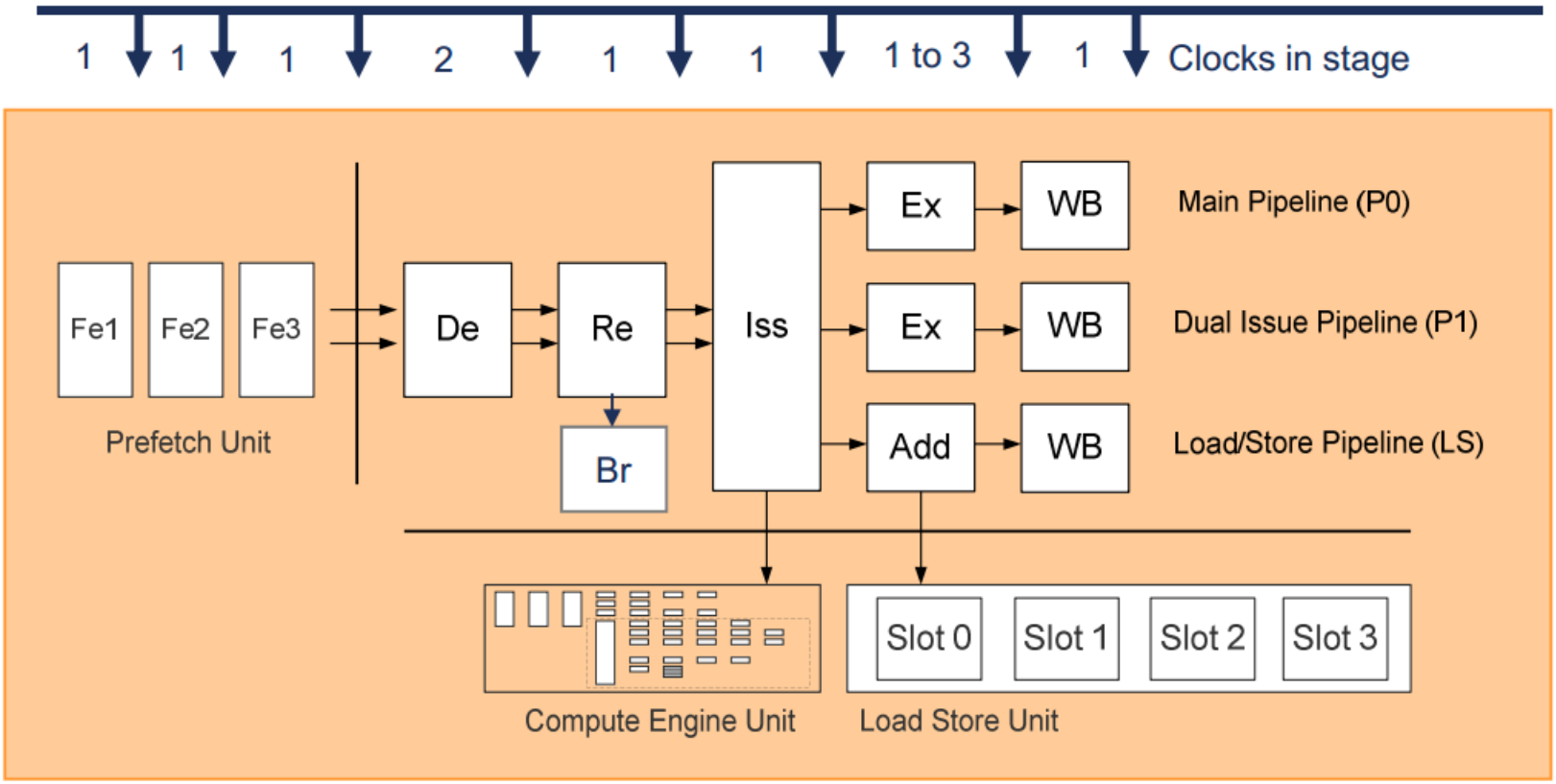



- Large Physical Address Extensions (LPAE) to ARMv7-A Architecture

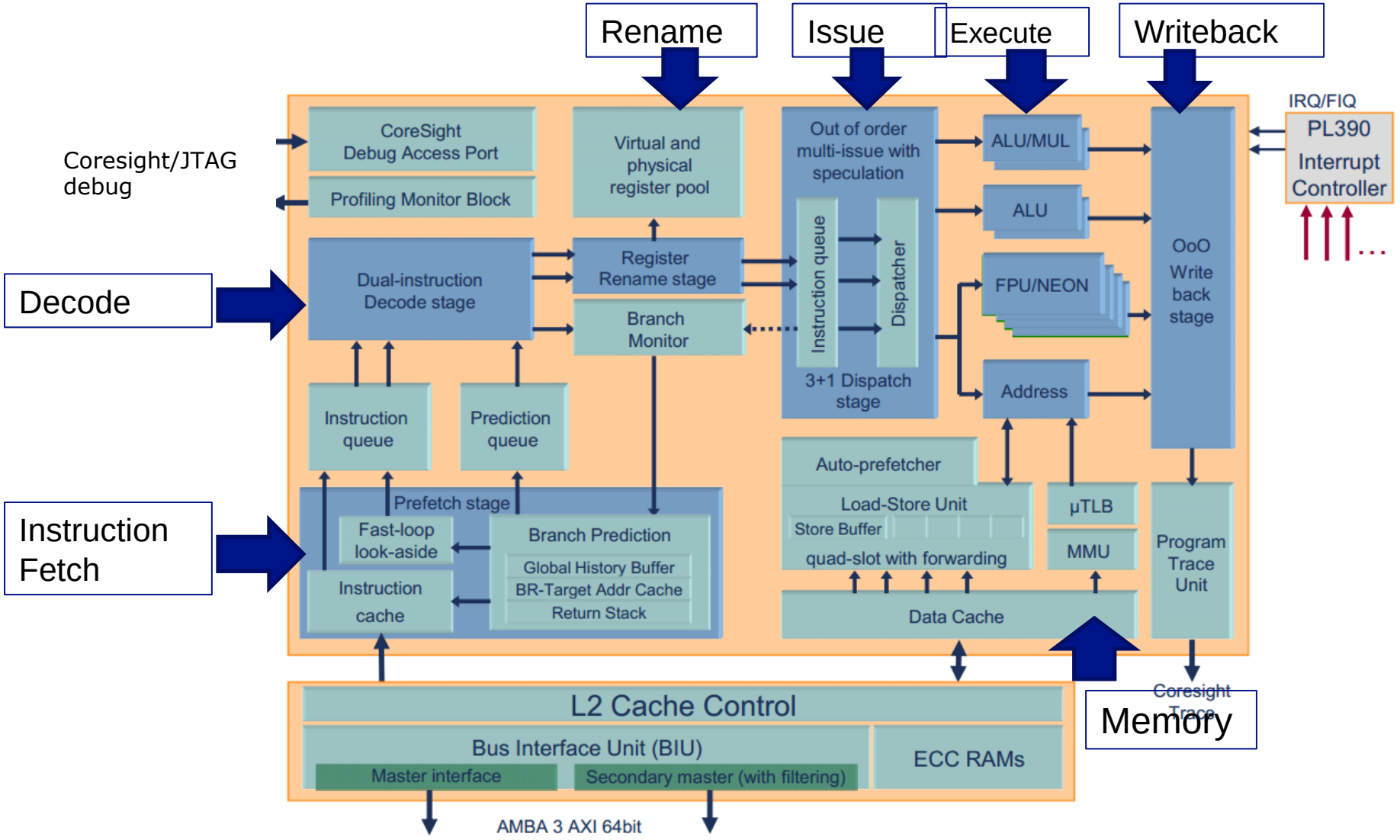
# Cortex A9 Microarchitecture Overview

- **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



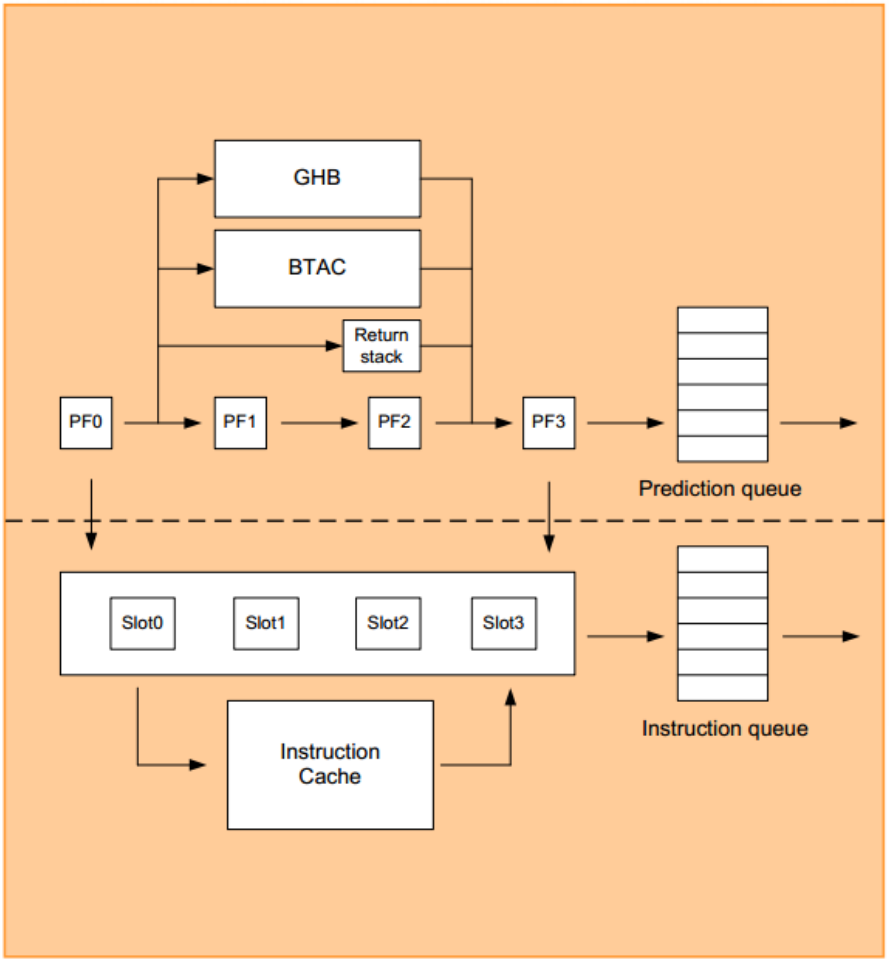
# CortexA9 Microarchitecture



# Instruction Fetch

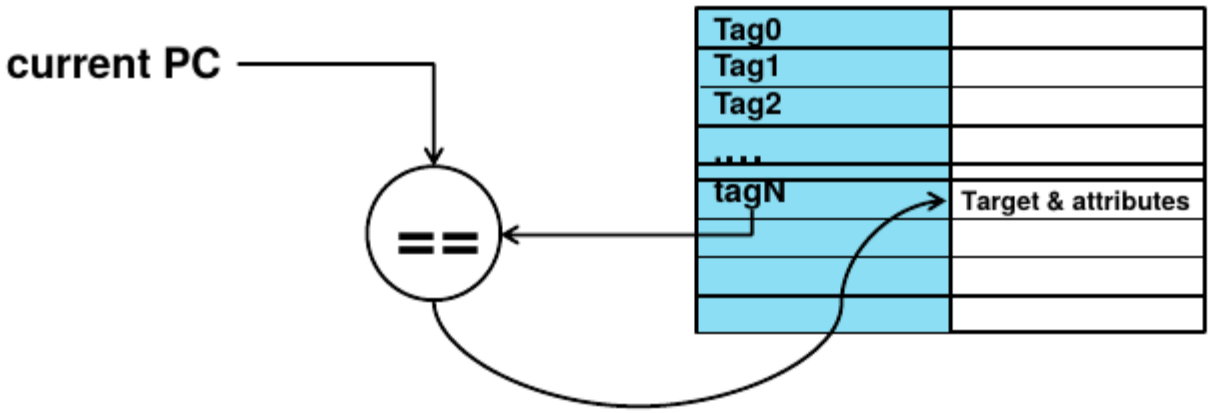

- Instruction cache size:
  - 16KB, 32KB, or 64KB
- Superscalar pipeline:
  - fetching two instructions at once
- Dynamic Branch Prediction

# Branch Prediction



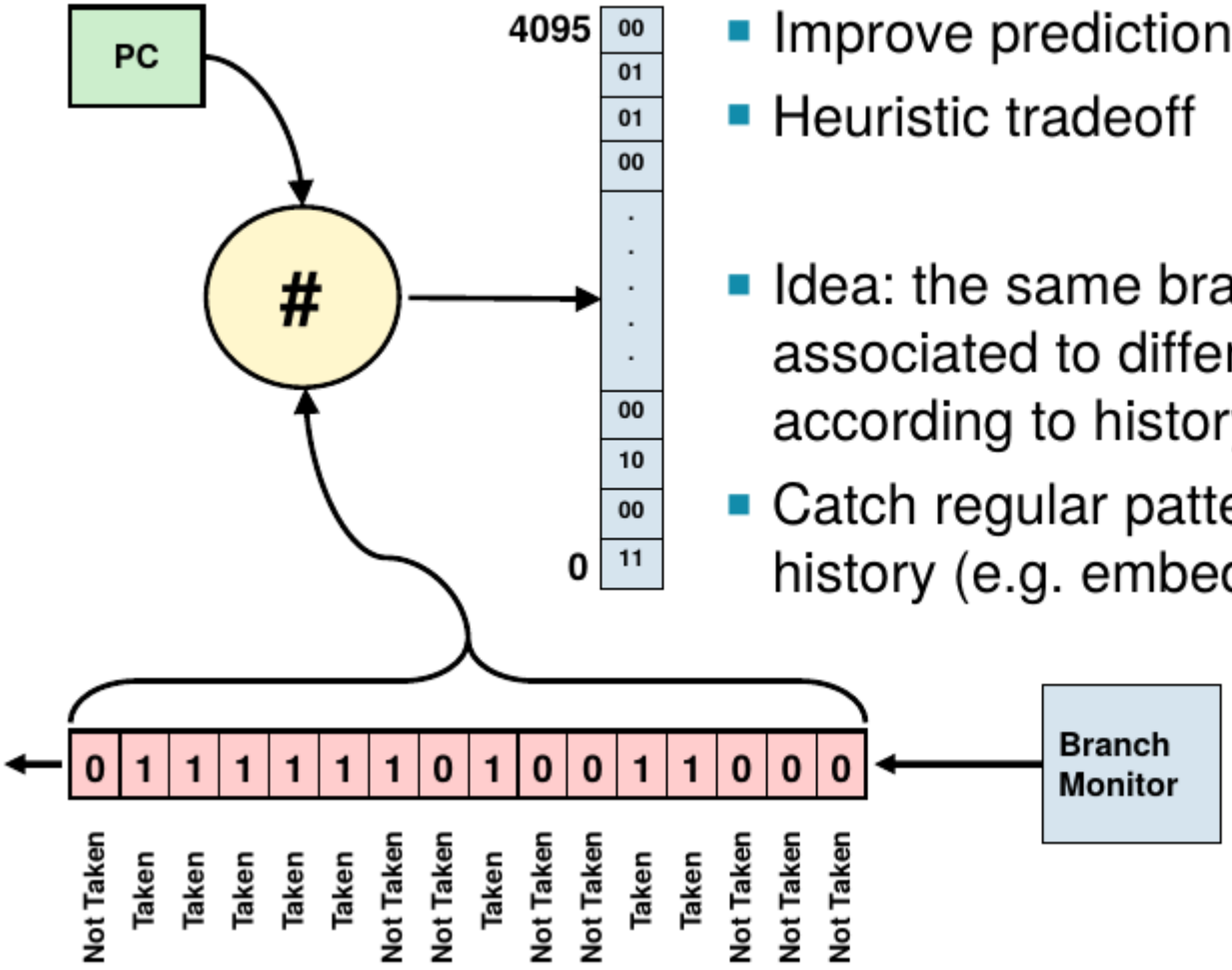

- **Dynamic Branch Prediction:**
  - Global History Buffer: 1K ~ 16K entries
  - Branch-Target Address Cache: 512 ~ 4K entries
  - Return stack of 4 x 32 bits
- **Fast-loop mode:**
  - instruction loop that are smaller than 64 bytes often complete without additional instruction cache accesses

# Branch Prediction : BTAC cache



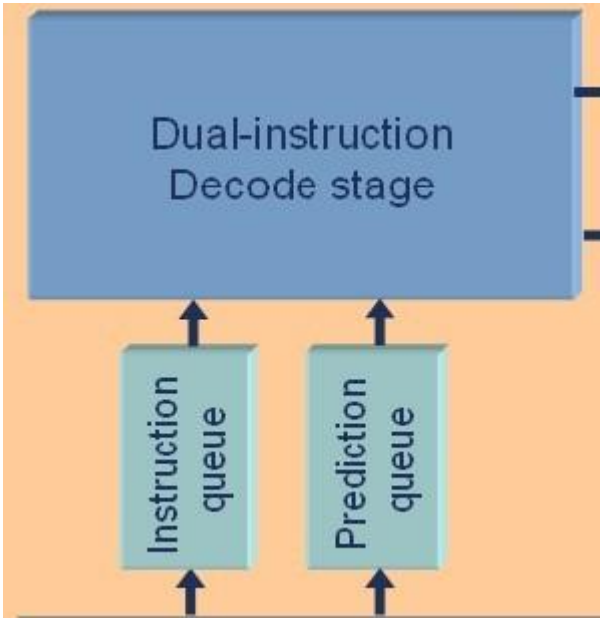

- **B**ranch **T**arget **A**ddress **C**ache (BTAC) associates PC with:
  - predicate 'is a branch'
  - target address and state (ARM,THUMB)
  - Other information (like 'is a func call', 'is a func return', etc)

# Branch Prediction : History Buffer



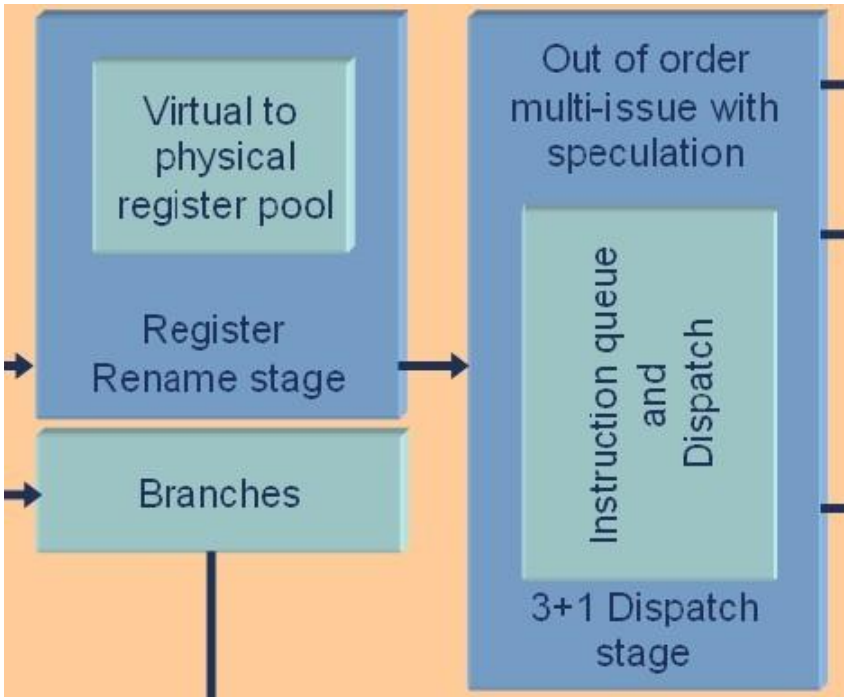

- Improve prediction further !
- Heuristic tradeoff
- Idea: the same branch gets associated to different predictors according to history of control
- Catch regular patterns in branch history (e.g. embedded loops)

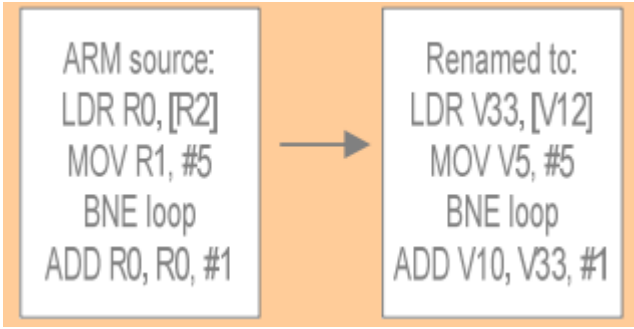
# Instruction Decode




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

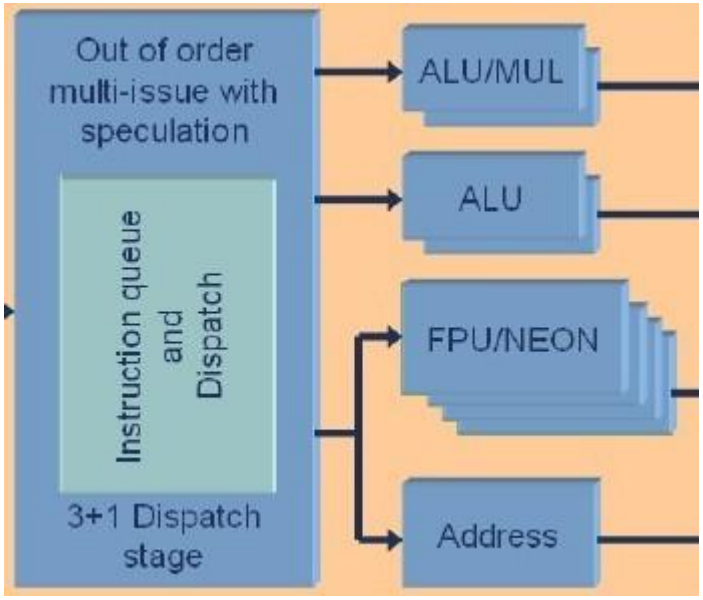
# Renaming



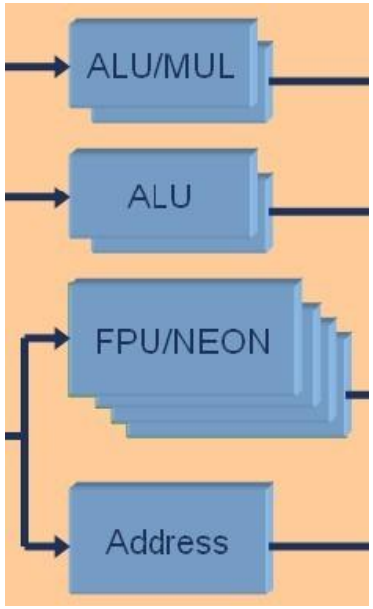

- Removes hazard in pipeline by solving data dependencies
- Use simplified stalling logic
- Can unroll simple code loops

# 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

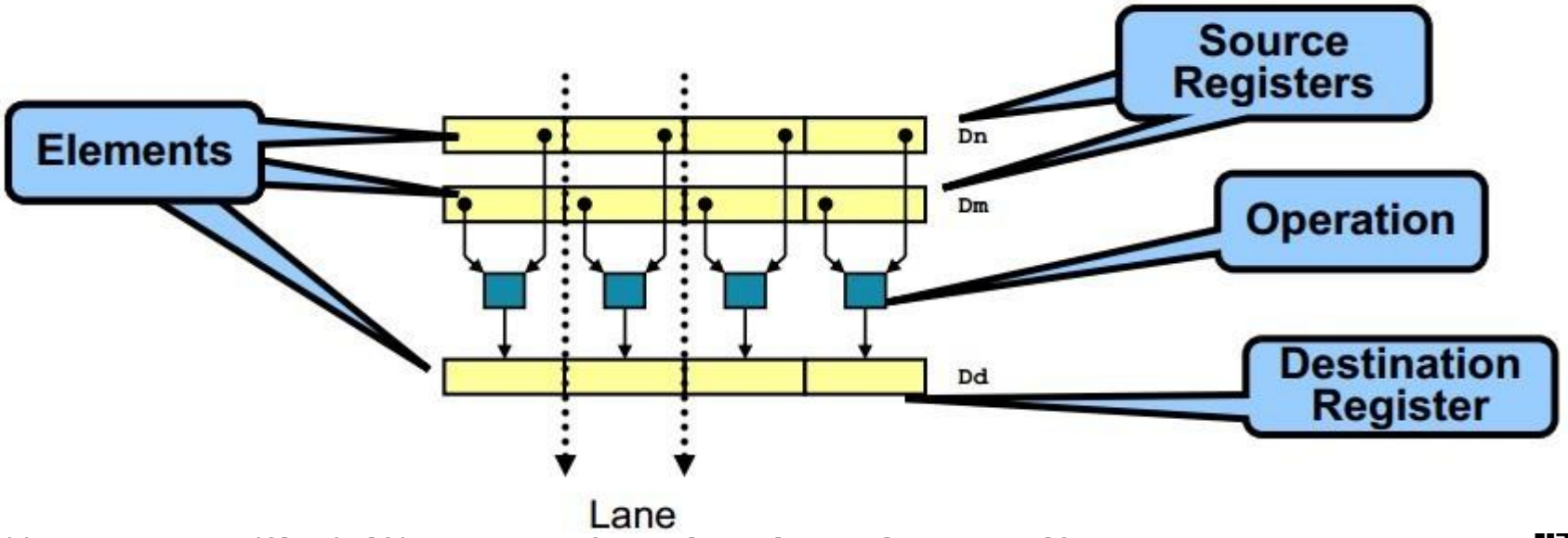
# Execute




- 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 x 64 bit wide or 16 registers x 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



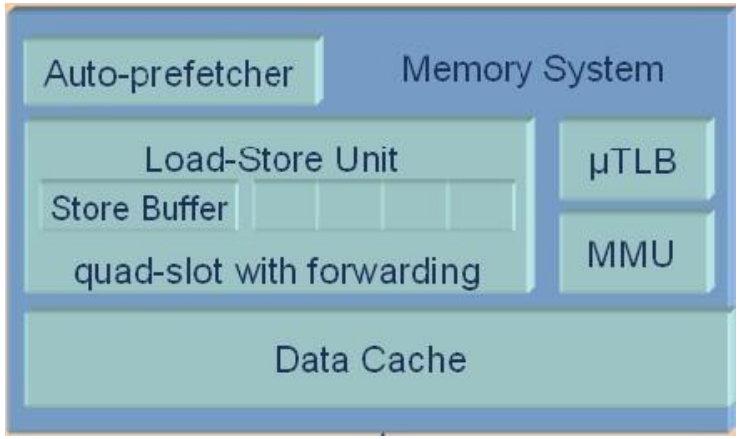

http://www.arm.com/files/pdf/AT_-_NEON_for_Multimedia_Applications.pdf

# 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

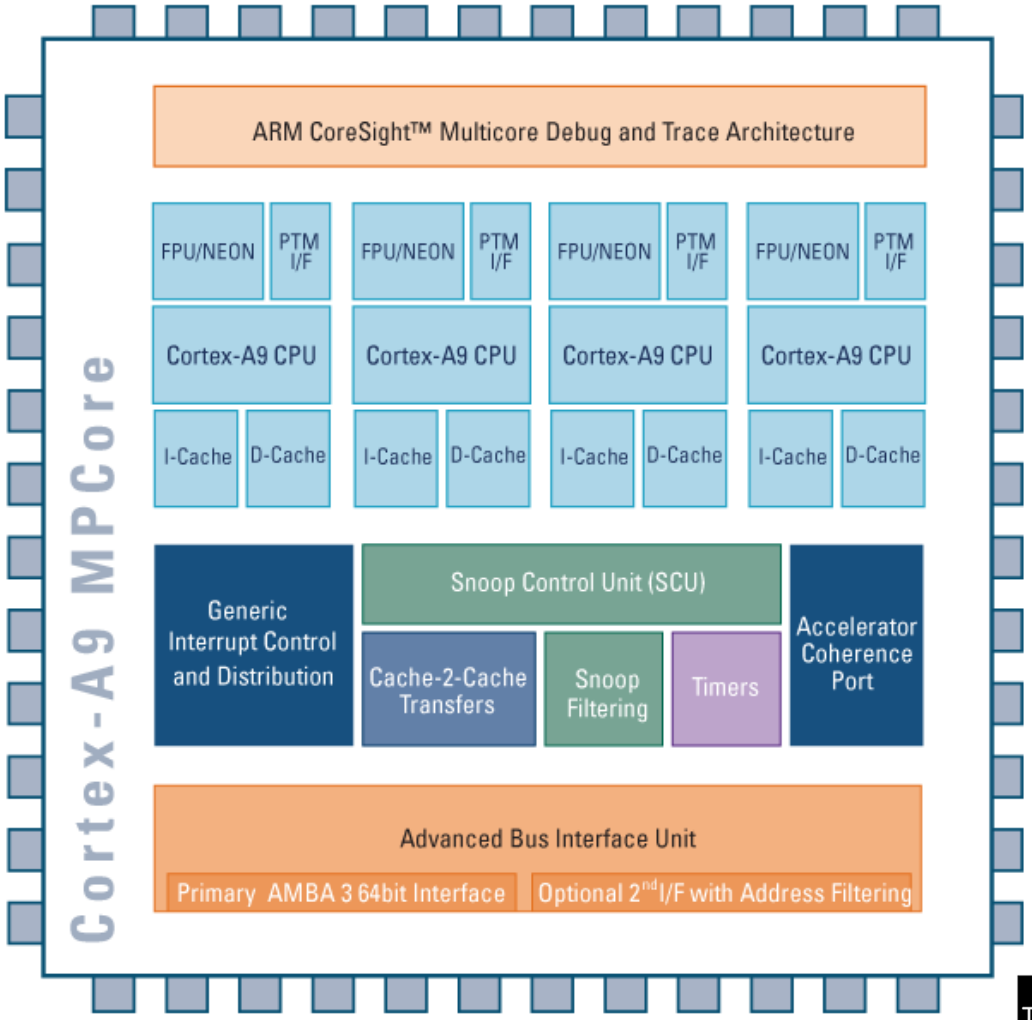
# Memory



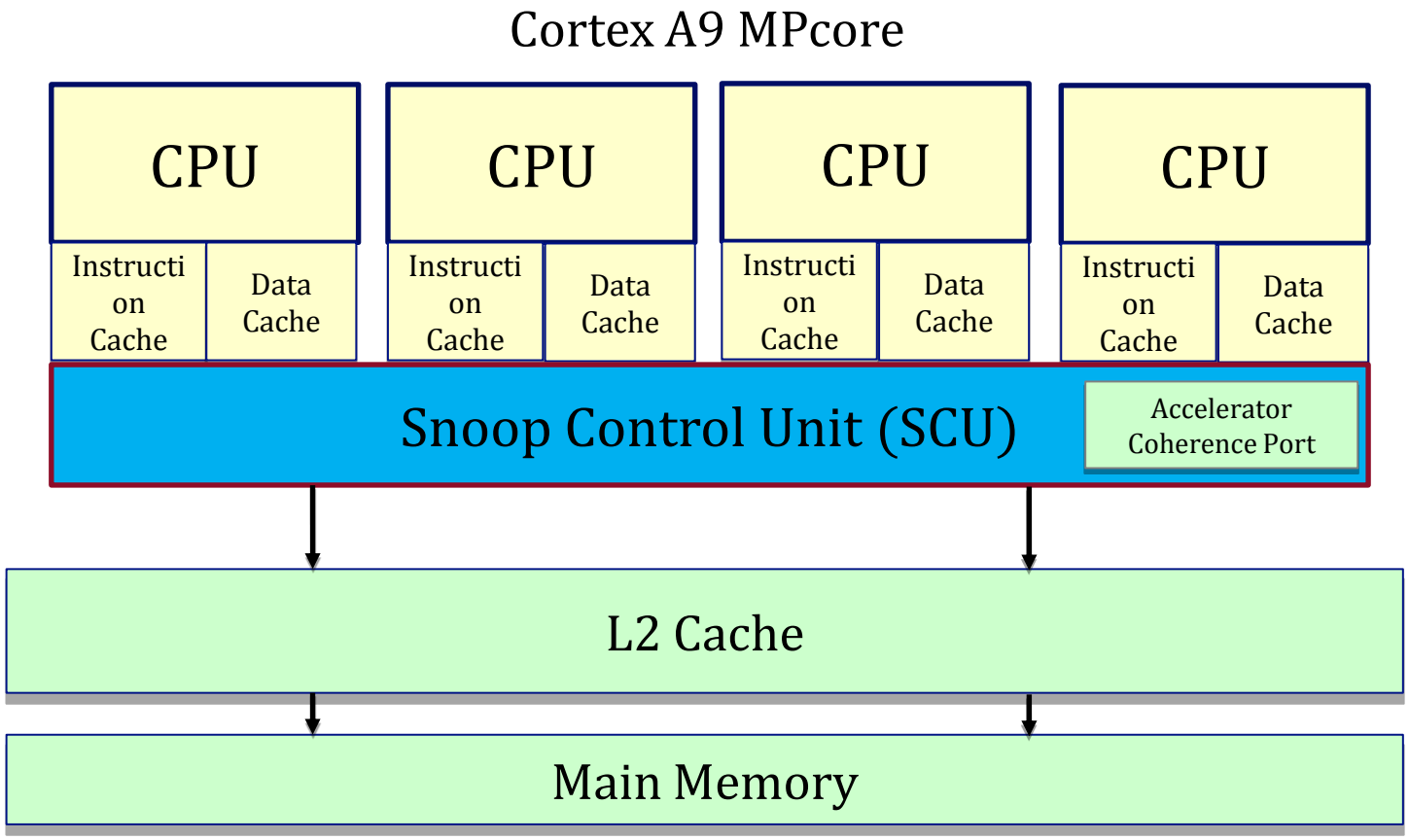

- **Data prefetcher**
  - monitor cache line requests by processor and cache misses to determine how much data to prefetch
  - can prefetch up to 8 independent data streams
- **load-store instructions forwarded for resolution within memory system**
- **2-level TLB structure (Translation Lookaside Buffer)**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

# Cortex MPCore Processors

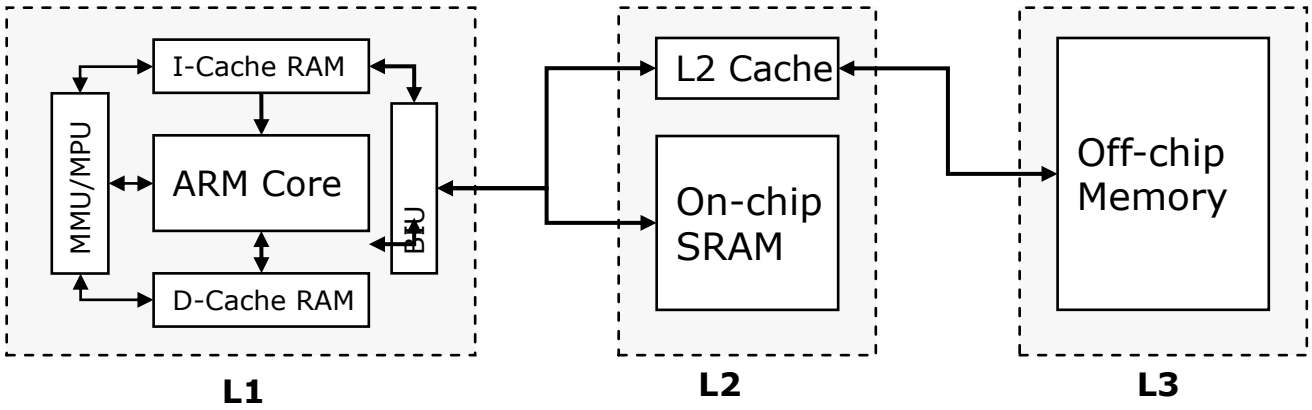
- **Standard Cortex cores, with additional logic to support MPCore**
  - Available as 1-4 CPU variants
- **Include integrated**
  - Interrupt controller
  - Snoop Control Unit (SCU)
  - Timers and Watchdogs



# Memory Hierarchy

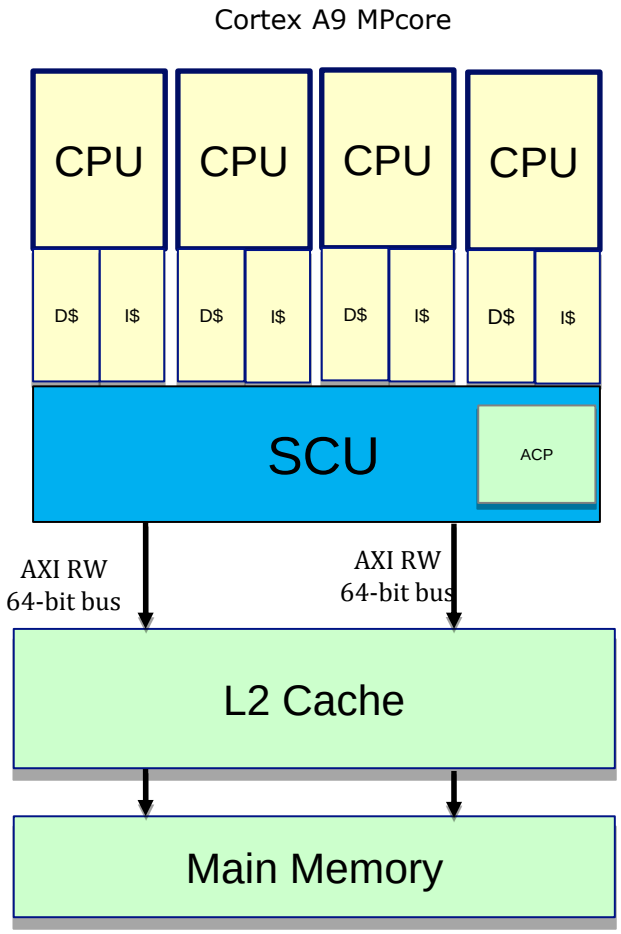


# L1 and L2 Caches



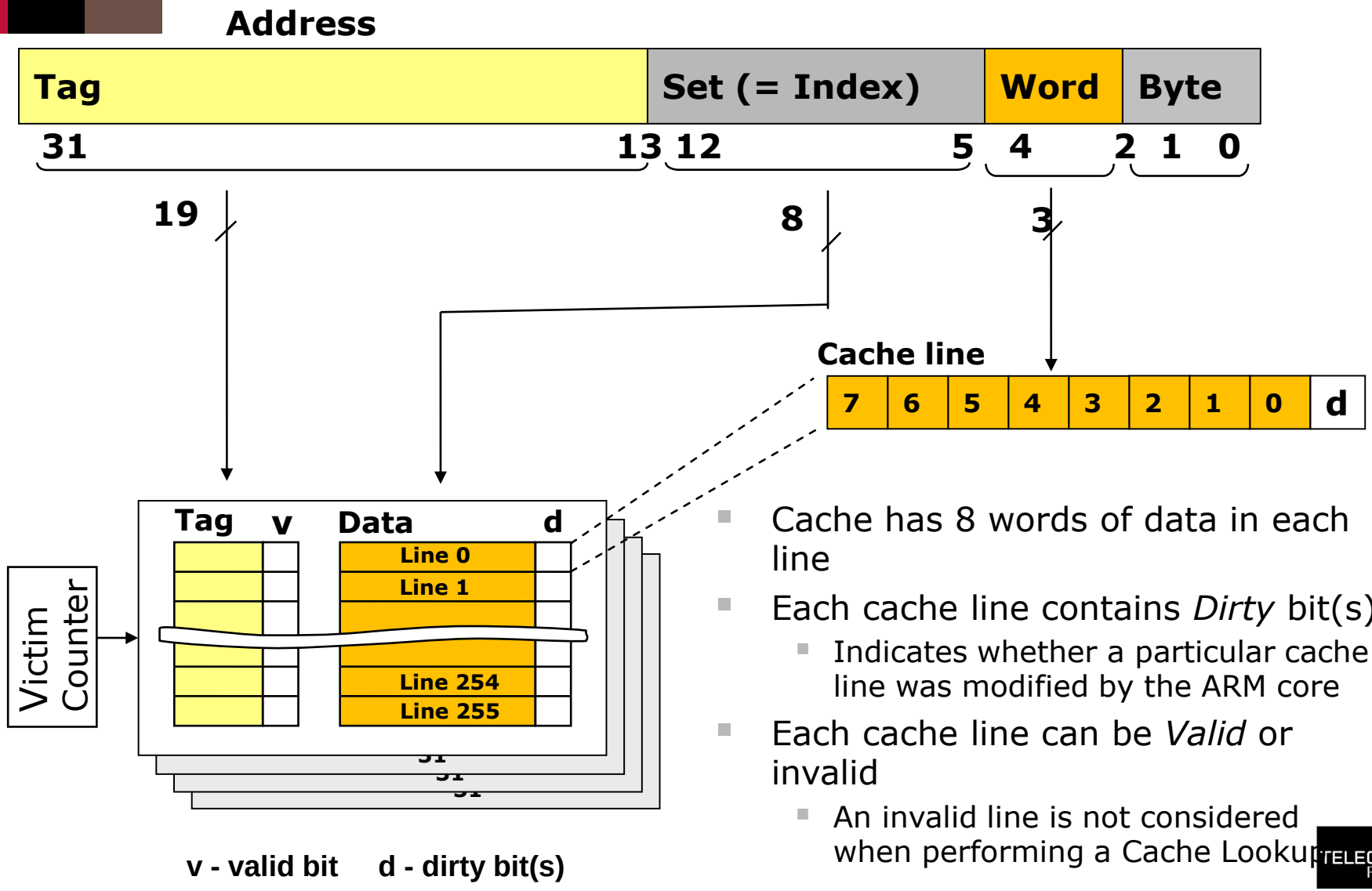

- **Typical memory system can have multiple levels of cache**
  - Level 1 memory system typically consists of L1-caches, MMU/MPU and TCMs
  - Level 2 memory system depends on the system design
- **Memory attributes determine cache behavior at different levels**
  - Controlled by the MMU/MPU
  - Inner 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)

# L1 caches



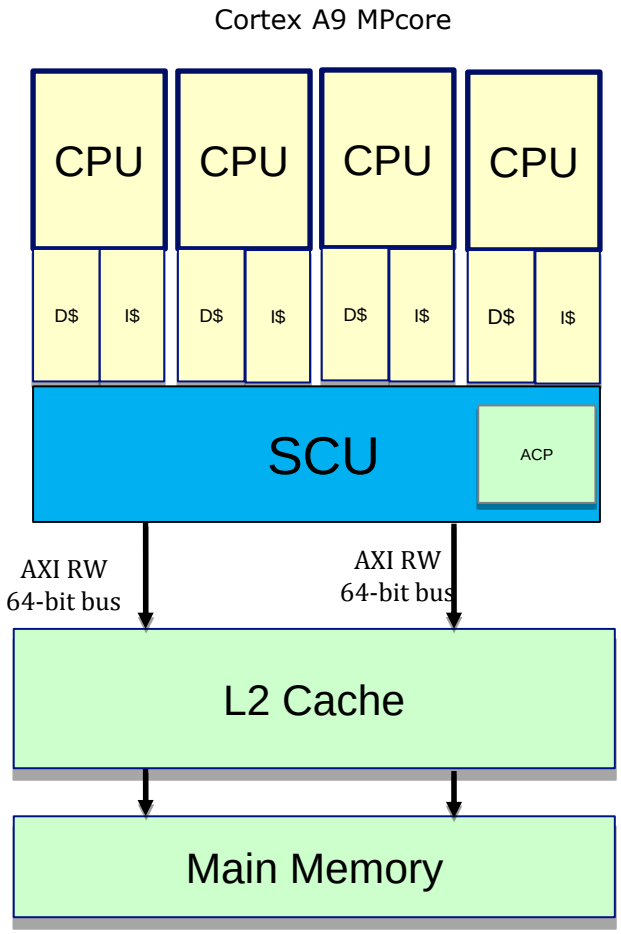

- Non-unified
  - 32 bytes line length
  - can be disabled independently
- 16, 32 or 64KB
- 4 - way associative
- support for Security Extensions
- I cache: VIP(virtual address)
- D cache: PIPT (physical address)
  - reduce number of caches flushes and refills and save energy

# Example 32KB ARM cache



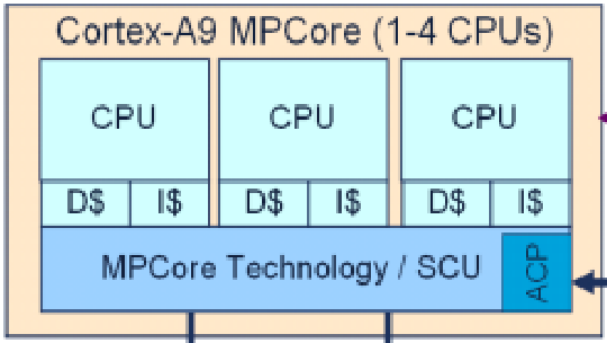

- Cache has 8 words of data in each line
- Each cache line contains *Dirty* bit(s)
  - Indicates whether a particular cache line was modified by the ARM core
- Each cache line can be *Valid* or invalid
  - An invalid line is not considered when performing a Cache Lookup

# L2 cache



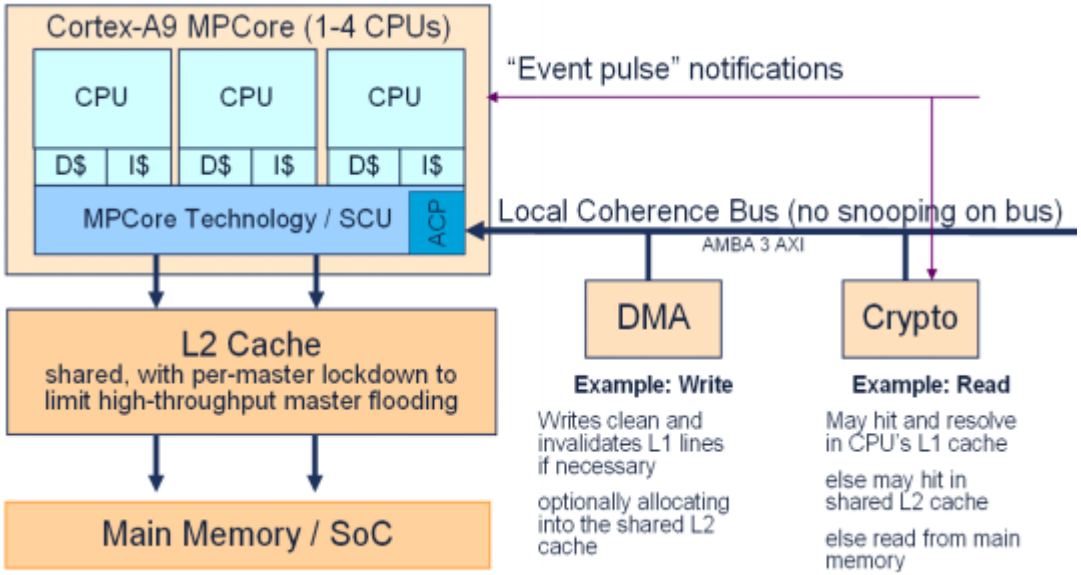

- shared, unified
- Off-chip
- 128KB to 8MB
- 4 to 16-way associative

# 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



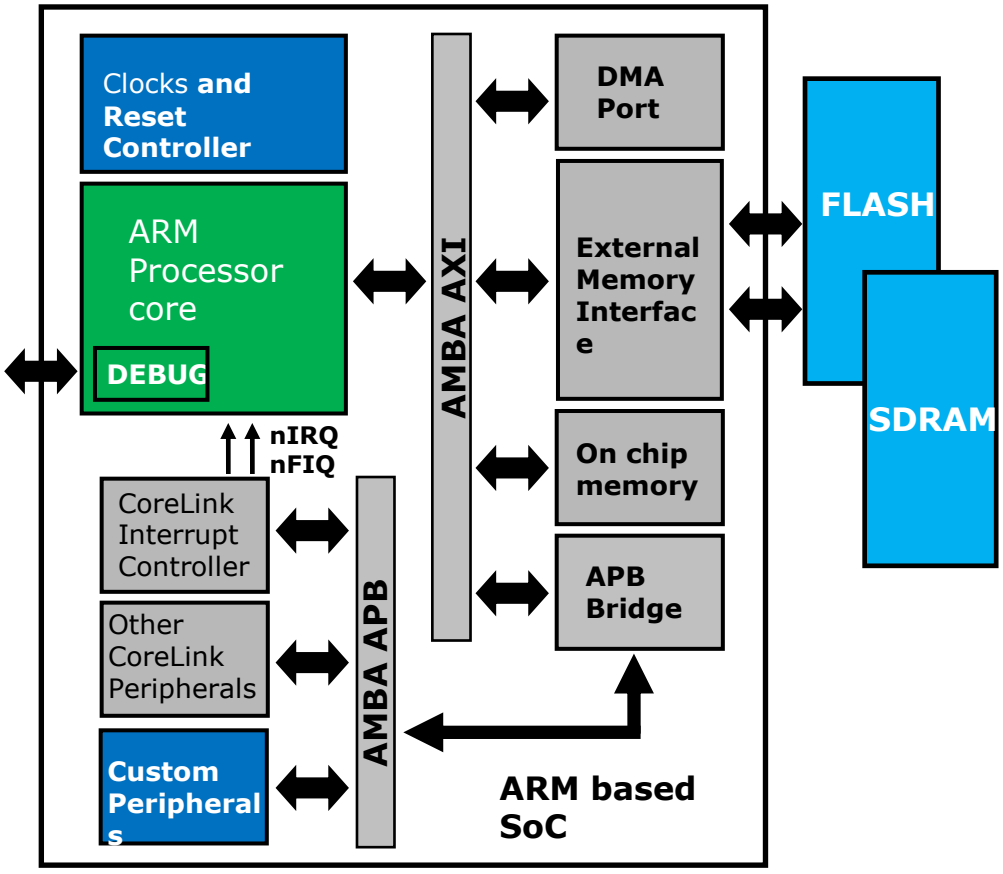

- optional AXI 64-bit slave port
- allows to connect to non-cached system mastering peripherals and accelerators
  - —DMA engine or cryptographic accelerator
- SCU enforces memory coherency

# Plan

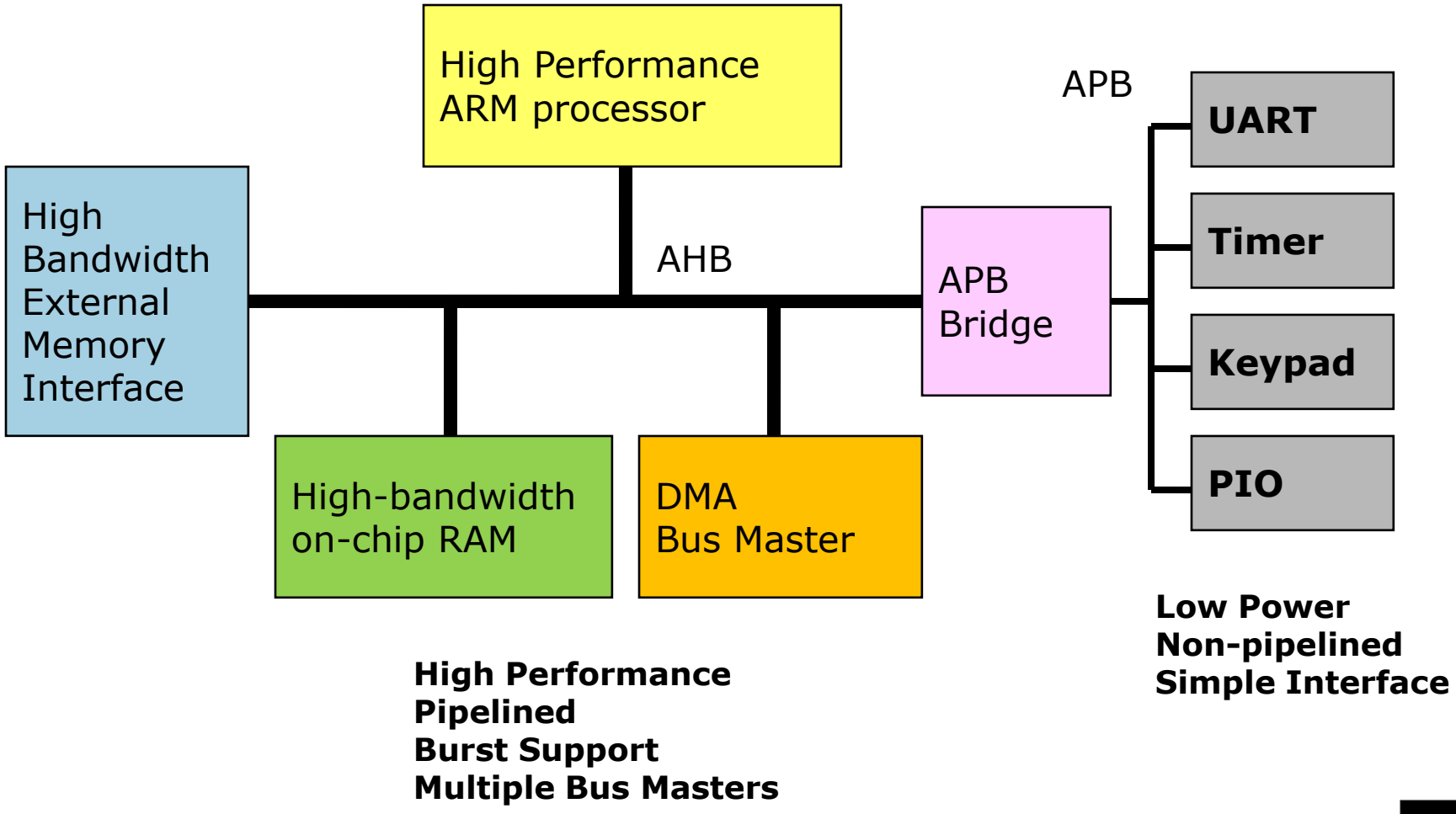
- **ARM historique : l'ARM7TDMI**
- **Les Cortex**
  - Cortex M family
  - Cortex A family
- **System bus**

# ARM-based system

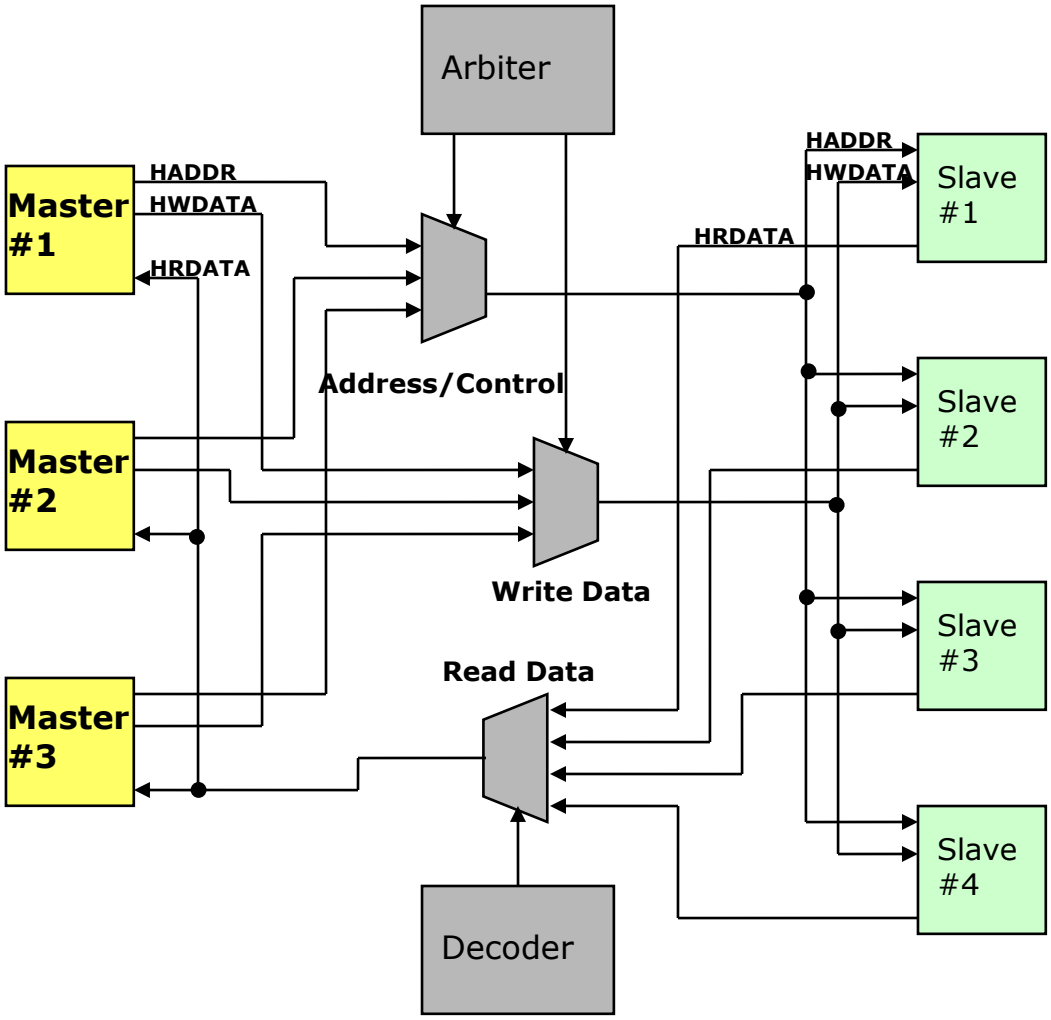
- **ARM core deeply embedded within an SoC**
  - External debug and trace via JTAG or CoreSight interface
- **Design can have both external and internal memories**
  - Varying width, speed and size – depending on system requirements
- **Can include ARM licensed CoreLink peripherals**
  - Interrupt controller, 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 of AMBA System

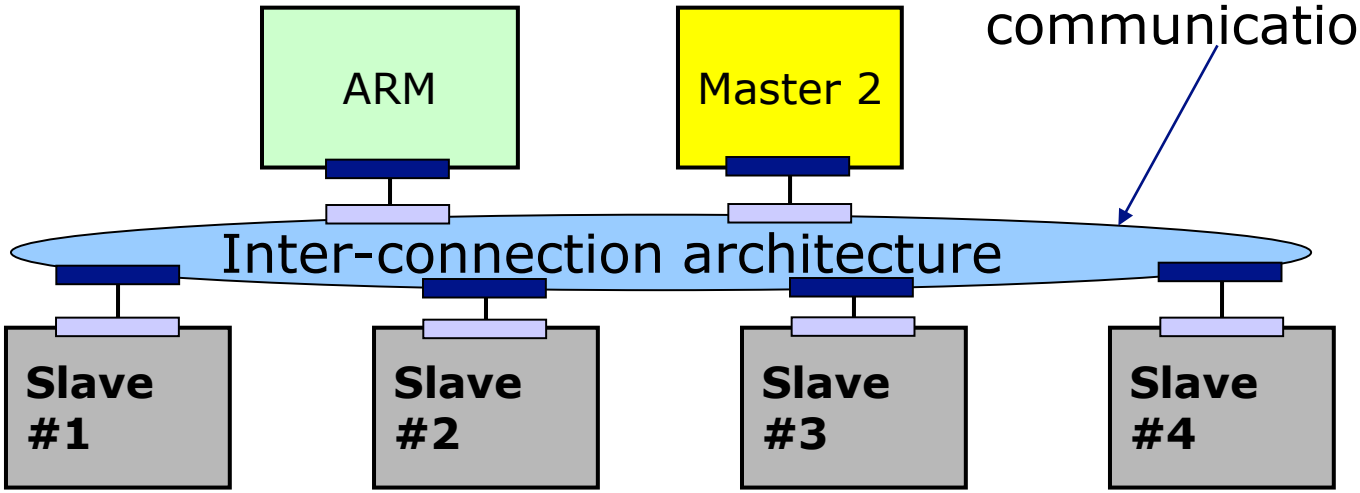


# AHB Structure



# AXI Multi-Master System Design

Compared to AHB, AXI allows multi-channel communication




Master interface

Slave interface