

Informatique embarquée

L'interface logiciel / matériel

Philippe.Plasson@obspm.fr

L'interface logiciel / matériel

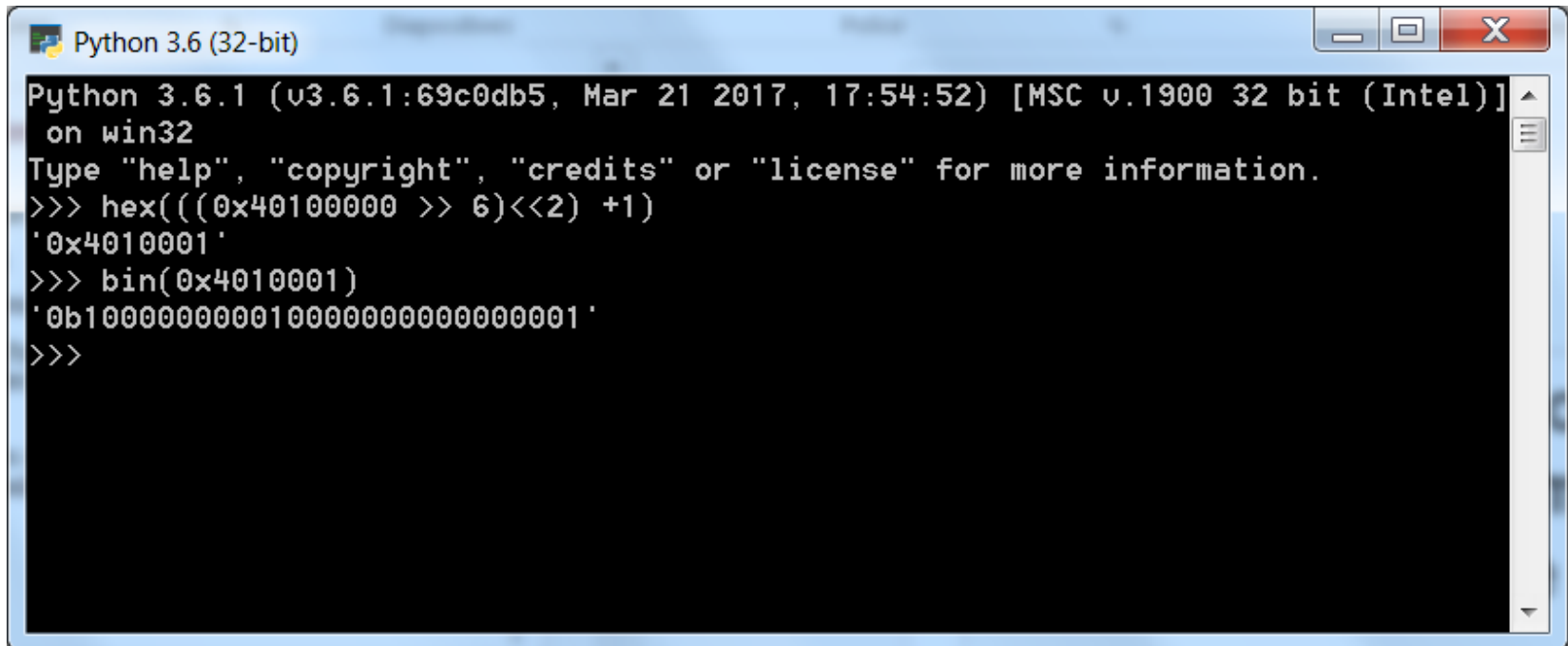
1. Préambule
2. Architecture des processeurs
3. Les registres
4. Les exceptions et les interruptions
5. La mémoire (types de mémoire physique, organisation de la mémoire, DMA, mémoire cache, MMU, endianisme)
6. L'accès aux périphériques

L'interface logiciel / matériel

1. Préambule
2. Architecture des processeurs
3. Les registres
4. Les exceptions et les interruptions
5. La mémoire (types de mémoire physique, organisation de la mémoire, DMA, mémoire cache, MMU, endianisme)
6. L'accès aux périphériques

Préambule

- Dans ce cours, on fait appel souvent à des opérations logiques sur la mémoire : masquages, décalages sur la droite, décalages sur la gauche, etc.
- Une console Python est très pratique pour rapidement tester les opérations logiques.

A screenshot of a Windows command prompt window titled "Python 3.6 (32-bit)". The window has standard Windows window controls (minimize, maximize, close) in the top right corner. The text inside the window shows the Python 3.6.1 startup message and several interactive commands. The first command is `hex(((0x40100000 >> 6)<<2) +1)`, which returns `'0x4010001'`. The second command is `bin(0x4010001)`, which returns `'0b10000000000100000000000000001'`. The prompt `>>>` is shown at the end of the line.

```
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(((0x40100000 >> 6)<<2) +1)
'0x4010001'
>>> bin(0x4010001)
'0b10000000000100000000000000001'
>>>
```

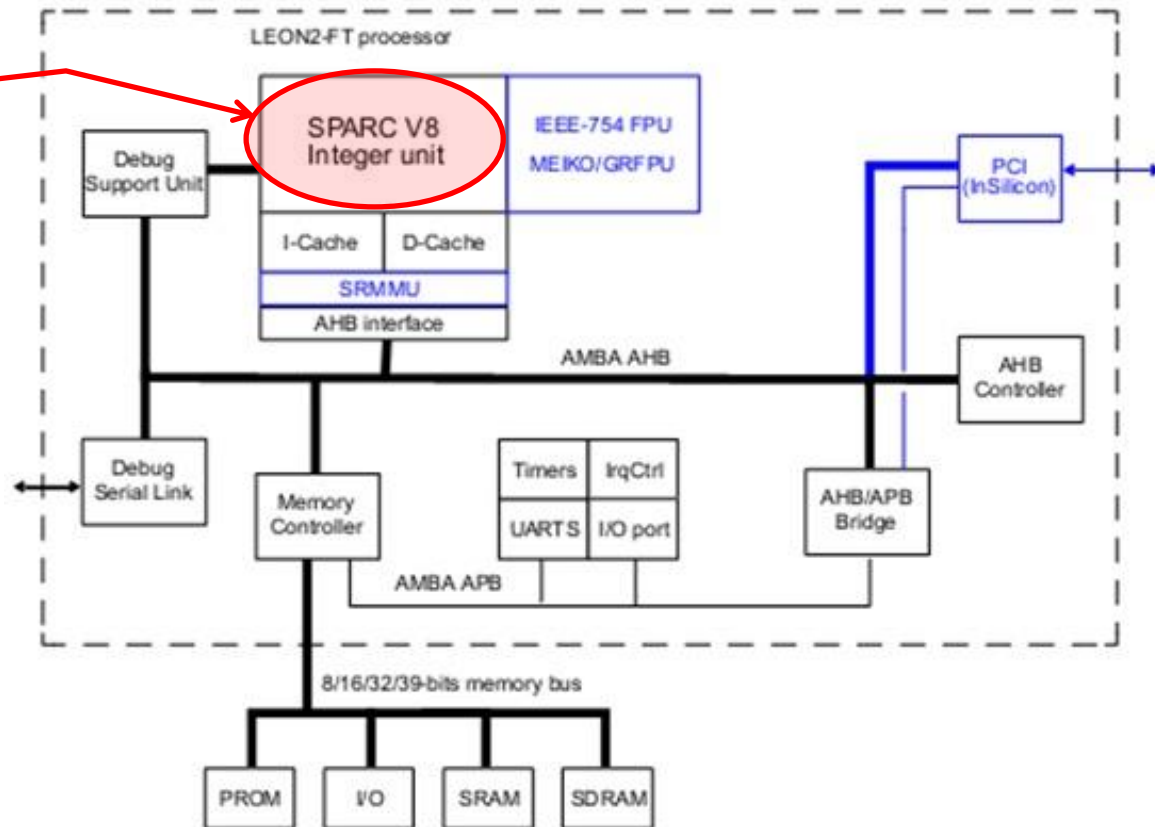
L'interface logiciel / matériel

1. Preamble
2. Architecture des processeurs
3. Les registres
4. Les exceptions et les interruptions
5. La mémoire (types de mémoire physique, organisation de la mémoire, DMA, mémoire cache, MMU, endianisme)
6. L'accès aux périphériques

Architecture des processeurs

Cœur de processeur contenant :

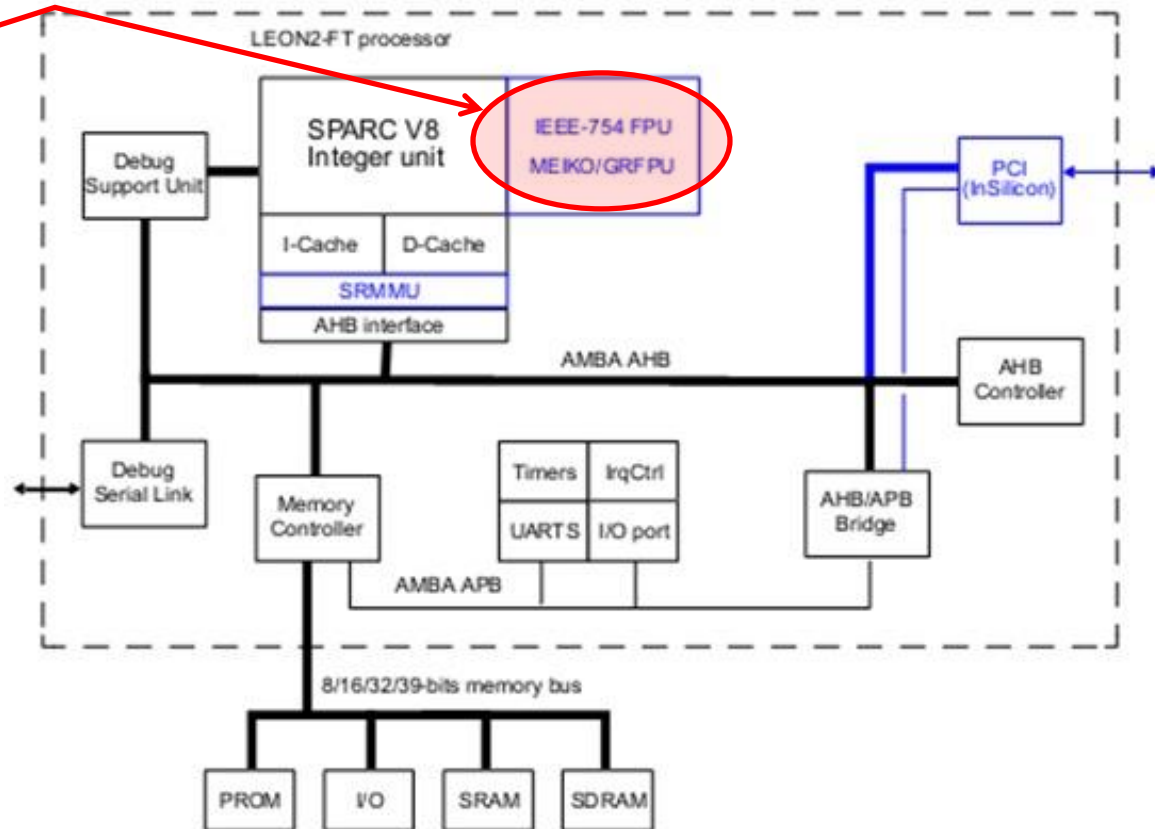
- une unité arithmétique et logique en charge des calculs (addition, soustraction, changement de signe, opérations logiques, comparaisons, décalages et rotations)
- une unité de contrôle gérant l'exécution du programme



Crédits : http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Microprocessors

Architecture des processeurs

FPU = Unité de Calcul en Virgule Flottante (Floating Point Unit – FPU) : réalise les opérations arithmétiques sur les réels. Elle supporte également des fonctions non arithmétiques : trigonométrie, exponentielle, logarithme, ...

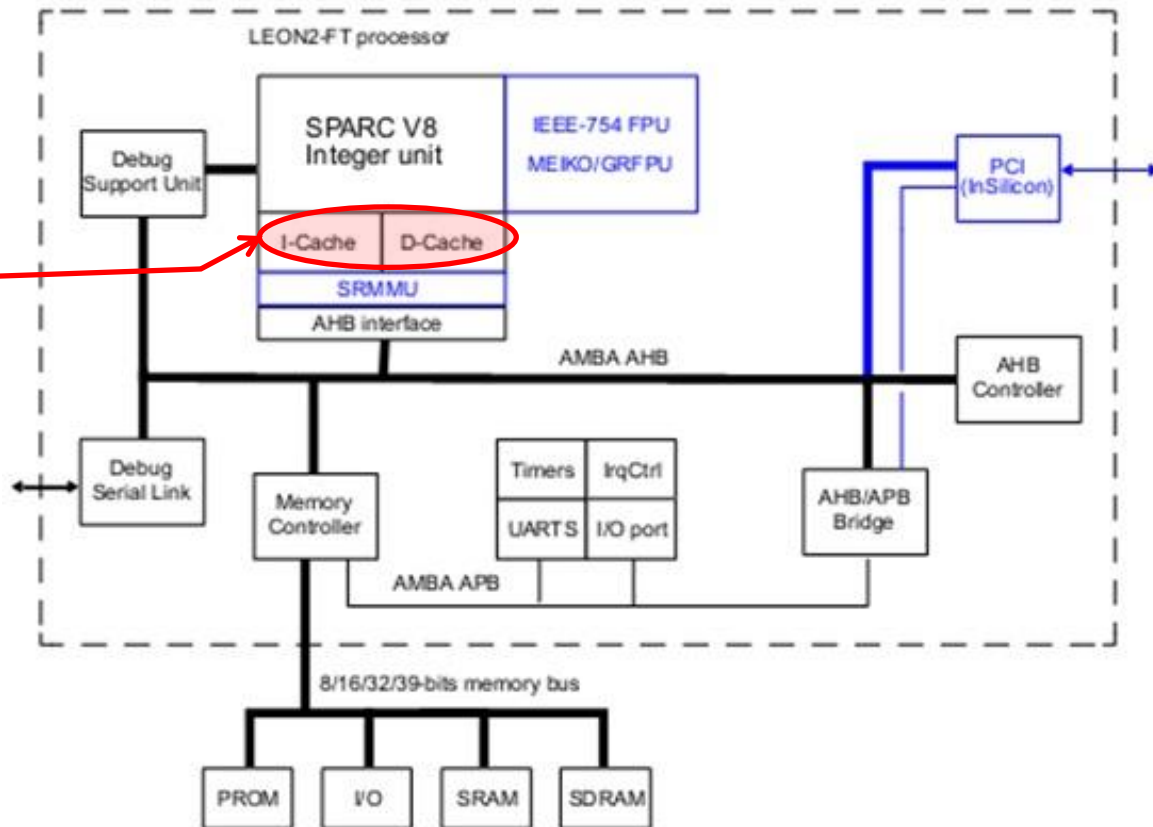


Crédits : http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Microprocessors

Architecture des processeurs

Caches de données et caches d'instructions.

- Mémoires cache = mémoires de petite taille, spécialisées (données ou instructions), rapides et proches du processeur dans lesquelles sont stockées le code ou les données fréquemment utilisés par le processeur
- Les caches peuvent être activés / désactivés ou encore vidés (flush) via des registres dédiés.
- Sur certains processeurs (multi-core), plusieurs niveaux de cache

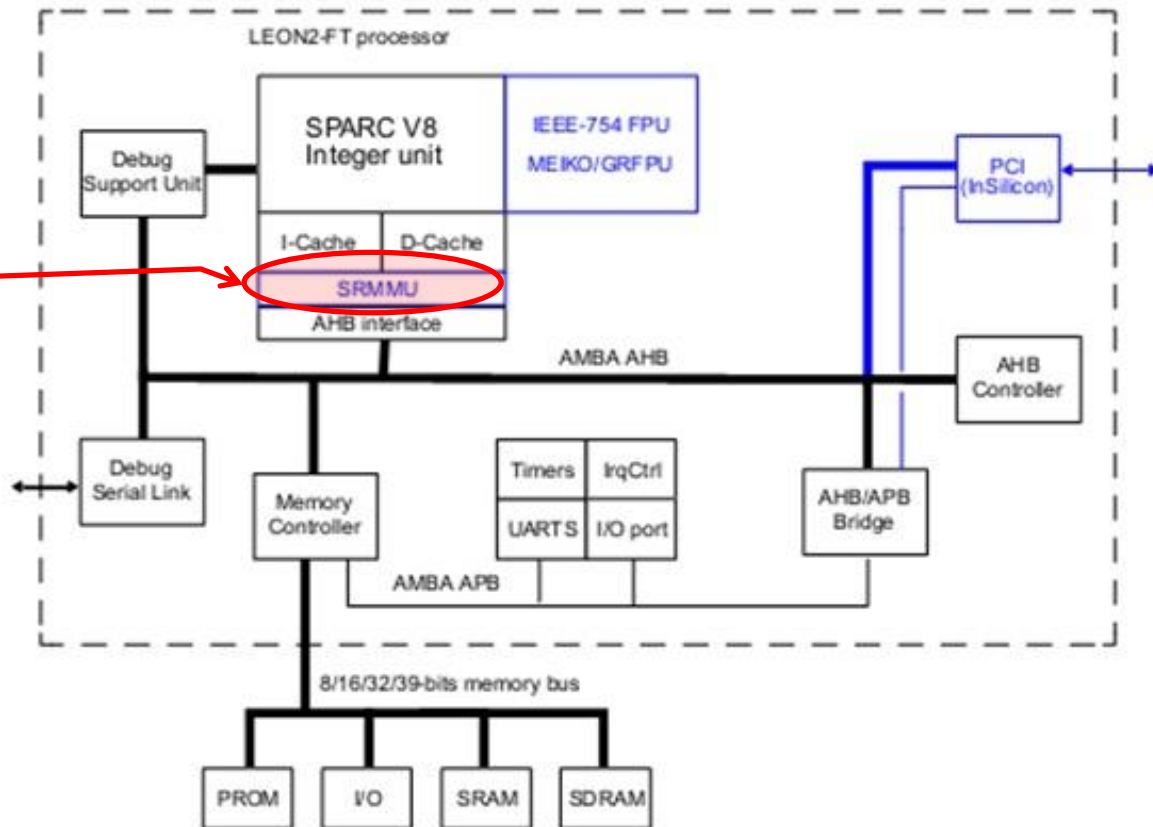


Crédits : http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Microprocessors

Architecture des processeurs

MMU = Memory Management Unit

- Gestionnaire de mémoire virtuelle (mémoire paginée)
- Unité réalisant des translations d'adresses de mémoires virtuelles vers des adresses physiques via des systèmes de tables
- Fonctions de protection de la mémoire (accès R, W, R/W...)
- Intègre un système de cache interne améliorant les performances (TLB = Translation Lookaside Buffer)



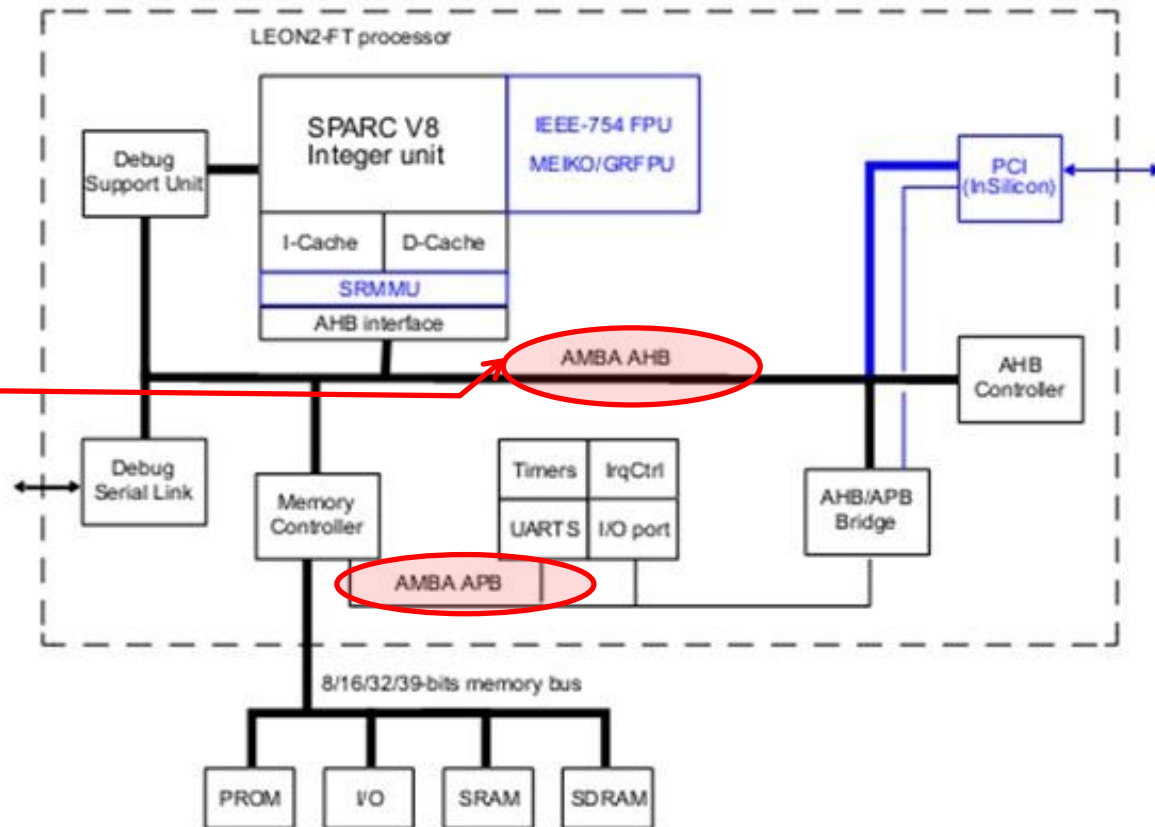
Crédits : http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Microprocessors

Architecture des processeurs

Bus interne AMBA
(standard pour
l'interconnexion et la
gestion des blocs
fonctionnels au sein d'un
SoC.

- AHB = Advanced High-Performance Bus = bus système principal dans les microcontrôleurs
- APB = Advanced Peripheral Bus = bus pour l'interconnexion des périphériques

<https://www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications>

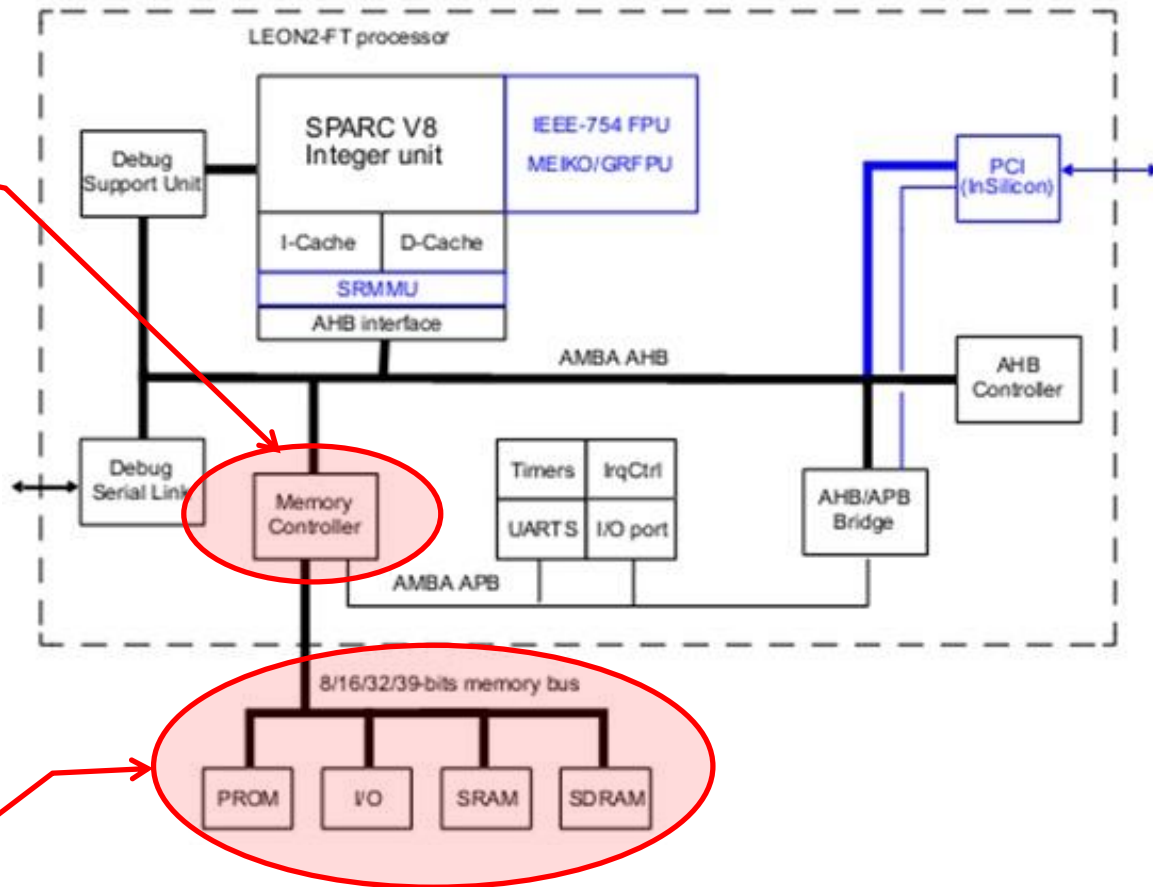


Crédits : http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Microprocessors

Architecture des processeurs

Contrôleur mémoire :

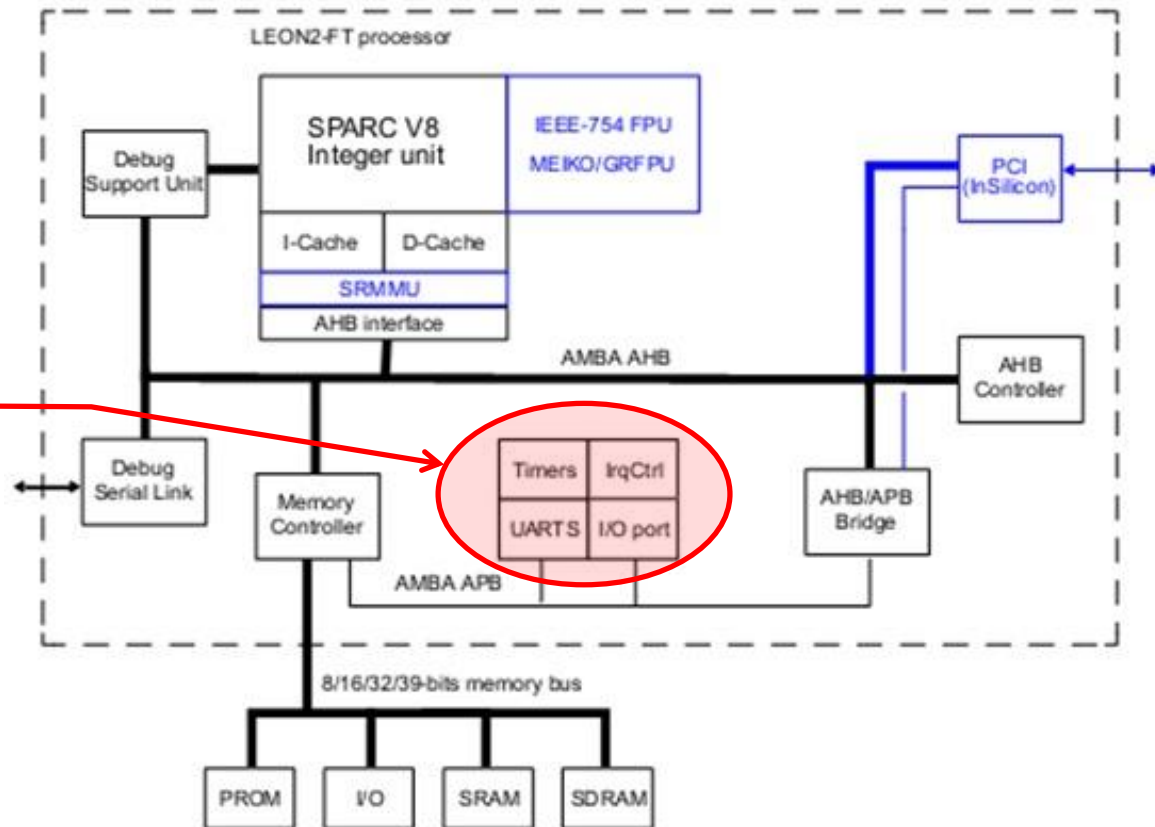
- fait le lien entre la mémoire externe (SRAM, SDRAM, PROM, ...) et le bus interne AMBA AHB,
- configurable via des registres en fonctions des caractéristiques des mémoires interfacées (taille, vitesse,...)



Bus mémoire externe et mémoires externes

Architecture des processeurs

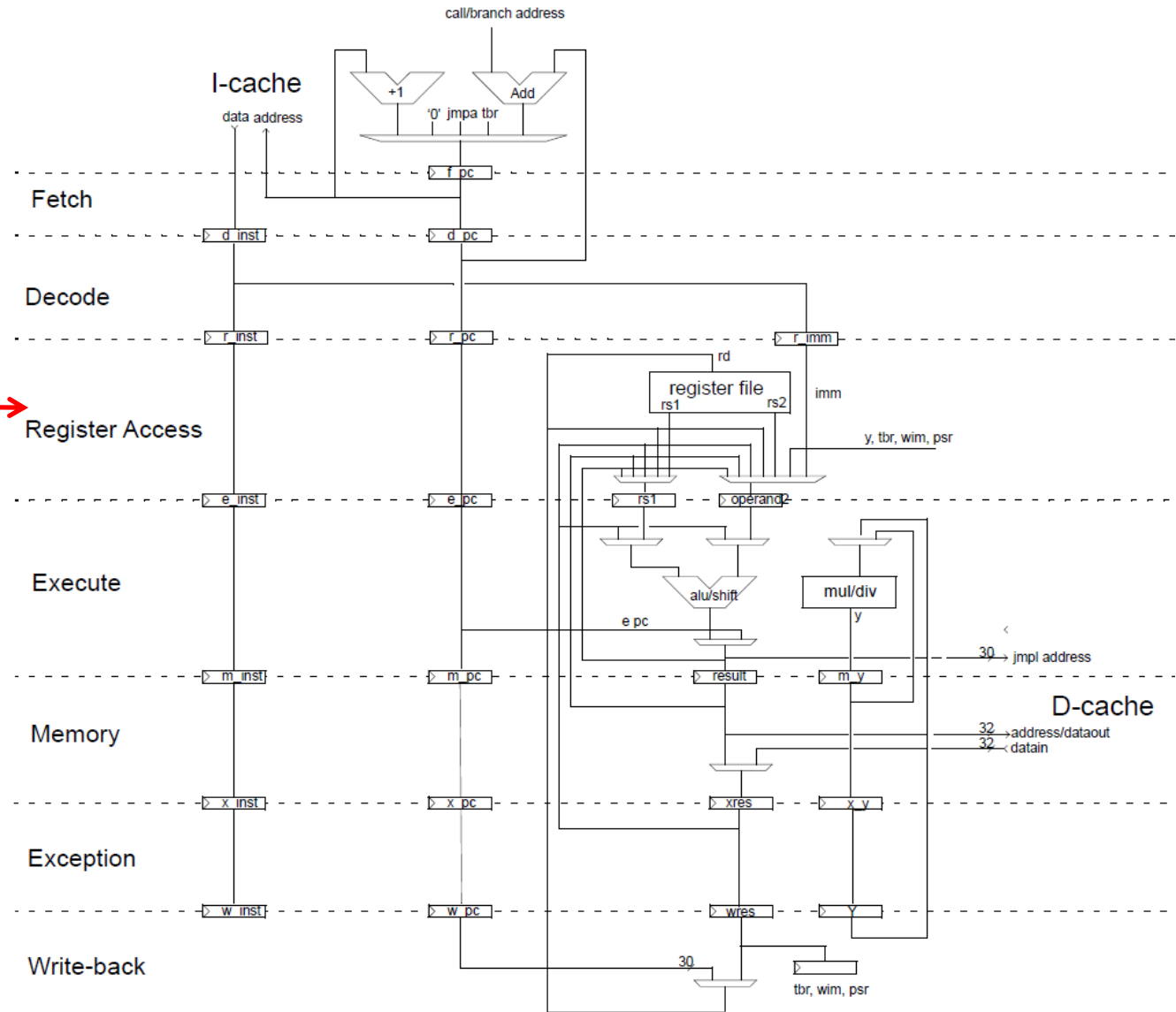
Unités périphériques (timers, UART, contrôleur d'interruptions, ports I/O) connectées sur le bus AMBA. Ces unités sont accessibles via des registres dédiés mappés dans l'espace mémoire du processeur.



Crédits : http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Microprocessors

Architecture des processeurs

Aperçu du pipeline à 7 étages de l'unité arithmétique et logique du processeur LEON3-FT



Architecture Von Neumann versus Architecture Harvard

■ Architecture Von Neumann

- Un seul bus mémoire pour les données et les instructions => limite de l'architecture : le transfert des données et la lecture des instructions ne peuvent pas être faits simultanément

■ Architecture Harvard

- Séparation physique du stockage et du bus mémoire pour le code et les données
- Le CPU peut simultanément lire des instructions et réaliser des accès mémoire (lecture / écriture).
- Architecture implémentée dans certains DSP et micro-contrôleurs

Architecture Von Neumann versus Architecture Harvard

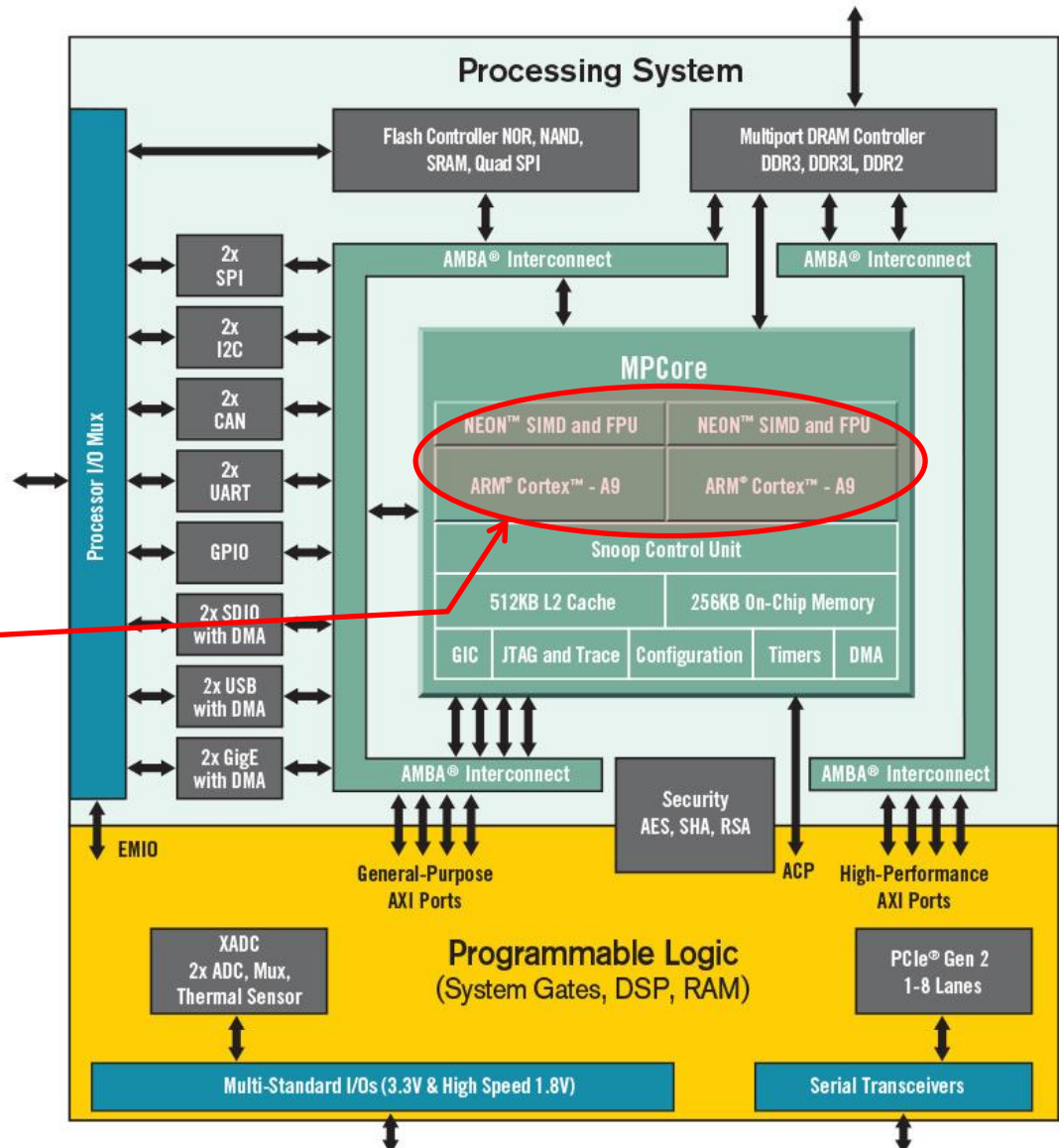
- Un grand nombre de processeurs actuels implémentent une version modifiée de l'architecture Harvard :
 - Pas de séparation physique de la mémoire principale (RAM) dédiée au code et de celle dédiée aux données.
 - Mais séparation physique de la mémoire cache instructions et données

Architecture des processeurs

Processeur multi-cœurs

- Ex. : Le processeur Xilinx Zynq-7000 (ARM Cortex-A9 dual-core)

Deux cœurs ARM Cortex A9 intégrés dans le même chip



L'interface logiciel / matériel

1. Préambule
2. Architecture des processeurs
- 3. Les registres**
4. Les exceptions et les interruptions
5. La mémoire (types de mémoire physique, organisation de la mémoire, DMA, mémoire cache, MMU, endianisme)
6. L'accès aux périphériques

Les registres

- Un registre est une zone mémoire (un mot de 32 bits généralement pour les processeurs 32 bits) intégrée au processeur, donc avec un accès rapide, permettant de :
 - stocker des données intermédiaires nécessaires aux différents calculs et opérations que doit réaliser le CPU
 - configurer le fonctionnement du processeur, changer son état, lire son état, en bref, interagir avec lui
 - accéder aux périphériques matériels (configuration, lecture des états, écriture / lecture des données transmises, etc.)

Les registres

- On distingue généralement :
 - Les registres accessibles par l'utilisateur :
 - General Purpose Registers (GPR)
 - Floating Point Registers (FPR)
 - Special Purpose Registers (SPR)
 - Program Counter (PC)
 - Status register
 - ...
 - Les registres internes qui ne sont pas accessibles via des instructions : Instruction Register, Memory Buffer Register, Memory Data Register, Memory Address Register
 - Les registres matériels permettant d'accéder aux périphériques et fonctions externes au CPU

Les registres

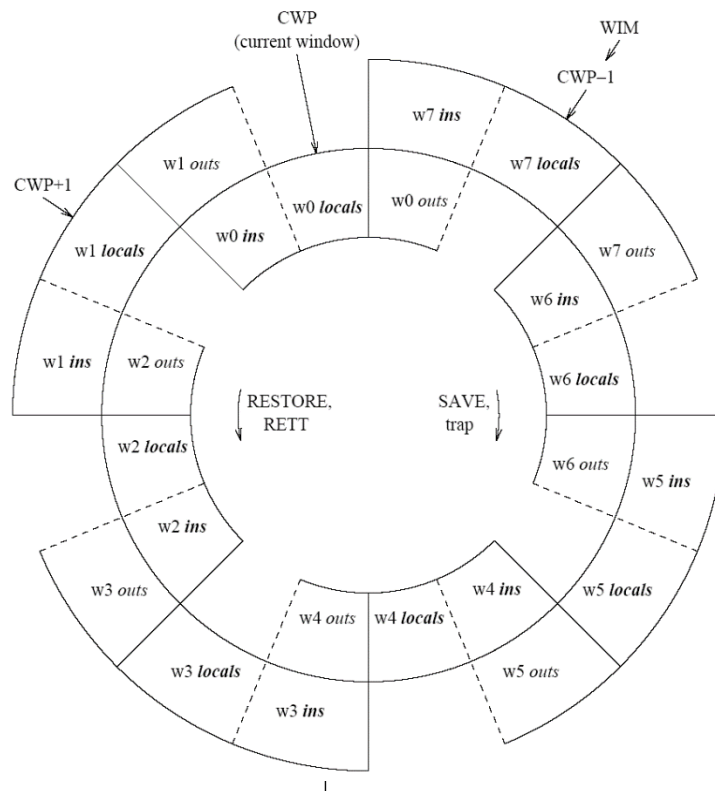
- Exemple : un SoC à base de processeur LEON possède les registres suivants :
 - Registres de l'IU (Integer Unit) : registre d'état du processeur, adresse de base de la trap table, compteur de programme (PC), registres fenêtrés : 8x [8x3] registres locaux + 8 registres globaux
 - Registres du FPU (Floating Point Unit)
 - Registres du contrôleur mémoire
 - Registres de la mémoire cache
 - Registres des Timers
 - Registres des UARTs
 - Registres de gestion des interruptions
 - Registres des différents IP cores intégrés dans le SOC (dépend de la configuration)

Les registres

Table 37. Window Registers

Type	Name	Definition
in	i7	return address
	i6	frame pointer
	i5	incoming parameter register 5
	i4	incoming parameter register 4
	i3	incoming parameter register 3
	i2	incoming parameter register 2
	i1	incoming parameter register 1
	i0	incoming parameter register 0
local	l7	local register 7
	l6	local register 6
	l5	local register 5
	l4	local register 4
	l3	local register 3
	l2	nPC (for RETT)
	l1	PC (for RETT)
	l0	local register 0
out	o7	temp
	o6	stack pointer
	o5	outgoing parameter register 5
	o4	outgoing parameter register 4
	o3	outgoing parameter register 3
	o2	outgoing parameter register 2
	o1	outgoing parameter register 1
	o0	outgoing parameter register 0

Les 8x[8x3] registres fenêtrés du LEON permettent de gérer de façon efficace les appels de fonctions (passage des paramètres d'entrée et de sortie), la sauvegarde de contexte et la restauration de contexte lors des traps / interruptions.



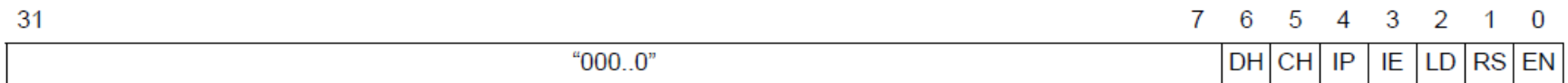
Les registres hardware

■ Exemple : les registres timers du LEON3-FT

APB address	Register	Reset value
0x80000300	Scaler value	0xFFFF
0x80000304	Scaler reload value	0xFFFF
0x80000308	Configuration register	0x0044
0x80000310	Timer 1 counter value register	-
0x80000314	Timer 1 reload value register	-
0x80000318	Timer 1 control register	IE=0, EN=0
0x80000320	Timer 2 counter value register	-
0x80000324	Timer 2 reload value register	-
0x80000328	Timer 2 control register	IE=0, EN=0
0x80000330	Timer 3 counter value register	-
0x80000334	Timer 3 reload value register	-
0x80000338	Timer 3 control register	IE=0, EN=0
0x80000340	Timer 4 counter value register	0xFFFF
0x80000344	Timer 4 reload value register	0xFFFF
0x80000348	Timer 4 control register	0x0009

Les registres hardware

■ Exemple : le registre Timer1 control register



- 31: 7 Reserved. Always reads as '000...0'.
- 6 Debug Halt (DH): Value of GPTI.DHALT signal which is used to freeze counters (e.g. when a system is in debug mode). Read-only.
- 5 Chain (CH): Chain with preceding timer. If set for timer n , timer n will be decremented each time when timer $(n-1)$ underflows.
- 4 Interrupt Pending (IP): The core sets this bit to '1' when an interrupt is signalled. This bit remains '1' until cleared by writing '0' to this bit.
- 3 Interrupt Enable (IE): If set the timer signals interrupt when it underflows.
- 2 Load (LD): Load value from the timer reload register to the timer counter value register.
- 1 Restart (RS): If set, the timer counter value register is reloaded with the value of the reload register when the timer underflows
- 0 Enable (EN): Enable the timer.

L'interface logiciel / matériel

1. Préambule
2. Architecture des processeurs
3. Les registres
- 4. Les exceptions et les interruptions**
5. La mémoire (types de mémoire physique, organisation de la mémoire, DMA, mémoire cache, MMU, endianisme)
6. L'accès aux périphériques

Les interruptions et les exceptions

- Une interruption est un signal émis par le matériel ou le logiciel indiquant qu'un événement vient de se produire et qu'il doit être traité immédiatement.
 - Une interruption (IRQ = Interrupt Request) peut être générée par le matériel, c'est-à-dire par un des modules de gestion des périphériques et des I/O interfacés au CPU (UART, timer, USB, I2C, SPI, CAN, ...)
 - Une interruption peut correspondre à l'occurrence d'un événement interne au processeur se produisant lors de l'exécution d'un programme (division par 0, accès mémoire non alignés, ...)
 - Les interruptions internes sont appelées des traps ou encore des exceptions.
 - Une interruption peut être déclenchée aussi par le logiciel en agissant directement sur le contrôleur d'interruptions ou en utilisant des instructions dédiées du processeur (software trap).

Les interruptions et les exceptions

- On associe à une interruption donnée une routine de traitement de l'interruption (ISR = Interrupt Service Routine).
- Les adresses des routines d'interruption, ou pour être plus précis, les instructions de branchement sur les routines d'interruption, sont localisées dans la table des vecteurs d'interruption :
 - Pour les processeurs LEON, dans la « trap table »
 - Pour les processeurs ARM, dans la « vector table »

Les interruptions et les exceptions

- Ex. : aperçu de la vector table du processeur ARM STM32F3 :

Table 30. Vector table (continued)

Position	Priority	Type of priority	Acronym	Description	Address
27	34	settable	TIM1_CC	TIM1 capture compare interrupt	0x0000 00AC
28	35	settable	TIM2	TIM2 global interrupt	0x0000 00B0
29	36	settable	TIM3	TIM3 global interrupt	0x0000 00B4
30	37	settable	TIM4	TIM4 global interrupt	0x0000 00B8
31	38	settable	I2C1_EV_EXTI23	I2C1 event interrupt & EXTI Line23 interrupt	0x0000 00BC
32	39	settable	I2C1_ER	I2C1 error interrupt	0x0000 00C0
33	40	settable	I2C2_EV_EXTI24	I2C2 event interrupt & EXTI Line24 interrupt	0x0000 00C4
34	41	settable	I2C2_ER	I2C2 error interrupt	0x0000 00C8
35	42	settable	SPI1	SPI1 global interrupt	0x0000 00CC
36	43	settable	SPI2	SPI2 global interrupt	0x0000 00D0
37	44	settable	USART1_EXTI25	USART1 global interrupt & EXTI Line 25	0x0000 00D4
38	45	settable	USART2_EXTI26	USART2 global interrupt & EXTI Line 26	0x0000 00D8
39	46	settable	USART3_EXTI28	USART3 global interrupt & EXTI Line 28	0x0000 00DC
40	47	settable	EXTI15_10	EXTI Line[15:10] interrupts	0x0000 00E0
41	48	settable	RTCAlarm	RTC alarm interrupt	0x0000 00E4
42 ⁽¹⁾	49	settable	USB_WKUP	USB wakeup from Suspend (EXTI line 18)	0x0000 00E8

Les interruptions et les exceptions

- Ex : aperçu de la trap table d'un processeur LEON :

Exceptions
processeur

Trap	TT	Pri	Description
reset	0x00	1	Power-on reset
write error	0x2b	2	write buffer error
instruction_access_error	0x01	3	Error during instruction fetch
illegal_instruction	0x02	5	UNIMP or other un-implemented instruction
privileged_instruction	0x03	4	Execution of privileged instruction in user mode
fp_disabled	0x04	6	FP instruction while FPU disabled
cp_disabled	0x24	6	CP instruction while Co-processor disabled
watchpoint_detected	0x0B	7	Hardware breakpoint match
window_overflow	0x05	8	SAVE into invalid window
window_underflow	0x06	8	RESTORE into invalid window
register_hadrware_error	0x20	9	Uncorrectable register file EDAC error
mem_address_not_aligned	0x07	10	Memory access to un-aligned address
fp_exception	0x08	11	FPU exception
data_access_exception	0x09	13	Access error during load or store instruction
tag_overflow	0x0A	14	Tagged arithmetic overflow
divide_exception	0x2A	15	Divide by zero
interrupt_level_1	0x11	31	Asynchronous interrupt 1
interrupt_level_2	0x12	30	Asynchronous interrupt 2
interrupt_level_3	0x13	29	Asynchronous interrupt 3
interrupt_level_4	0x14	28	Asynchronous interrupt 4
interrupt_level_5	0x15	27	Asynchronous interrupt 5
interrupt_level_6	0x16	26	Asynchronous interrupt 6

Interruptions
matérielles

Les interruptions et les exceptions

- Quand une interruption est déclenchée (changement d'état d'une des lignes d'interruption du processeur) :
 1. Le processeur interrompt l'exécution du programme en cours.
 2. Il saute à l'emplacement de la trap table (ou vector table) où l'instruction de branchement sur la routine de traitement de l'interruption est stockée.
 3. Il sauvegarde le contexte d'exécution du programme interrompu (sauvegarde dans la pile ou dans des jeux de registres dédiés).
 4. Il exécute l'instruction de branchement vers la routine de traitement de l'interruption (i.e. il saute à l'emplacement mémoire correspondant à cette routine).
 5. Il exécute la routine d'interruption.
 6. Il restaure le contexte d'exécution du programme interrompu.
 7. Il saute à l'emplacement mémoire où le programme a été interrompu.
 8. Il reprend l'exécution du programme interrompu.

Les interruptions et les exceptions

- La gestion des interruptions se fait à l'aide :
 - des registres du contrôleur d'interruption :
 - Registre indiquant qu'une interruption donnée s'est produite : Interrupt Pending Register (IPR)
 - Registre permettant d'activer ou désactiver une interruption : Interrupt Mask Register (IMR)
 - Registre permettant de forcer une interruption : Interrupt Force Register (IFR)
 - Registre permettant de nettoyer une interruption : Interrupt Clear Register
 - de fonctions de l'OS ou des bibliothèques BSP (Board Support Package) permettant d'installer des routines de traitements d'interruptions (interrupt handler)
 - La sauvegarde et la restauration du contexte d'exécution est pris en charge par l'OS ou les fonctions de la bibliothèque BSP.

Les interruptions et les exceptions

- Exemple : les registres du gestionnaire d'interruption du processeur LEON :



Figure 34. Interrupt pending register

- [31:17] Extended Interrupt Pending n (EIP[n]).
- [15:1] Interrupt Pending n (IP[n]): Interrupt pending for interrupt n .
- [0] Reserved

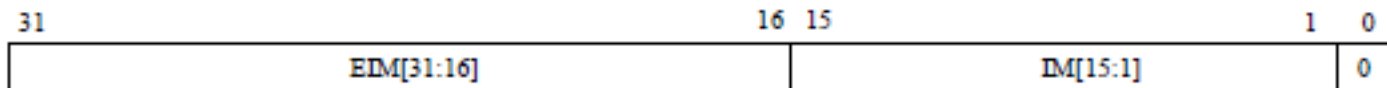



Figure 38. Processor interrupt mask register

- [31:16] Interrupt mask for extended interrupts
- [15:1] Interrupt Mask n (IM[n]): If IM n = 0 the interrupt n is masked, otherwise it is enabled.
- [0] Reserved.

Les interruptions et les exceptions

- Accès en C aux registres du contrôleur d'interruption (mot clé volatile, opérations logiques, masquage, ...)

```
#define INTERRUPT_PENDING_REGISTER 0x80000204
#define INTERRUPT_FORCE_REGISTER 0x80000208
#define INTERRUPT_MASK_REGISTER 0x80000240
#define INTERRUPT_CLEAR_REGISTER 0x8000020C
#define INTERRUPT_NUMBER 16
```



```
volatile uint32_t* interrupt_pending_register = (uint32_t*)INTERRUPT_PENDING_REGISTER;
volatile uint32_t* interrupt_force_register = (uint32_t*)INTERRUPT_FORCE_REGISTER;
volatile uint32_t* interrupt_mask_register = (uint32_t*)INTERRUPT_MASK_REGISTER;
volatile uint32_t* interrupt_clear_register = (uint32_t*)INTERRUPT_CLEAR_REGISTER;
```

```
void force_interrupt(uint32_t irq) {
    *interrupt_force_register = *interrupt_force_register | (1 << irq);
}
```

```
void enable_interrupt(uint32_t irq) {
    *interrupt_mask_register = *interrupt_mask_register | (1 << irq);
}
```

```
void disable_interrupt(uint32_t irq) {
    *interrupt_mask_register = *interrupt_mask_register & ~(1 << irq);
}
```

- Le mot clé « volatile » indique que le contenu de la mémoire peut changer entre 2 accès au pointeur indépendamment du programme lui-même (état du registre dépend d'une cause exogène liée aux I/O) → le mot clé indique au compilateur que les accès en lecture et écriture sur cette variable ne doivent pas être optimisés.

Les interruptions et les exceptions

■ Fonctions d'affichage du contenu des registres et des compteurs d'interruptions

```
void print_interrupt_registers() {
    printf("INTERRUPT_PENDING_REGISTER = %08X\n", *interrupt_pending_register);
    printf("INTERRUPT_FORCE_REGISTER = %08X\n", *interrupt_force_register);
    printf("INTERRUPT_MASK_REGISTER = %08X\n", *interrupt_mask_register);
}

void print_interrupt_counters() {
    uint32_t i;

    for (i=1; i < INTERRUPT_NUMBER ; i++) {
        printf("IRQ%d = %d\t", i, interrupt_counter[i]);
    }
    printf("\n");
}
```

Les interruptions et les exceptions

```
void interrupt_handler(int irq) {  
    interrupt_counter[irq]++;  
}  
  
int main(void) {  
  
    print_interrupt_registers();  
  
    catch_interrupt((int)interrupt_handler,1);  
    catch_interrupt((int)interrupt_handler,3);  
  
    enable_interrupt(1);  
    disable_interrupt(3);  
  
    force_interrupt(1);  
    print_interrupt_counters();  
    print_interrupt_registers();  
  
    force_interrupt(3);  
    force_interrupt(1);  
    print_interrupt_counters();  
    print_interrupt_registers();  
  
    enable_interrupt(3);  
    print_interrupt_counters();  
    print_interrupt_registers();  
  
    return EXIT_SUCCESS;  
}
```

- Fonction gestionnaire d'interruptions

- Enregistrement de la fonction interrupt_handler() comme gestionnaire de l'interruption n°1.
- Enregistrement de la fonction interrupt_handler() comme gestionnaire de l'interruption n°3.
- catch_interrupt() est une fonction de la bibliothèque BSP du LEON qui permet d'installer des routines d'interruption (en modifiant la trap table)

- L'interruption n°1 est autorisée
- L'interruption n°3 est désactivée

- L'interruption n°1 est déclenchée en agissant sur le registre IFR.

Les interruptions et les exceptions

```
void interrupt_handler(int irq) {
    interrupt_counter[irq]++;
}

int main(void) {

    print_interrupt_registers();

    catch_interrupt((int)interrupt_handler,1);
    catch_interrupt((int)interrupt_handler,3);

    enable_interrupt(1);
    disable_interrupt(3);

    force_interrupt(1);
    print_interrupt_counters();
    print_interrupt_registers();

    force_interrupt(3);
    force_interrupt(1);
    print_interrupt_counters();
    print_interrupt_registers();

    enable_interrupt(3);
    print_interrupt_counters();
    print_interrupt_registers();

    return EXIT_SUCCESS;
}
```

Diagram illustrating the state of interrupt registers at different points in the program execution:

Initial State:

```
INTERRUPT_PENDING_REGISTER = 00000000
INTERRUPT_FORCE_REGISTER = 00000000
INTERRUPT_MASK_REGISTER = 00000000

IRQ1 = 1  IRQ2 = 0  IRQ3 = 0  IRQ4 = 0
IRQ5 = 0  IRQ6 = 0  IRQ7 = 0  IRQ8 = 0
IRQ9 = 0  IRQ10 = 0  IRQ11 = 0  IRQ12 = 0
IRQ13 = 0  IRQ14 = 0  IRQ15 = 0
```

After `enable_interrupt(1)` and `disable_interrupt(3)`:

```
INTERRUPT_PENDING_REGISTER = 00000000
INTERRUPT_FORCE_REGISTER = 00000000
INTERRUPT_MASK_REGISTER = 00000002

IRQ1 = 2  IRQ2 = 0  IRQ3 = 0  IRQ4 = 0
IRQ5 = 0  IRQ6 = 0  IRQ7 = 0  IRQ8 = 0
IRQ9 = 0  IRQ10 = 0  IRQ11 = 0  IRQ12 = 0
IRQ13 = 0  IRQ14 = 0  IRQ15 = 0
```

After `force_interrupt(1)` and `force_interrupt(3)`:

```
INTERRUPT_PENDING_REGISTER = 00000000
INTERRUPT_FORCE_REGISTER = 00000008
INTERRUPT_MASK_REGISTER = 00000002

IRQ1 = 2  IRQ2 = 0  IRQ3 = 1  IRQ4 = 0
IRQ5 = 0  IRQ6 = 0  IRQ7 = 0  IRQ8 = 0
IRQ9 = 0  IRQ10 = 0  IRQ11 = 0  IRQ12 = 0
IRQ13 = 0  IRQ14 = 0  IRQ15 = 0
```

After `enable_interrupt(3)`:

```
INTERRUPT_PENDING_REGISTER = 00000000
INTERRUPT_FORCE_REGISTER = 00000000
INTERRUPT_MASK_REGISTER = 0000000A
```

Les interruptions et les exceptions

- Note : dans l'exemple précédent, on aurait pu utiliser le registre « pending » (IPR) à la place du registre « force » (IFR) : le résultat aurait été le même.
 - Le registre « pending » sert pour les interruptions déclenchées par une source externe.

Les interruptions et les exceptions

Exercice - Enoncé

- Que produit comme sortie le programme suivant ?

```
int main(void) {
    catch_interrupt((int)interrupt_handler,4);
    catch_interrupt((int)interrupt_handler,5);
    catch_interrupt((int)interrupt_handler,8);

    enable_interrupt(1);
    enable_interrupt(5);
    disable_interrupt(4);
    disable_interrupt(8);

    force_interrupt(1);
    force_interrupt(5);
    print_interrupt_counters();
    print_interrupt_registers();

    force_interrupt(4);
    force_interrupt(8);
    print_interrupt_counters();
    print_interrupt_registers();

    *interrupt_pending_register = * interrupt_pending_register | (1 << 5);
    enable_interrupt(8);
    print_interrupt_counters();
    print_interrupt_registers();

    return EXIT_SUCCESS;
}
```

Les interruptions et les exceptions

Exercice - Solution

```
IRQ1 = 0  IRQ2 = 0  IRQ3 = 0  IRQ4 = 0
IRQ5 = 1  IRQ6 = 0  IRQ7 = 0  IRQ8 = 0
IRQ9 = 0  IRQ10 = 0 IRQ11 = 0 IRQ12 = 0
IRQ13 = 0 IRQ14 = 0 IRQ15 = 0
```

```
INTERRUPT_PENDING_REGISTER = 00000000
INTERRUPT_FORCE_REGISTER = 00000000
INTERRUPT_MASK_REGISTER = 00000022
```

```
IRQ1 = 0  IRQ2 = 0  IRQ3 = 0  IRQ4 = 0
IRQ5 = 1  IRQ6 = 0  IRQ7 = 0  IRQ8 = 0
IRQ9 = 0  IRQ10 = 0 IRQ11 = 0 IRQ12 = 0
IRQ13 = 0 IRQ14 = 0 IRQ15 = 0
```

```
INTERRUPT_PENDING_REGISTER = 00000000
INTERRUPT_FORCE_REGISTER = 00000110
INTERRUPT_MASK_REGISTER = 00000022
```

```
IRQ1 = 0  IRQ2 = 0  IRQ3 = 0  IRQ4 = 0
IRQ5 = 2  IRQ6 = 0  IRQ7 = 0  IRQ8 = 1
IRQ9 = 0  IRQ10 = 0 IRQ11 = 0 IRQ12 = 0
IRQ13 = 0 IRQ14 = 0 IRQ15 = 0
```

```
INTERRUPT_PENDING_REGISTER = 00000000
INTERRUPT_FORCE_REGISTER = 00000010
INTERRUPT_MASK_REGISTER = 00000122
```

L'interface logiciel / matériel

1. Préambule
2. Architecture des processeurs
3. Les registres
4. Les exceptions et les interruptions
5. La mémoire (types de mémoire physique, organisation de la mémoire, DMA, mémoire cache, MMU, endianisme)
6. L'accès aux périphériques

Types de mémoire

■ SRAM = Static Random Access Memory

- Mémoire vive utilisant des bascules pour mémoriser les données
- Temps d'accès faible
- Pas besoin de rafraîchir périodiquement son contenu
- Interfaçage simple
- Utilisée en interne dans les processeurs (caches, registres, mémoires on-chip) et comme mémoire principale pour l'exécution du code

■ SDRAM = Synchronous Dynamic Random Access Memory

- Mémoire vive utilisant des transistors et des pico-condensateurs pour mémoriser les données
- Temps d'accès plus longs que ceux de la SRAM
- Densité plus élevée que la SRAM pour une consommation plus faible
- Besoin de rafraîchir périodiquement son contenu
- Interfaçage plus complexe

Types de mémoire

- PROM = Programmable Read Only Memory
 - Mémoire morte : cellules mémoires = fusibles détruits par claquement diélectrique
 - Non reprogrammable
- EEPROM = Electrically-Erasable Programmable Read-Only Memory
 - Peut être effacée par une simple courant électrique et donc peut être reprogrammée
 - Permet d'enregistrer des informations (code, données) qui ne doivent pas être perdues quand l'appareil est hors tension

Types de mémoire

■ FLASH

- Mémoire de masse à semi-conducteurs ré-inscriptibles,
- Mémoire possédant les caractéristiques d'une mémoire vive mais dont les données ne disparaissent pas lors d'une mise hors tension
- Temps d'accès environ 7 fois supérieurs à celui de la SRAM

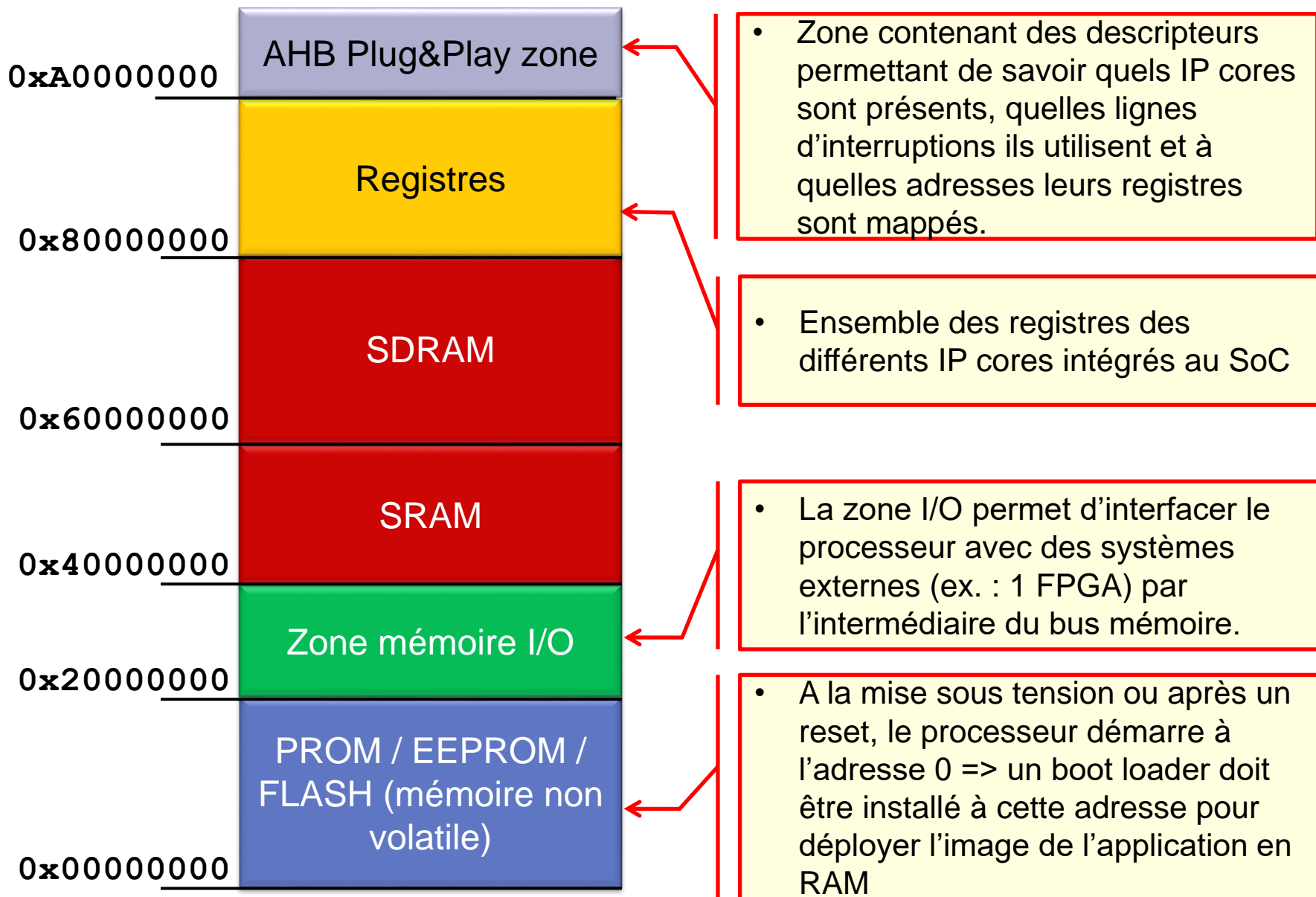
■ MRAM = Magnetic Random Access Memory

- Mémoire non volatile de type magnétique.
- Les données ne sont pas stockées sous forme d'une charge électrique mais d'une orientation magnétique : ceci confère à la MRAM un fort niveau d'immunité face aux SEU.
- Combine les performances d'une mémoire vive (type SRAM), et la non-volatilité d'une EEPROM ou d'une FLASH.

Types de mémoire

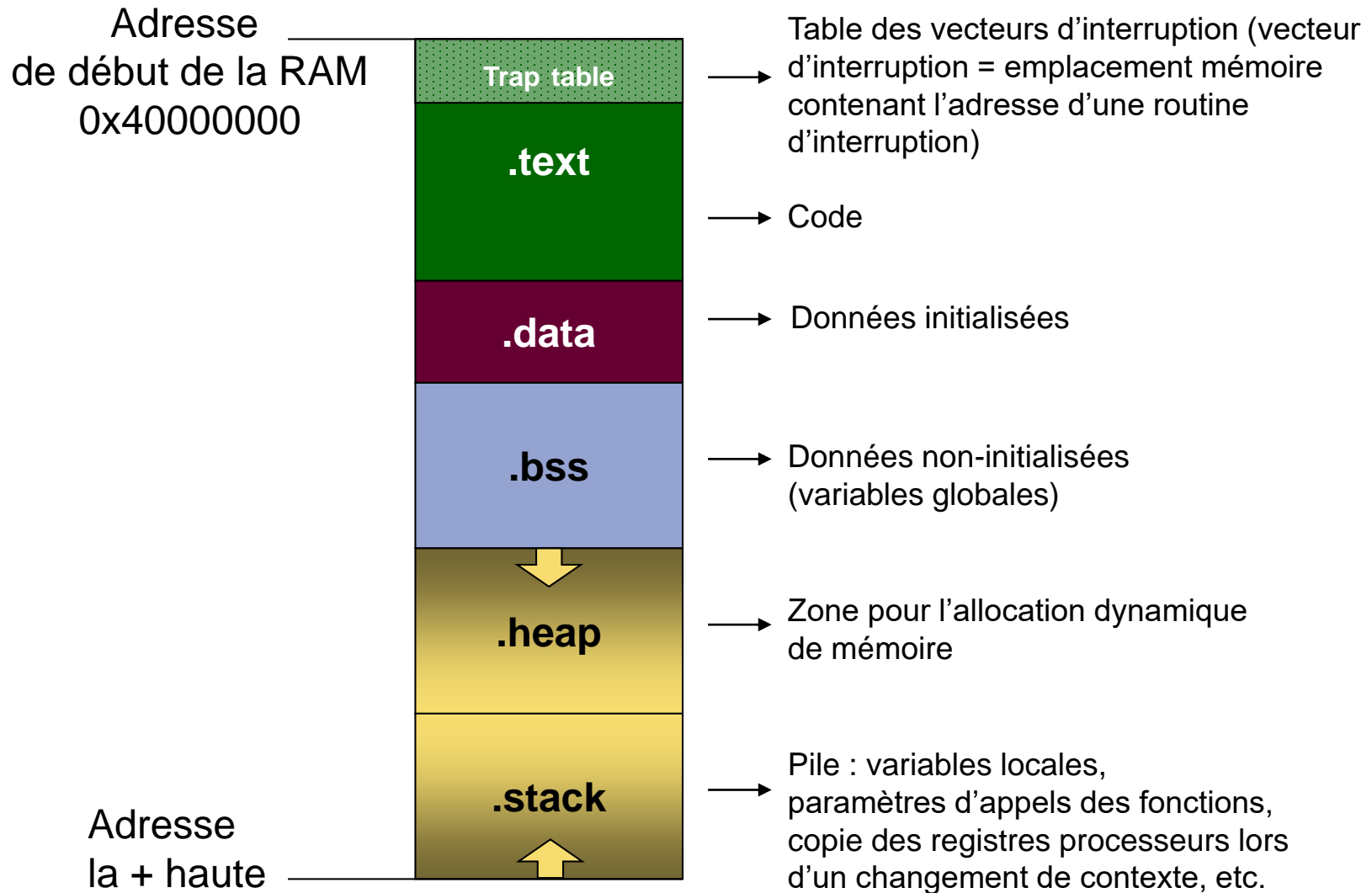
- Un processeur peut être interfacé à différents types de mémoire :
 - SRAM pour l'exécution du code
 - SDRAM pour stocker des volumes de données importants
 - FLASH ou EEPROM (mémoire non volatile) pour stocker l'image de l'exécutable
 - ...
- Ces différentes mémoires sont « mappées » sur des espaces mémoire différents ce qui permet au logiciel d'y accéder sans conflit.

Organisation de l'espace mémoire



Organisation de l'espace mémoire

Zoom sur l'organisation de la RAM

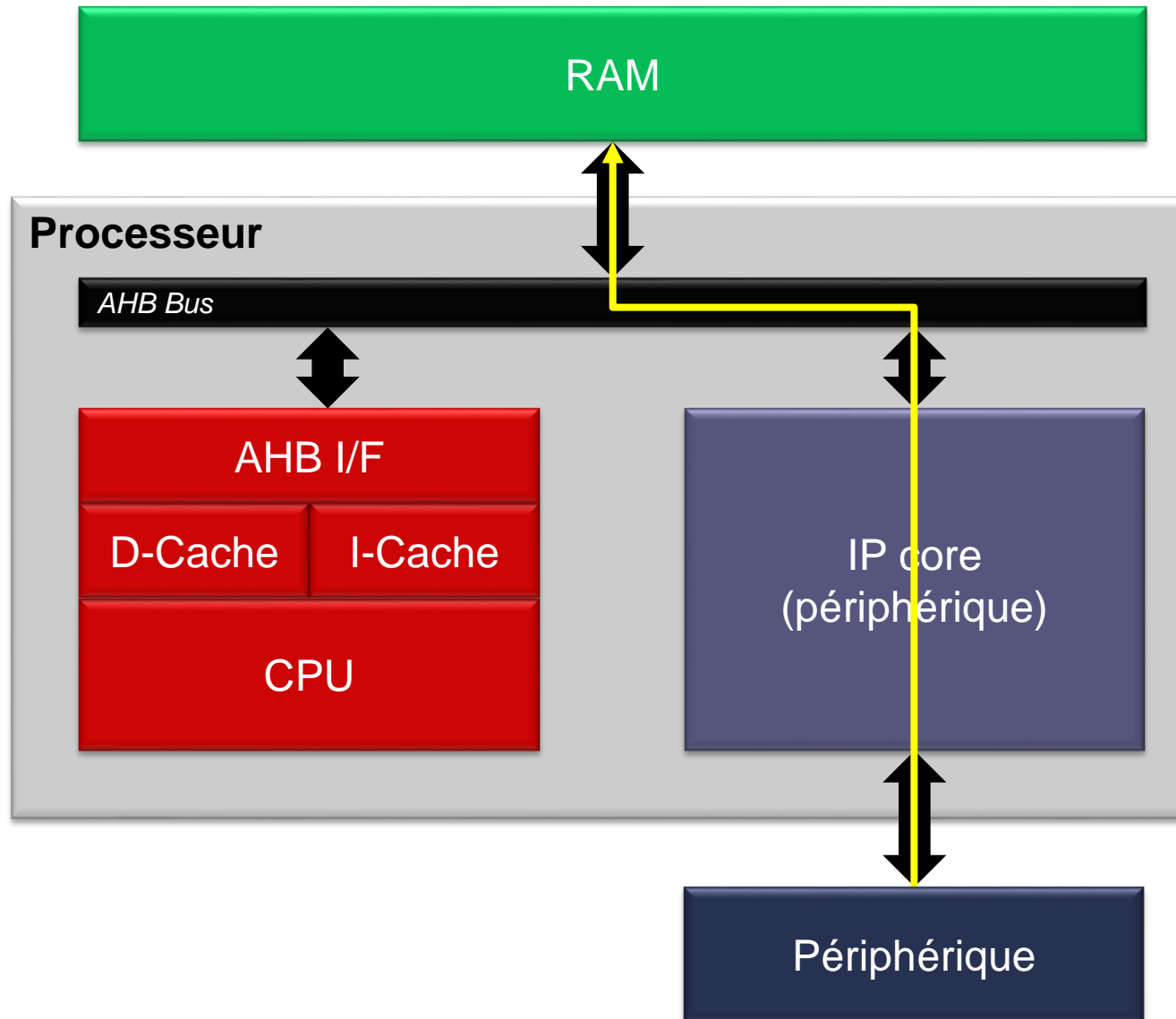


DMA – Direct Memory Access

■ Technologie DMA = Direct Memory Access

- Avec la technologie DMA, les données sont transmises par un périphérique vers la mémoire du processeur (RAM) sans intervention du processeur.
- Contrôleur DMA généralement embarqué au sein du processeur.
- Concrètement, cela signifie qu'on n'a pas besoin d'avoir un morceau de logiciel implémentant une boucle de lecture des données reçues via une mémoire dédiée à l'interface (registres du périphérique, FIFO, DPRAM, mémoire mappée, ...) → ceci améliore considérablement les performances des applications développées

DMA – Direct Memory Access



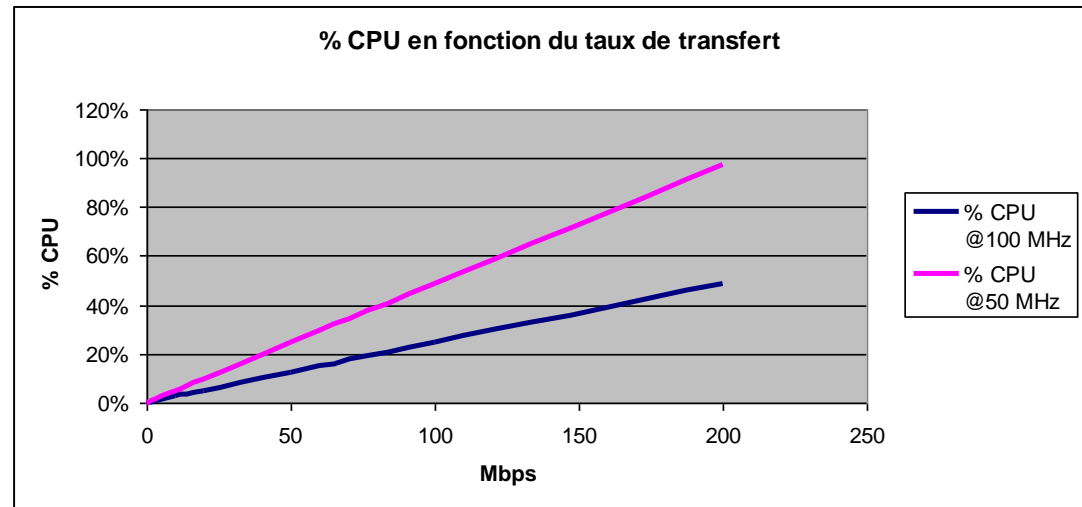
DMA – Direct Memory Access

■ Technologie DMA = Direct Memory Access

- Exemple : $128 \text{ Mbps} = 128 \text{ Mbits/s} = 16 \text{ MBytes / s} =$
 - 16 millions d'instructions READ par seconde (accès au niveau byte) + 16 millions d'instructions WRITE par seconde + instructions de gestion de la boucle.
- Fonctionnement identique pour la transmission.
- Sans DMA, le transfert des données représente un coût CPU directement proportionnel au taux de transfert.

DMA – Direct Memory Access

- Le graphe ci-dessous donne le taux d'occupation d'un processeur LEON en fonction du débit de données (entrant ou sortant).
 - Le processeur n'est pas couplé à un contrôleur DMA.
 - Il communique avec son périphérique via une mémoire partagée de 10 kO ; les données sont lues en utilisant des instructions de lecture / écriture 64 bits.



- Dans l'exemple ci-dessus, si l'interface était une FIFO 8 bits, le taux d'occupation CPU serait 8 fois plus important pour le même taux de transfert !

Mémoires cache

- Mémoires de petite taille (quelques kO), spécialisées (données ou instructions), rapides et proches du processeur dans lesquelles sont stockées le code ou les données fréquemment utilisées par le processeur
 - Principe de localité spatiale (ex. : parcours d'un tableau)
 - Principe de localité temporelle (ex. : boucle de traitement)
- Améliorent grandement les temps d'accès moyens (facteur 5).
- Quand le processeur veut lire une donnée ou instructions, il s'adresse au cache
 - Si la donnée est présente, on parle de succès de cache (cache hit)
 - Si la donnée est absente, on parle de défaut de cache (cache miss)

Mémoires cache

- Les caches sont organisés en lignes (1 ligne = 1 bloc) contenant plusieurs mots de 32 bits contigus.
 - Ligne = plus petit élément de données qui peut être transféré entre la mémoire cache et la mémoire de niveau supérieur
 - Mot = plus petit élément de données qui peut être transféré entre le processeur et la mémoire cache.
- Les caches peuvent être activés / désactivés, gelés ou encore vidés (flush) via des registres dédiés du contrôleur de cache.
- Sur certains processeurs (multi-core), plusieurs niveaux de cache :
 - Cache L1 = cache propre à chaque CPU
 - Cache L2 = cache de taille plus importante partagé par les CPU et localisé entre le bus AMBA et la mémoire RAM externe

Mémoire cache

Cohérence du cache

- Avec le mécanisme DMA, l'écriture dans la mémoire du processeur n'est pas réalisé par le processeur lui-même et le mécanisme de cache est dès lors contourné.
- Par exemple, si à un instant donné, le cache données du processeur contient le contenu a_1 de l'adresse A et qu'un périphérique (IP core) accède en écriture à l'adresse A en écrivant une valeur a_2 , alors les lignes du cache correspondant à l'adresse A ne sont pas invalidées et le processeur quand il va accéder à l'adresse A va lire la valeur se trouvant dans le cache (a_1) et pas la valeur contenue dans la mémoire (a_2).

Mémoire cache

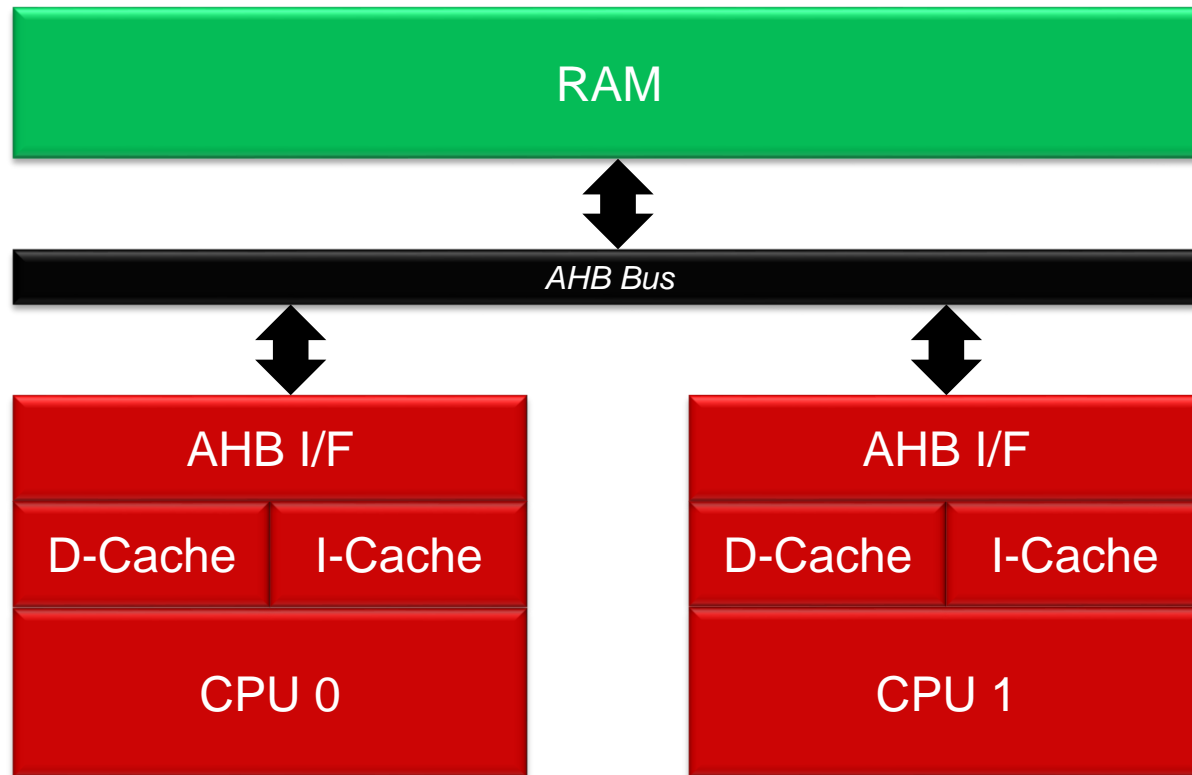
Cohérence du cache

- Certains processeurs proposent des mécanismes dit de « bus-snooping » permettant d'assurer de façon transparente pour le logiciel la cohérence du cache.
 - Monitoring des accès en écriture sur le bus AMBA
 - Si un accès en écriture est réalisé à une adresse contenue dans le cache, alors la ligne de cache correspondante est marquée comme invalide.
 - Au prochain accès en lecture à cette adresse, le processeur sera forcé d'aller lire la donnée dans la mémoire et non dans le cache.
- Sans ces mécanismes de « bus snooping », le logiciel doit gérer la cohérence du cache de données en commandant par exemple un vidage du cache (flush) quand un transfert DMA a été signalé via une interruption.

Mémoire cache

Cohérence du cache

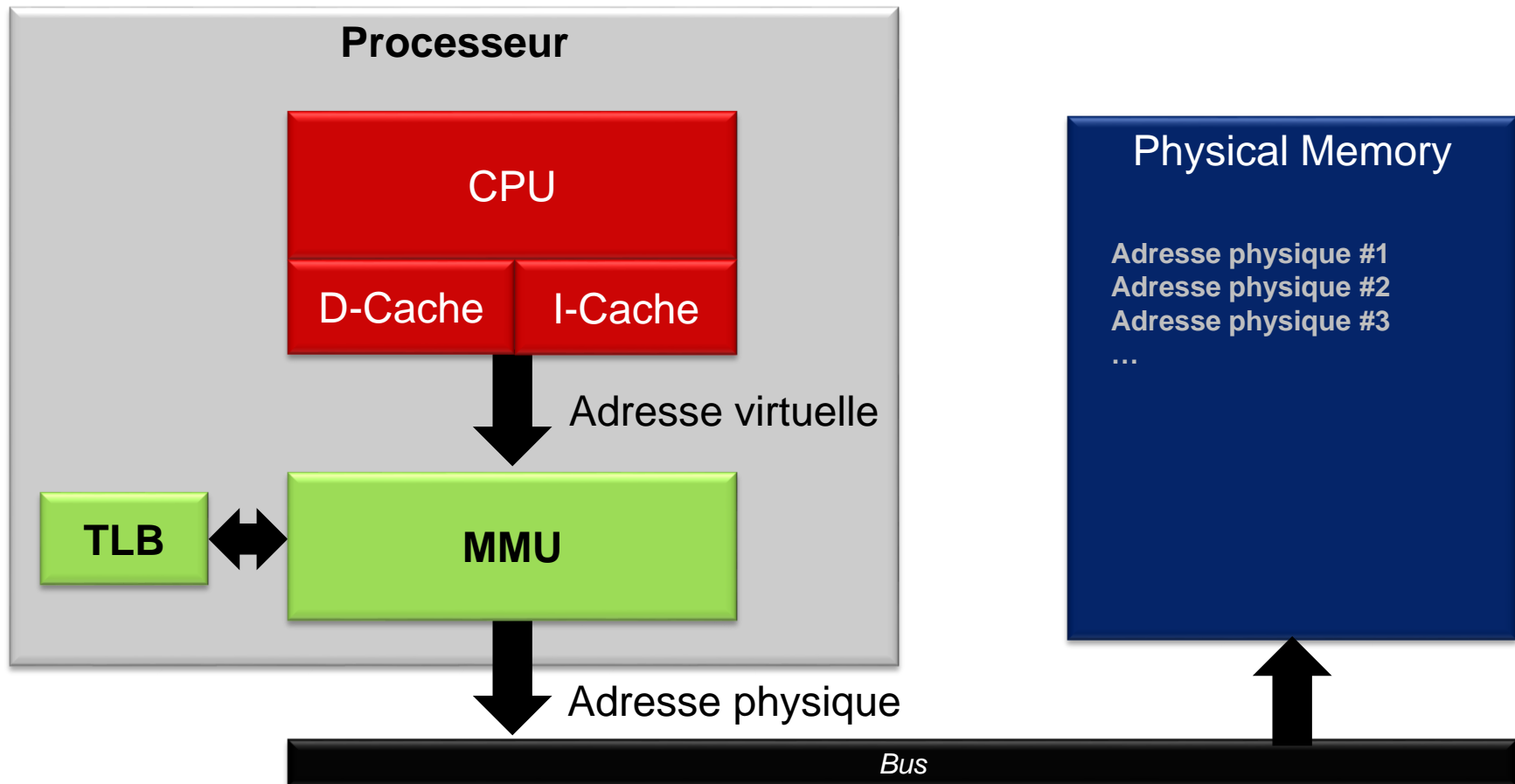
- Ces problèmes de cohérence du cache se posent aussi dans le cas des architectures multi-core.
- Le mécanisme du bus-snooping permet là aussi d'assurer la cohérence des caches données de chaque cœur de processeur.



MMU

Memory Management Unit

- Unité permettant de traduire des adresses virtuelles en adresses physiques
 - Cas d'utilisation : OS multi-processus, OS multi-core, virtualisation, ...



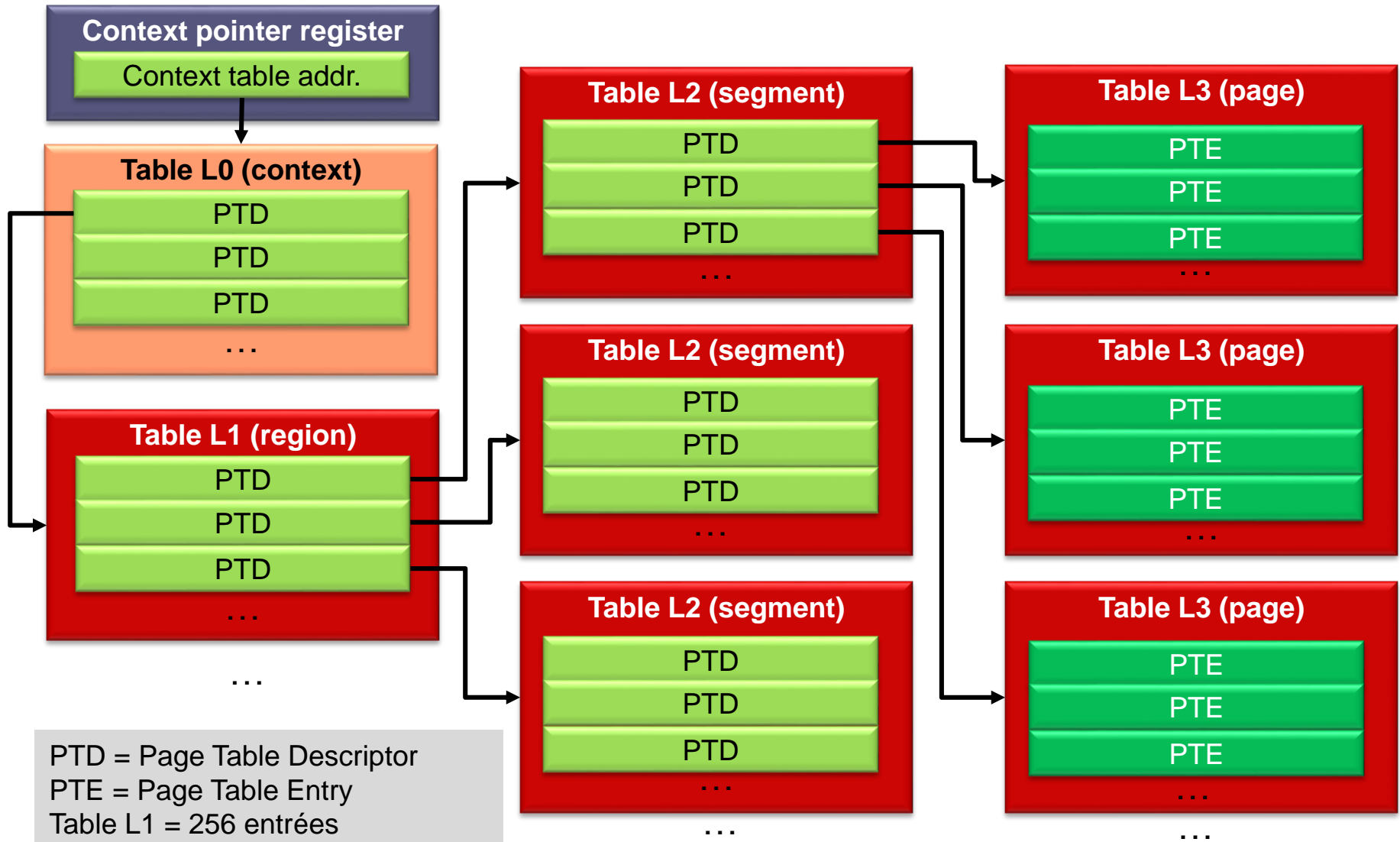
MMU

Memory Management Unit

- Le MMU du SPARC utilise des tables de translation organisées en 4 étages :
 - Table L3 : chaque entrée définit une page (4 kB)
 - Table L2 : chaque entrée définit un segment ($64 * 4 \text{ kB} = 256 \text{ kB}$)
 - Table L1 : chaque entrée définit une région ($64 * 256 \text{ kB} = 16 \text{ MB}$)
 - Table L0 : chaque entrée définit un contexte ($256 * 16 \text{ MB} = 4 \text{ GB}$)
- Ces tables de translation sont stockées en RAM.
- Afin d'accélérer les mécanismes de translation, un mécanisme de cache est intégré au MMU : le TLB (Translation Lookaside Buffer).

MMU - Memory Management Unit

Organisation des tables de translation



PTD = Page Table Descriptor
PTE = Page Table Entry
Table L1 = 256 entrées
Table L2 = 64 entrées
Table L3 = 64 entrées

MMU - Memory Management Unit

Organisation des tables de translation

■ Taille des tables :

- Taille de la table L0 = 256 entrées PTD soit $256 * 32 \text{ bits} = 1 \text{ kB}$
- Taille des tables L1 = 256 entrées PTD soit $256 * 32 \text{ bits} = 1 \text{ kB}$
- Taille des tables L2 = 64 entrées PTD soit $64 * 32 \text{ bits} = 256 \text{ B}$
- Taille des tables L3 = 64 entrées PTE soit $64 * 32 \text{ bits} = 256 \text{ B}$

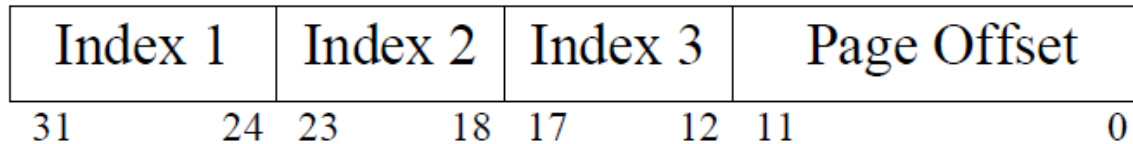
■ Taille des pages = $4 \text{ kB} = 2^{12} \text{ bits}$

■ Pour chaque page, il y a une entrée PTE

MMU

Memory Management Unit

- Une adresse virtuelle est composée ainsi :



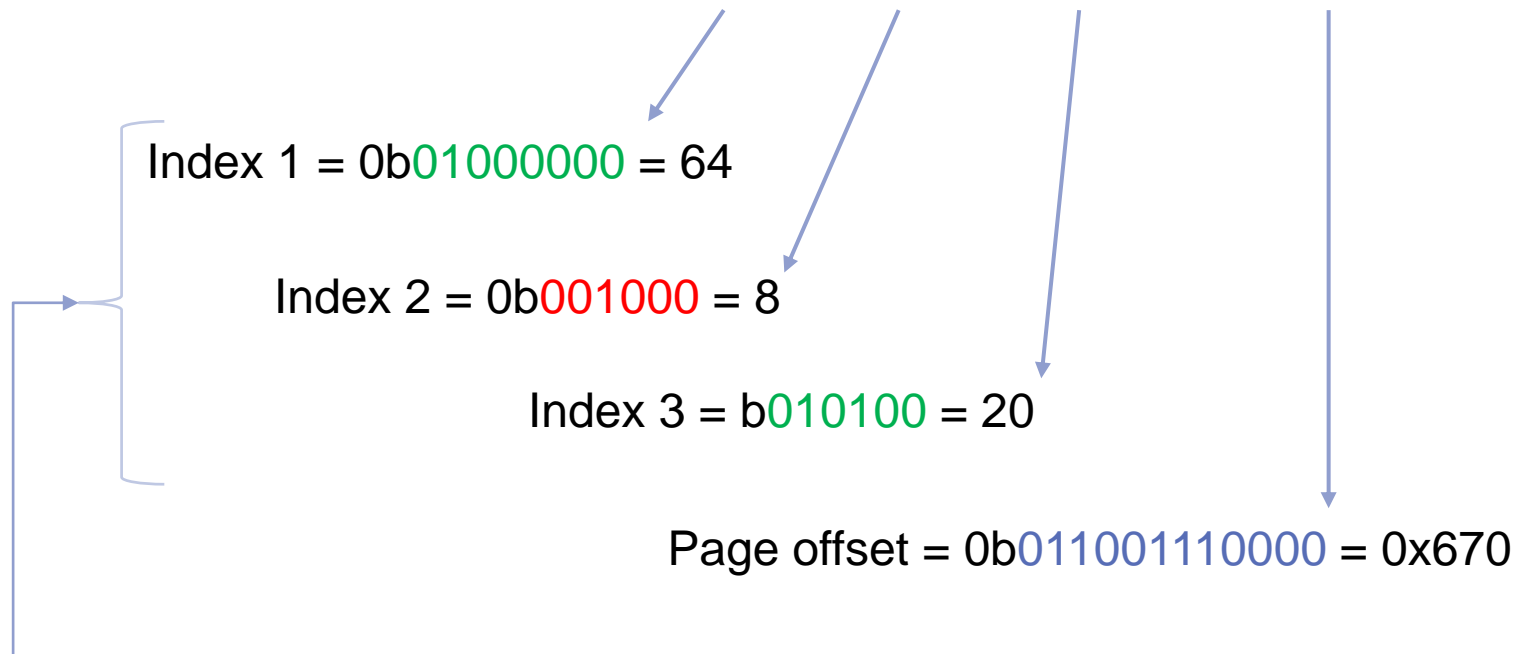
- Chaque champ index fournit un déplacement dans la table de niveau correspondant
 - Index 1 codé sur 8 bits soit 256 entrées (taille table L1)
 - Index 2 codé sur 6 bits soit 64 entrées (taille table L2)
 - Index 3 codé sur 6 bits soit 64 entrées (taille table L3)
- Le champ Page Offset est codé sur 12 bits, ce qui correspond à la taille de 4 kB : ce champ est utilisé tel quel dans le mécanisme de translation => une adresse virtuelle et une adresse physique ont les 12 bits de poids faibles identiques.

MMU

Memory Management Unit

■ Exemple :

- Soit l'adresse virtuelle suivante : 0x40214670
- 0x40214670 = 0b01000000001000010100011001110000



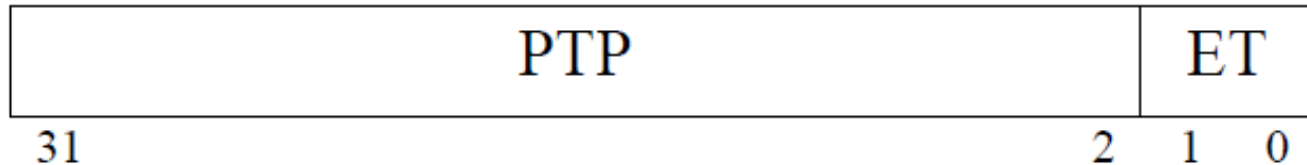
Définit l'adresse de début de la page virtuelle de 4 kB dans laquelle se trouve l'adresse 0x40214670 :

0b01000000001000010100000000000000 = 0x40214000

MMU

Memory Management Unit

■ Page Table Descriptor (PTD)

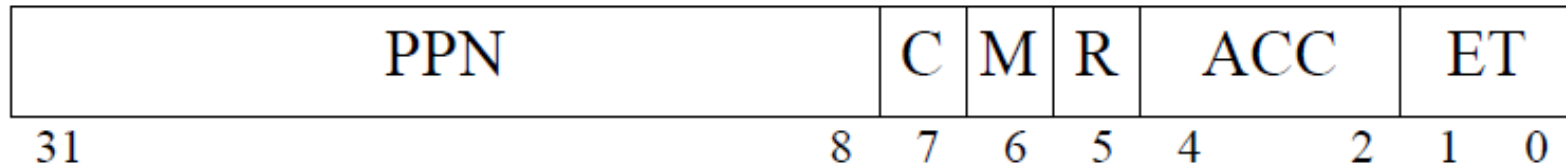


- ET = Entry Type = 1 pour une PTD
- PTP = adresse physique de la table de page pointée
 - bits 32 à 6 de cette adresse physique
 - les 6 bits de poids faibles sont ignorés puisque la table a une taille de 2^6 entrées = 64 entrées

MMU

Memory Management Unit

■ Page Table Entry



- ET = Entry Type = 2 pour une PTE
- ACC = Permission d'accès (3 = Read/Write/Execute)
- R = positionné à 1 par le MMU quand la page est accédée
- M = positionné à 1 par le MMU quand la page est accédée en écriture
- C = si le bit est positionné, alors la page est « cacheable » par le cache donnée ou le cache instruction => pour les zone I/O, mettre C à 0
- PPN = Physical Page Number = bits 32 à 12 de l'adresse physique utilisée pour la translation

MMU

Memory Management Unit

- Supposons que l'on souhaite réaliser la translation suivante :

Page	Adresse virtuelle	Adresse Physique
N°1	0x40214000-0x40214FFF	0x40314000-0x40314FFF

- L'adresse virtuelle de début de page 0x40214000 se décompose en (voir slides précédents) :
 - Index 1 = 64
 - Index 2 = 8
 - Index 3 = 20
- On a besoin de définir 4 tables (L0, L1, L2, L3) :
 - On positionne la table L0 à l'adr. 0x40100000
 - On positionne la table L1 à l'adr. $0x40100000 + 1\text{kB} = 0x40100400$
 - On positionne la table L2 à l'adr. $0x40100400 + 1\text{kB} = 0x40100800$
 - On positionne la table L3 à l'adr. $0x40100800 + 256\text{ B} = 0x40100900$

MMU

Memory Management Unit

■ Programmation du contenu des tables

■ Table L0 :

- A l'adr. phys. $0x40100000 + 0$ (context 0), on doit écrire une entrée PTD pointant sur l'adresse de la table L1, soit la valeur $((0x40100400 \gg 6) \ll 2) + 1 = 0x4010041$

■ Table L1 :

- A l'adr. phys. $0x40100400 + \text{index } 1 = 0x40100400 + 64 \cdot 4 = 0x40100500$, on doit écrire une entrée PTD pointant sur l'adresse de la table L2, soit la valeur $((0x40100800 \gg 6) \ll 2) + 1 = 0x4010081$

■ Table L2 :

- A l'adr. phys. $0x40100800 + \text{index } 2 = 0x40100800 + 8 \cdot 4 = 0x40100820$, on doit écrire une entrée PTD pointant sur l'adresse de la table L3, soit la valeur $((0x40100900 \gg 6) \ll 2) + 1 = 0x4010091$



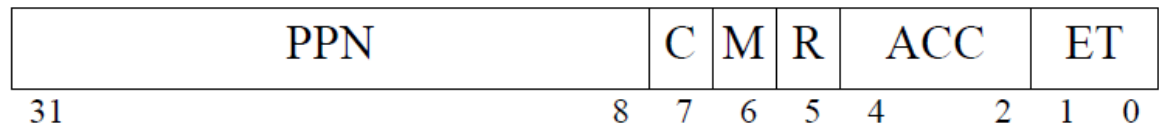
MMU

Memory Management Unit

■ Programmation du contenu des tables

■ Table L3 :

- A l'adr. phys. $0x40100900 + \text{index } 3 = 0x40100900 + 20 \times 4 = 0x40100950$, on doit écrire une entrée PTE contenant :
 - le PPN (Physical Page Number), c'est-à-dire les bits 32 à 12 de l'adresse physique utilisée pour la translation, soit la valeur $\text{PPN} = 0x40314000 \gg 12$
 - $C = 0$
 - $M = 0$
 - $R = 0$
 - $\text{ACC} = 3$
 - $\text{ET} = 2$
 - $\text{PTE} = \text{PPN} \ll 8 + C \ll 7 + M \ll 6 + R \ll 5 + \text{ACC} \ll 2 + \text{ET} = ((0x40314000 \gg 12) \ll 8) + (3 \ll 2) + 2 = 0x403140E$



■ Programmation du registre de contexte :

- Il doit pointer sur la table L0 => sa valeur est :
 $((0x40100000 \gg 6) \ll 2) = 0x4010000$

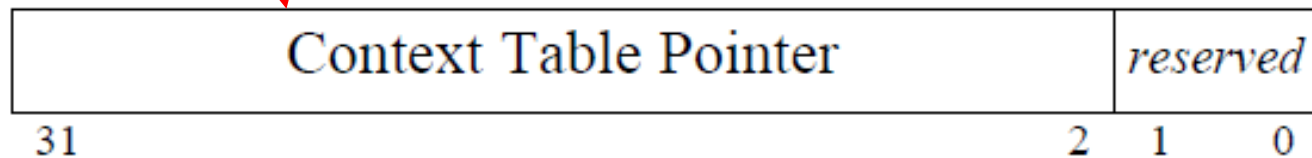
MMU

Memory Management Unit

■ Registres de programmation (Sparc V8) :

Table 20. MMU registers (ASI = 0x19)

Address	Register
0x000	MMU control register
0x100	Context pointer register
0x200	Context register
0x300	Fault status register
0x400	Fault address register



MMU

Test sur cible LEON avec TSIM

- Le script TSIM ci-dessous permet de programmer le MMU du LEON3 selon l'exemple présenté précédemment

```
#set table L0 (context)
wmem 0x40100000 0x4010041
#set table L1 (region)
wmem 0x40100500 0x4010081
#set table L2 (segment)
wmem 0x40100820 0x4010091
#set table L3 (table) : 0x40214000 (virtual) -> 0x40314000 (physical)
wmem 0x40100950 0x403140E

#set Context pointer register
xwmem 0x19 0x100 0x4010000
#set Context register (first L1 table)
xwmem 0x19 0x200 0
#set Control register (enable MMU)
xwmem 0x19 0x000 1

walk 0x40214000
vmem 0x40214000
```

- Programmation des registres du MMU avec la commande xwmem permettant d'accéder aux registres mappés sur l'espace ASI
- La command 'walk' permet de vérifier le décodage d'une adresse virtuelle
- La command 'vmem' permet d'afficher le contenu d'une adresse virtuelle

MMU

Test sur cible LEON avec TSIM

```
system frequency: 50.000 MHz
serial port A on stdin/stdout
allocated 4096 KiB SRAM memory, in 1 bank
allocated 32 MiB SDRAM memory, in 1 bank
allocated 2048 KiB ROM memory
icache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
dcache: 1 * 4 KiB, 16 bytes/line (4 KiB total)
tsim> wmem 0x40100000 0x4010041
tsim> wmem 0x40100500 0x4010081
tsim> wmem 0x40100820 0x4010091
tsim> wmem 0x40100950 0x403140E
tsim>
tsim> xwmem 0x19 0x100 0x4010000
tsim> xwmem 0x19 0x200 0
tsim> xwmem 0x19 0x000 1
tsim>
tsim> walk 0x40214000
Tablewalk: (40) (8) (14), dcache,write:n,su:n
+ctx(0):40100000 ctx->4010041
+region(40):40100500 region->4010081
+segment(8):40100820 segment->4010091
+page(14):40100950 page->403140e
= 40314000 (pte:403140e,cachable:no)
=>0x40314000
tsim> vmem 0x40214000
```

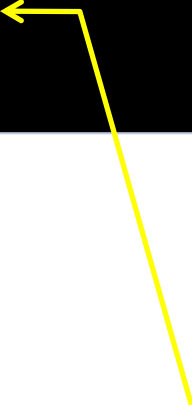
40314000	00000000	00000000	00000000	00000000
40314010	00000000	00000000	00000000	00000000
40314020	00000000	00000000	00000000	00000000
40314030	00000000	00000000	00000000	00000000

- La translation 0x40214000 (adr. Virtuelle) → 0x40314000 (adr. Physique) a bien eu lieu

MMU

Test sur cible LEON avec TSIM

```
tsim> walk 0x40215000
Tablewalk: (40) (8) (15), dcache,write:n,su:n
+ctx(0):40100000 ctx->4010041
+region(40):40100500 region->4010081
+segment(8):40100820 segment->4010091
+page(15):40100954 page->0
*!Page fault
=>0x0
```



- La translation de l'adresse virtuelle 0x40215000 échoue (page fault) car l'entrée dans la table L2 correspondant à l'index3 0x15 = 21 n'a pas été programmée

MMU

Exercice 1

■ Enoncé exercice MMU 1 :

1. Comment faut-il programmer le MMU pour que la translation de l'adresse virtuelle 0x40215000 donne l'adresse physique 0x40315000 ?
2. Quelle serait alors l'adresse physique correspondant à l'adresse virtuelle 0x40215F00 ?
3. Quelle serait alors l'adresse physique correspondant à l'adresse virtuelle 0x402160A0 ?
4. Comment faut-il programmer le MMU pour que la translation de l'adresse virtuelle 0x403DB000 donne l'adresse physique 0x40010000 ?
5. Comment faut-il programmer le MMU pour que la translation de l'adresse virtuelle 0x60000000 donne l'adresse physique 0x40000000 ?

MMU

Exercice 2

■ Enoncé exercice MMU 2 :

1. A quelle adresse physique correspond l'adresse virtuelle 0x40214000 quand on programme le MMU de cette façon :

```
wmem 0x40100000 0x4010041
wmem 0x40100500 0x4010081
wmem 0x40100820 0x4010091
wmem 0x40100950 0x418140E

xwmem 0x19 0x100 0x4010000
xwmem 0x19 0x200 0
xwmem 0x19 0x000 1
```

MMU

Exercice

■ Solution exercice MMU 1 :

1. Comment faut-il programmer le MMU pour que la translation de l'adresse virtuelle **0x40215000** donne l'adresse physique **0x40315000** ?
 - Calcul des index :
 - $\text{bin}(0x40215000) = '0b1000000\ 001000\ 010101\ 0000000000000'$
 - $\text{Index1} = 0b1000000 = 64$
 - $\text{Index2} = 0b001000 = 8$
 - $\text{Index3} = 0b010101 = 21$
 - Adresse des tables (arbitraire)
 - L0 à 0x40100000
 - L1 à 0x40100400
 - L2 à 0x40100800
 - L3 à 0x40100900
 - Programmation du contenu des tables
 - Table L0 :
 - A l'adr. phys. $0x40100000 + 0$ (context 0), on doit écrire une entrée PTD pointant sur l'adresse de la table L1, soit la valeur $((0x40100400 \gg 6) \ll 2) + 1 = 0x4010041$
 - Table L1 :
 - A l'adr. phys. $0x40100400 + \text{index } 1 = 0x40100400 + 64*4 = 0x40100500$, on doit écrire une entrée PTD pointant sur l'adresse de la table L2, soit la valeur $((0x40100800 \gg 6) \ll 2) + 1 = 0x4010081$
 - Table L2 :
 - A l'adr. phys. $0x40100800 + \text{index } 2 = 0x40100800 + 8*4 = 0x40100820$, on doit écrire une entrée PTD pointant sur l'adresse de la table L3, soit la valeur $((0x40100900 \gg 6) \ll 2) + 1 = 0x4010091$
 - Table L3 :
 - A l'adr. phys. $0x40100900 + \text{index } 3 = 0x40100900 + 21*4 = 0x40100954$, on doit écrire une entrée PTE contenant :
 - le PPN (Physical Page Number), c'est-à-dire les bits 32 à 12 de l'adresse physique utilisée pour la translation, soit la valeur $\text{PPN} = 0x40315000 \gg 12$
 - $C = 0, M = 0, R = 0, \text{ACC} = 3, \text{ET} = 2$
 - $\text{PTE} = \text{PPN} \ll 8 + C \ll 7 + M \ll 6 + R \ll 5 + \text{ACC} \ll 2 + \text{ET} = ((0x40315000 \gg 12) \ll 8) + (3 \ll 2) + 2 = 0x403150E$
 - Programmation du registre de contexte :
 - Il doit pointer sur la table L0 => sa valeur est : $((0x40100000 \gg 6) \ll 2) = 0x4010000$

MMU

Exercice

■ Solution exercice MMU 1 (suite) :

2. Quelle serait alors l'adresse physique correspondant à l'adresse virtuelle 0x40215F00 ?
 - L'adresse physique correspondant à l'adresse virtuelle 0x40215F00 est 0x40315F00 (on est dans la même page que celle de l'adresse 0x40215000 puisque les 20 bits de poids forts sont identiques).
3. Quelle serait alors l'adresse physique correspondant à l'adresse virtuelle 0x402160A0 ?
 - L'adresse virtuelle 0x402160A0 n'est pas définie dans les tables du MMU, donc aucune adresse physique ne correspond à cette adresse virtuelle. Y accéder provoquerait une erreur « Page fault ».

MMU

Exercice

■ Solution exercice MMU 1 (suite):

4. Comment faut-il programmer le MMU pour que la translation de l'adresse virtuelle **0x403DB000** donne l'adresse physique **0x40010000** ?

- Calcul des index :
 - $\text{bin}(0x403DB000) = '0b10000000\ 001111\ 011011\ 0000000000000'$
 - $\text{Index1} = 0b10000000 = 64$
 - $\text{Index2} = 0b001111 = 15$
 - $\text{Index3} = 0b011011 = 27$
- Adresse des tables (arbitraire)
 - L0 à 0xA0000000
 - L1 à 0xA0000400
 - L2 à 0xA0000800
 - L3 à 0xA0000900
- Programmation du contenu des tables
 - Table L0 :
 - A l'adr. phys. $0xA0000000 + 0$ (context 0), on doit écrire une entrée PTD pointant sur l'adresse de la table L1, soit la valeur $((0xA0000400 \gg 6) \ll 2) + 1 = 0xA000041$
 - Table L1 :
 - A l'adr. phys. $0xA0000400 + \text{index } 1 = 0xA0000400 + 64*4 = 0xA0000500$, on doit écrire une entrée PTD pointant sur l'adresse de la table L2, soit la valeur $((0xA0000800 \gg 6) \ll 2) + 1 = 0xA000081$
 - Table L2 :
 - A l'adr. phys. $0xA0000800 + \text{index } 2 = 0xA0000800 + 15*4 = 0xA000083C$, on doit écrire une entrée PTD pointant sur l'adresse de la table L3, soit la valeur $((0xA0000900 \gg 6) \ll 2) + 1 = 0xA000091$
 - Table L3 :
 - A l'adr. phys. $0x40100900 + \text{index } 3 = 0xA0000900 + 27*4 = 0xA000096C$, on doit écrire une entrée PTE contenant :
 - le PPN (Physical Page Number), c'est-à-dire les bits 32 à 12 de l'adresse physique utilisée pour la translation, soit la valeur $\text{PPN} = 0x40010000 \gg 12$
 - $C = 0, M = 0, R = 0, \text{ACC} = 3, \text{ET} = 2$
 - $\text{PTE} = \text{PPN} \ll 8 + C \ll 7 + M \ll 6 + R \ll 5 + \text{ACC} \ll 2 + \text{ET} = ((0x40010000 \gg 12) \ll 8) + (3 \ll 2) + 2 = 0x400100E$
- Programmation du registre de contexte :
 - Il doit pointer sur la table L0 => sa valeur est : $((0xA0000000 \gg 6) \ll 2) = 0xA000000$

MMU

Exercice

■ Solution exercice MMU 1 (suite):

5. Comment faut-il programmer le MMU pour que la translation de l'adresse virtuelle **0x60000000** donne l'adresse physique **0x40000000** ?
- Calcul des index :
 - $\text{bin}(0x60000000) = '0b1100000\ 000000\ 000000\ 00000000000000'$
 - $\text{Index1} = 0b1100000 = 96$
 - $\text{Index2} = 0b000000 = 0$
 - $\text{Index3} = 0b000000 = 0$
 - Adresse des tables (arbitraire)
 - L0 à 0xA0000000
 - L1 à 0xA0000400
 - L2 à 0xA0010800
 - L3 à 0xA0010900
 - Programmation du contenu des tables
 - Table L0 :
 - A l'adr. phys. $0xA0000000 + 0$ (context 0), on doit écrire une entrée PTD pointant sur l'adresse de la table L1, soit la valeur $((0xA0000400 \gg 6) \ll 2) + 1 = 0xA000041$
 - Table L1 :
 - A l'adr. phys. $0xA0000400 + \text{index } 1 = 0xA0000400 + 96*4 = 0xA0000580$, on doit écrire une entrée PTD pointant sur l'adresse de la table L2, soit la valeur $((0xA0010800 \gg 6) \ll 2) + 1 = 0xA001081$
 - Table L2 :
 - A l'adr. phys. $0xA0010800 + \text{index } 2 = 0xA0010800 + 0*4 = 0xA0010800$, on doit écrire une entrée PTD pointant sur l'adresse de la table L3, soit la valeur $((0xA0010900 \gg 6) \ll 2) + 1 = 0xA001091$
 - Table L3 :
 - A l'adr. phys. $0xA0010900 + \text{index } 3 = 0xA0010900 + 0*4 = 0xA0010900$, on doit écrire une entrée PTE contenant :
 - le PPN (Physical Page Number), c'est-à-dire les bits 32 à 12 de l'adresse physique utilisée pour la translation, soit la valeur $\text{PPN} = 0x40000000 \gg 12$
 - $C = 0, M = 0, R = 0, \text{ACC} = 3, \text{ET} = 2$
 - $\text{PTE} = \text{PPN} \ll 8 + C \ll 7 + M \ll 6 + R \ll 5 + \text{ACC} \ll 2 + \text{ET} = ((0x40000000 \gg 12) \ll 8) + (3 \ll 2) + 2 = 0x400000E$
 - Programmation du registre de contexte :
 - Il doit pointer sur la table L0 \Rightarrow sa valeur est : $((0xA0010000 \gg 6) \ll 2) = 0xA001000$

MMU

Exercice

■ Solution exercice MMU 2 :

1. A quelle adresse physique correspond l'adresse virtuelle 0x40214000 quand on programme le MMU de cette façon :

```
wmem 0x40100000 0x4010041
wmem 0x40100500 0x4010081
wmem 0x40100820 0x4010091
wmem 0x40100950 0x418140E

xwmem 0x19 0x100 0x4010000
xwmem 0x19 0x200 0
xwmem 0x19 0x000 1
```

Réponse : 0x41814000

L'endianisme (Endianness)

- Deux façons d'organiser en mémoire ou dans une transmission les octets représentant une donnée :
 - Big endian
 - L'octet de poids fort (MSB = Most Significant byte) est stocké à l'adresse la plus petite (mémoire) ou transmis en premier (transmission).
 - Les octets suivants sont stockés / transmis dans l'ordre décroissant de leur poids.
 - L'octet de poids faible (LSB = Less Significant Byte) est stocké à l'adresse la plus grande (mémoire) ou transmis en dernier (transmission).
 - Little endian
 - L'octet de poids faible (LSB = Less Significant Byte) est stocké à l'adresse la plus petite (mémoire) ou transmis en premier (transmission).
 - Les octets suivants sont stockés / transmis dans l'ordre croissant de leur poids.
 - L'octet de poids fort (MSB = Most Significant Byte) est stocké à l'adresse la plus grande (mémoire) ou transmis en dernier (transmission).

L'endianisme (Endianness)

- Utilisation de big endian :
 - Processeurs SPARC, Motorola 68000, Freescale ColdFire, Xilinx MicroBlaze, ATMEL AVR32 (32-bit RISC microcontrôleur), ...
 - Protocoles internet : IPv4, IPv6, TCP, UDP
- Utilisation de little endian :
 - Processeurs Intel x86, AMD64 / x86-64
 - Processeurs ARM (ex. : série des STM32F3, STM32F4, ...)
- Bi-endianisme :
 - Capacité du processeur à fonctionner avec les 2 modes d'organisation mémoire
 - Configuration pouvant parfois être réalisée par logiciel au démarrage ou configuration figée dans le silicon
 - Processeurs ARM à partir de la V3, SPARC V9, MIPS

L'endianisme

Big endian

- Exemple : la valeur 32 bits 0x12345678 est écrite à l'adresse 0x40000000
 - En C, cela donne
 - `uint32_t* addr = (uint32_t *)0x40000000;`
 - `addr[0] = 0x12345678;`
 - 0x12 est l'octet de poids le plus fort : il est enregistré en premier à l'adresse mémoire la plus petite (soit 0x40000000)
 - 0x78 est l'octet de poids le plus faible : il est enregistré en dernier à l'adresse mémoire la plus grande (soit 0x40000003)

addr+0	addr+1	addr+2	addr+3
0x12	0x34	0x56	0x78

L'endianisme

Big endian

- Exemple : la valeur 32 bits 0x12345678 est stockée à l'adresse 0x40000000

addr+0	addr+1	addr+2	addr+3
0x12	0x34	0x56	0x78

- Si on définit un pointeur `uint16_t* addr = (uint16_t*)0x40000000`, alors `addr[0]` vaut 0x1234 et `addr[1]` vaut 0x5678
- Si on définit un pointeur `uint8_t* addr = (uint8_t*)0x40000000`, alors `addr[0]` vaut 0x12, `addr[1]` vaut 0x34, `addr[2]` vaut 0x56 et `addr[3]` vaut 0x78

L'endianisme

Big endian

- Exemple : la valeur 16 bits 0x1234 est écrite à l'adresse 0x40000000 et la valeur 16 bits 0x5678 à l'adresse 0x40000002
 - En C, cela donne :
 - `uint16_t* addr = (uint16_t *)0x40000000;`
 - `addr[0] = 0x1234;`
 - `addr[1] = 0x5678;`

addr+0	addr+1	addr+2	addr+3
0x12	0x34	0x56	0x78

L'endianisme

Little endian

- Exemple : la valeur 32 bits 0x12345678 est écrite à l'adresse 0x40000000
 - En C, cela donne
 - `uint32_t* addr = (uint32_t *)0x40000000;`
 - `addr[0] = 0x12345678;`
 - 0x78 est l'octet de poids le plus faible : il est enregistré en premier à l'adresse mémoire la plus petite (soit 0x40000000)
 - 0x12 est l'octet de poids le plus fort : il est enregistré en dernier à l'adresse mémoire la plus grande (soit 0x40000003)

addr+0	addr+1	addr+2	addr+3
0x78	0x56	0x34	0x12

L'endianisme

Little endian

- Exemple : la valeur 32 bits 0x12345678 est stockée à l'adresse 0x40000000

addr+0	addr+1	addr+2	addr+3
0x78	0x56	0x34	0x12

- Si on définit un pointeur `uint16_t* addr = (uint16_t*)0x40000000`, alors `addr[0]` vaut 0x5678 et `addr[1]` vaut 0x1234
- Si on définit un pointeur `uint8_t* addr = (uint8_t*)0x40000000`, alors `addr[0]` vaut 0x78, `addr[1]` vaut 0x56, `addr[2]` vaut 0x34 et `addr[3]` vaut 0x12

L'endianisme

Little endian

- Exemple : la valeur 16 bits 0x1234 est écrite à l'adresse 0x40000000 et la valeur 16 bits 0x5678 à l'adresse 0x40000002
 - En C, cela donne
 - `uint16_t* addr = (uint16_t *)0x40000000;`
 - `addr[0] = 0x1234;`
 - `addr[1] = 0x5678;`

addr+0	addr+1	addr+2	addr+3
0x34	0x12	0x78	0x56

L'endianisme (Endianness)

- La gestion de l'endianisme doit être prise en compte quand on produit des bibliothèques de fonctions C/C++ pouvant être utilisées sur différentes cibles matérielles ou quand on fait communiquer entre eux des systèmes ayant des processeurs n'ayant pas le même endianisme.
- Il existe des API standardisées permettant de réaliser des conversions little endian \Leftrightarrow big endian
 - Voir fonctions htonl() et htons() de l'API Berkeley sockets

```
uint32_t htonl(uint32_t value)
{
    uint32_t result = 0;
    result |= (value & 0x000000FF) << 24;
    result |= (value & 0x0000FF00) << 8;
    result |= (value & 0x00FF0000) >> 8;
    result |= (value & 0xFF000000) >> 24;
    return result;
}
```

L'endianisme (Endianness)

Exercice - Enoncé

1. Sur un processeur little endian, on écrit la valeur 32 bits 0xAE47FCD0 à l'adresse 0x600FD000
 - Si on définit un pointeur `uint16_t* addr = (uint16_t*)0x600FD000`, alors que valent `addr[0]` et `addr[1]` ?
 - Si on définit un pointeur `uint8_t* addr = (uint8_t*) 0x600FD000`, alors que valent `addr[0]`, `addr[1]`, `addr[2]` et `addr[3]` ?
2. Même question dans le cas d'un processeur big endian.
3. Sur un processeur big endian, on écrit la valeur 32 bits 0xA5B11B32 à l'adresse 0x50026000.
 - Si on définit un pointeur `uint16_t* addr = (uint16_t*)0x50026000`, alors que valent `addr[0]` et `addr[1]` ?

L'endianisme (Endianness)

Exercice - Solution

1. Sur un processeur little endian, on écrit la valeur 32 bits 0xAE47FCD0 à l'adresse 0x600FD000
 - Si on définit un pointeur `uint16_t* addr = (uint16_t*)0x600FD000`, alors que valent `addr[0]` et `addr[1]` ?
 - `addr[0] = 0xFCD0`
 - `addr[1] = 0xAE47`
 - Si on définit un pointeur `uint8_t* addr = (uint8_t*) 0x600FD000`, alors que valent `addr[0]`, `addr[1]`, `addr[2]` et `addr[3]` ?
 - `addr[0] = 0xD0`
 - `addr[1] = 0xFC`
 - `addr[2] = 0x47`
 - `addr[3] = 0xAE`

L'endianisme (Endianness)

Exercice - Solution

2. Sur un processeur big endian, on écrit la valeur 32 bits 0xAE47FCD0 à l'adresse 0x600FD000
- Si on définit un pointeur `uint16_t* addr = (uint16_t*)0x600FD000`, alors que valent `addr[0]` et `addr[1]` ?
 - `addr[0] = 0xAE47`
 - `addr[1] = 0xFCD0`
 - Si on définit un pointeur `uint8_t* addr = (uint8_t*) 0x600FD000`, alors que valent `addr[0]`, `addr[1]`, `addr[2]` et `addr[3]` ?
 - `addr[0] = 0xAE`
 - `addr[1] = 0x47`
 - `addr[2] = 0xFC`
 - `addr[3] = 0xD0`

L'endianisme (Endianness)

Exercice - Solution

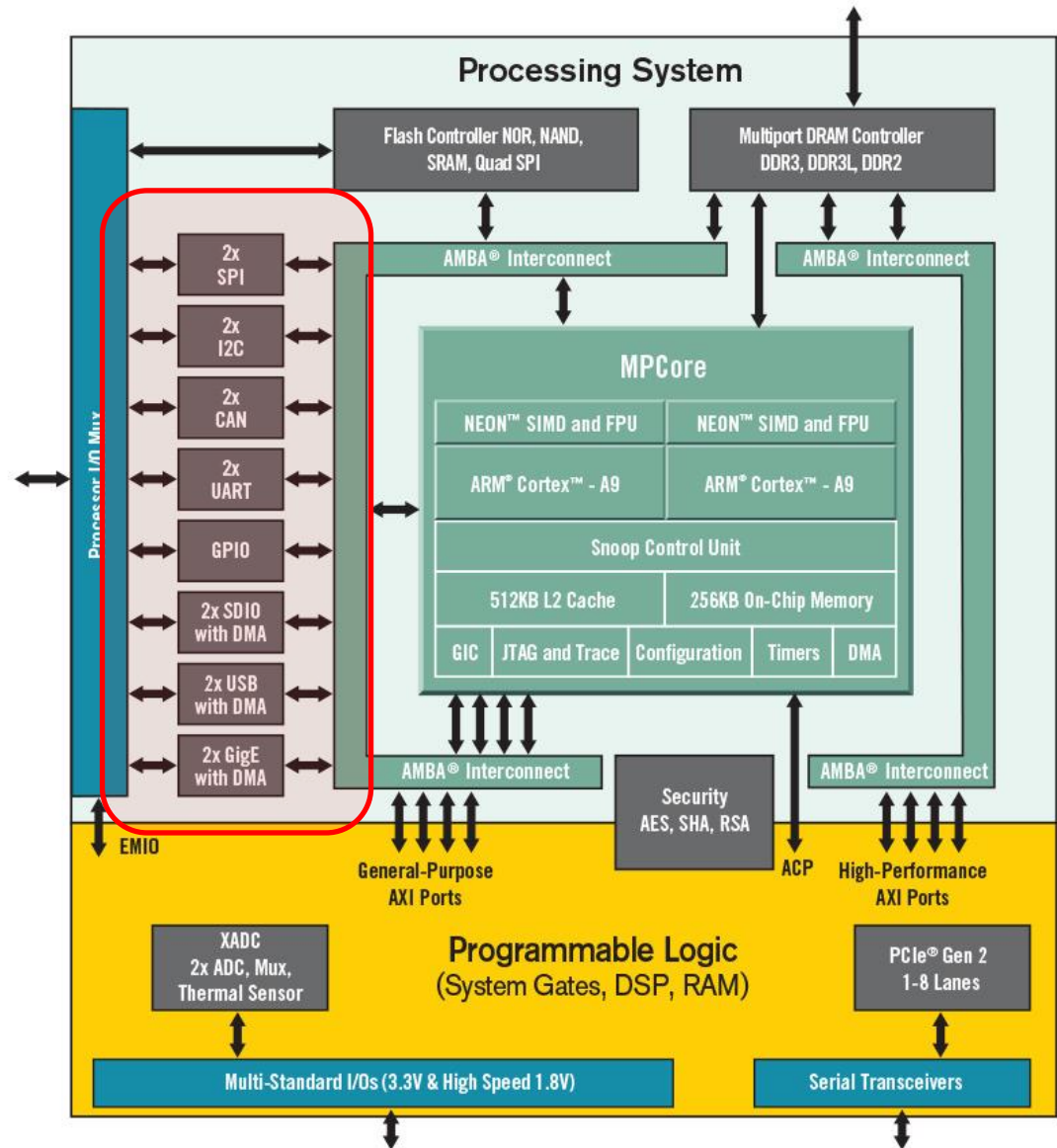
3. Sur un processeur big endian, on écrit la valeur 32 bits 0xA5B11B32 à l'adresse 0x50026000.
- Si on définit un pointeur `uint16_t* addr = (uint16_t*)0x50026000`, alors que valent `addr[0]` et `addr[1]` ?
 - `addr[0] = 0xA5B1`
 - `addr[1] = 0x1B32`

L'interface logiciel / matériel

1. Préambule
2. Architecture des processeurs
3. Les registres
4. Les exceptions et les interruptions
5. La mémoire (types de mémoire physique, organisation de la mémoire, DMA, mémoire cache, MMU, endianisme)
6. L'accès aux périphériques

L'accès aux périphériques

- Dans les microcontrôleurs à base d'ARM ou encore dans les processeurs LEON, l'accès aux périphériques (timers, ADC/DAC, UART, SPI, CAN, I2C, USB, Ethernet, SpaceWire, 1553, etc.) se fait désormais via des IP cores dédiés, intégrés à la puce processeur (System On Chip) et communiquant avec le processeur via un bus de type AMBA.
- Ex. : Le processeur Xilinx Zynq-7000 (ARM Cortex-A9 dual-core)



L'accès aux périphériques

Registres dédiés

- L'accès par logiciel aux périphériques se fait via des registres de l'IP core qui sont mappés sur l'espace mémoire du processeur
 - Registres d'état
 - Registres de configuration
 - Registres de données

12.13.14 ADC regular Data Register (ADCx_DR, x=1..4)

Address offset: 0x40

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RDATA[15:0]															
r	r	r	r	r	r	r	r	r	r	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **RDATA[15:0]**: Regular Data converted

These bits are read-only. They contain the conversion result from the last converted regular channel. The data are left- or right-aligned as described in [Section 12.5: Data management on page 236](#).

Ex. : les registres
de lecture des
données ADC du
processeur
STM32F3

L'accès aux périphériques

Transferts DMA

- Le transfert des données en provenance du périphérique ou à destination du périphérique se fait :
 - soit directement à travers des registres quand le volume de données est faible (ex. : ADC)
 - soit via des transferts DMA :
 - C'est le cas par exemple des liens de communication : quand un paquet est reçu, il est copié directement dans la mémoire RAM du processeur.
 - Des registres et/ou des tables de descripteurs permettent généralement de configurer l'adresse à laquelle les paquets reçus seront copiés
 - La transmission des paquets se fait aussi via le DMA : le logiciel construit le paquet à transmettre dans la RAM et indique au périphérique, via des registres dédiés, que le paquet est prêt à être transmis.

L'accès aux périphériques

Notification par interruption

- On associe généralement à chaque IP core en charge d'un périphérique une ou plusieurs lignes d'interruption qui sont utilisées pour notifier le logiciel que :
 - une donnée ou un bloc de données est disponible à la lecture
 - dans un registre
 - ou dans la RAM en cas de transfert DMA
 - une donnée ou un bloc de données a été transmis
 - un événement particulier s'est produit (erreur pendant le transfert, ...)

L'accès aux périphériques

Driver logiciel

- Un driver logiciel est constitué par un ensemble de fonctions permettant de :
 - configurer les registres des périphériques
 - activer / désactiver l'accès au périphérique
 - lire les données transmises
 - écrire les données transmises