

# Programmation des systèmes

Enseignant: Pierre Leone

Assistant:

Moniteur: Jérémie Martin

Contacts:

x@(etu.)unige.ch

# Plan du cours

## Systèmes informatiques et programmation en assembleur

- Éléments des systèmes informatiques
- Etude du jeu d'instruction d'un microcontrôleur (ARM7TDMI)
- Programmation en assembleur
- Appels système (LINUX)

# Evaluation

- Un examen écrit à la fin de l'année
- Deux contrôle continus, rendre le premier contrôle continu valide l'inscription au contrôle continu.
- Les travaux pratiques ne sont pas notés mais **ils font partie du champ de l'examen.**

# 1<sup>ère</sup> partie

## Programmation assembleur

Systèmes informatiques:

Les composants principaux d'un ordinateur (système informatique) sont

1. **Le processeur**
2. **La mémoire**
3. **Les périphériques**

Ces éléments constituent un système, ils interagissent pour réaliser une fonction. Généralement ils utilisent un ou plusieurs **bus** pour échanger des informations. On distingue

1. Le/les bus de données
2. Le/les bus d'adresses
3. Le/les bus de contrôle

# Les bus

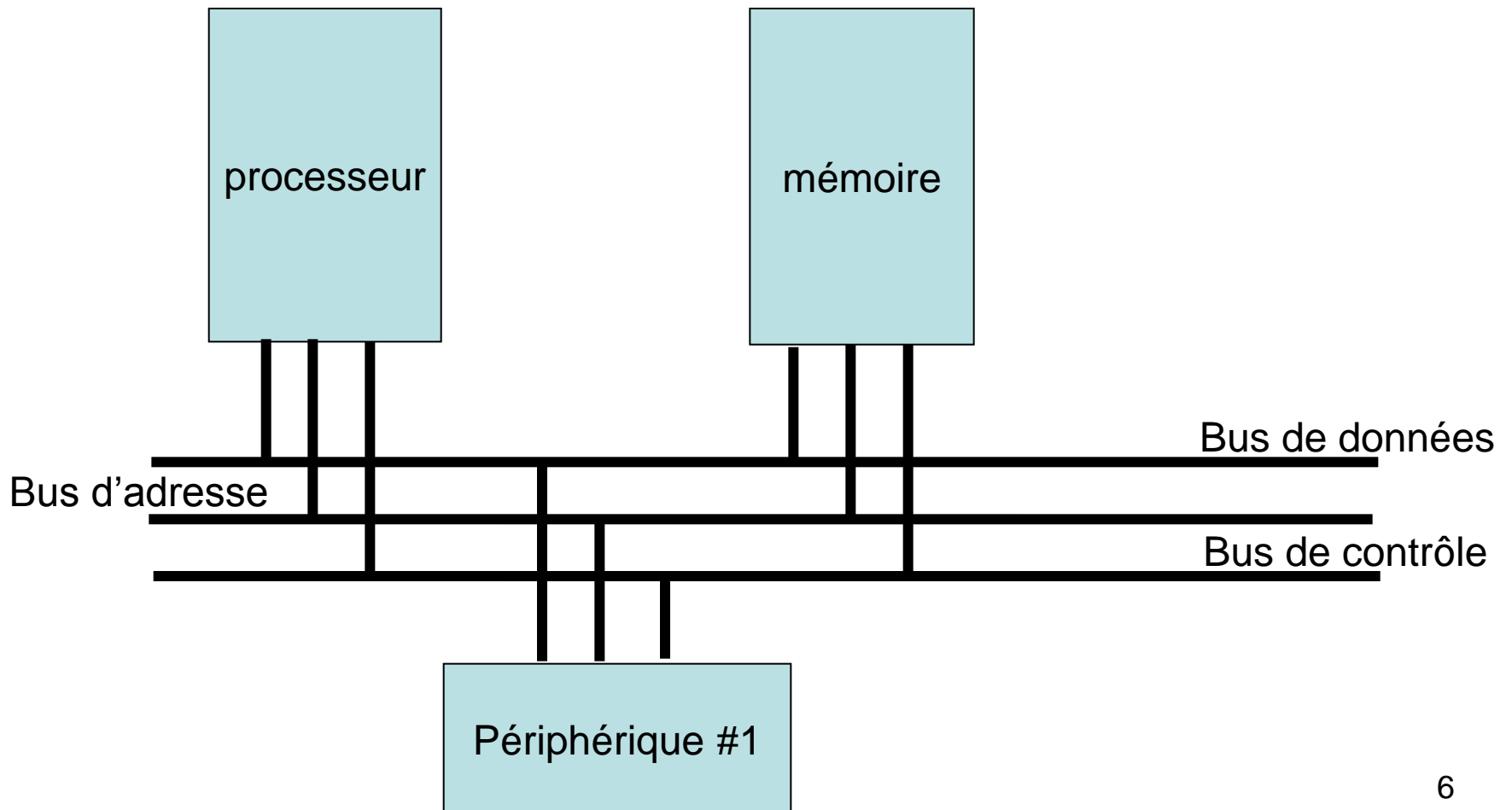
On peut résumer les différentes fonctionnalités des bus

**Bus d'adresse:** permet de spécifier l'élément concerné par le transfert, un périphérique particulier (un registre d'un périphérique particulier), une position de la mémoire externe, ... **Chaque élément constituant le système informatique est désigné par une adresse ou une plage d'adresses**

**Bus de données:** assure le transfert des informations (données) entre le processeur et les périphériques

**Bus de contrôle:** les signaux de ce bus permettent de valider les signaux qui se trouvent sur les autres bus, par exemples l'adresse qui se trouve sur le bus d'adresse maintenant est valide, on désire réaliser un cycle de lecture/écriture

# schématiquement



# Les bus

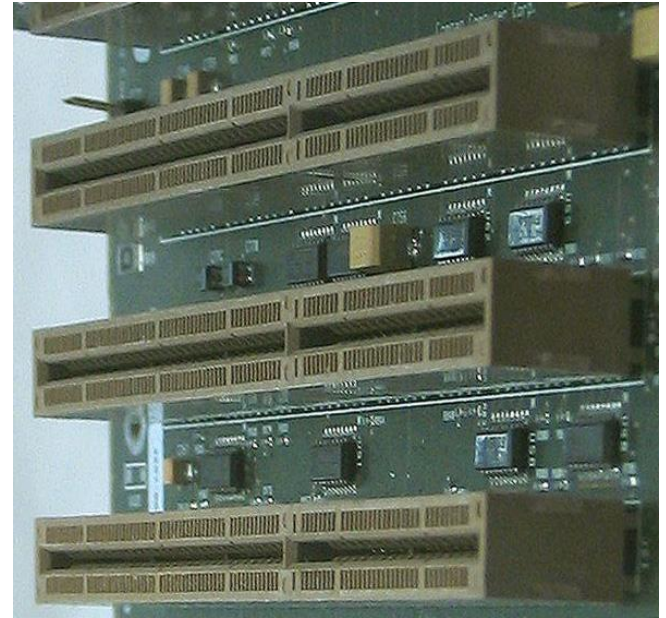
Il existe beaucoup de standard différents qui définissent des bus.

- Le bus PCI Peripheral Component Interconnect, c'est le bus utilisé par les processeur de la famille Intel x86
- AMBA un bus pour les applications embarquées utilisé par le processeur ARM7
- Les bus ISA, MCA, EISA, les anciens bus x86
- Le bus AGP Accelerated Graphics port est un bus optimisé pour les transferts d'images et de son (signal vidéo)
- Des bus optimisés pour les environnement industriels
- ...

# Les bus exemples



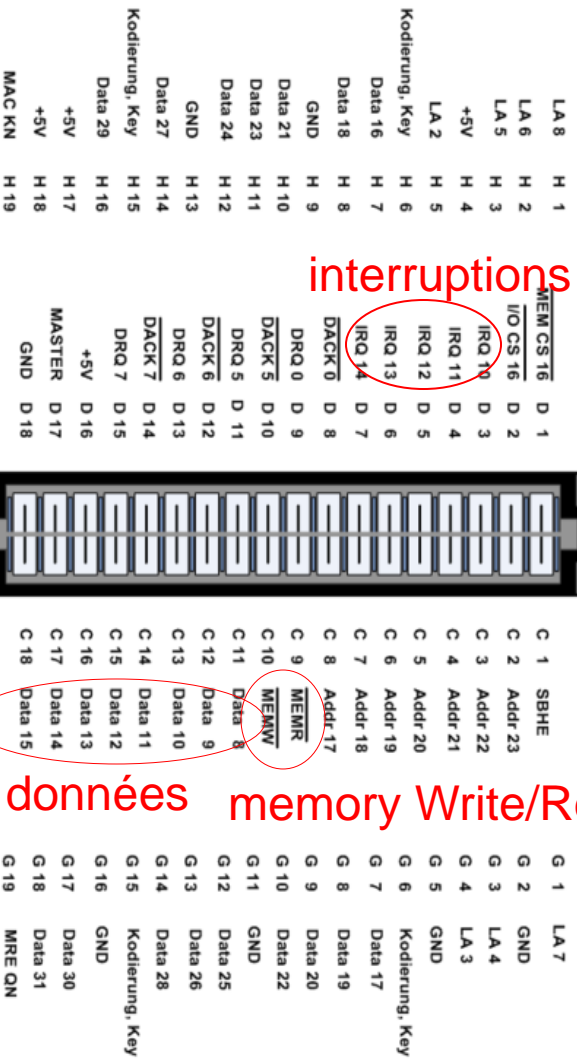
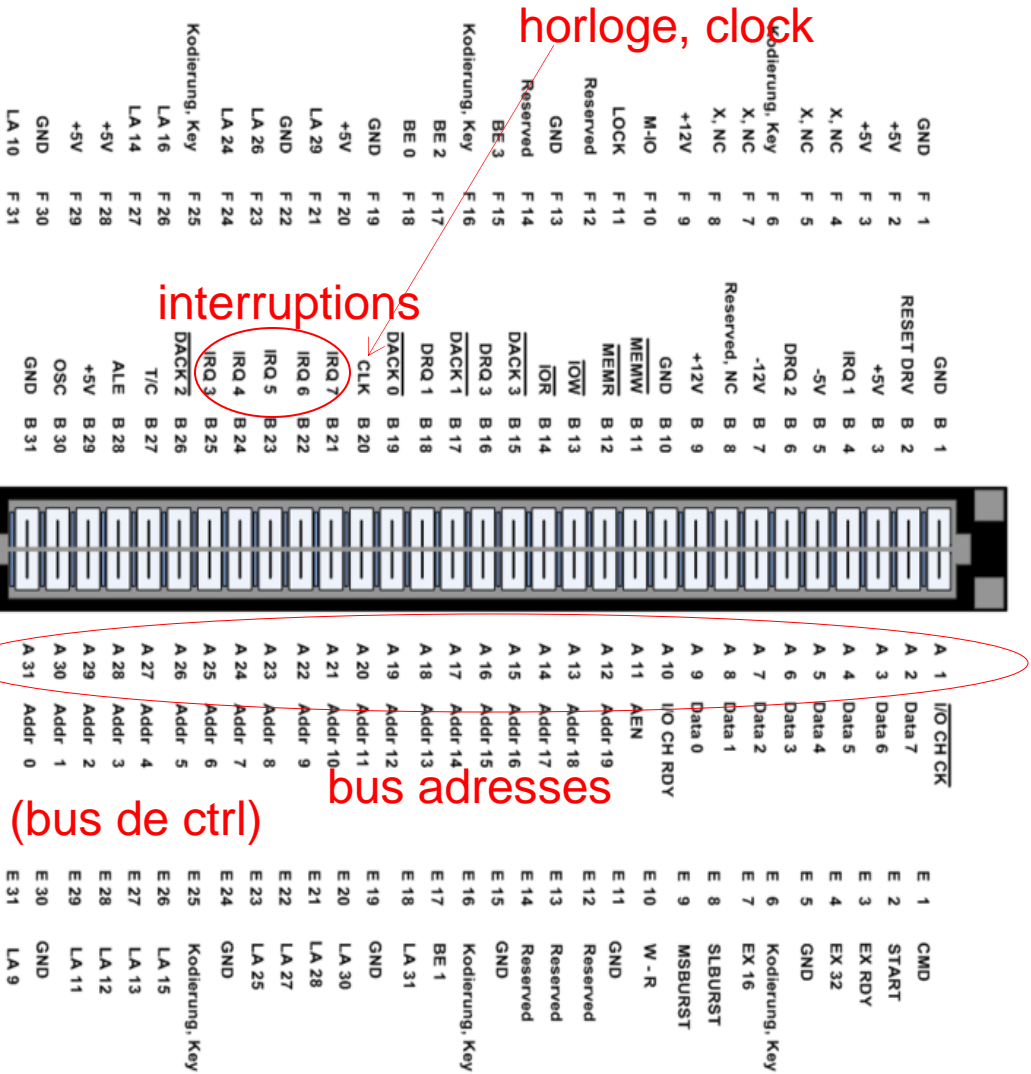
Carte graphie EISA (source wikipedia)



Connecteurs



# Les bus - exemple



# Processeur

Le processeur est le composant d'un ordinateur qui exécute les programmes, c'est-à-dire

- Exécute des instructions, l'ensemble des instructions (machine) exécutables constitue le **jeu d'instruction du processeur**.
- Effectue des **opérations sur les données**, par exemple des opérations arithmétiques (addition,...), des opérations sur les adresses pour retrouver les données en mémoire (pour permettre la gestion des tableaux de données, des pointeurs,...), des comparaisons.

abréviation **CPU**: **C**entral **P**rocessing **U**nit

# Les registres

Les **registres** constituent une petite zone mémoire (de l'ordre d'une dizaine de mots). Ils sont internes aux processeurs, généralement utilisés par l'unité d'exécution comme opérande source et destination des instructions.

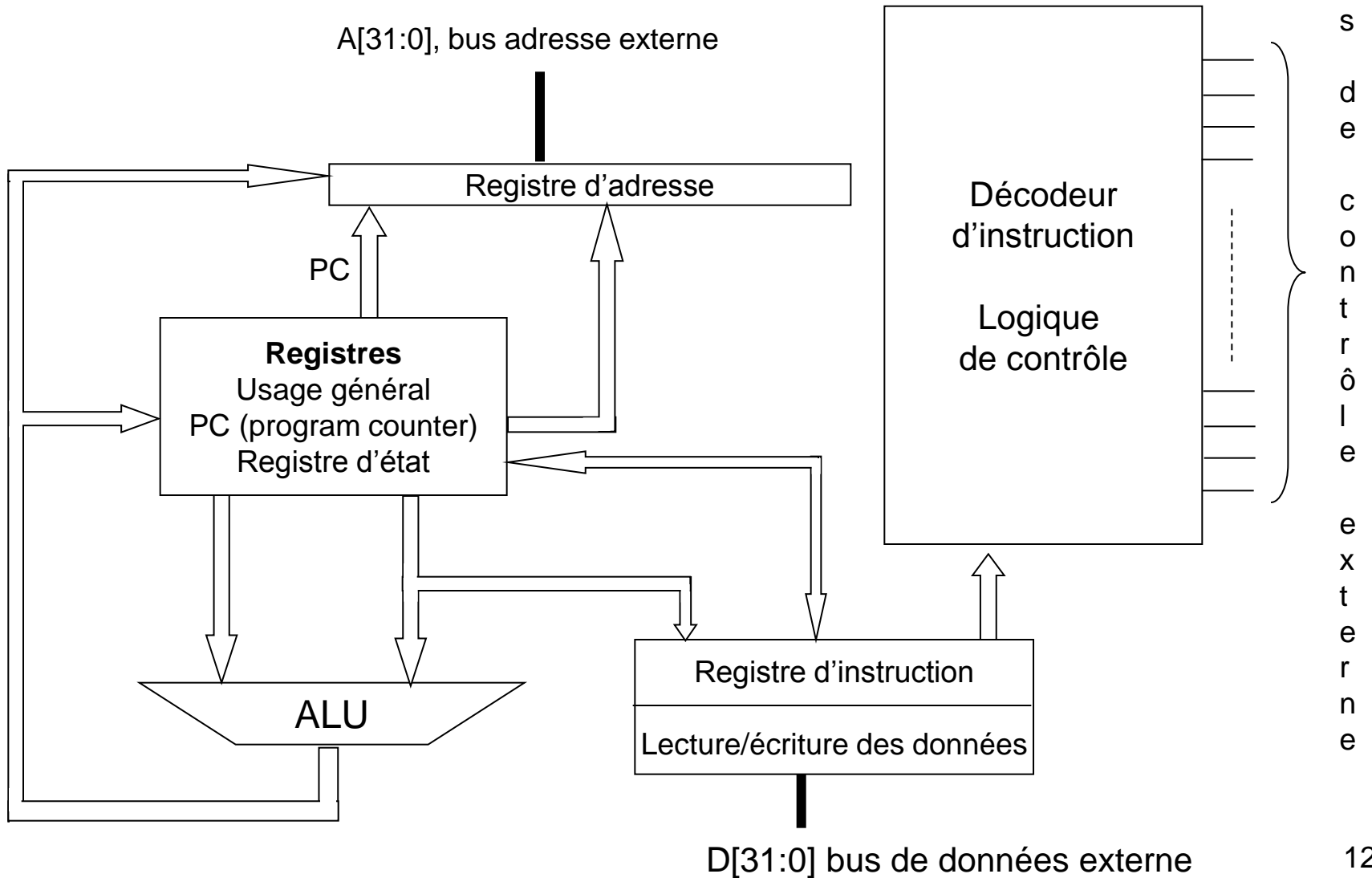
Leur accès est très rapide, beaucoup plus rapide qu'un accès en mémoire externe et c'est la raison pour laquelle on préfère les utiliser comme opérande plutôt que des mots qui se trouvent en mémoire externe.

Les registres sont pour le programmeur les seules variables sur lesquelles il peut agir.

```
ADD    r0,r1,r2    r0=r1+r2
```

On dispose de 16 registres par exemple, r0, ..., r15.

# Schématiquement



# La mémoire

Les informations sont repérées par des adresses. Par convention une adresse pointe sur une **donnée de 8 bits**.

Selon **la taille du bus de données** les informations peuvent être organisées par

- Byte, groupe de 8 bits (par ex. un caractère)
- halfword, groupe de 16 bits (par ex. un entier)
- word, groupe de 32 bits (par ex. un pointeur)

La valeur codée sur 8bits  
(1byte) stockée à  
L'adresse 0x00

Bus de données 8 bits:

Les adresses

0x00

0x01

0x02

0x03

0x04

0x05

0x06

0xBC

0x01

0xFF

0xFC

0x25

0xD2

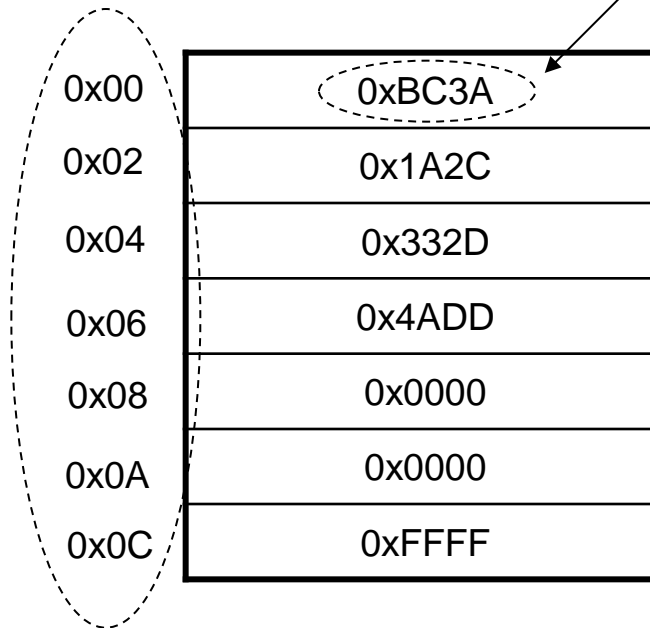
0x10

# La mémoire

Bus de données 16 bits:

Un halfword = 2bytes = 16 bits

Adresses paires  
*car on veut toujours  
pouvoir adresser de  
bytes!*

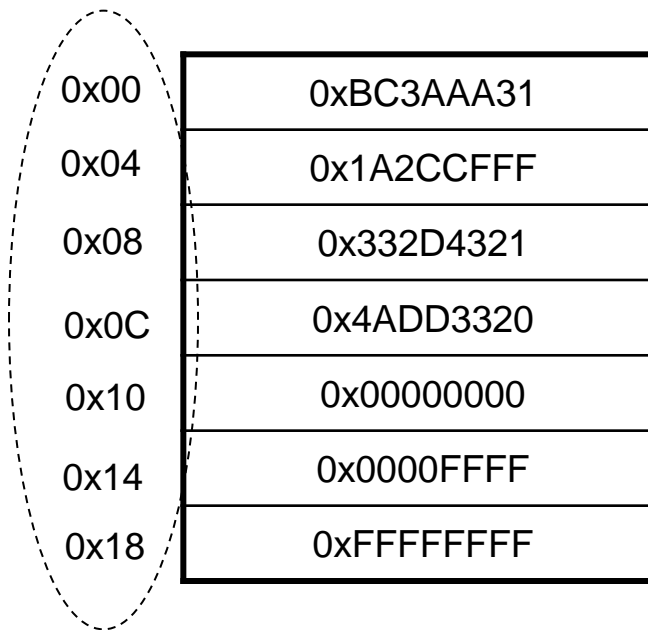


0x00	0xBC3A
0x02	0x1A2C
0x04	0x332D
0x06	0x4ADD
0x08	0x0000
0x0A	0x0000
0x0C	0xFFFF

# La mémoire

Bus de données 32 bits (word)

Adresses divisibles par 4  
*car on veut toujours  
pouvoir adresser de  
bytes!*



0x00	0xBC3AAA31
0x04	0x1A2CCFFF
0x08	0x332D4321
0x0C	0x4ADD3320
0x10	0x00000000
0x14	0x0000FFFF
0x18	0xFFFFFFFF

# La mémoire

Pour que les lectures/écritures en mémoire soient efficaces:

- Les adresses des halfwords doivent être divisible par 2
- Les adresses des mots des words doivent être divisible par 4

Certains processeurs refusent (Bus Error) d'effectuer des cycles de lectures/écritures si ces règles ne sont pas respectées

Lorsque l'on écrit un mot de 32 bits, par exemple 0x4A3B2C1D les chiffres à gauche sont les plus significatifs.

Lorsque les mots de 32 bits sont écrits en mémoire on a généralement deux possibilités



# La mémoire

Big-endian (big end first): Les bytes sont ordonnés en mémoire tel que les bytes les plus significatifs possèdent les adresses les plus petites

		0x0D	0x0E	0x0F
0x0C	4A	3B	2C	1D
0x10				

little-endian (little end first): Les bytes sont ordonnés en mémoire tel que les bytes les moins significatifs possèdent les adresses les plus petites

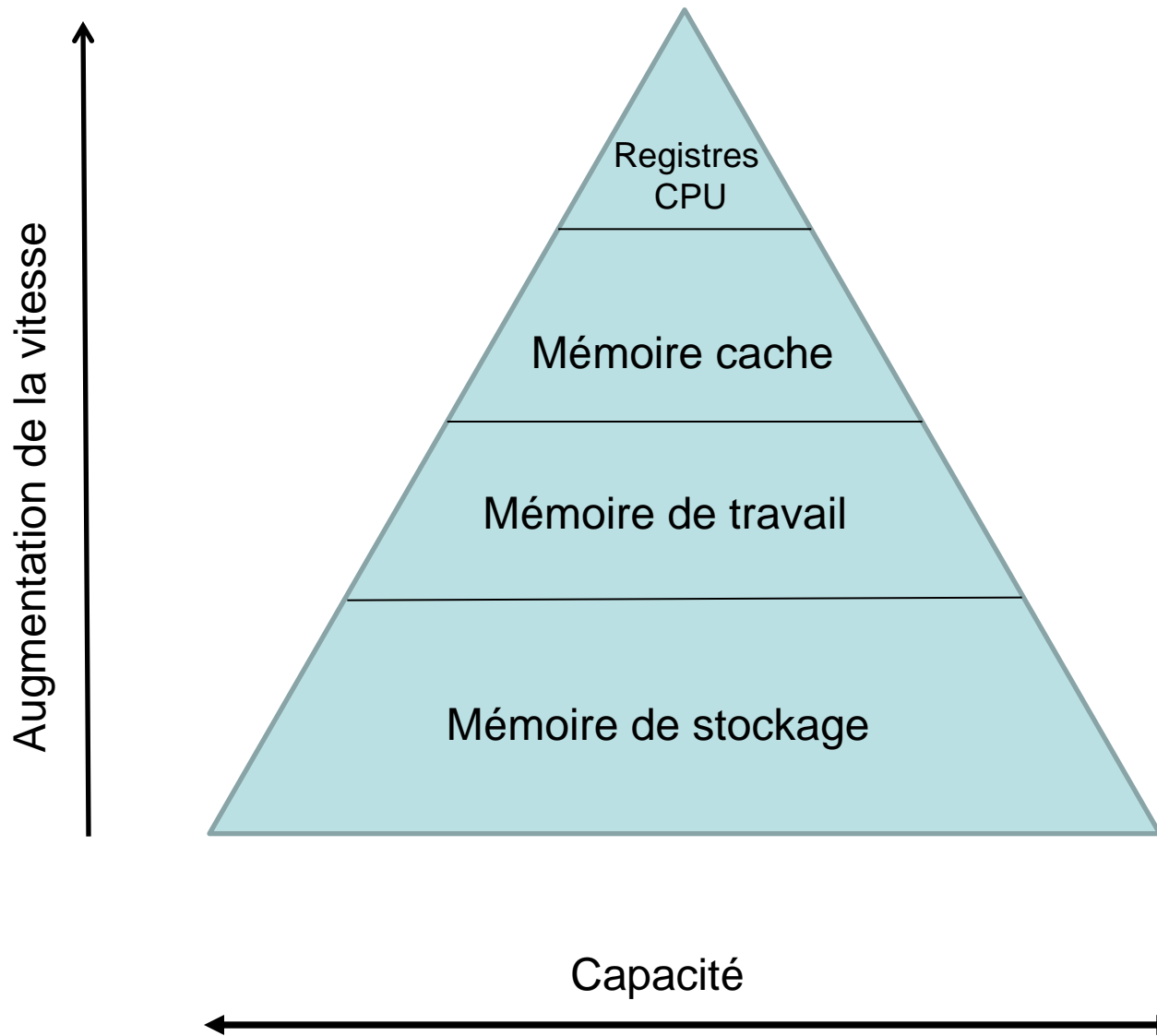
		0x0D	0x0E	0x0F
0x0C	1D	2C	3B	4A
0x10				

# Les différents types de mémoire

On peut classer les différents types de mémoire par la capacité de stockage qu'ils permettent. Les mémoires à la capacité la plus limitée sont aussi celles qui sont accessibles le plus rapidement.

- **La mémoire de travail**, c'est la mémoire qui est accédée durant l'exécution d'un programme. Elles doivent être accessibles rapidement pour permettre une exécution performante du programme. Il s'agit des registres internes du processeur, de la mémoire cache, de la mémoire externe, de la mémoire morte. **Vitesse d'accès, taux d'intégration**

- **La mémoire de stockage**, permet de conserver d'importante quantité de donnée, ce sont les mémoires de masses comme les disques magnétiques, ou les CD. **Capacité**



# Temps d'accès – ordre de grandeur

1. Les registres internes, l'accès le plus rapide, la capacité la plus faible
2. La mémoire cache, 6-35 ns pour lire/écrire un mot (150-30 MHz)
3. La mémoire centrale, 35-120 ns
4. Disque durs, 10-20 ms

La mémoire cache est généralement une mémoire qui est plus rapide que la mémoire centrale (capacité plus faible car plus chère) et est utilisée pour dupliquer les informations souvent accédées (en mémoire centrale) pour diminuer les temps d'accès.

# Mémoires volatiles

Volatile = l'information est perdue lorsqu'on coupe l'alimentation

- Plutôt rapide, mémoire cache, mémoire de travail

On distingue **SRAM** , Static RAM et **DRAM** Dynamic RAM.

**RAM:** Random Acces Memory par opposition aux mémoires qu'on accède selon un ordre défini, par exemple séquentiel.

Static et Dynamic font référence à la construction des mémoires

# Mémoires non-volatiles

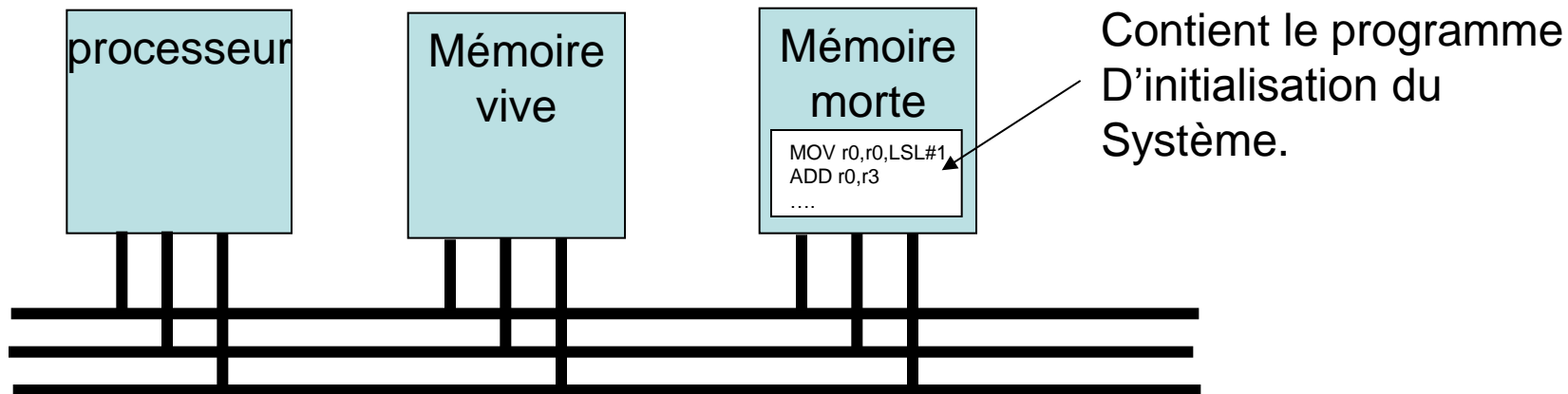
- Les mémoires mortes ROM (Read Only Memory)
- Les disques durs
- Bandes magnétiques
- Disques optiques

Ils utilisent tous des éléments mécaniques. Ils sont les plus lents, utilisés pour la mémoire de masse.

# Les mémoires mortes

Appelées aussi ROM (Read Only memory), elles ont l'avantage de conserver l'information même lorsque le circuit n'est plus alimenté.

Elles sont utilisées pour stocker le logiciel de démarrage du système (ordinateur) qui s'appelle aussi programme "boot".



Une des fonctions du programme de boot sur un ordinateur est de démarrer le système d'exploitation qui se trouve mémorisé sur le disque dur

# Les périphériques

Le système processeur + mémoire est le système fonctionnel minimum. Pour pouvoir accéder 'à l'extérieur' on utilise les périphériques.

On distingue périphériques interne (à l'ordinateur):

- Carte vidéo
- Carte son

Les périphériques externes (à l'ordinateur):

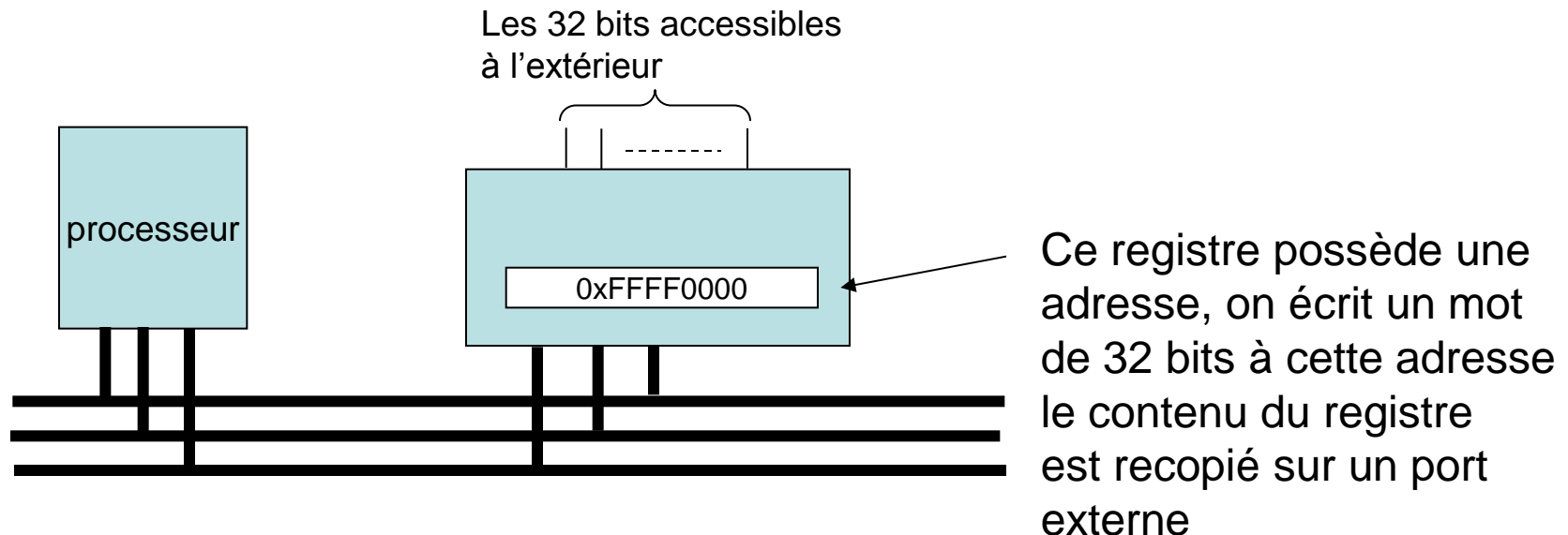
- Écran
- Clavier
- Souris



# Les périphériques

En général les périphériques possèdent des registres que l'on accède de la même manière qu'on accède la mémoire\*. En écrivant les valeurs adéquates dans ces registres on active/désactive le périphérique.

Par exemple. Un port parallèle simplifié



\* Memory-mapped peripheral

# Exemple - GBA

## Memory Map

### General Internal Memory

0000:0000-0000:3FFF BIOS - System ROM (16 KBytes)

0000:4000-01FF:FFFF Not used

0200:0000-0203:FFFF WRAM - On-board Work RAM (256 KBytes) 2 Wait

0204:0000-02FF:FFFF Not used

0300:0000-0300:7FFF WRAM - In-chip Work RAM (32 KBytes)

0300:8000-03FF:FFFF Not used

0400:0000-0400:03FE I/O Registers

0400:0400-04FF:FFFF Not used

### Internal Display Memory

0500:0000-0500:03FF BG/OBJ Palette RAM (1 Kbyte)

0500:0400-05FF:FFFF Not used

0600:0000-0617:FFFF VRAM - Video RAM (96 KBytes)

0618:0000-06FF:FFFF Not used

0700:0000-0700:03FF OAM - OBJ Attributes (1 Kbyte)

0700:0400-07FF:FFFF Not used

# Exemple

## **External Memory (Game Pak)**

0800:0000-09FF:FFFF Game Pak ROM/FlashROM (max 32MB) - Wait State 0  
0A00:0000-0BFF:FFFF Game Pak ROM/FlashROM (max 32MB) - Wait State 1  
0C00:0000-0DFF:FFFF Game Pak ROM/FlashROM (max 32MB) - Wait State 2  
0E00:0000-0E00:FFFF Game Pak SRAM (max 64 KBytes) - 8bit Bus width  
0E01:0000-0FFF:FFFF Not used

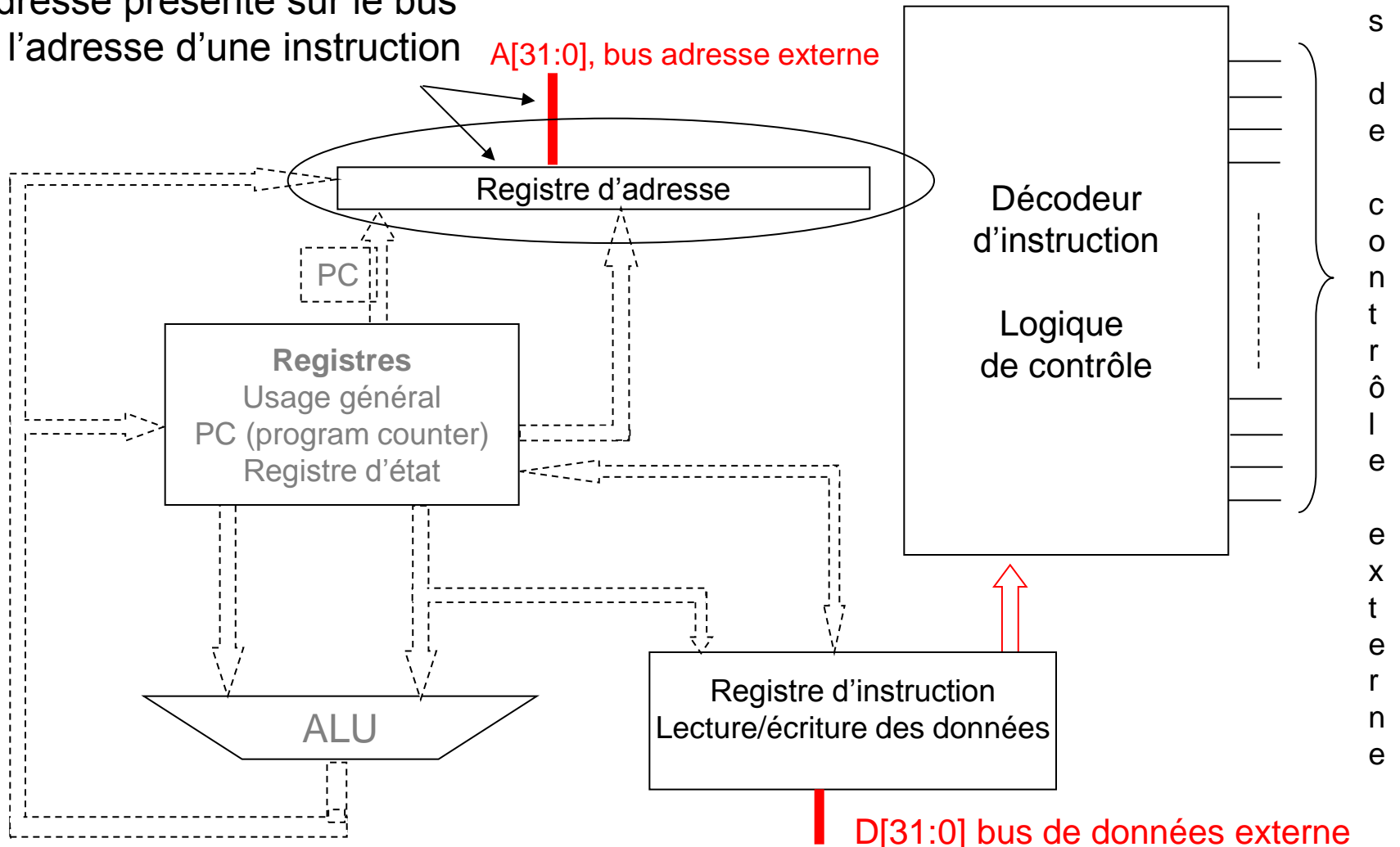
## **Unused Memory Area**

1000:0000-FFFF:FFFF Not used (upper 4bits of address bus unused)

# Cycle de lecture

L'adresse présente sur le bus  
est l'adresse d'une instruction

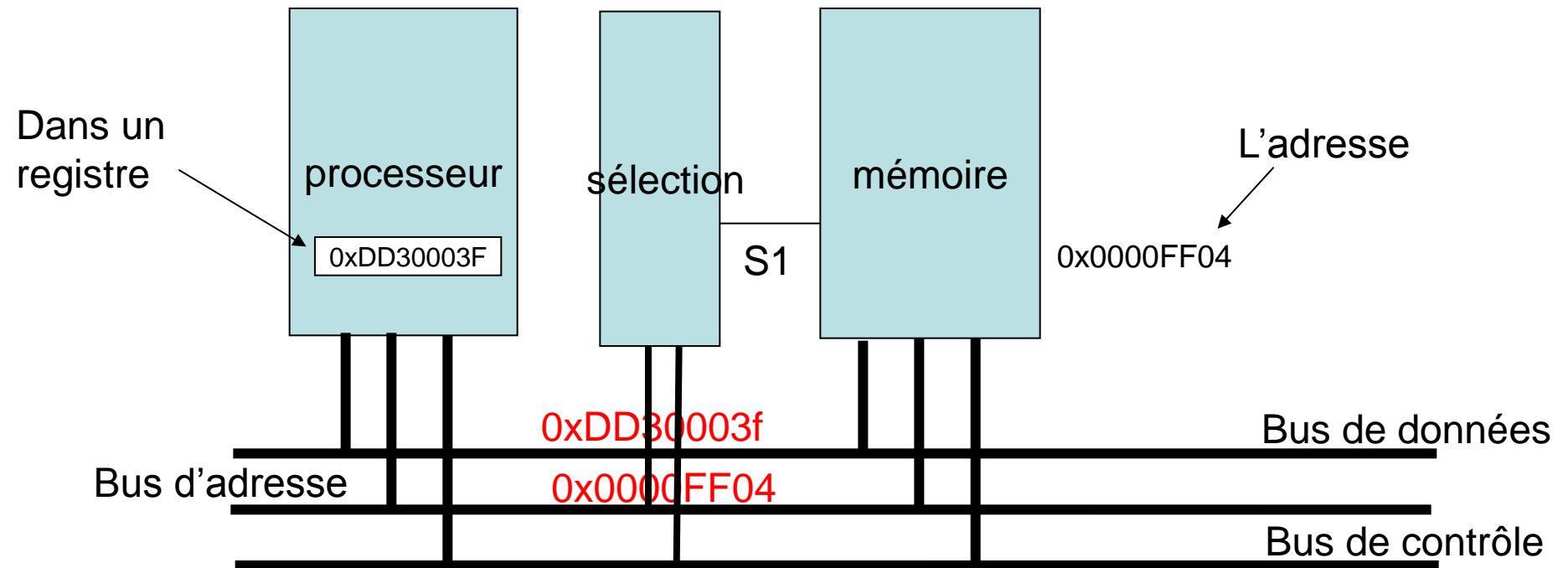
A[31:0], bus adresse externe



L'instruction à exécuter est fournie au processeur par  
Le bus de données

B  
u  
s  
  
d  
e  
  
c  
o  
n  
t  
r  
ô  
l  
e  
  
e  
x  
t  
e  
r  
n  
e

# Cycle de lecture



# Cycle de lecture

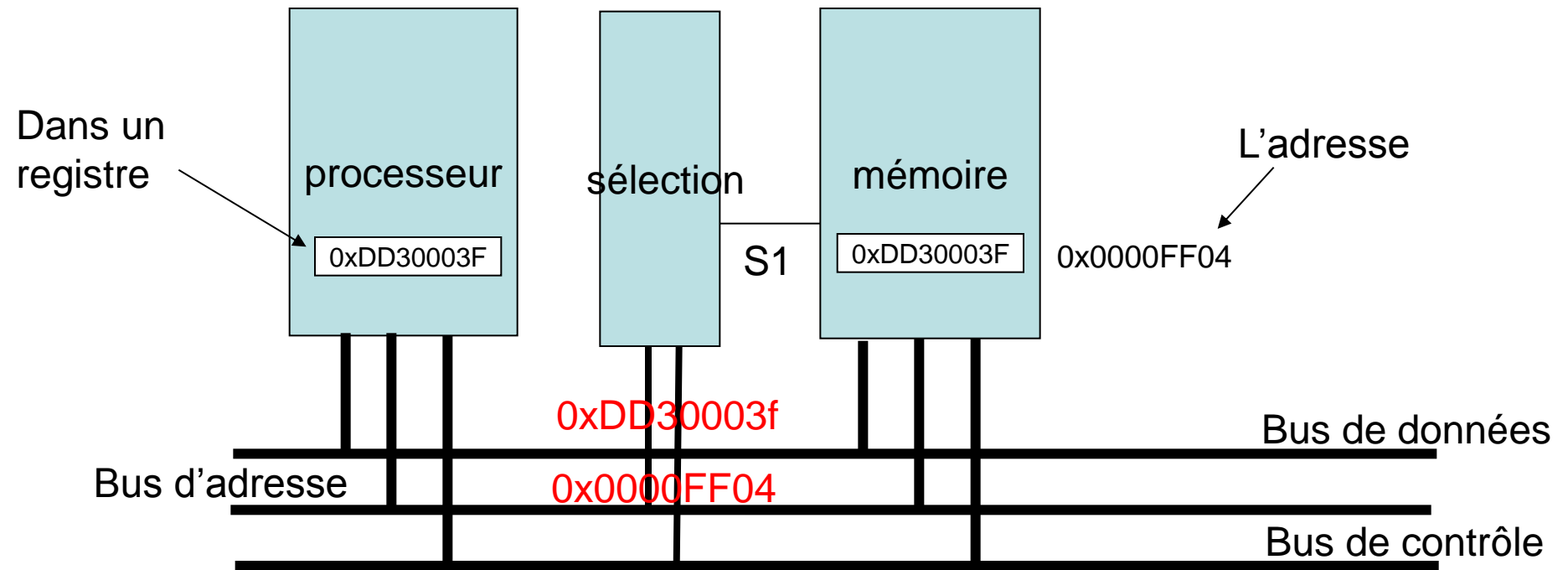
on peut décomposer un cycle **de lecture**

1. Le processeur dépose l'adresse sur le bus d'adresse
2. En utilisant l'adresse ou un signal dédié le circuit mémoire est sélectionné, ce qui lui indique qu'il doit prendre en charge le transfert (bus de contrôle)
3. Le circuit mémoire dépose sur le bus de donnée la donnée qui se trouve à l'adresse
4. Le processeur saisit la donnée et libère les bus pour le prochain transfert

La donnée peut être une information telle que la valeur d'une variable, un pointeur, etc. mais aussi la prochaine instruction à exécuter.

Un cycle de lecture est un cycle pendant lequel la donnée est transférée depuis le périphériques/la mémoire vers les registres internes du processeur

# Cycle d'écriture



# Cycle d'écriture

on peut décomposer un **cycle d'écriture**

1. Le processeur dépose l'adresse sur le bus d'adresse ainsi que la donnée à écrire
2. En utilisant l'adresse ou un signal dédié le circuit mémoire est sélectionné. Un signal dédié spécifie que le transfert est une écriture (bus de contrôle)
3. Le circuit mémoire transfère la donnée sur le bus de donnée à l'adresse spécifiée par le bus d'adresse

Un cycle d'écriture est un cycle pendant lequel la donnée est transférée depuis un registre du processeur vers le périphériques/la mémoire.



# Le Microcontrôleur ARM7TDMI

# Nomenclature des processeur ARM

## ARM7TDMI

T: jeu d'instructions Thumb

D: debbugger hardware JTAG

M: multiplicateur rapide

I: macrocell ICE embarqué (debug)

# RISC

Dans un processeur RISC les instructions doivent s'exécuter en **un seul cycle d'horloge** qui doit être à haute fréquence. La complexité des instructions est donc limitée ( par exemple les modes d'adressages). La complexité d'un algorithme est donc assurée par l'ensemble des instructions et les performances du compilateurs sont déterminantes

- **Instructions:** Reduced Instructions Set Computer, le compilateur doit réaliser des opérations simples (typ. pas de divisions). Le temps d'exécution des instructions est similaire (1 cycle). Pour un processeur CISC certaines instructions peuvent durer plusieurs cycles d'horloges.
- **Pipelines:** Les instructions ayant toutes le même format il est possible de fractionner leur exécution en sous-étapes d'exécution (lire l'instruction, décodage, exécution).
- **Registres:** Grand nombre de registres à usage général (pas de registres dédiés). Les registres peuvent contenir des données ou des adresses indifféremment.

# RISC

- **Registres (suites):** les registres sont utilisés par les instructions comme paramètres d'entrées/sorties. Dans certains processeurs RISC, certaines instructions peuvent s'exécuter seulement avec certains registres pour améliorer les performances.
- **Architecture Load/Store:** Les instructions s'exécutent uniquement sur des données contenues dans les registres. Les transferts entre mémoire externe et registres s'effectuent par des instructions spécifiques (Load et Store) . Les processeurs CISC permettent par exemple la multiplication de données en mémoire externes et le placent résultat en mémoire externe (3 accès mémoire)

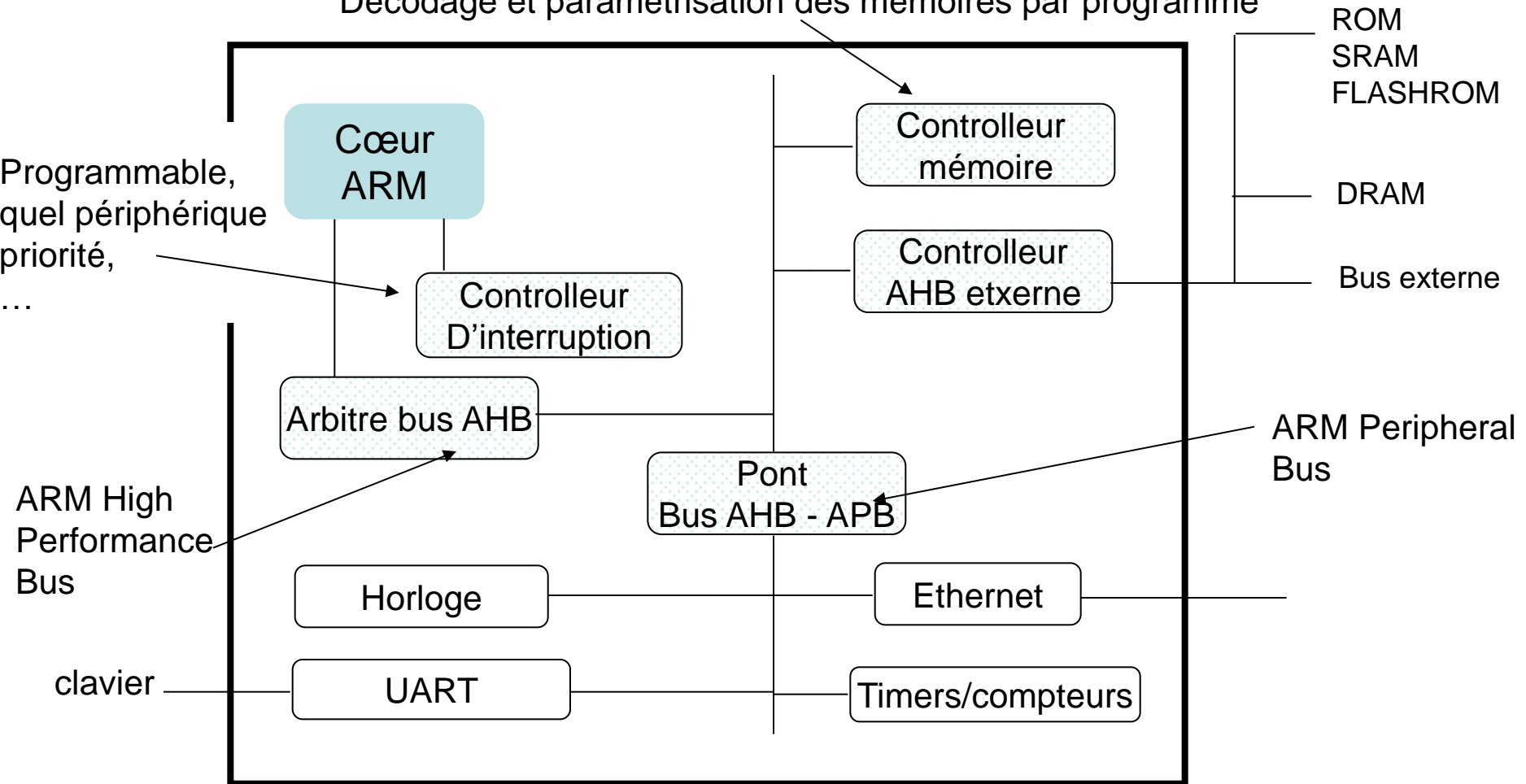
Les architectures RISC cherchent à optimiser les performances en implémentant des instructions simples et optimisées et laissent le compilateur responsable d'une part des performances. Actuellement, beaucoup de processeurs CISC utilisent des aspects de conception RISC.

# ARM-RISC ou CISC

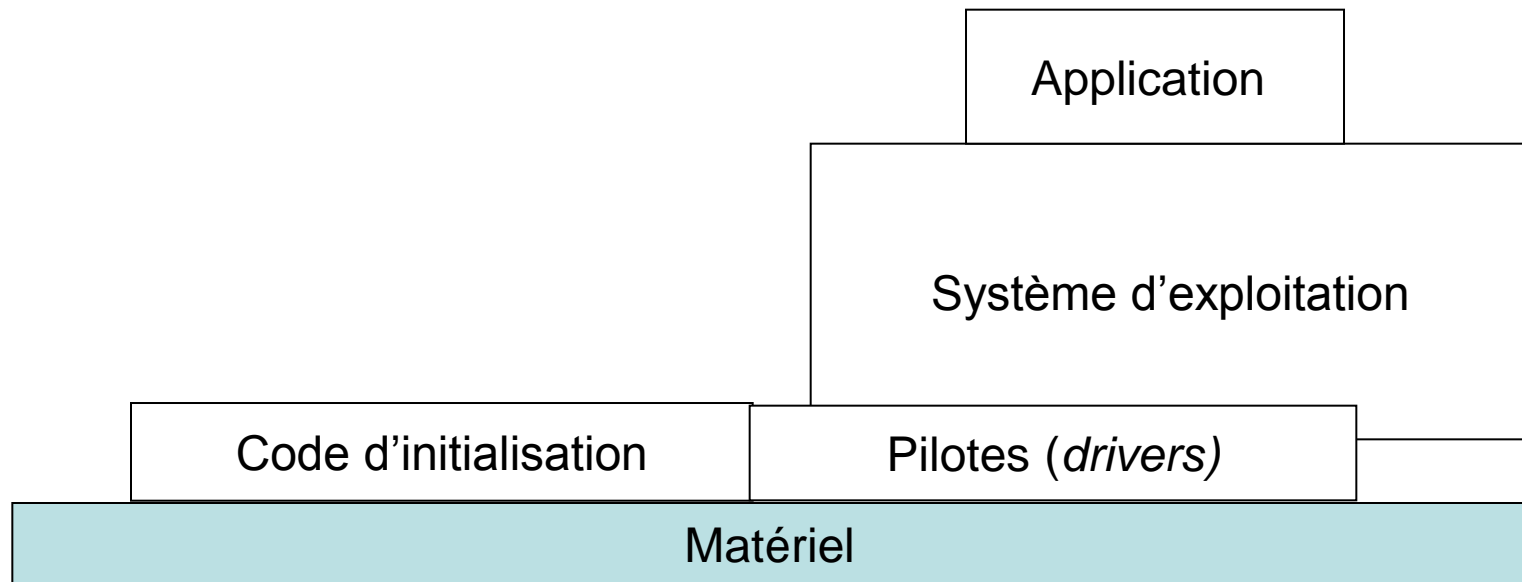
- Les instructions ne s'exécutent pas toutes en un seul cycle d'horloge. Par exemple, les instructions load/store peuvent transférer plusieurs registres à des adresses séquentielles (augmentation de la densité de code, accès séquentiels sont généralement plus rapides)
- Un registre à décalage '*barrel register*' permet d'effectuer un précalcul sur un registre avant qu'il soit utilisé par une instruction (décalage)
- Un jeu d'instruction codé sur 16 bits (augmentation de la densité d'intégration ~30%)
- Les instructions peuvent être conditionnelles, ce qui réduit l'utilisation des instructions de branchement
- Certaines instructions sont spécialisées, comme les instructions DSP

# Exemple d'architecture

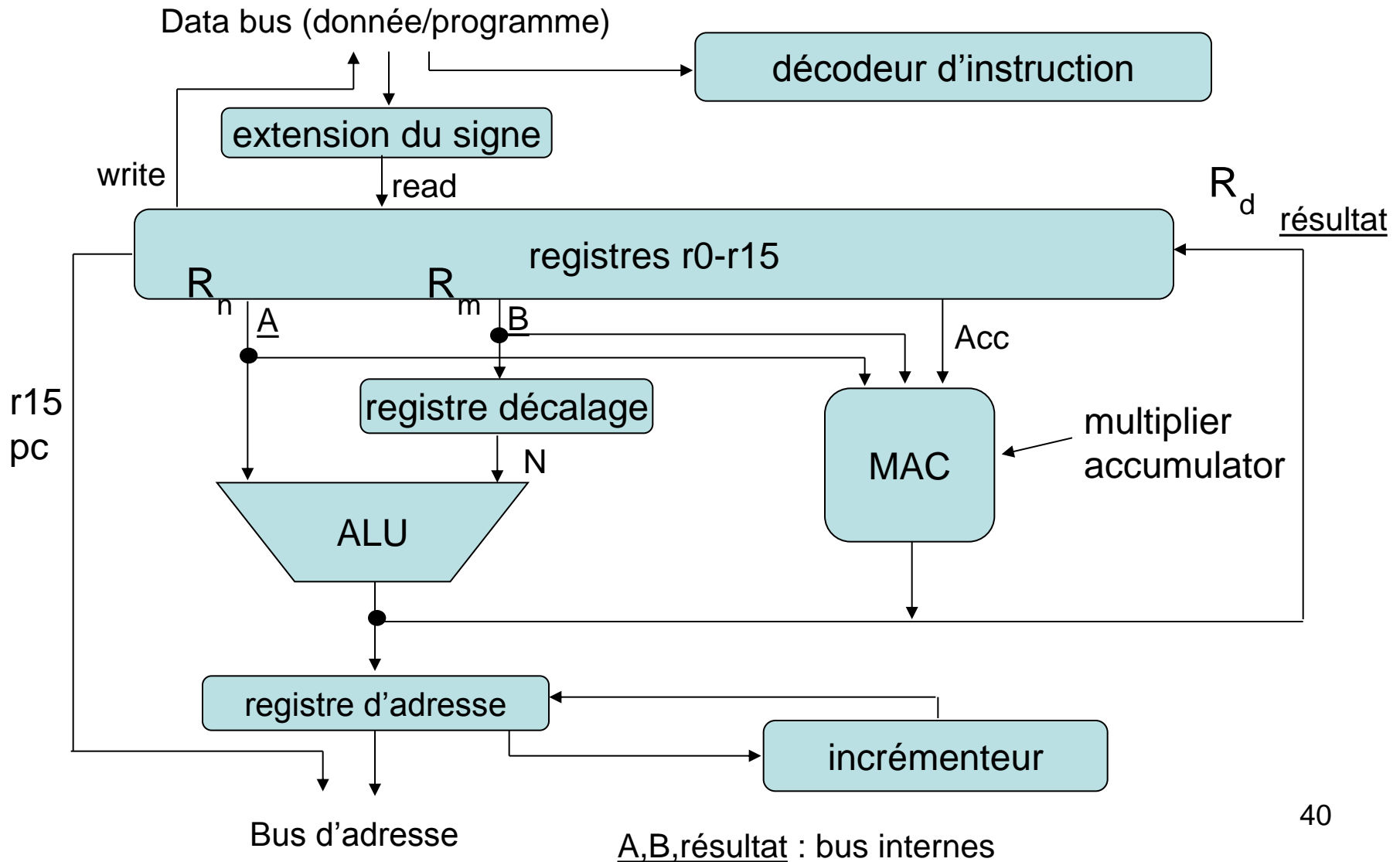
Décodage et paramétrisation des mémoires par programme



# Software



# Flot des données





# ARM

- Le schéma représente une implémentation de *Von Neumann* de l'architecture ARM (un seul bus de données pour les instructions/données). Il existe des implémentations de Harvard.
- L'architecture est de type *load/store* (typique des RISC). C'est-à-dire que les instructions s'exécutent uniquement sur les registres et les accès externes sont seulement pour transférer des données vers/depuis les registres.
- Les données sont sur 32 bits, les entiers peuvent être signés. Lorsqu'une donnée sur 8 ou 16 bits est transférée vers les registres sur 32 bits et en étendant le signe.
- Les instruction load/store utilisent l'ALU pour générer une adresse qui est maintenue dans le registre d'adresse et sur le bus d'adresse externe.  $R$
- Le registre  $R_m$  peut être modifié avec le registre à décalage avant d'être utilisé par l'ALU

# Les registres

Les registres accessibles dépendent du mode de fonctionnement du processeur.

**Mode Utilisateur (user mode):** les registres sont 32 bits, les registres rx sont à usage général (general purpose). Les registres r13,r14 et r15 sont utilisés comme pointeur de pile (sp=stack pointer), adresse de retour de sous-routine (lr=link register) et de compteur de programme (pc=program counter) pointe sur l'adresse de la prochaine instruction à charger.

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 sp
r14 lr
r15 pc

Les registres r0-r13 sont **orthogonaux** c'est-à-dire que toutes les instructions qui s'exécutent sur r0 peuvent s'exécuter sur les autres registres.

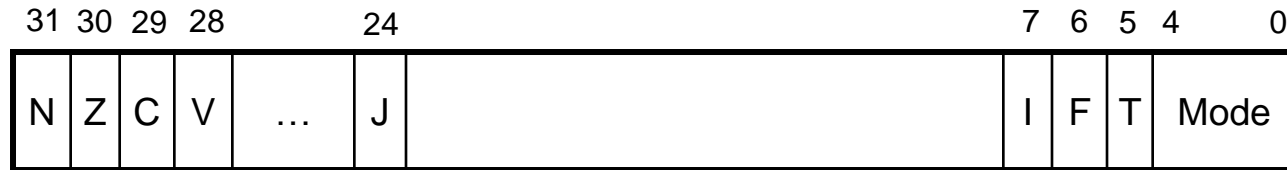
cpsr
-



registres d'états (un registre et sa copie)

# Registre cpsr

## Current Program Status Register



**N,Z,C,V:** drapeau de condition (condition flags) positionné par l'exécution d'instruction selon le résultat (le résultat est nul, overflow, ...)

**I,F:** masques d'interruptions

**T:** Thumb state (voir plus loin)

**Mode:** mode de fonctionnement du processeur (abort, fast interrupt request, interrupt request, supervisor, system, undefined et user, **les six premiers sont des modes privilégiés**)

# Modes de fonctionnement du processeur

**abort:** lorsque le processeur détecte qu'il n'arrive pas à accéder la mémoire externe

**(fast) interrupt request:** correspondent aux deux niveaux de priorités disponibles pour le traitement des interruptions

**supervisor.** généralement le mode d'opération du système d'exploitation, c'est le mode après le reset (initialisation du proc.)

**system:** identique au mode user, excepté qu'on a accès en lecture-écriture au registre cpsr (en particulier on peut changer de mode)

**undefined:** état du processeur lorsqu'il doit exécuter une instruction qu'il ne peut pas décoder (undefined) ou qui n'est pas supportée par l'implémentation

**user:** mode de fonctionnement des applications le seul qui ne peut pas modifier les bits mode du cpsr.

**les différents modes correspondent à des accès privilégiés et à des registres particuliers.**

# Les registres - Modes

user/system

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 sp
r14 lr
r15 pc

registres cachés selon le mode courant du processeur

fast interrupt

r8_fiq
r9_fiq
r10_fiq
r11_fiq
r12_fiq
r13_fiq
r14_fiq

interrupt

r13_irq
r14_irq

supervisor

r13_svc
r14_svc

undefined

r13_undef
r14_undef

abort

r13_abt
r14_abt

cpsr
-

spsr_fiq	spsr_irq	spsr_svc	spsr_undef	spsr_abt
----------	----------	----------	------------	----------

saved program status register

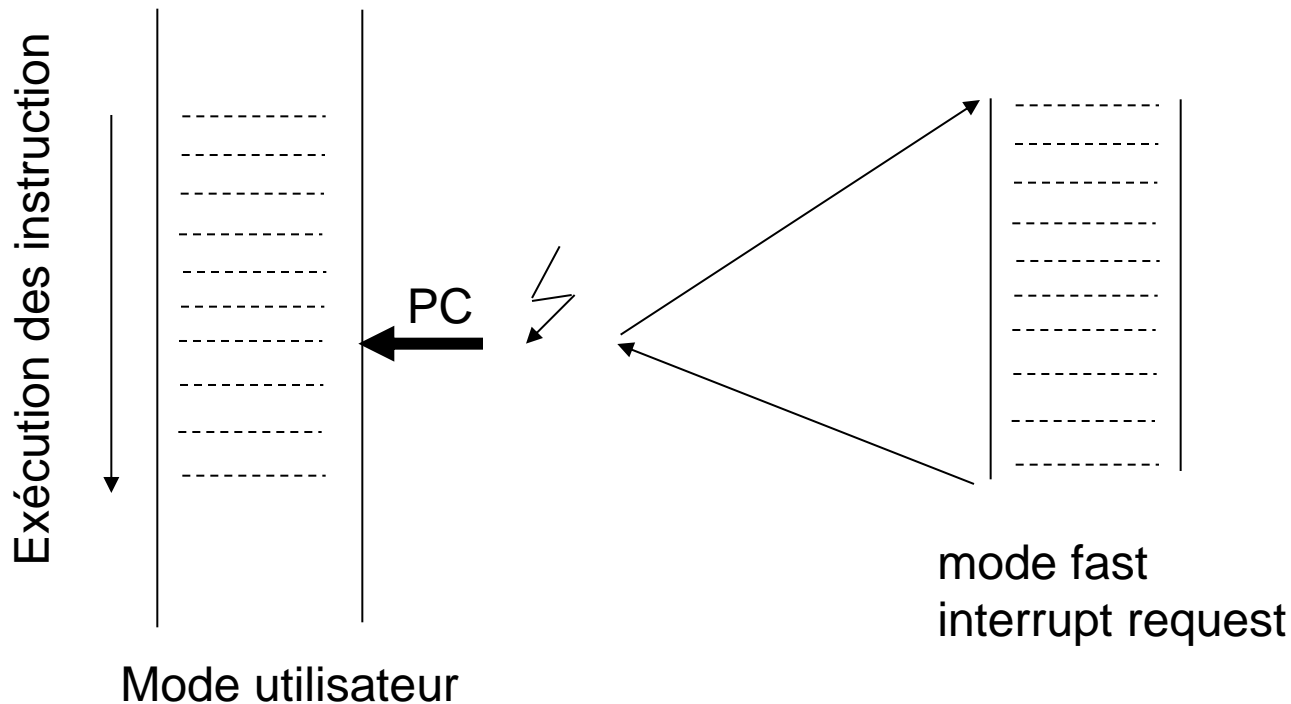
# Les registres - modes

lorsque l'on change de mode de fonctionnement les registres affectés par l'exécution des instructions peuvent changer.

Par exemple, en mode fast Interrupt request, les instructions qui modifient les registres r8 - r14 modifient réellement les registres r8\_fiq - r14\_fiq.

L'intérêt de ce mécanisme est que dans la routine d'interruption, on peut utiliser (modifier) les registres r8-r14 sans se soucier de préserver les valeurs de ces registres. Lorsque l'on quitte la routine de traitement de l'interruption et que l'on change de mode les registres sont restaurés.

# Les registres - modes



la routine peut modifier les registres r8-r14 sans se soucier de restaurer les valeurs a la fin du traitement. Les autres registres doivent être sauvegardés puis restaurer.

# Notion de contexte

Le **contexte d'exécution** d'une tâche ou d'une application est constitué des registres utilisés par la tâche pour réaliser sa fonction. Pour le processeur ARM, il s'agit des registre r0-r15 et des registres d'états.

Pour traiter une interruption, exécuter une routine, ou changer d'utilisateur (système partagé ou multitâche) il faut commuter (changer) le contexte, c'est-à-dire

1. sauvegarder les registres en mémoire du processus courant
2. retrouver et restaurer le contexte du nouveau processus (ou routine) à exécuter.
3. exécuter les instruction du nouveau processus

Si on se restreint à utiliser les registres dupliqués lors des changement de mode cette opération est prise en charge par le processeur (au niveau matériel) sinon on doit programmer l'opération.



# Modes résumé

Mode	Abbréviation	Privilégié	Mode[4:0]
abort	abt	oui	10111
fast interrupt request	fiq	oui	10001
interrupt request	irq	oui	10010
superviseur	svc	oui	10011
system	sys	oui	11111
undefined	und	oui	11011
utilisateur	usr	non	10000

# Etat du processeur et jeu d'instructions

Le processeur peut se trouver dans trois états différents: **ARM**, **Thumb**, **Jazelle**. Les changements d'états sont provoqués par l'exécution d'instructions particulières

A chacun des états correspond un jeu d'instruction particulier

ARM: jeu d'instruction sur 32 bits que l'on étudiera

Thumb: jeu d'instructions codé sur 16 bits, permet de programmer un système avec des mémoires 16 bits sans doubler les accès en mémoire externe, augmente la densité du code d'un facteur 30%.

Jazelle: jeu d'instruction sur 8 bits destiné à améliorer les performances de programmes écrits en bytecodes Java (environ 60% du bytecode est implémenté en hardware, le reste est software)

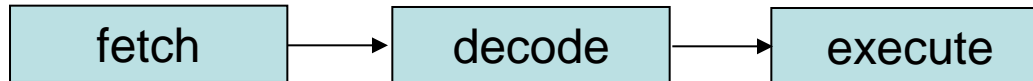
# comparaison

	<b>ARM</b>	<b>Thumb</b>
<b>taille des instructions</b>	32 bits	16 bits
<b>#instructions</b>	58	30
<b>instructions conditionnelles</b>	la plupart	les instructions de branchement
<b>instructions traitement des données</b>	accès au registre à décalage et à l'ALU	accès séparé à l'ALU et registre à décalage
<b>CPSR</b>	lecture/écriture en mode privilégié	pas d'accès direct
<b>utilisation des registres</b>	15 registres généraux, pc	8 registres généraux, 7 registres 'hauts', pc

# Pipeline

L'exécution d'une instruction se décompose en

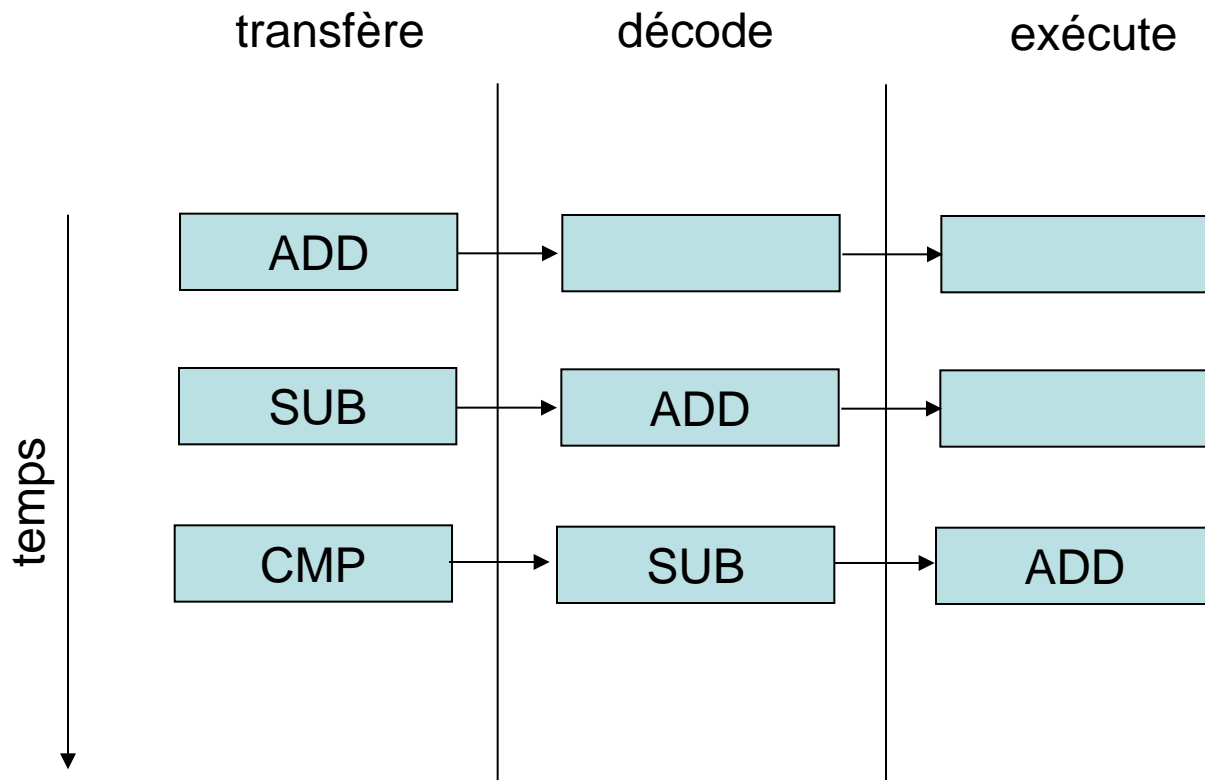
1. transférer l'instruction de la mémoire dans le registre dédié au décodage es instructions
2. décodage de l'instruction
3. exécution de l'instruction



le pipeline de l'ARM7

# Pipeline

Exemple: on a trois instruction ADD, SUB, CMP qui se suivent



# Pipeline

- la longueur du pipeline détermine les caractéristiques de l'exécutions
  - une instruction est exécutée uniquement après qu'elle ait passé tout les étages du pipeline. A l'initialisation le pipeline doit se 'remplir' avant la première exécution, a cela correspond un temps de **latence**. (*idem lorsqu'il y a rupture de séquence avec une instruction de branchement*)
  - Il se peut qu'il y ait des **dépendances** entre les données à l'intérieur du pipeline. Un étage du pipeline est bloqué car il doit connaître le résultat d'une instruction elle aussi dans le pipeline. Le compilateur est responsable de limiter de telles dépendances .
  - Les cœurs ARM9 et ARM10 utilisent des pipelines de 5 et 6 étages respectivement

# Pipeline - pc

Le registre pc contient l'adresse de la prochaine instruction à transférer de la mémoire vers le processeur. Si l'instruction en cours d'exécution se trouvait à l'adresse  $n$ , le pc pointe sur l'adresse  $n + 8$  ( mode thumb c'est  $+ 4$ ).

Si le pc est modifié, directement en utilisant une instruction ADD par exemple ou en exécutant une instruction de rupture de séquence (B), le pipeline est vidé et la prochaine instruction chargée depuis la mémoire. Dans ce cas, 2 cycles supplémentaires seront nécessaires pour 'remplir' le pipeline (temps de latence).

Si une interruption est levée, le processeur termine l'exécution de l'instruction en cours, vide le pipeline et traite l'interruption.

# Exceptions - interruptions

Lorsqu'une interruption ou une exception est levée le processeur suspend l'exécution du programme courant et exécute l'instruction située à l'adresse correspondante dans la table des vecteurs d'interruptions.

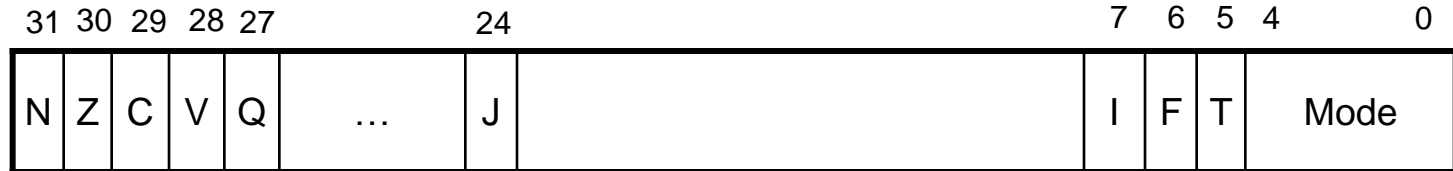
- **Reset:** adresse du code d'initialisation après mise sous tension du processeur
- **Undefined instruction:** le processeur est incapable de décoder une instruction
- **Prefetch abort:** le processeur n'est pas autorisé à transférer une instruction à l'adresse désirée
- **Data abort:** idem à prefetch abort excepté que l'accès mémoire est pour une donnée
- **Interrupt request:** interruption matérielle externe
- **Fast interrupt request:** idem interrupt request mais temps de réponse plus court



# Exceptions - interruptions

Exception/interruption	mnémonique	adresse	adresse2
reset	RESET	0x00000000	0xffff0000
undefined instruction	UNDEF	0x00000004	0xffff0004
software interrupt	SWI	0x00000008	0xffff0008
prefetch abort	PABT	0x0000000c	0xffff000c
data abort	DABT	0x00000010	0xffff0010
reserved	----	0x00000014	0xffff0014
interrupt request	IRQ	0x00000018	0xffff0018
fast interrupt request	FIQ	0x0000001c	0xffff001c

# Retour sur le cpsr



**F,T:** masque d'interruption pour les *fast interrupt request* et *interrupt request*

Flags (drapeaux) de conditions **NZCV:**

ces drapeaux sont mis-à-jour en exécutant les **instructions de comparaison** et par les **résultats obtenus par l'ALU**.

Par exemple: si le résultat de l'exécution d'une instruction *SUBS* (soustraction) est nul alors le flag Z est positionné à 1

**Q: Saturation et/ou overflow**

**V: Overflow signé**

**C: Carry (retenu)**

**Z: Zero**

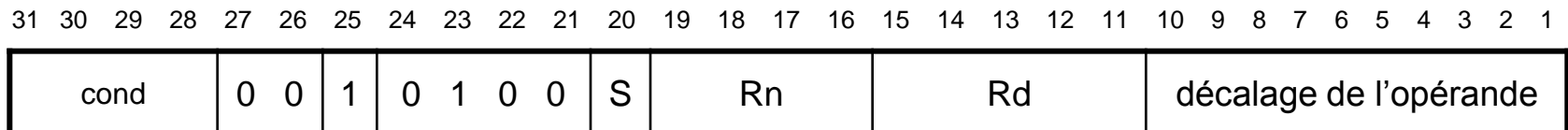
**N: Negative**

# les instructions ARM

Nous ne considérons que le jeu d'instruction ARM codé sur 32 bits. Les caractéristiques générales sont:

- exécution conditionnelle
- instructions comprenant deux opérandes sources et une opérande destination
- opération de décalage sur une opérande source (barrel register)
- saut de programme (relatif) de + ou – 32 Mbytes

codage des instructions, exemple une instruction ADD



# Exemple ADD

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
cond				0	0	1	0	1	0	0	S	Rn				Rd				décalage de l'opérande										

cond : il s'agit d'une instruction conditionnelle, elle est exécutée seulement si cette condition est remplie

27-21 : code de l'opération (opcode)

20 : si S=1 le résultat modifie le registre d'état cpsr

19-16 : numéro du registre source 1

15-11 : numéro du registre de destination

10-0 : source 2, peut-être une constante, un décalage par rapport au contenu d'un registre

# exemple ADD

ADD        R1,R1,#1         $R1 = R1 + 1$

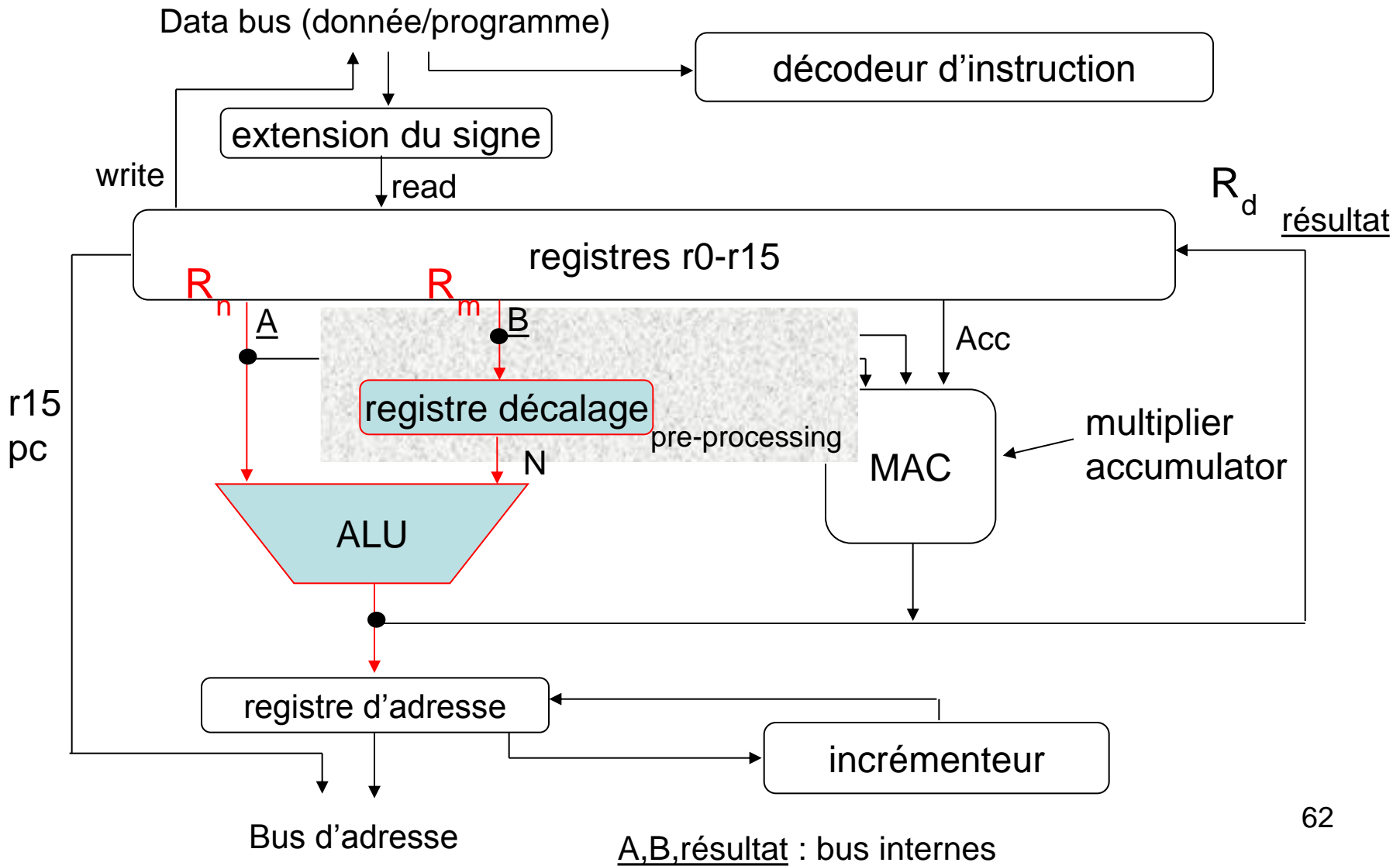
ADDNE     R2,R3,R4        si  $Z=0$   $R2 = R3 + R4$  sinon NOP

ADD        R1,R6,R6,LSL,#5  $R1 = R6 + R6 \ll 5 = R6(32+1)$

ADDEQS    R1,R2,R3         $R1 = R2 + R3$  si  $Z=1$  sinon NOP

les indicateurs N,Z,C,V sont positionnés  
dans le registre cpsr

# pré-processing des données



# Les conditions

cond [31:28]	mnémonique	signification	drapeau testé (cpsr)
0000	EQ	égal	Z=1
0001	NE	pas égal	Z=0
0010	CS/HS	retenue/plus grand ou égal (non signé)	C=1
0011	CC/LO	pas de retenue/plus petit (strict et non signé)	C=0
0100	MI	moins/négatif	N=1
0101	PL	plus/positif ou nul	N=0
0110	VS	overflow/ dépassement de capacité	V=1
0111	VC	no overflow/pas de dépassement de capacité	V=0
1000	HI	plus grand non signé	C=1 et Z=0
1001	LS	plus petit ou égal non signé	C=0 ou Z=1
1010	GE	plus grand ou égal signé	N=V
1011	LT	plus petit signé	N != V
1100	GT	plus grand signé	Z=0 et (N=V)
1101	LE	plus petit ou égal signé	Z=1 ou (N != V)
1110	AL	pas de condition	toujours

# Les conditions

EQ = equal

NE = not equal

CS = carry set (=1)

CC = carry clear (=0)

MI = minus

PL = plus

VS = V set (=1)

VC = V clear (=0)

HI = higher

LS = lower or same

GE = greater or equal

LT = less than

GT = greater than

LE = less or equal

AL = always



# Exemples

ADDNE R1,R2,R3       $R1=R2+R3$  si  $Z=0$

ADDVS R8,R9,R10     $R8=R9+R10$  si  $V=1$

ADDAL R4,R5,R6       $R4=R5+R6$  toujours

ADD     R2,R5,R8       $R2=R5+R8$  toujours par défaut

Ce mécanisme permet d'éviter les branchements conditionnels

	BNE	suite	}	= ADDNE R1,R2,R3
	ADD	R1,R2,R3		
suite	.....			

# Les drapeaux

les conditions portent sur des drapeaux (V=1, C=0, etc) qui s'exprime sous la forme overflow/dépassement de capacité signé, etc... Le code général est:

**<SignedOverflow>**: le résultat d'une opération arithmétique provoque un dépassement de capacité signé par exemple:  $0x7ffffff + 1 = 0x80000000$ , la somme des deux nombres positifs est négative. typiquement le drapeau **V** du cpsr indique un tel dépassement

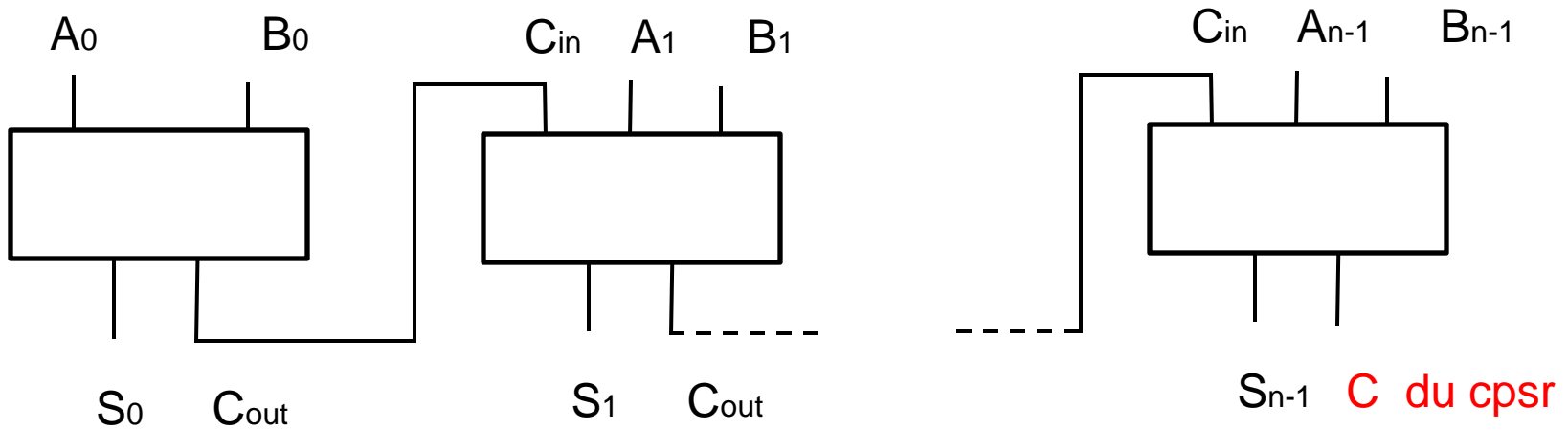
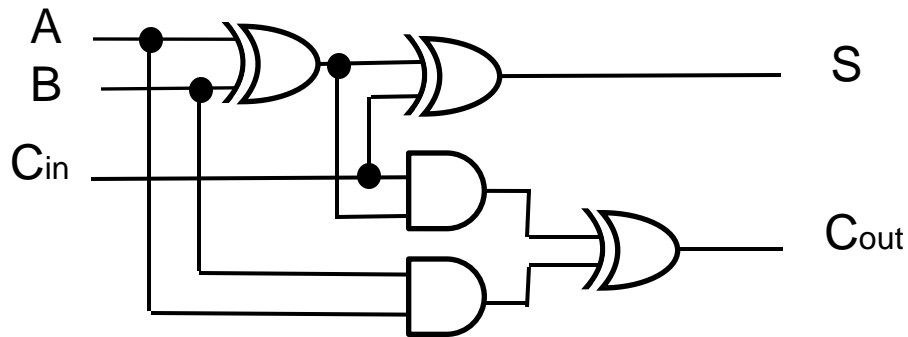
**<UnsignedOverflow>**: le résultat d'une opération arithmétique provoque un dépassement de capacité nonsigné par exemple:  $0xffffffff + 1 = 0$  le résultat ne peut pas être codé sur 32 bits. Typiquement le drapeau **C** du cpsr indique un tel dépassement

# Les drapeaux

**<NoUnsignedOverflow>**:  $1 - \text{<UnsignedOverflow>}$

**<Zero>**: le résultat d'une opération arithmétique ou logique est nul.  
Typiquement le drapeau **Z** indique un tel résultat

**<Negative>**: le résultat d'une opération arithmétique ou logique est négatif = le bit 31 du résultat est positionné à un ou encore **<Negative>** = bit 31 du résultat. typiquement, le bit **N** du cpsr indique un tel résultat



$$A+B=S$$

## Bit C

- Pour une addition s'il y a retenue,  $a+b > \text{max}$
- Pour une soustraction  $a-b$  s'il y a une retenue

Le bit C suppose une arithmétique **non signée**  $[0, 2^{32}-1]$

## Bit V

- Pour une addition, est positionné si les opérandes sont du même signe qui n'est pas celui du résultat.

Le bit V suppose une arithmétique en **complément à deux**

En complément à deux  $-b = \text{max} - b + 1$  où  $\text{max}$  est la valeur non-signée la plus grande ( $2^n - 1$  sur  $n$  bits).

La définition en complément à deux utilise le fait que

(1)  $\text{max} + 1 = 0$ , alors  $-b = 0 - b = \text{max} + 1 - b$ .

La définition (1) est vraie si les nombre sont non-signés.  
On peut l'utiliser pour calculer une soustraction en utilisant un additionneur.

$a - b = a + (\text{max} - b + 1)$  on a réduit la soustraction à une addition de deux nombres positifs. Il n'est pas nécessaire de créer un soustracteur.

Comment calculer  $\max-b+1$  ? En binaire  $\max = 111..111$

$$\begin{array}{ccccccc}
 \text{---} & 1 & 1 & 1 & 1 & \dots & 1 \\
 & b_{n-1} & b_{n-2} & b_{n-3} & b_{n-4} & & b_0 \\
 \hline
 & \overline{b_{n-1}} & \overline{b_{n-2}} & \overline{b_{n-3}} & \overline{b_{n-4}} & & \overline{b_0}
 \end{array}$$

Ensuite on ajoute 1.

Pour calculer le complément à deux on doit inverser les bits puis ajouter 1.

Les nombres sont contenus dans l'intervalle  $[-2^{(n-1)}, 2^{(n-1)}-1]$

En complément à deux, on a overflow ( $V=1$ ) si le résultat de l'addition ne satisfait pas:

$$- 2^{(n-1)} \leq a+b \leq 2^{(n-1)}-1$$

$$a+b > 2^{(n-1)}-1 \iff a > 2^{(n-1)}-1 - b$$

$$\Rightarrow (\text{comme } b < 2^{(n-1)}) \quad a > 2^{(n-1)}-1 - 2^{(n-1)} \geq 0, \text{ } a \geq 0$$

De même on a  $b \geq 0$

La condition  $a+b > 2^{(n-1)}-1$  entraîne que le bit de poids fort de  $a+b$  est 1, i.e. **le nombre est négatif en complément à deux.**



De même on montre que l'autre conditions n'est pas satisfaite seulement si  $a < 0$ ,  $b < 0$  et le résultat de  $a+b$  est positif.

En résumé, le bit  $V=1$  indique que les deux opérandes de  $a+b$  ont le même signe qui est opposé au signe du résultat.

Instruction **CMP r0,r1** positionne les drapeaux comme le résultat de  $r0-r1$ .

EQ/NE: si le résultat est nul (**Z=1**) on a bien  $r0=r1$  sinon  $r0 \neq r1$ .

CS/HS: Pour calculer  $r0-r1$  le processeur calcule  $r0+\max(-r1+1, 0)$ . La retenue (Carry) **C=1** si  $r0-r1 \geq 0$ , c'est-à-dire  $r0 \geq r1$ .  $C=1$  indique bien que  $r0 \geq r1$ .

CC/LO: La négation d'au-dessus. **C=0** si  $r0-r1 < 0$ , c'est-à-dire  $r0 < r1$ .

HI: Le résultat de  $r0-r1$  implique  $C=1$  et  $Z=0$ .  $C=1$  implique  $r0 \geq r1$  et  $Z=0$  implique que l'inégalité est stricte, donc  $r0 > r1$  (higher). Le résultat est non signé car on teste  $C$ .

LS:  $C=0$  ou  $Z=1$ . c'est-à-dire  $r0 < r1$  ou  $r0 = r1$ , soit  $r0 \leq r1$ .

GE: Si  $N=V=1$ . le résultat de  $r0-r1$  est négatif ( $N=1$ ) et  $r0 \geq 0$  et  $-r1 \geq 0$  ( $V=1$ ). Donc  $r1 \leq 0$  et on a bien  $r0 \geq r1$  signé (en fait  $r0 > r1$ )

Si  $N=V=0$ . le résultat de  $r0-r1$  est positif ( $N=0$ ). Comme  $V=0$  alors le résultat de  $r0-r1$  est correct (pas d'overflow) alors on a bien  $r0-r1 \geq 0$ , i.e.  $r0 \geq r1$ .

LT:  $N \neq V$ . Si  $N=0$  et  $V=1$ . Alors le résultat de  $r_0-r_1$  est positif ( $N=0$ ) et  $r_0 < 0$  et  $-r_1 < 0$  ( $V=1$ ). Alors  $r_1 > 0$  et on a bien  $r_0 < r_1$ .

Si  $N=1$  et  $V=0$ . Alors le résultat de  $r_0-r_1 < 0$  (pas d'overflow) et donc  $r_0 < r_1$ .

GT: ( $Z=0$ ) et ( $N=V$ ), plus grand ou égal ( $N=V$ ) et pas égal ( $Z=0$ )

LE:  $Z=1$  ou  $N \neq V$ , plus petit ( $N \neq V$ , LT) ou égal ( $Z=1$ )

# Les instructions de manipulation des données

manipulation des données entre registres.

- déplacement (MOVE)
- instructions arithmétiques
- instructions logiques
- instructions de comparaison
- instruction multiplication

le registre à décalage (barrel shifter) est généralement disponible

## Déplacement/MOVE

syntaxe: <instruction>{<cond>}{S}

↑  
optionnel

Rd, N

↑    ↑  
registre destination

opérande source

Mise à jour du registre cpsr (S=Statut)

# Exemple

MOV	$Rd = N$
MOVN	$Rd = \sim N$

MOV    r7,r5      r7=r5 copie le contenu de r5 dans r7

↑  
l'opérande source est un simple **registre**

MOV    r7,#34      r7=34

↑  
une constante, **valeur immédiate**

# MOV valeur immédiate

Lorsque l'opérande source de l'instruction MOV est une valeur immédiate, cette valeur ne peut pas être quelconque. La valeur immédiate est une valeur sur 32 bits qui est obtenue en utilisant

- une valeur immédiate sur 8 bits `imm_8`
- une valeur immédiate sur 4 bits `imm_4`, qui permet de calculer le nombre de décalage vers la droite (ROR) de `imm_8`

Plus précisément,

`valeur_immédiate = imm_8 ROR 2*imm_4`

par exemple: `0xff`, `0x104`, `0xe0000005`, `0xbc00000`

et pas: `0x101`, `0x102`

# Mise à jour du cpsr

Si l'instruction le spécifie (MOVS, par exemple) le registre d'état cpsr est mis à jour en fonction de la valeur du registre de destination (résultat).

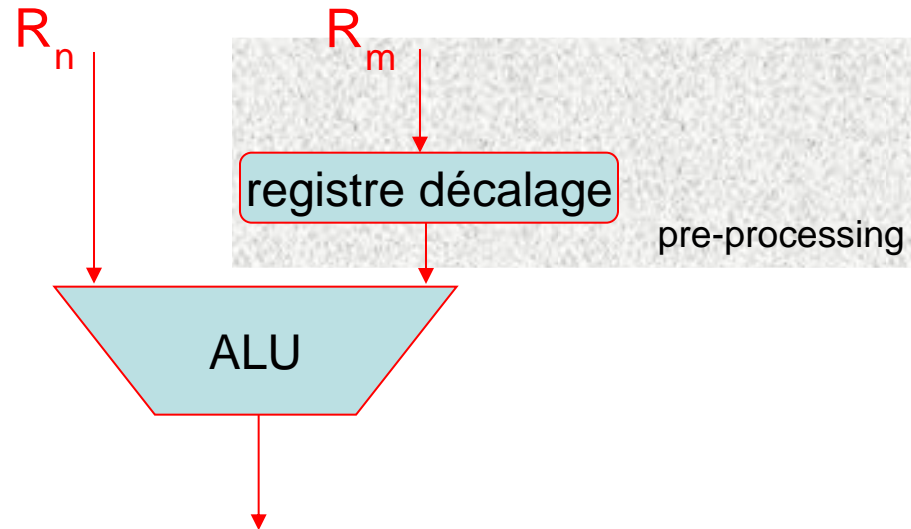
N = <negative> (bit 31)

Z = <zero>

V n'est pas modifié



# Registre à décalage



le pré-processing des données est exécuté sans augmenter la durée d'exécution des instructions

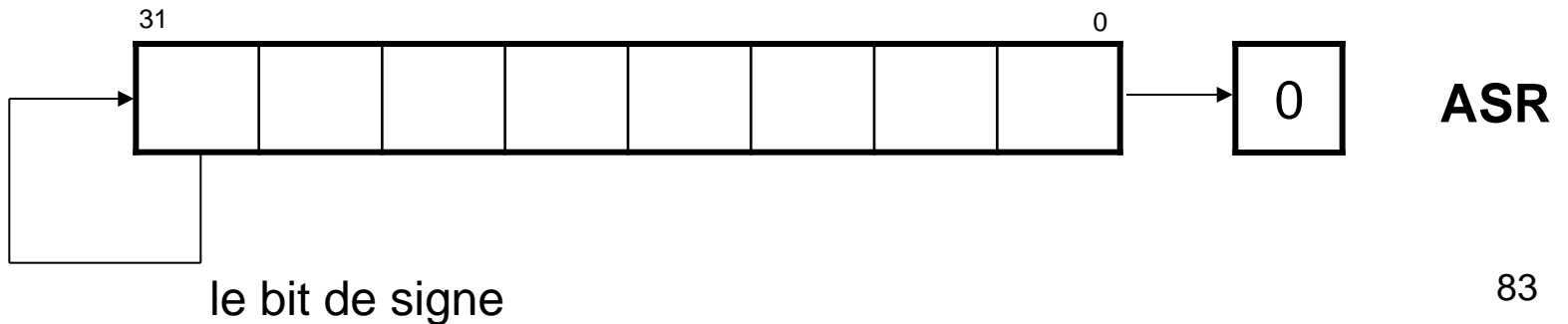
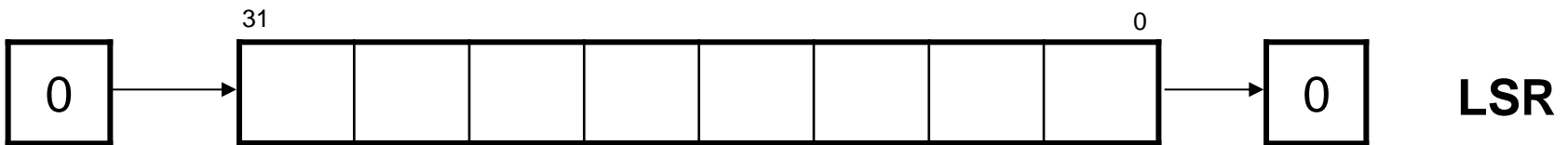
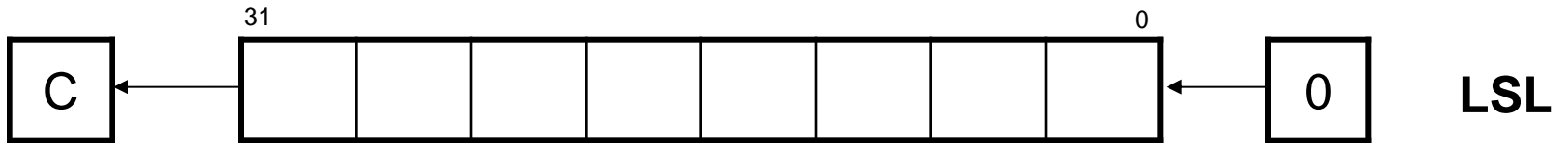
# Registre à décalage

constante ou valeur immédiate

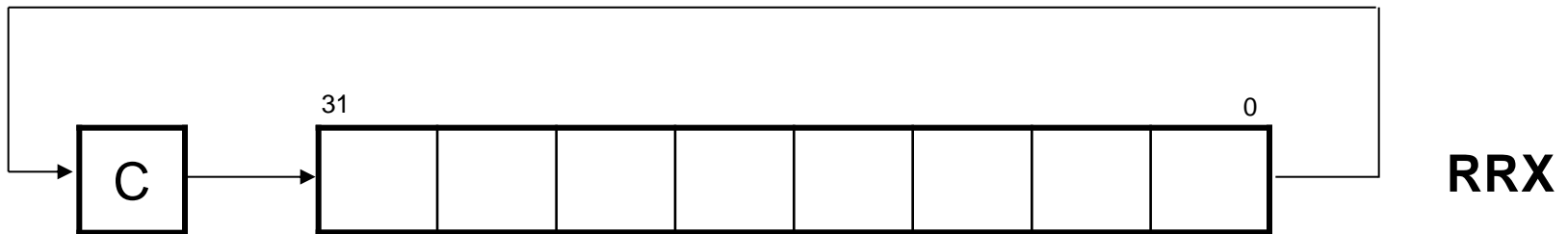
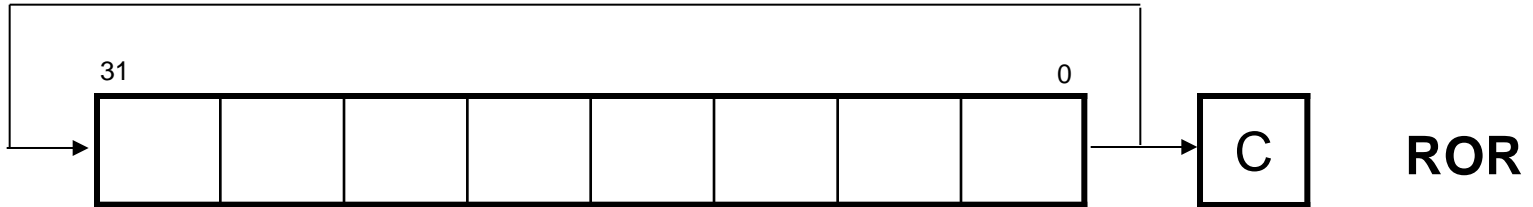


mnémonique	description	décalage	résultat	nombre de décalage
LSL	décalage logique à gauche	$xLSLy$	$x \ll y$	#0-31 ou Rs
LSR	décalage logique à droite	$xLSRy$	$(\text{unsigned})x \gg y$	#1-32 ou Rs
ASR	décalage arithmétique à droite	$xASRy$	$(\text{signed})x \gg y$	#1-32 ou Rs
ROR	décalage à droite	$xRORy$	$((\text{unsigned})x \gg y) \mid (x \ll (32-y))$	#1-31 ou Rs
RRX	décalage à droite étendu	$xRRX y$	$(c \text{ flag} \ll 31) \mid ((\text{unsigned})x \gg 1)$	----

# Registre à décalage



# Registre à décalage



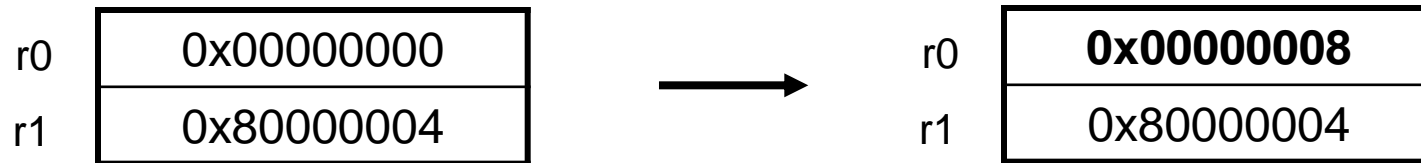
# 2<sup>ème</sup> opérande source

<instruction>{<cond>}{S} Rd, **N**

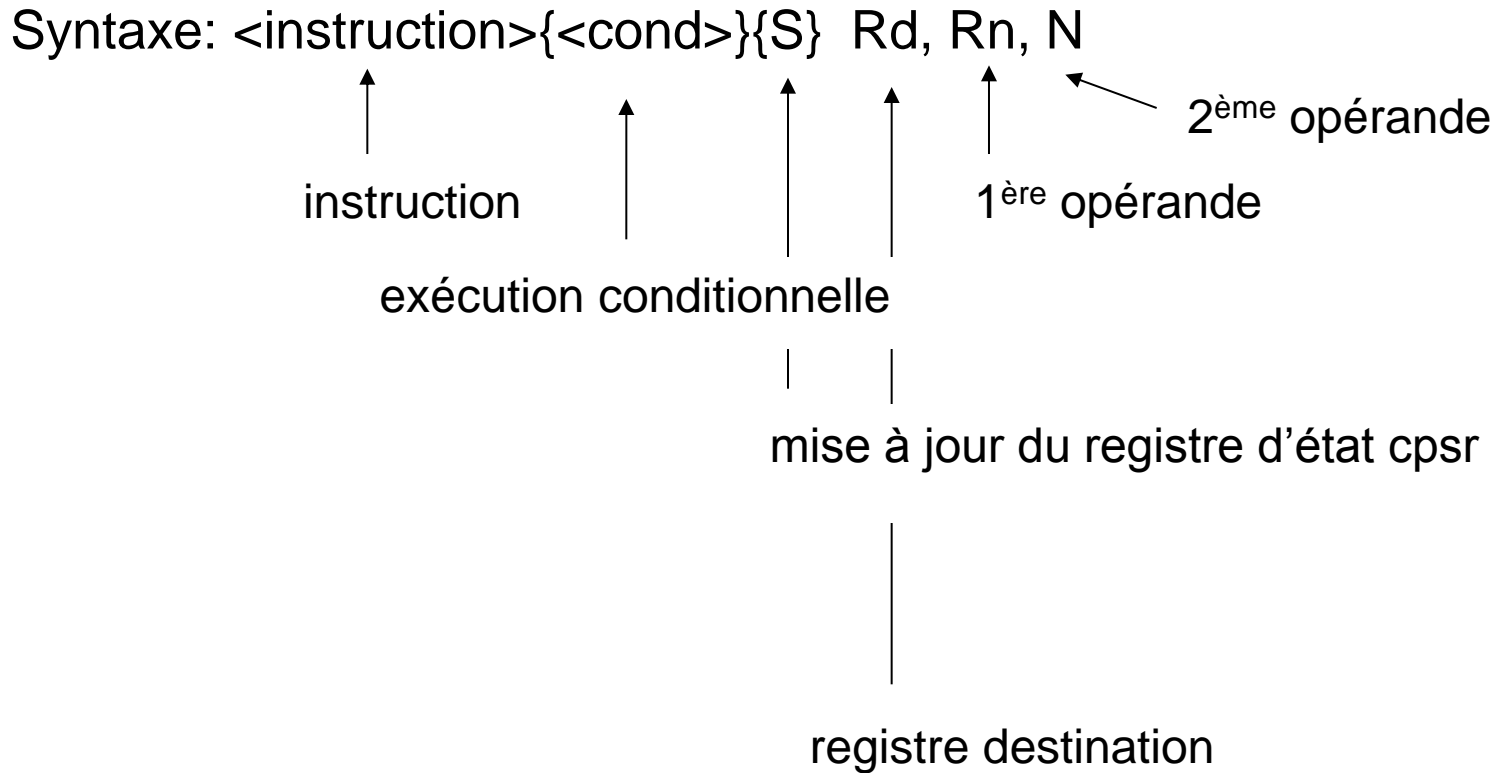
<b>N</b>	<b>syntaxe</b>
valeur immédiate	#valeur
registre	Rm
décalage logique gauche immédiate	Rm, LSL #val_imm
décalage logique gauche registre	Rm, LSL Rs
décalage logique droite immédiate	Rm, LSR #val_imm
décalage logique droite registre	Rm, LSR Rs
décalage arithmétique droite immédiate	Rm, ASR #val_imm
décalage arithmétique droite registre	Rm, ASR Rs
rotation droite immédiate	Rm, ROR #val_imm
rotation droite registre	Rm, ROR Rs
Rotation droite avec extension	Rm, RRX

# Example

MOVS r0,r1, LSL #1



# Instructions arithmétiques



# Instructions arithmétiques

ADC	adition 2 opérandes* + retenue	$Rd = Rn + N + C$
ADD	addition 2 opérandes*	$Rd = Rn + N$
RSB	soustraction inverse 2 opérandes*	$Rd = N - Rn$
RSC	soustraction inverse 2 opérandes* avec retenue	$Rd = N - Rn - !(C)$
SBC	soustraction 2 opérandes* avec retenue	$Rd = Rn - N - !(C)$
SUB	soustraction 2 opérandes*	$Rd = Rn - N$

\* opérandes sur 32 bits



# Exemples

SUB r0, r1, r2

r0	0x00000000
r1	0x00000002
r2	0x00000001



r0	<b>0x00000001</b>
r1	0x00000002
r2	0x00000001

RSB r0, r1, #0

r0	0x00000000
r1	0x00000077



r0	<b>0xffffffff89</b>
r1	<b>0x00000077</b>

complément à 2

les bits C=<UnsignedOverflow>, V=<SignedOverflow>, N=<Negative> et Z = <Zero> sont positionnés

# Exemple

Programmation de boucles:

```
MOV    r1, N
```

loop

; le corps de la boucle  $r1 = N, N-1, \dots, 1$

```
SUBS   r1, r1, #1
```

```
BGT    loop
```

met à jour les drapeaux **Z**, **N**, **V** du cpsr.

B = branch,  
instruction de branchement

instruction conditionnelle, GT test **Z=0** et (N=V)

# Exemple

Programmation de boucles 2:

```
SUBS  r1, N, #1
```

loop

; le corps de la boucle  $i = N-1, N-2, \dots, 0$

```
SUBS  r1, r1, #1
```

$\#2, \#3, \dots$  pour modifier le pas

```
BGE   loop
```

met à jour le drapeaux **Z,N,V** du cpsr.

B = branch,  
instruction de branchement

instruction conditionnelle, GE test **N=V**  
(**V** overflow signé)

# Exemples

1. ADD r1, r2, r3, LSL #3

$$r1 = r2 + 8 * r3$$

2. ADD r1, r2, r3, LSR #4

$$r1 = r2 + r3/16 \text{ (non signé)}$$

3. ADD r1, r2, r3, ASR #4

$$r1 = r2 + r3/16 \text{ (signé)}$$

# Examples

4.      ADD    r1,r2,r3, LSL r4

$$r1 = r2 + 2^{r4} * r3$$

# Instructions logiques

Les instructions logiques effectuent des opérations bit-à-bit (bitwise)

Syntaxe: <instruction>{<cond>}{S} Rd, Rn, N

AND	ET logique bit-à-bit de deux opérandes*	$Rd = Rn \& N$
ORR	OU logique bit-à-bit de deux opérandes*	$Rd = Rn   N$
EOR	OU Exclusif bit-à-bit de deux opérandes*	$Rd = Rn \wedge N$
BIC	Bit Clear (AND NOT) de deux opérandes*	$Rd = Rn \& \sim N$

\* de 32-bits

les bits **Z** et **N** du cpsr sont mis-à-jour

# Exemples

ORR r0, r1, r2

r0	0x00000000
r1	0x02040608
r2	0x10305070



r0	<b>0x12345678</b>
r1	0x02040608
r2	0x10305070

BIC r0, r0, #1<<22

0x00400000



r0	0xffff0000
----	------------



r0	0xffbf0000
----	------------

l'opération consiste à effacer (= mettre à zéro) du registre Rn les bits qui sont positionnés à des positions correspondant à des 1 dans l'opérande N. L'opérande N agit comme un masque. Une application est l'activation/désactivation des interruptions.

# Instructions de comparaisons

Les instructions de comparaisons permettent de comparer ou tester un registre avec une valeur sur 32 bits. Le registre *cpsr* est mis à jour et **aucun** autre registre est modifié (le suffixe S n'est pas nécessaire pour indiquer la mise à jour des drapeaux du cpsr). Pour les tests avec des valeurs immédiate, utiliser la même convention que pour MOV

**Syntaxe:** <instruction>{<cond>} Rn, N

CMN	comparaison négative	positionne les drapeaux de cpsr selon le résultat de $Rn + N$
CMP	comparaison	positionne les drapeaux de cpsr selon le résultat de $Rn - N$
TEQ	test si égalité opérandes 32 bits	positionne les drapeaux du cpsr selon le résultat de $Rn \wedge N$ (ou-ex)
TST	test bit-à-bit	positionne les drapeaux du cpsr selon le résultat de $Rn \& N$



# registre cpsr

**CMN:** positionne les bits **N**, **Z**, **C** = <unsignedOverflow>, **V** = <signedOverflow>

**CMP:** positionne les bits comme CMN excepté **C**=<NoUnsignedOverflow>

En effet,

$x-y = x+\sim y+1$ , il y a retenue ( $C=1$ ) si l'opération dépasse la capacité (non signé), c'est-à-dire si  $y \leq x$  entraîne un <UnsignedOverflow>. Donc,  $x-y$  ne dépasse pas la capacité.

$x=0011$ ,  $y=0010$   $x+\sim y+1 = 0011 + 1110 = 0001 + \text{retenue}$

$x=0011$ ,  $y=0100$   $x+\sim y+1 = 0011 + 1100 = 1111$  pas de retenue et résultat signé faux.

**TEQ, TST:** positionne les bits **Z**=<zero> et **N**=<negative>, sous certaines conditions (c.f. manuel) TEQ permet de tester l'égalité et de préserver le bit C.

# Instructions de multiplication

On a deux classes, les multiplications et les multiplications longues (le résultat est sur 64 bits). Le résultat est placé dans un registre ou une pair de registres

**Syntaxe:** MLA{<cond>}{S} Rd, Rm, Rs, Rn  
          MUL{<cond>}{S} Rd, Rm, Rs

MLA	Multiplication et accumulation	$Rd = (Rm * Rs) + Rn$
MUL	Multiplication	$Rd = Rm * Rs$

# Instructions de multiplication

Multiplications longues

**Syntaxe:** <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs

SMLAL	multiplication signée longue avec accumulation	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	multiplication signée longue	$[RdHi, RdLo] = Rm * Rs$
UMLAL	multiplication non signée longue avec accumulation	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	multiplication non signée longue	$[RdHi, RdLo] = Rm * Rs$

# example

MUL r0, r1, r2       $r0 = r1 * r2$

r0	0x00000000
r1	0x00000002
r2	0x00000002



r0	<b>0x00000004</b>
r1	0x00000002
r2	0x00000002

UMULL r0, r1, r2, r3       $[r1, r0] = r2 * r3$

r0	0x00000000
r1	0x00000000
r2	0xf0000002
r3	0x00000002



r0	<b>0xe0000004</b>
r1	0x00000001
r2	0xf0000002
r3	0x00000002

# Remarque sur les multiplications

le temps nécessaire à l'exécution d'une multiplication dépend de

1. l'implémentation du processeur = certaines réalisations sont plus optimum que d'autres
2. la valeur du registre Rs, on détermine
  1.  $M=1$  pour  $-2^{11} \leq R_s < 2^{11}$
  2.  $M=2$  pour  $-2^{23} \leq R_s < 2^{23}$
  3.  $M=3$  pour les autres valeurs de Rs

le nombre de cycle pour exécuter l'instruction est M pour les instructions MUL/MLA, 4 pour MULS/MLAS,  $1+M$  xMULL/xMLAL et 5 pour xMULLS/xMLALS

Attention, après l'exécution de MUL/MLA et xMULL/XMLAL on doit attendre 1 cycle supplémentaire avant de pouvoir effectuer une nouvelle multiplication

# Remarque sur les multiplications

Pour un processeur 68'000 la différence de temps d'exécution entre une addition et une multiplication est un facteur 8-9

MLAS, MULS mettent à jour N=<Negative, Z = <Zero>, C n'est pas prévisible, V est inchangé. En général il faut éviter de mettre à jour le cpsr car les implémentations peuvent nécessiter des cycles supplémentaires (utiliser une comparaison).

# Instructions de branchement

les instructions de branchement permettent de modifier le déroulement du programme (initialement séquentiel) pour effectuer des tests ou des appels de sous-routines.

Syntaxe: B{<cond>} label

BL{<cond>} label

BX{<cond>} Rm

BLX{<cond>} label | Rm

B	branch	$pc = label$
BL	branch + link	$pc = label$ $lr = \text{adresse de la prochaine instruction après BL}$
BX	branch exchange	$pc = Rm \ \& \ 0xffffffe, T = Rm \ \& \ 1$
BLX	branch exchange + link	$pc = label, T = 1$ $pc = Rm \ \& \ 0xffffffe, T = Rm \ \& \ 1$ $lr = \text{adresse de la prochaine instruction après BLX}$

# Instructions de branchement

Le **bit T** désigne le bit T du *cpsr* qui indique que le processeur est dans l'état **Thumb**, les instructions BX et BLX permettent donc de changer l'état du processeur selon que l'adresse est paire ou impaire.

Les branchements avec lien (BL et BLX) permettent d'effectuer des appels de sous-routine. Le label ou le registre indique l'adresse du début de la sous-routine et le registre lr du cpsr contient l'adresse de retour de la sous-routine, c'est-à-dire l'adresse de la prochaine instruction à exécuter au retour de la sous-routine.

Lorsque l'adresse est indiquée par un label, le déplacement est relatif par rapport à la valeur courante du pc, se déplacement doit être au maximum de +- 32MBytes.



# Instructions de branchement

**exemple:**

```
B    forward  
ADD  r1, r2, #4  
ADD  r0, r6, #2  
ADD  r3, r7, 34
```

forward

```
SUB  r1, r2, #4
```

# Instructions de branchement

exemple:

backward

ADD r1, r2, #4

SUB r1, r2, #4

ADD r4, r6, r7

B backward boucle infinie

# instructions de branchements

exemple: appel à une sous-routine

```
BL      sousroutine
```

```
CMP     r1, #5
```

```
MOVEQ   r1, #0
```

```
.
```

```
.
```

```
.
```

```
.
```

sousroutine

```
<code de la sous-routine>
```

```
MOV      pc, lr    move le contenu du registre lr dans le pc
```

# Instructions load- Store

les instructions **Load et Store** permettent le transfert de données entre les registres et la mémoire externes (périphériques,...).

il y a trois types d'instructions load-store

- transfert registre simple
- transfert registre multiples
- échange (swap)

# transfert registre simple

Single-register transfer

Syntaxe: <LDR | STR>{<cond>}{B} Rd, adresse1

LDR{<cond>}{SB | H | SH} Rd, adresse2

STR{<cond>}H Rd, adresse2

LDR	charge un mot vers registre	Rd <= mem32[adresse]
STR	sauve byte/mot depuis registre	Rd => mem32[adresse]
LDRB	charge un byte vers registre	Rd <= mem8[adresse]
STRB	sauve un byte depuis un registre	Rd => mem8[adresse]

# transfert registre simple

LDRH	charge un demi-mot vers registre	$Rd \leftarrow \text{mem16}[\text{adresse}]$
STRH	sauve un demi-mot depuis registre	$Rd \Rightarrow \text{mem16}[\text{adresse}]$
LDRSB	charge un byte signé vers registre	$Rd \leftarrow \text{signe étendu}(\text{mem8}[\text{adresse}])$
LDRSH	charge un demi-mot signé vers registre	$Rd \leftarrow \text{signe étendu}(\text{mem16}[\text{adresse}])$

# modes d'adressages

Le processeur ARM permet d'accéder la mémoire selon différents modes (modes d'adressages). Chacun de ces modes utilise une des méthodes suivantes pour l'indexage

- pré indexe avec mise à jour
- pré index
- post index

méthode	donnée	registre de base	exemple
pré index avec mise à jour	mem[base + offset]	base + offset	LDR r0, [r1, #4]!
pré index	mem[base + offset]	pas mis à jour	LDR r0,[r1, #4]
post index	mem[base]	base + offset	LDR r0, [r1],#4

# modes d'adressages

## adressage1 et méthode d'indexage:

valeur immédiate sur 12 bits

pré index avec offset immédiat:

$[Rn, \# \pm \text{offset}_{12}]$

pré index avec registre d'offset:

$[Rn, \pm Rm]$

pré index avec registre d'offset et registre à décalage:

le registre d'adresse de base  
 $[Rn, \pm Rm, \text{shift } \# \text{shift}_{\text{imm}}]$

pré index avec offset immédiat et mise à jour :

$[Rn, \# \pm \text{offset}_{12}]!$  LSL,...

pré index avec registre d'offset et mise à jour:

$[Rn, \pm Rm]!$



# modes d'adressages

## **adressage1 et méthode d'indexage (suite):**

pré index avec registre d'  
offset et registre à décalage  
et mise à jour:

$[R_n, \pm R_m, \text{shift } \# \text{shift\_imm}]!$

mode immédiat post indexé:

$[R_n], \# \pm \text{offset}_{12}$

post indexé avec registre:

$[R_n], \pm R_m$

Post indexé avec registre  
d'offset et registre à décalage:

$[R_n], \pm R_m, \text{shift } \# \text{shift\_imm}$

# modes d'adressages

Les modes d'adressages présentés sur les deux derniers transparents sont disponibles pour des instructions load/store de mot de 32 bits ou de byte non signé

+ - indique que l'offset est signé, l'offset spécifie le nombre de bytes

le registre d'adresse de base pointe sur un byte en mémoire

le mot *immédiate* (par exemple offset immédiat) indique que les calculs utilisent un offset de 12 bits encodé dans l'instruction (par d'accès mémoire supplémentaire)

# modes d'adressages

## mode d'adressage 2:

pré index avec offset immédiat:  $[Rn, \#+-offset\_8]$

pré index avec registre d'offset;  $[Rn, +-Rm]$

pré index avec offset immédiat  
et mise à jour:  $[Rn, \#+-offset]!$

pré index avec registre d'offset  
et mise à jour:  $[Rn, +-Rm]!$

post index immédiat:  $[Rn], \#+-offset\_8$

post index registre:  $[Rn], +-Rm$

# exemple

LDR r1, [r2, -r3]!

r1	0x00000000
r2	0x00100014
r3	0x00000008

adresse      valeur

100014	
100010	
1000C	0x9876C4A1
10008	
10004	

$0x00100014 - 0x8 = 0x0010000C$

mémoire

r1	<b>0x9876C4A1</b>
r2	<b>0x0010000C</b>
r3	0x00000008

# exemple

LDR r1, [r2], r3, LSL #3

r1	0x00000000
r2	0x00100000
r3	0x00000001

0x1 LSL #3 = 0x8

r1	<b>0x9876C4A1</b>
r2	<b>0x00100008</b>
r3	0x00000001

adresse      valeur

100010	
10000C	
100008	
100004	
100000	0x9876C4A1

mémoire

# exemple

lors d'un chargement d'un byte non signé les 24 bits de poids fort sont mis à 0

LDRB r1, [r2, #2]

r1	0xxxxxxxxx
r2	0x00100004

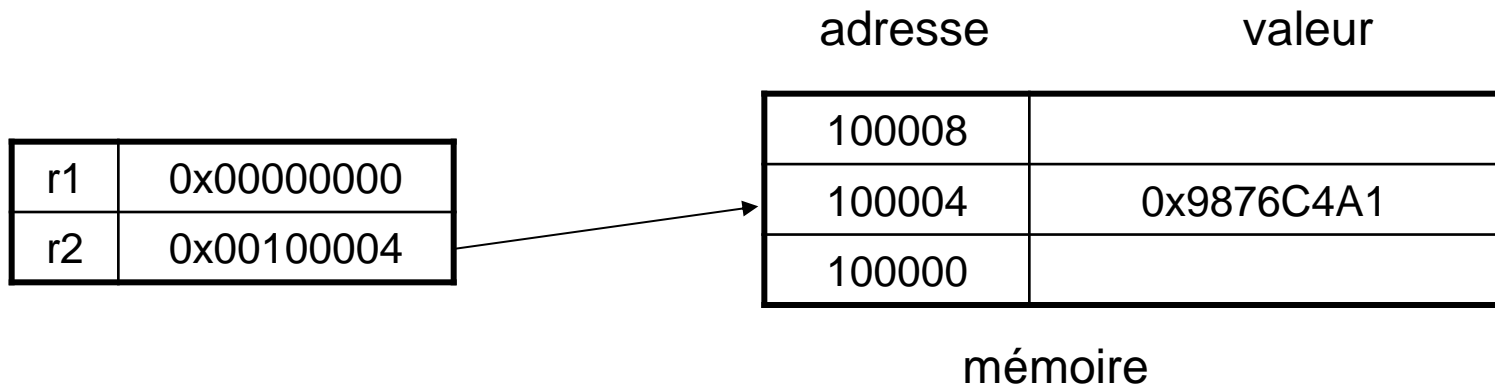
adresse	valeur
100008	
100004	0x9876C4A1
100000	

r1	0x000000C4
r2	0x00100004

# exemple

**Mot non-aligné:** lorsque l'adresse n'est pas multiple de 4, le mot entier est transféré dans le registre, mais le byte désigné par l'adresse devient le byte de poids fort

LDR r1, [r2, #2]



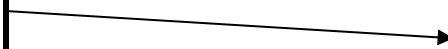
r1	0xC4A19876
r2	0x00100004

# exemple

**Mot non aligné:** lorsqu'un mot est stocké en mémoire à une adresse non divisible par 4, les deux derniers bits de l'adresse sont mis à zéro et le mot stocké à cette adresse

STR r1,[r2,#0]

r1	0x12345678
r2	0x00100007



adresse	valeur
100008	
100004	<b>0x12345678</b>
100000	



# exemples

pré index avec mise à jour

LDR r0, [r1, #0x4]!      r0=mem32[r1+0x4]      r1 += 0x4

LDR r0, [r1,r2]!      r0=mem32[r1+r2]      r1 += r2

LDR r0, [r1, r2, LSR#0x4]!    r0=mem32[r1+(r2 LSR 0x4)]    r1 += (r2 LSR 0x4)

pré index

LDR r0,[r1,#0x4]    r0=mem32[r1+0x4]      r1 pas mis à jour

LDR r0,[r1,r2]      r0=mem32[r1+r2]      r1 pas mis à jour

LDR r0,[r1, -r2,LSR #0x4]    r0=mem32[r1-(r2 LSR 0x4)]    r1 pas mis à jour

# examples

post index

LDR r0, [r1], #0x4    r0=mem32[r1]    r1 += 0x4

LDR r0,[r1],r2    r0=mem32[r1]    r1 += r2

LDR r0, [r1], r2, LSL #0x4    r0 = mem32[r1]    r1 += r2 LSL 0x4

# transferts de multiples registres

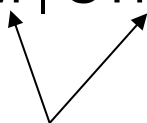
Une seule instructions peut transférer le contenu de plusieurs registres vers/depuis la mémoire. Le transfert débute depuis une adresse de base contenue dans un registre  $R_n$ .

Une interruption n'interrompt pas l'exécution d'une instruction de transfert multiple. Si  $N$  est le nombres de registres à transférer et  $t$  est le nombre de cycle nécessaires par accès alors les  $2+Nt$  cycles sont in-interruptibles.

Certains compilateurs permettent de limiter le nombre maximum de registres transférés ce qui permet de diminuer le temps maximum de latence du système (par exemple réponse à une interruption dans les systèmes temps-réels).

# transferts de multiples registres

syntaxe: <LDM | STM>{<cond>}<mode adressage> Rn{!}, <registres>{^}

  
multiple

mode adressage	description	adresse début	adresse fin	Rn!
IA	Incrémente après	$R_n$	$R_{n+4*n-4}$	$R_{n+4*N}$
IB	incrémente avant	$R_n + 4$	$R_{n+4*N}$	$R_{n+4*N}$
DA	décrémente après	$R_{n-4*N+4}$	$R_n$	$R_{n-4*N}$
DB	décrémente avant	$R_{n-4*N}$	$R_{n-4}$	$R_{n-4*N}$

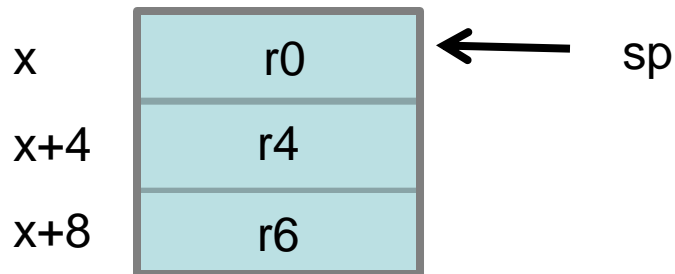
# transferts de multiples registres

Remarquez que l'adresse de début < l'adresse de fin

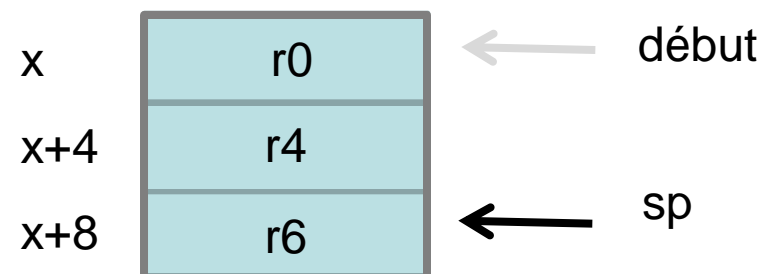
Pour un **STM**:

Les registres sont transféré dans l'ordre de leur index, i.e. {r0,r4,r6} -> d'abord r0, puis r4, puis r6.

L'adresse de **début** correspond au premier transfert ensuite les adresse sont toujours incrémentées



stmia sp,{r0,r4,r6}



stmda sp,{r0,r4,r6}

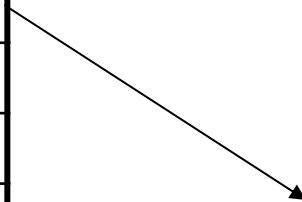
# exemple

LDMIA r0!, {r1-r3}

r0 est le registre de base

! indique que le registre r0 est mis à jour après l'exécution de l'instruction

r0	0x00080010
r1	0x00000000
r2	0x00000000
r3	0x00000000



adresse                      valeur

0x80020	0x00000005
0x8001c	0x00000004
0x80018	0x00000003
0x80014	0x00000002
0x80010	0x00000001

# example

r0	0x8001c
r1	0x00000001
r2	0x00000002
r3	0x00000003

r0 →

0x80020	0x00000005
0x8001c	0x00000004
0x80018	0x00000003
0x80014	0x00000002
0x80010	0x00000001

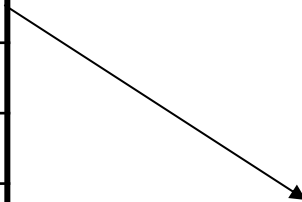
# exemple

LDMIB r0!, {r1-r3}

r0 est le registre de base

! indique que le registre r0 est mis à jour après l'exécution de l'instruction

r0	0x00080010
r1	0x00000000
r2	0x00000000
r3	0x00000000



adresse                      valeur

0x80020	0x00000005
0x8001c	0x00000004
0x80018	0x00000003
0x80014	0x00000002
0x80010	0x00000001



# example

r0	0x8001c
r1	0x00000002
r2	0x00000003
r3	0x00000004

r0 →

0x80020	0x00000005
0x8001c	0x00000004
0x80018	0x00000003
0x80014	0x00000002
0x80010	0x00000001

# exemple

les instructions STM<cond> et LDM<cond> doivent être utilisés par paires pour lire/écrire les mêmes registres, par exemple lors d'une sauvegarde sur la pile

STMIA – LDMDB

STMIB – LDMDA

STMDA – LDMIB

STMDB – LDMIA

# exemple

r0	0x00090000
r1	0x00000009
r2	0x00000008
r3	0x00000007

avant

STMIB r0!, {r1,r2,r3}

MOV r1, #1

MOV r2, #2

MOV r3, #3

r0	0x0009000c
r1	0x00000001
r2	0x00000002
r3	0x00000003

après

→	0x0009010	
	0x000900c	0x00000007
	0x0009008	0x00000008
	0x0009004	0x00000009
→	0x0009000	

# exemple

r0	0x0009000c
r1	0x00000001
r2	0x00000002
r3	0x00000003

avant

LDMDA r0!, {r1,r2,r3}

r0	0x00090000
r1	0x00000009
r2	0x00000008
r3	0x00000007



0x0009010	
0x000900c	0x00000007
0x0009008	0x00000008
0x0009004	0x00000009
0x0009000	

après

# exemple

copie d'une zone mémoire vers une autre zone

; le registre r9 pointe sur le début de la zone source

; le registre r10 pointe sur le début de la zone destination

; le registre r11 pointe sur la fin de la zone source

; les transferts sont effectués par blocs de 32 bytes

loop

; charger un bloc de 32 bytes depuis la mémoire

LDMIA r9!, {r0-r7}

; transférer le bloc vers la mémoire

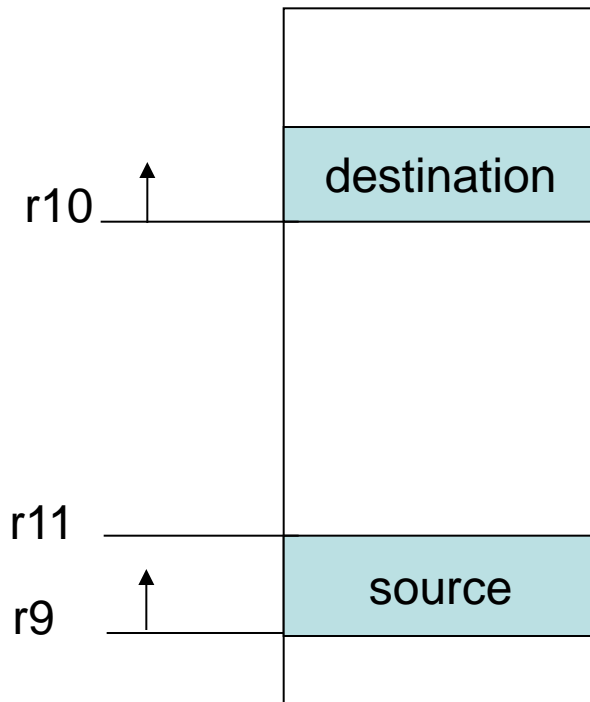
STMIA r10!, {r0-r7}

; a-t-on terminé?

CMP r9, r11

BNE loop

# example



# la pile

0x1000c		
0x10008		← 3
0x10004		← 2
0x10000		← 1
0x0fffc		
0x0fff8		
0x0fff4		

**pile ascendante *A***

0x1000c		
0x10008		
0x10004		
0x10000		← 1
0x0fffc		← 2
0x0fff8		← 3
0x0fff4		

**pile descendante *D***

# la pile

On parle de *pile pleine* **F** (*full stack*) si le pointeur de pile (*sp*) pointe sur le dernier élément qui a été empilé

On parle de *pile vide* **E** (*empty stack*) si le pointeur de pile (*sp*) pointe sur le premier élément non utilisé de la pile

les opérations de base sur la piles sont:

- *push* empilé un élément sur la pile
- *pop* dépile le dernier élément empilé

selon les options choisies pour définir la pile (A, D, F, E) les instructions pop et push sont codées différemment



# opérations sur la pile

Mode adressage	description	pop	=LDM	push	=STM
FA	full Ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

# exemple

r1 = 0x00000002

r4 = 0x00000003

sp = 0x00080014

STMFD sp!, {r1,r4}

sp →

0x00080018	0x00000001
0x00080014	0x00000002
0x00080010	empty
0x0008000c	empty

avant

sp →

0x00080018	0x00000001
0x00080014	0x00000002
0x00080010	0x00000003
0x0008000c	0x00000002

après

# exemple

r1=0x00000002

r4=0x00000003

sp=0x00080010

STMED sp!, {r1,r4}

sp →	0x00080018	0x00000001
	0x00080014	0x00000002
	0x00080010	empty
	0x0008000c	empty
	0x00080008	empty

avant

sp →	0x00080018	0x00000001
	0x00080014	0x00000002
	0x00080010	0x00000003
	0x0008000c	0x00000002
	0x00080008	empty

après

# attributs

les attributs à respecter pour une pile sont

- l'adresse de base
- le pointeur de pile
- l'adresse limite de la pile (stack limit)

pour une pile descendante, par exemple, le code suivant vérifie que la pile reste dans la limite, sinon appelle une routine `_stack_overflow`

```
SUB    sp, sp, #size    attention valeur immédiate sur 8 bits
CMP    sp, r10           registre r10 stack limit s/
BLLO   _stack_overflow
```



**Branch + Link + LO** est la condition plus petit strictement non signé

# exemples - remarques

le signe ^ modifie le comportement des instructions. Le processeur ne doit pas se trouver en mode utilisateur ou système (par exemple lors du traitement d'une interruption).

Dans ce cas les registres qui apparaissent dans la liste sont les registres en mode utilisateur. Si le pc est dans la liste, le spsr est copié dans le cpsr

LDMFD sp!, {r0-pc}^ retour d'une exception et mise a jour du cpsr

# Instruction swap

L'instruction *swap* est une instruction du type load/store spécialisée. Elle permute le contenu d'une position mémoire avec le contenu d'un registre. L'instruction est **atomique**, c'est-à-dire que les cycles de lecture et d'écriture s'effectuent pendant le même cycle bus. Aucune instruction peut modifier le registre ou la position mémoire avant la fin de l'instruction, le système garde le contrôle du bus.

Syntaxe: SWP{B}{<cond>} Rd, Rm, [Rn]

SWP	permute les mots d'une position mémoire et d'un registre	tmp = mem32[Rn] mem32[Rn] = Rm Rd = tmp
SWPB	permutation d'un byte	tmp = mem8[Rn] mem8[Rn] = Rm Rd = tmp

# exemple

mem32[0x9000] = 0x12345678

r0 = 0x00000000

r1 = 0x11112222

r2 = 0x00009000

SWP    r0, r1, [r2]

mem32[0x9000] = **0x11112222**

r0 = **0x12345678**

r1 = 0x11112222

r2 = 0x00009000

# exemple

L'instruction *swap* permet d'implémenter un mécanisme de sémaphore pour gérer l'accès à une ressource partagée.

## algorithme:

- l'adresse sémaphore contient #1 si la ressource est déjà utilisée et ne peut pas être accédée.
- Si la ressource peut être accédée l'adresse sémaphore contient une autre valeur.
- Une tâche doit donc tester la valeur contenue à l'adresse sémaphore et réserver l'accès à la ressource si elle est disponible.
- Le test et la modification de la valeur doit s'effectuer de manière **atomique** c'est-à-dire sans qu'aucune tâche puisse accéder la valeur pendant la lecture et l'écriture de la valeur (situation idem à celle discutée avec les interruptions)



# exemple

**1<sup>er</sup> processus**

**lecture** de la valeur  
valeur = 0x0 ok ressource  
accessible

**2<sup>ème</sup> processus**

**lecture** de la valeur  
valeur = 0x0 ok ressource  
accessible  
**écriture** 0x1 à l'adresse  
sémaphore pour réserver  
la ressource (**trop tard**)

commutation de tâche

temps

# exemple - sémaphore

Le code correct pour implémenter un sémaphore

spin

MOV	r1, sémaphore	r1 contient l'adresse
MOV	r2, #1	valeur à tester
SWP	r3, r2, [r1]	place #1 à l'adresse + r3
CMP	r3, #1	est-ce que la ressource est ok
BEQ	spin	

# Instruction d'Interruption logicielle

Les interruptions logicielles sont un mécanisme utilisé par exemple par les applications pour faire des appels aux routines du système d'exploitation.

syntaxe: SWI{<cond>} SWI\_number

voir les modes, superviseur  
adresse de la table

SWI	interruption logicielle	<p>lr_svc = adresse de l'instruction qui suit le SWI</p> <p>spsr_svc = cpsr</p> <p>pc = vectors + 0x8</p> <p>cpsr mode = SVC (change de mode)</p> <p>cpsr I = 1 (masque les interruptions)</p>
-----	-------------------------	--

# exemple

cpsr = n z c V q i f t\_user

pc = 0x00008000

lr = 0x003fffff; lr = r14

r0 = 0x12

SWI 0x123456

cpsr = n z c V q **l** f t\_svc

spsr = **n z c V q i f t\_user**

pc = **0x00000008**

lr = **0x00008004** pas de sauvegarde nécessaire  
r0 = **0x12** paramètre, le registre r0 n'est pas  
modifié

# exemple - suite

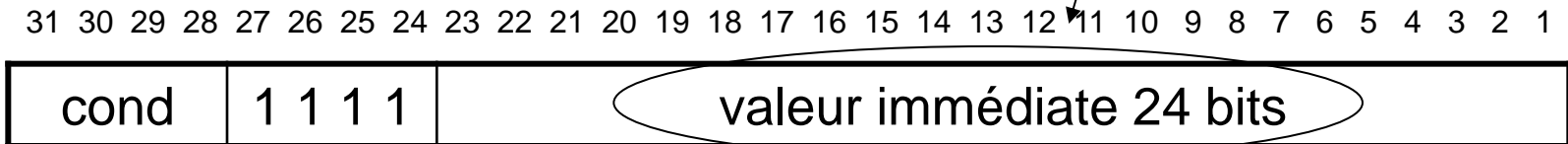
les registres peuvent être utilisés pour passer des paramètres à la routine d'interruption

pour retrouver le SWI\_number il faut exécuter

<SWI-instruction> **AND NOT** 0Xff000000

l'adresse de l'instruction SWI est connue grâce au registre lr

code de l'instruction SWI



# gestionnaire d'interruption

SWI\_handler

; sauvegarde des registres r0-r12 et lr

**STMFD** sp!, {r0-r12, lr}

; lecture de l'instruction SWI

**LDR** r10, [lr, #-4]

; on masque les 8 bits de poids fort

**BIC** r10, r10, #0xff000000 bit clear

; r10 contient le SWI\_number

**BL** routine

; retour

**LDMFD** sp!, {r0-r12, pc}^

indique que le registre  
cpsr doit être mis-à-jour  
avec le contenu de  
spsr\_svc

# Remarque sur l'exécution

Lorsque le processeur exécute l'instruction

SWI SWI\_number


L'exception correspondante (SWI) est levée et le processeur change la valeur du pc par 0x8, la valeur correspondante dans la table d'interruption. Il va donc exécuter cette instruction qui est dans notre cas une instruction

B SWI\_handler.

Cette instruction ne modifie pas le *link register* (lr) et donc lr pointe sur l'instruction qui suit l'instruction SWI.

# instructions et cpsr

Le jeu d'instruction ARM met à disposition deux instructions pour contrôler le registre cpsr. Il s'agit d'instructions qui permettent de sauver et restaurer le contenu du registre d'état. En mode utilisateur les bits peuvent être lus et seulement le champ flag peut être modifié.

Syntaxe: MRS{<cond>} Rd, <cpsr | spsr>  = psr  
MSR{<cond>} <cpsr | spsr>\_<fields>, Rm  
MSR{<cond>} <cpsr | spsr>\_fields, #immediate

MRS	sauvegarde d'un psr dans un registre	Rd = psr
MSR	copie le contenu (partiel)d'un registre vers un psr	psr[field] = Rn
MSR	copie une valeur immédiate (partiellement) vers un psr	psr[field] <sub>152</sub> = immédiate



# exemple - cpsr

cpsr = n z c v q I F t\_SVC

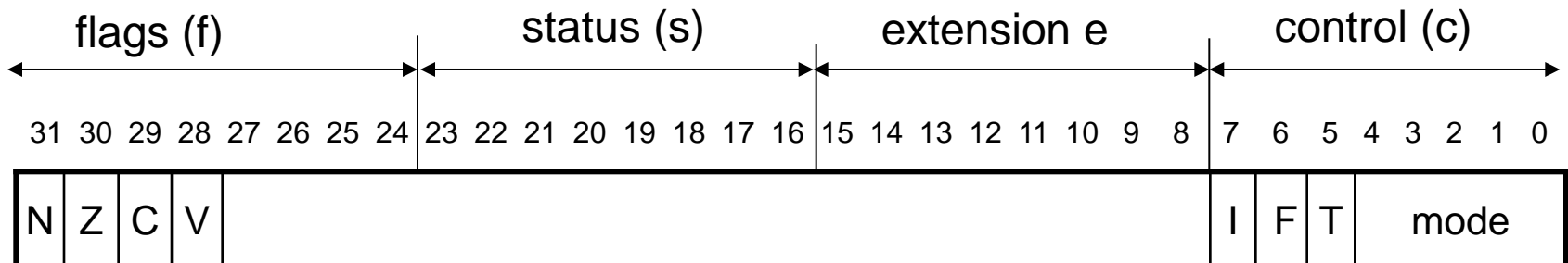
MRS r1, cpsr

BIC r1, r1, #0x80 0b01000000

MSR cpsr\_c, r1

cpsr = n z c v q i F t\_SVC

les différents champs sont



# instructions coprocesseur

les instructions coprocesseur étendent le jeu d'instruction du processeur. Le coprocesseur peut fournir des ressources de calculs supplémentaires ou contrôler la mémoire du système (mémoire cache et gestionnaire de mémoire).

Ces instructions dépendent des coprocesseur utilisés, elles ne font pas parties du processeur ARM.

# LDR Rd, #imm\_32

Les instructions ARM sont codées sur 32 bits, il est donc impossible de coder une valeur immédiate sur 32 bits à charger dans un registre. L'assembleur ARM propose néanmoins des pseudo-instructions.

Syntaxe: LDR Rd, =constant valeur immédiate  
ADR Rd, label

LDR	charge une valeur immédiate sur 32 bits	Rd = constante sur 32 bits
ADR	charge l'adresse effective d'un label	Rd = l'adresse sur 32 bits 155

# pseudo-instructions

Le compilateur effectue la traduction des pseudo-instructions en instructions ARM

LDR r0, =0xff devient MOV r0, #0xff valeur immédiate sur 8 bits

LDR r0, =0x55555555 devient LDR r0, [pc, #offset\_12]



valeur relative de l'adresse ou se trouve  
la valeur immédiate à charger dans r0

# instructions conditionnelles

L'intérêt des instructions conditionnelles est de réduire les instructions de rupture de séquence qui pénalisent le bon fonctionnement du pipeline.

calcul du plus grand commun diviseur

```
while(a!=b)
{
    if (a>b) a-=b; else b-=a;
}
```

# instructions conditionnelles

ce qui peut s'écrire en assembleur (r1 est a et r2 est b)

gcd

```
CMP    r1, r2
BEQ    complete
BLT    lessthan
SUB    r1, r1, r2
B      gcd
```

lessthan

```
SUB    r2, r2, r1
B      gcd
```

complete

.....

# instructions conditionnelles

ou encore

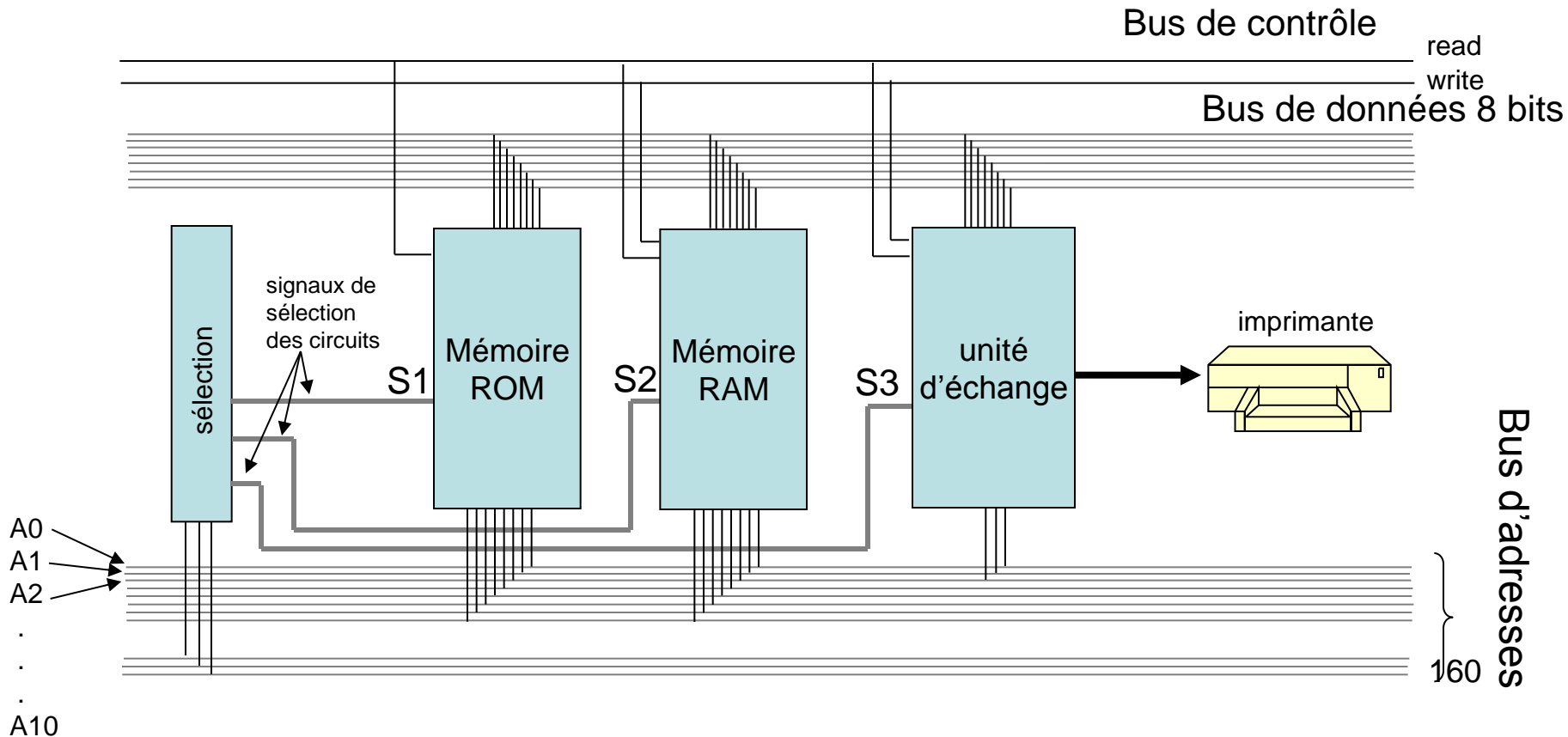
gcd

CMP	r1, r2
SUBGT	r1, r1, r2
SUBLT	r2, r2, r1
BNE	gcd

# Périphérique

## Les unités d'échanges

Par *Unités d'échanges*, on désigne les périphériques d'entrée/sortie qui permettent donc d'échanger de l'information entre les composants internes du système informatique et l'extérieur.





# Espace d'adressage

Sur le schéma précédent:

le **bus de donnée** est partagé par tous les éléments extérieurs (mémoire et périphérique) pour permettre le transfert de données depuis le processeur vers tous les circuits.

Le **bus d'adresse** est sur 11 bits, A0, A1, ..., A10.

Pour les circuits mémoires on utilise les 8 bits A0, ..., A7, les mémoires ont donc une capacité de  $2^8$  mots de 8 bits (byte).

L'**unité d'échange** utilise uniquement les bits A0, A1 et A2 on peut donc accéder au plus 8 byte (registres)

# Espace d'adressage

On utilise trois signaux S1, S2, S3 qui sont générés par l'unité de sélection pour désigner le circuit parmi les trois qui doit prendre en compte le transfert. C'est-à-dire le circuit à qui est destiné les données ou qui doit fournir les données.

La sélection des circuits se fait en utilisant les lignes d'adresses A8, A9 et A10. On décide par exemple d'activer

- S1, soit la mémoire ROM si  $A8=0$ ,  $A9=0$ ,  $A10=0$
- S2, soit la mémoire RAM si  $A8=1$ ,  $A9=0$ ,  $A10=0$
- S3, soit l'unité d'échange si  $A8=0$ ,  $A9=1$ ,  $A10=0$

Tout ça pour obtenir que depuis le processeur on accède

- *La mémoire ROM aux adresses de 0x000 à 0x0FF*
- *La mémoire RAM aux adresses de 0x100 à 0x1FF*
- *L'unité d'échange aux adresses de 0x200 à 0x207*

# Espace d'adressage

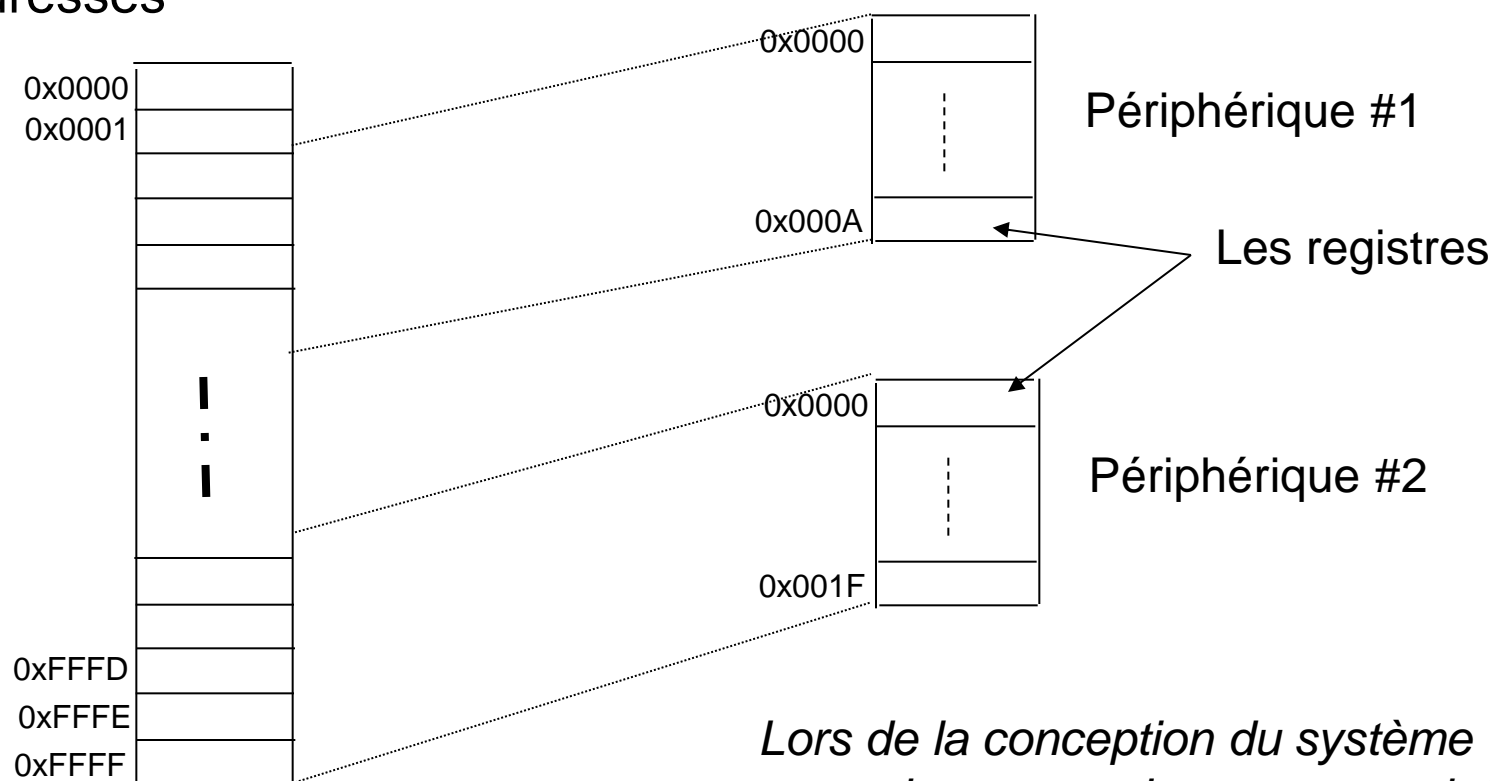
L'opération qui consiste à activer les signaux de sélection des circuits s'appelle **le décodage des adresses**.

Le décodage peut être **exhaustif** si une et une seule adresse permet de sélectionner un byte particulier. Sinon le décodage est non-exhaustif.

Dans notre exemple le décodage est-il exhaustif ou non-exhaustif?

# Espace d'adressage

On considère par exemple un processeur qui dispose de 16 lignes d'adresses



Adressable par le processeur

*Lors de la conception du système  
on assigne une adresse aux registres  
des périphériques dans l'espace adressé  
par le processeur*

# Exemple - GBA

## Memory Map

### General Internal Memory

0000:0000-0000:3FFF BIOS - System ROM (16 KBytes)

0000:4000-01FF:FFFF Not used

0200:0000-0203:FFFF WRAM - On-board Work RAM (256 KBytes) 2 Wait

0204:0000-02FF:FFFF Not used

0300:0000-0300:7FFF WRAM - In-chip Work RAM (32 KBytes)

0300:8000-03FF:FFFF Not used

0400:0000-0400:03FE I/O Registers

0400:0400-04FF:FFFF Not used

### Internal Display Memory

0500:0000-0500:03FF BG/OBJ Palette RAM (1 Kbyte)

0500:0400-05FF:FFFF Not used

0600:0000-0617:FFFF VRAM - Video RAM (96 KBytes)

0618:0000-06FF:FFFF Not used

0700:0000-0700:03FF OAM - OBJ Attributes (1 Kbyte)

0700:0400-07FF:FFFF Not used

# Example

## **External Memory (Game Pak)**

0800:0000-09FF:FFFF Game Pak ROM/FlashROM (max 32MB) - Wait State 0  
0A00:0000-0BFF:FFFF Game Pak ROM/FlashROM (max 32MB) - Wait State 1  
0C00:0000-0DFF:FFFF Game Pak ROM/FlashROM (max 32MB) - Wait State 2  
0E00:0000-0E00:FFFF Game Pak SRAM (max 64 KBytes) - 8bit Bus width  
0E01:0000-0FFF:FFFF Not used

## **Unused Memory Area**

1000:0000-FFFF:FFFF Not used (upper 4bits of address bus unused)

# Cache

Vitesse d'un processeur ~ 1GHz, temps de cycle 1 nanosecondes  
(=10<sup>-9</sup> secondes)

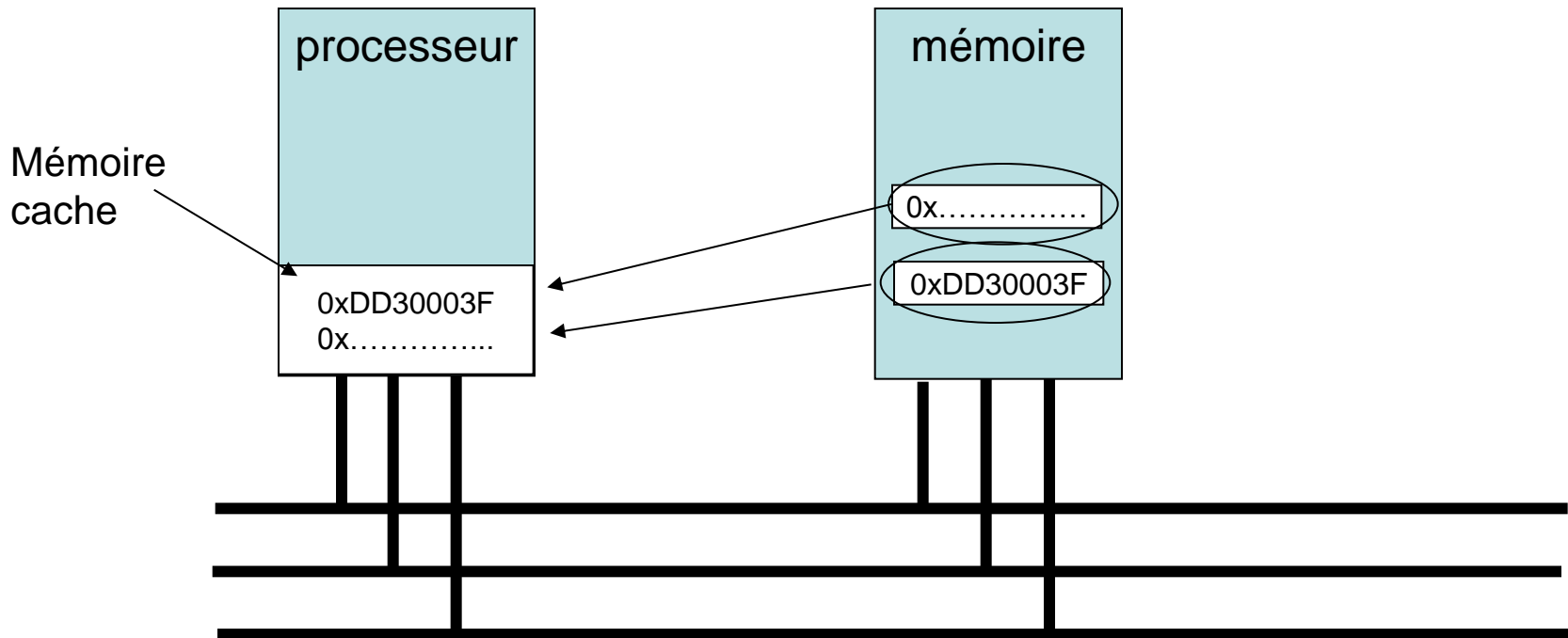
Accès mémoire SRAM 20 nanosecondes

Les accès mémoires sont très couteux en temps

On observe:

- Les performances des mémoires évoluent moins vite que celle des processeurs
- Augmenter la taille du bus pour transférer plus de bits par cycle bus. Cela fonctionne si les accès sont à des positions mémoires contigües (plus de complexité)
- Réduire les accès en mémoire externe en utilisant de la mémoire cache (antémémoire). Ces mémoires sont très rapides mais la capacité de stockage est limitée.

# Cache



Les informations (données ou programme) mémorisé en mémoire cache sont accédées beaucoup plus rapidement. L'utilisation de mémoire cache est plus complexe, par exemple on doit s'assurer de la cohérence des informations entre mémoire cache et mémoire centrale



# Cache

## principe de fonctionnement

Le principe de fonctionnement est le suivant.

Si une position mémoire est souvent accédée, l'information n'est plus sauvegardée en mémoire externe mais en mémoire cache.

L'information va donc être accédée beaucoup plus rapidement (pas de cycle bus)

1. Est-ce que le mot se trouve en mémoire cache, si oui la donnée est transférée au processeur (cache hit)
2. Sinon (cache miss), on effectue un cycle bus (lent) pour chercher l'information en mémoire externe (centrale) et on place l'information en mémoire cache pour un accès ultérieur. Si le cache est plein il faut choisir
  1. si la donnée doit en remplacer une autre
  2. La donnée est rarement accédée et n'est pas placée en cache

# Cache

## principe de fonctionnement

Les performances de l'utilisation de la mémoire cache dépendent des chance de succès (cache hit) et donc de l'algorithme de remplacement des données en mémoire cache. Pour comprendre comment s'effectuent les accès en mémoire on effectue un grand nombre d'étude statistique sur des programmes choisis.

On observe:

1. Lorsqu'une instruction référence une adresse il est très probable que les prochains accès soit dans un voisinage de cette adresse. Par exemple, accès à un tableau, les instructions sont disposées séquentiellement en mémoire. ***C'est le principe de localité spatiale.***
2. Lorsque le processeur référence un mot donné, il est très probable que ce mot soit référencé dans les instants qui suivent. ***C'est le principe de localité temporelle***

Si les accès mémoire étaient aléatoire, les mémoires cache ne seraient pas efficaces

# Cache - cohérence

Lorsque l'on utilise de la mémoire cache, on utilise deux positions mémoires différentes pour stocker la même information.

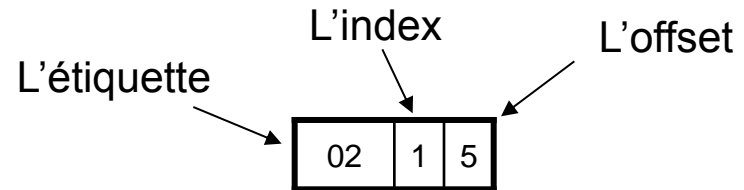
Lorsque l'on modifie la valeur d'une donnée qui se trouve en cache on a deux options:

1. On modifie la donnée en mémoire cache **et** en mémoire externe simultanément. C'est la stratégie **écriture immédiate (*write through*)**. L'information qui se trouve en mémoire cache et externe est donc **toujours cohérente** (c'est la même).
2. On modifie la donnée en mémoire externe quand c'est le moins pénalisant pour le système, c'est la stratégie **d'écriture différée (*write back*)**. On a plusieurs options. Par exemple, on écrit en mémoire externe lorsque le bus est inutilisé ou seulement lorsque l'information en mémoire cache doit être réutilisée. La cohérence n'est pas garantie en permanence, mais les temps d'écriture sont plus faibles.

# Exemple Cache direct

On suppose qu'on dispose d'une mémoire centrale d'une capacité de 10'000 mots, les adresses évoluent donc de 0000 à 9999. On décompose l'adresse en *l'étiquette*, *l'index* et *l'offset*. Par exemple, l'adresse 0215 est décomposé en :

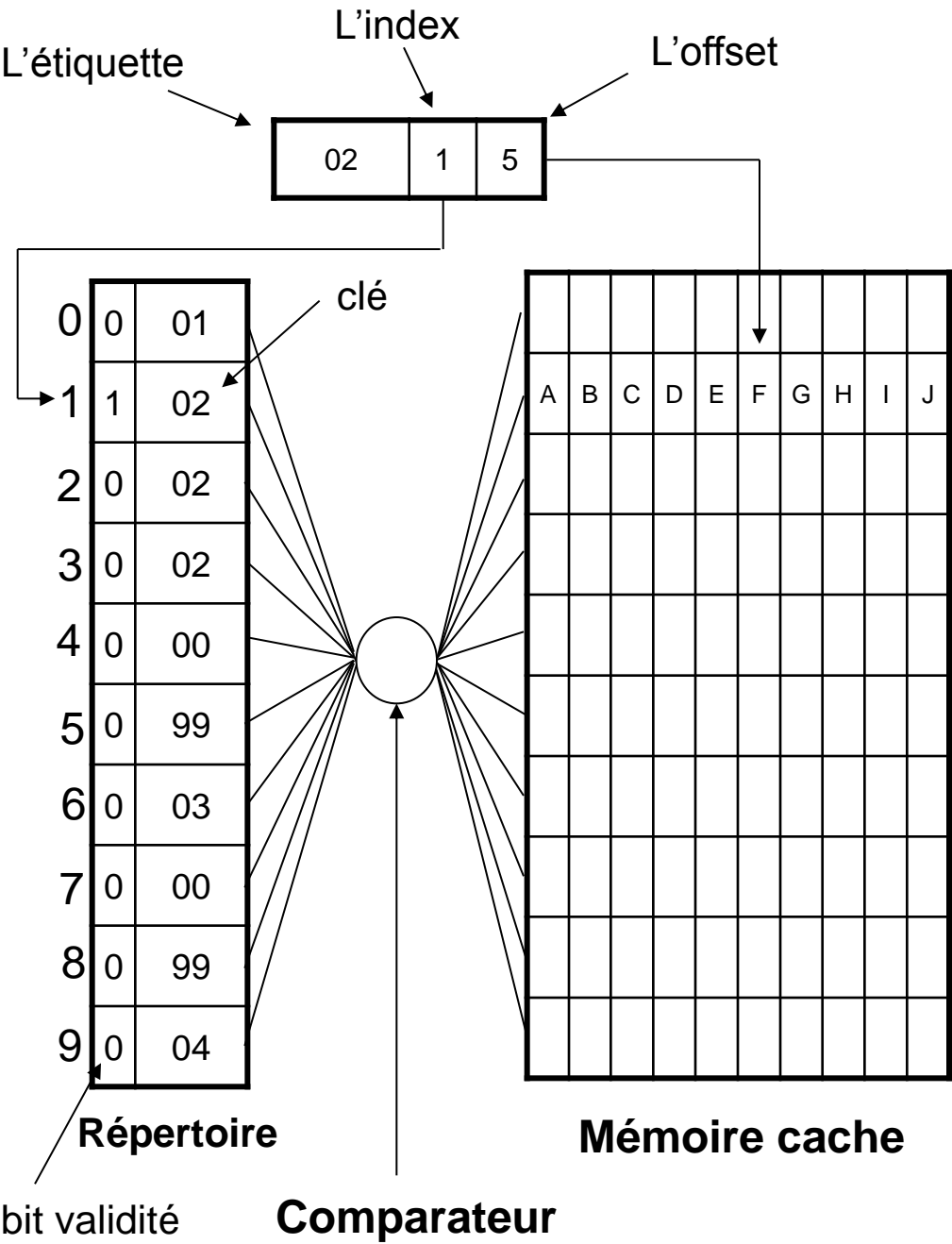
- 02 = l'étiquette
- 1 = l'index
- 5 = l'offset



On dispose d'un répertoire qui associe à chaque valeur possible de l'index (0,1,2,...,9) une clé permettant d'identifier une ligne (étiquette) et un bit qui indique si les données sont valides.

La mémoire cache est organisé par ligne de 10 mots (ici 10 lignes)

On dispose d'un comparateur qui vérifie si la valeur de l'étiquette est égales à la clé.



### Mémoire centrale

00	0	0																	
00	1	0																	
00	2	0																	
00	3	0																	
02	0	0																	
02	1	0	A	B	C	D	E	F	G	H	I	J							
02	2	0																	
02	3	0																	
99	9	0																	

adresses      données

# Cache direct pseudo-code

```
si repertoire[index] = étiquette & bit de validité alors  
    charger le processeur avec mémoire_cache[index,offset]  
sinon  
    sauvegarder mémoire_cache[index] dans la mémoire centrale  
    repertoire[index]=étiquette  
    charger mémoire_cache[index] depuis la mémoire centrale  
    charger le processeur avec mémoire_cache[index,offset]  
finsi
```

# Cache direct exemple

En supposant que la mémoire cache est dans l'état indiqué sur le précédent transparent, on désire accéder successivement les adresses:

- 0215 succès (cache hit) la donnée est transférée depuis la mémoire cache
- 0212 re-succès
- 0114 échec (cache miss) on modifie l'entrée de la mémoire cache qui correspond à l'index 1
- 0225 échec, à cause du bit de validité (=0), chargement de la ligne
- 0116 succès

On observe que toutes les adresses du type 011x, 021x, 031x, etc donnent lieu à des collisions car elles utilisent le même index.

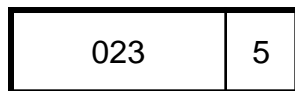
# Cache purement associative

Ces mémoires caches utilisent autant de comparateurs que de lignes de cache, ce qui permet de supprimer l'index dans le répertoire.



L'étiquette

L'offset



0	001
1	003
2	022
3	
4	023
5	
6	
7	000
8	998
9	

A	B	C	D	E	F	G	H	I	J

Répertoire

Mémoire cache

Comparateurs

validité

Mémoire centrale

0000									
0010									
0020									
0030									
0200									
0210									
0220									
0230	A	B	C	D	E	F	G	H	I
9990									

adresses

données

# Mémoire cache associative

## pseudo-code

```
si répertoire contient l'étiquette & bit de validité alors  
    transférer la donnée vers le processeur depuis la mémoire  
    cache  
sinon  
    si répertoire est plein alors  
        appliquer un algorithme pour choisir une ligne  
        modifier la valeur de la ligne choisie en mémoire cache  
        avec les données qui se trouvent en mémoire centrale  
        transférer les données vers le processeur  
    sinon  
        choisir une ligne vide et transférer les données  
        correspondantes depuis la mémoire centrale  
        transférer les données vers le processeur  
    finsi  
finsi
```

# Mémoire cache associative

- Ce type de mémoire est plus couteux que la mémoire cache directe, principalement à cause des comparateurs
- Ce type de mémoire est plus flexible à l'usage (plus d'index)
- L'algorithme de choix d'une ligne a remplacer est critique pour les performances de fonctionnement du cache. Un algorithme classique est LRU qui consiste à remplacer la ligne qui à été le moins récemment utilisée

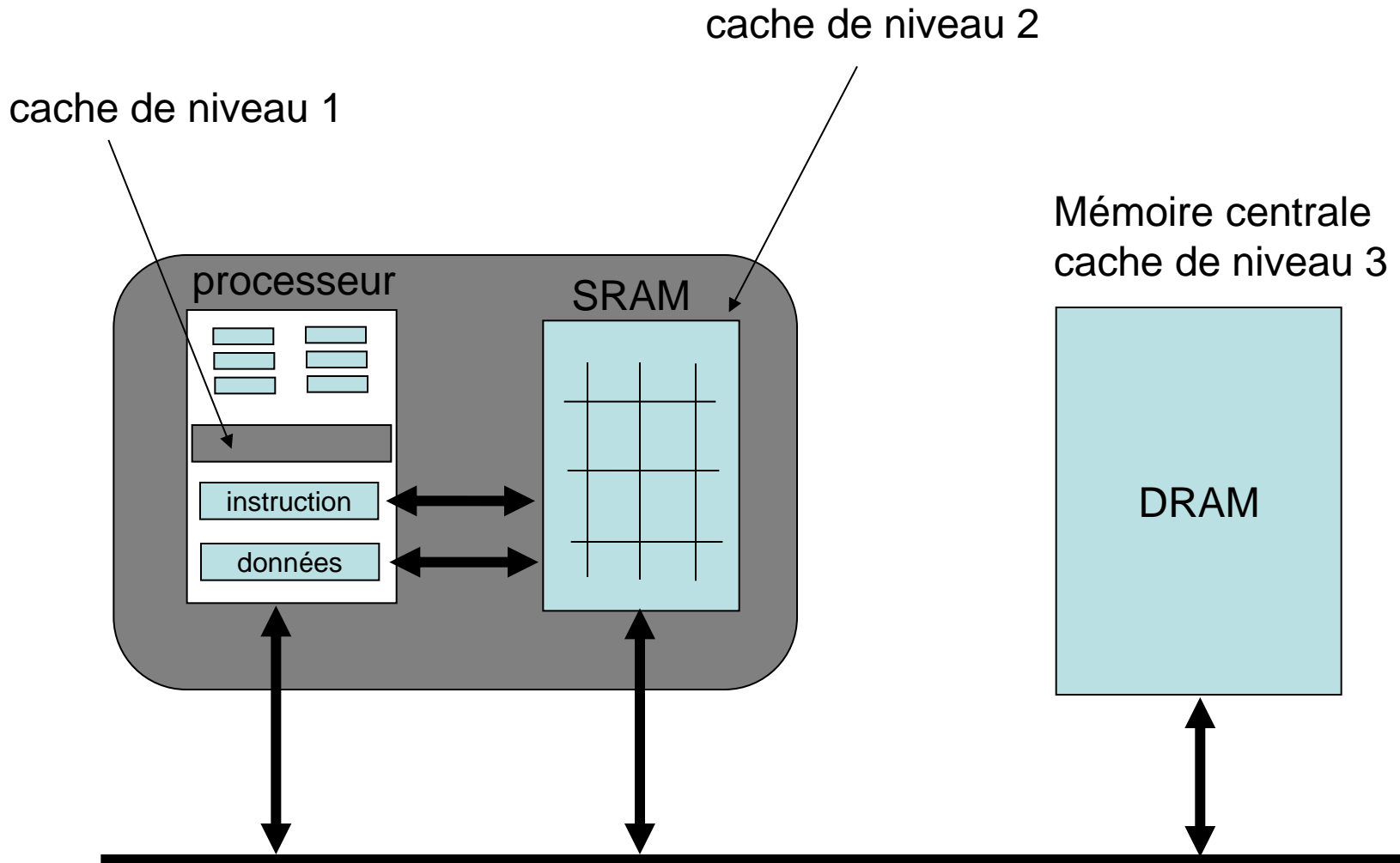
# Localisation des caches

On distingue trois niveau de mémoire cache

1. Un cache de petite capacité, très rapide intégré au processeur
2. Un cache de plus grande capacité, situé à l'extérieur du processeur et relié par un bus dédié
3. La mémoire centrale

vitesse ↑

↓ capacité



# Conclusion – mémoire cache

Les mémoires caches permettent d'améliorer les performances d'un système informatique en réduisant les temps d'accès (lecture/écriture) en mémoire centrale

La question de la cohérence entre l'information contenue dans la cache et la mémoire centrale est critique et peut engendrer un mauvais fonctionnement du système. Par exemple:

- Si on accède une position mémoire qui correspond à un registre d'un périphérique (pour commander ou configurer) alors le cache prévient le processeur de tout accès au périphérique....
- Dans un environnement multiprocesseur, certaines variables sont partagées entre les processeurs et la cohérence doit toujours être garantie.
- Mis-à-part les processeurs, certains périphériques nécessitent d'assurer la cohérence en permanence.

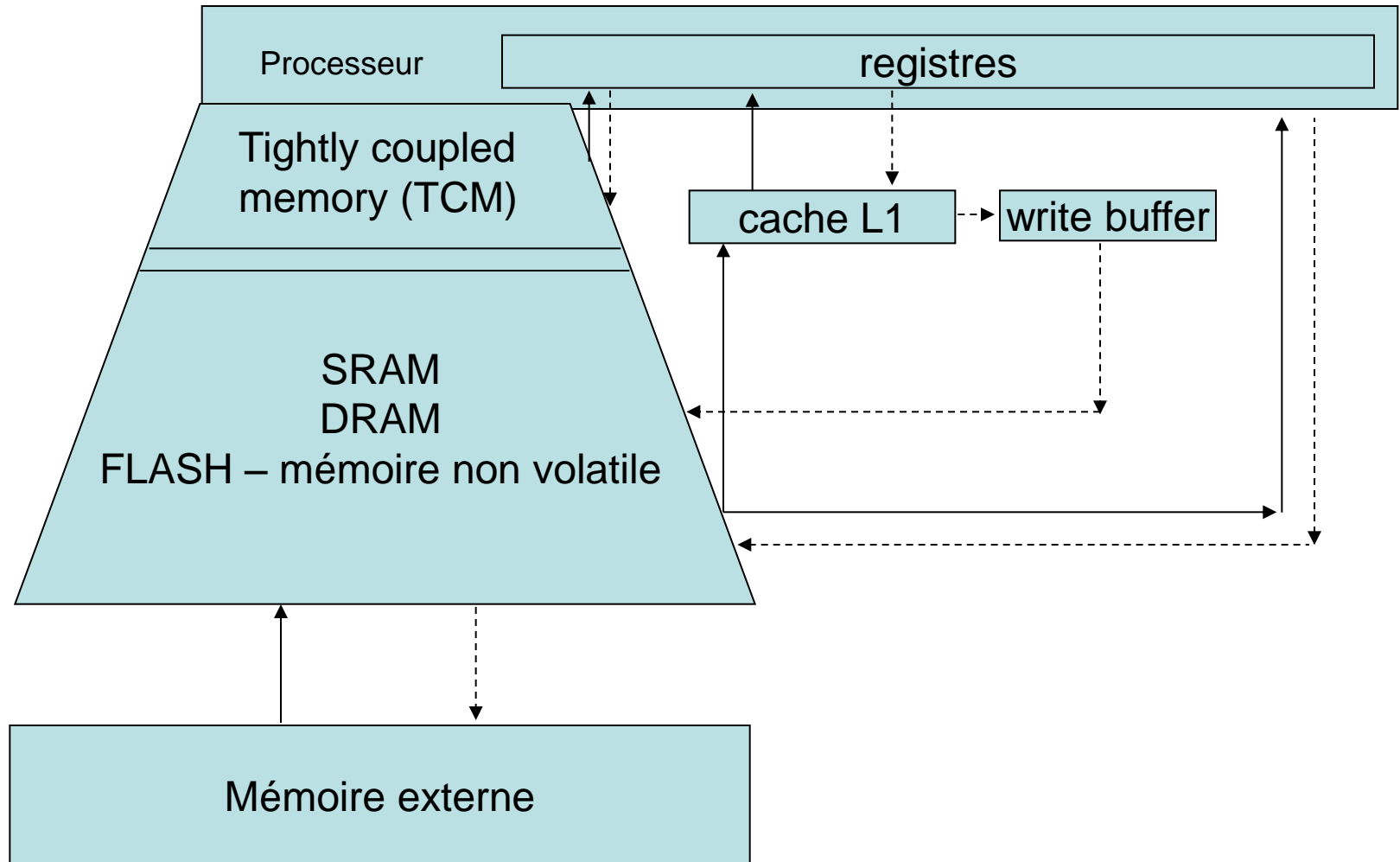
# Retour sur les mémoires caches

Les systèmes ARM propose des mémoires caches. On se propose de passer en revue les différents systèmes. On détaillera aussi certains mécanismes suggérés au début du cours.

La fonction de la mémoire cache est de conserver des données qui se trouvent en mémoire centrale et permettre des accès beaucoup plus rapides.

Pour accélérer les écritures en mémoire centrale le système dispose d'un *write buffer*. Il s'agit d'une mémoire tampon de type FIFO, placée entre le processeur et la mémoire centrale. Les écritures en mémoire centrale sont effectuées par le buffer pour libérer, autant que possible, le processeur.

# Hiérarchie





# Hiérarchie

La mémoire TCM se trouve sur le chip.

Le cache représenté est un cache de niveau 1 (L1). certains systèmes ont plusieurs niveaux de cache.

Un cache de niveau 2, L2, se trouve entre le cache L1 et une mémoire. Pour la gestion de plusieurs caches on doit se préoccuper des problèmes de cohérence entre les différents caches.

De manière générale, le processeur cherche une donnée (instruction) dans le cache de niveau 1 (L1) puis, si échec, dans le cache de niveau suivant. On parle de cache à plusieurs niveaux.

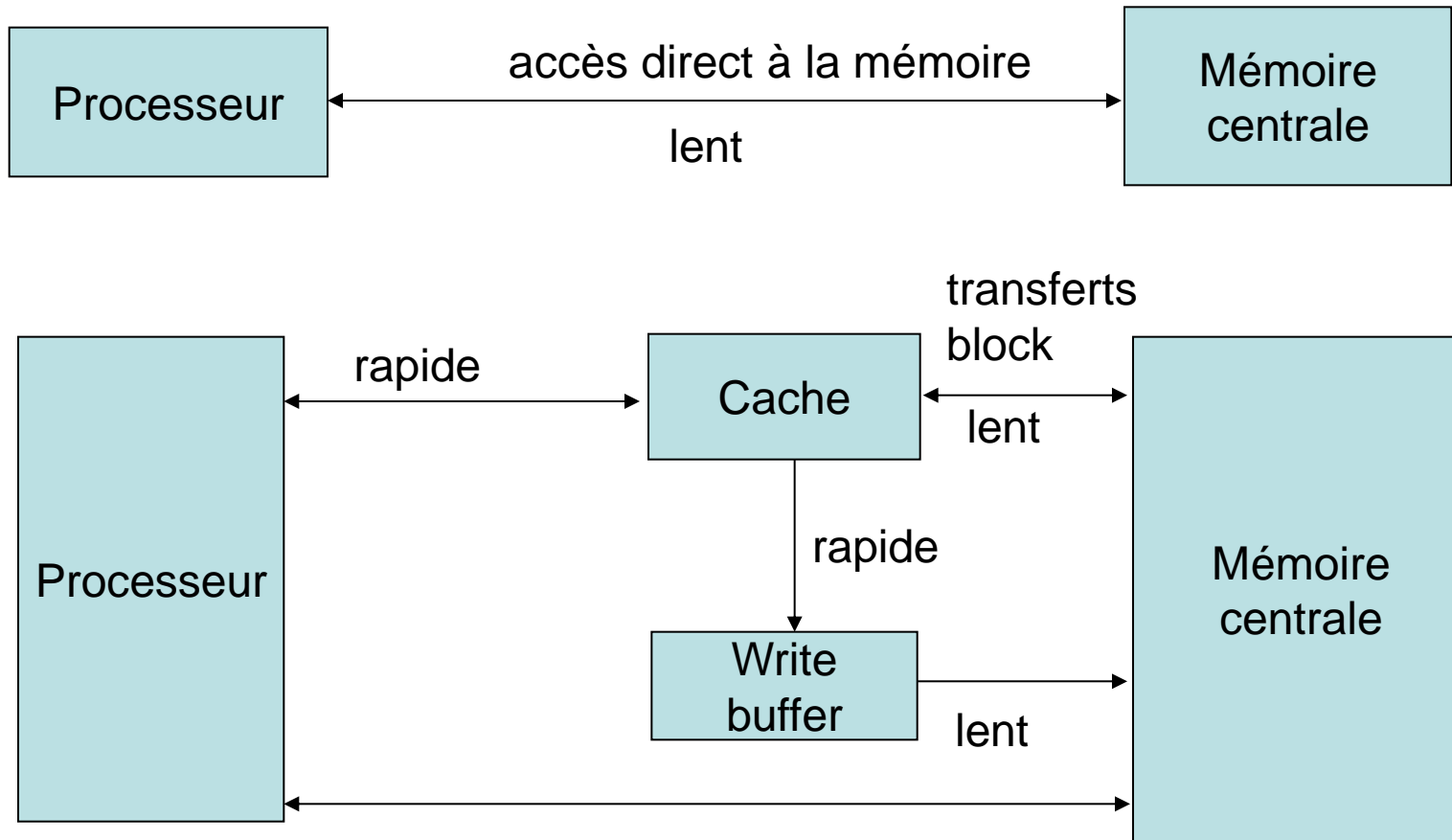
Une mémoire cache peut-être efficace entre deux niveaux de mémoire dont les temps d'accès sont très différents.

# Transferts

Les transferts de données entre le cache (L1) et la mémoire centrale se fait par blocks, appelés aussi *cache lines*.

Les écritures de la mémoire cache s'effectuent *rapidement* à travers le *Write buffer*. Les écritures de ce tampon vers la mémoire centrale se font à la vitesse de la mémoire (lentement).

# Transferts



# Caches virtuels

Certains systèmes disposent d'un circuit MMU, Memory Management Unit, qui s'occupe de traduire les adresses (virtuelles) manipulées par le processeur en adresse physique. La mémoire cache peut-être placée soit

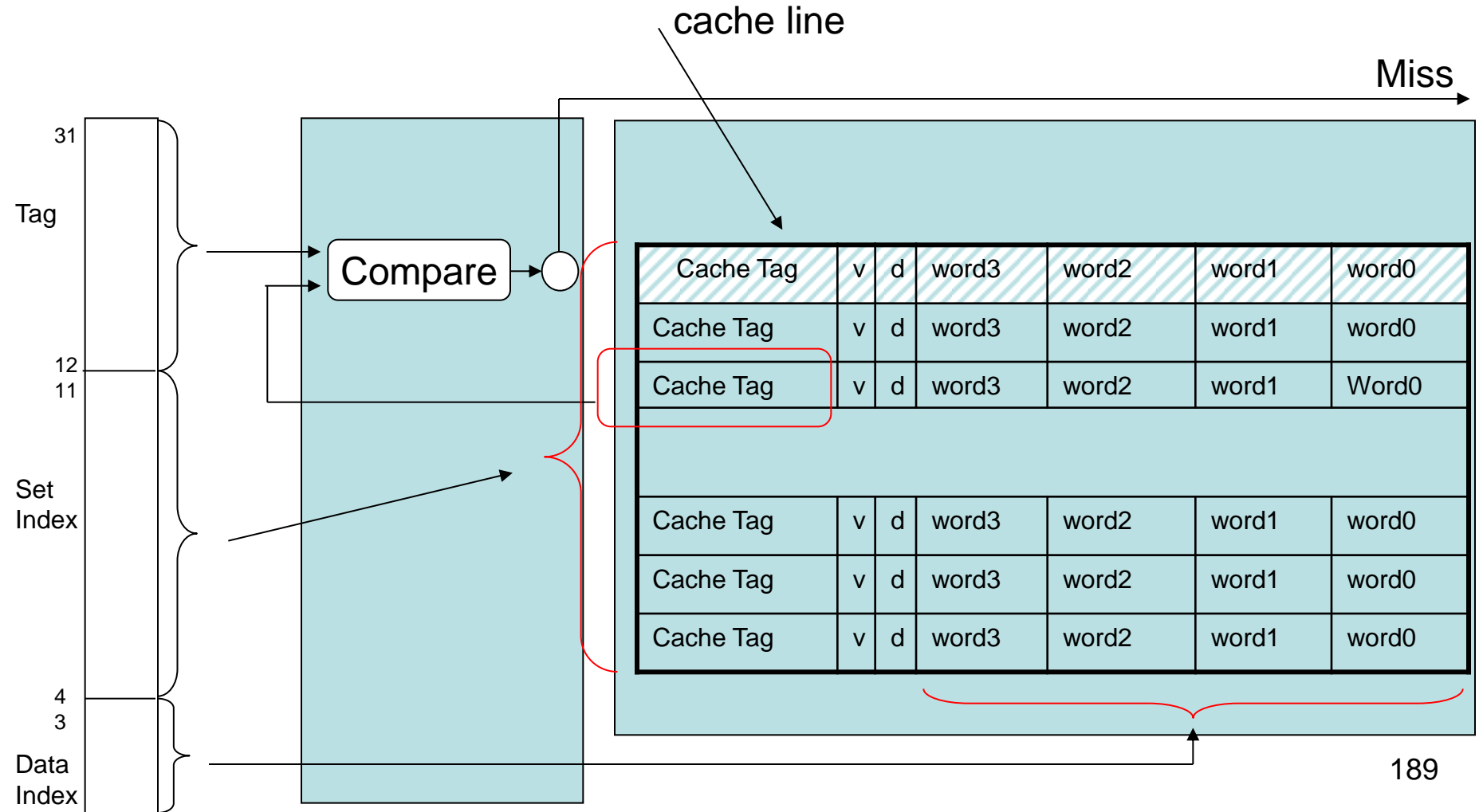
- Entre le processeur et le circuit MMU, on parle de cache virtuel ou logique.
- Entre le circuit MMU et la mémoire central, on parle de cache physique.

Les cœurs ARM7- ARM10 utilisent des caches logiques.  
(comme les processeurs Intel XScale)

La famille ARM11 utilise un cache physique.

# Architecture de base

L'architecture de cache de base proposée pour les systèmes ARM est un cache à accès direct.



# Architecture de base

Comme précédemment, le Set Index (bits 4-11) pointe sur une entrée de la mémoire cache et le Tag (bits 12-31) est comparé avec l'entrée de la cache.

Le bit de validité *v* indique si la ligne de donnée (cache line) est valide.

Le bit *d* (dirty) indique que la donnée a été modifiée par le processeur (écriture). Ce bit indique que l'entrée correspondante de la mémoire centrale doit être mise-à-jour (si writeback).

# Fonctionnement

On a vu que les transferts entre la mémoire centrale et la mémoire cache étaient réalisés par blocks, (4 mots de 32 bits).

La mémoire cache implémente une technique appelée *Data Streaming* lorsqu'une nouvelle ligne du cache est chargée. La mémoire cache est capable de simultanément fournir la donnée au processeur et charger la ligne du cache.

Le *Data Streaming* permet de ne pas pénaliser un accès en mémoire centrale après un *cache miss*.

# Fonctionnement

Lorsqu'une ligne de la mémoire cache est valide et qu'on doit la remplacer, les mémoires cache à accès direct peuvent être inefficace.

Par exemple on considère l'exécution de la boucle

```
do {  
    routineA();  
    routineB();  
    x--;  
} while (x>0)
```



# Fonctionnement

Avec les routines routineA() et routineB() qui se trouvent aux adresses

0x00000480 et 0x00001480 respectivement

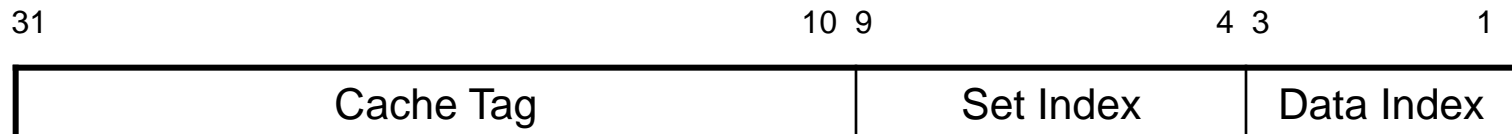
et les données accédées en 0x00002480

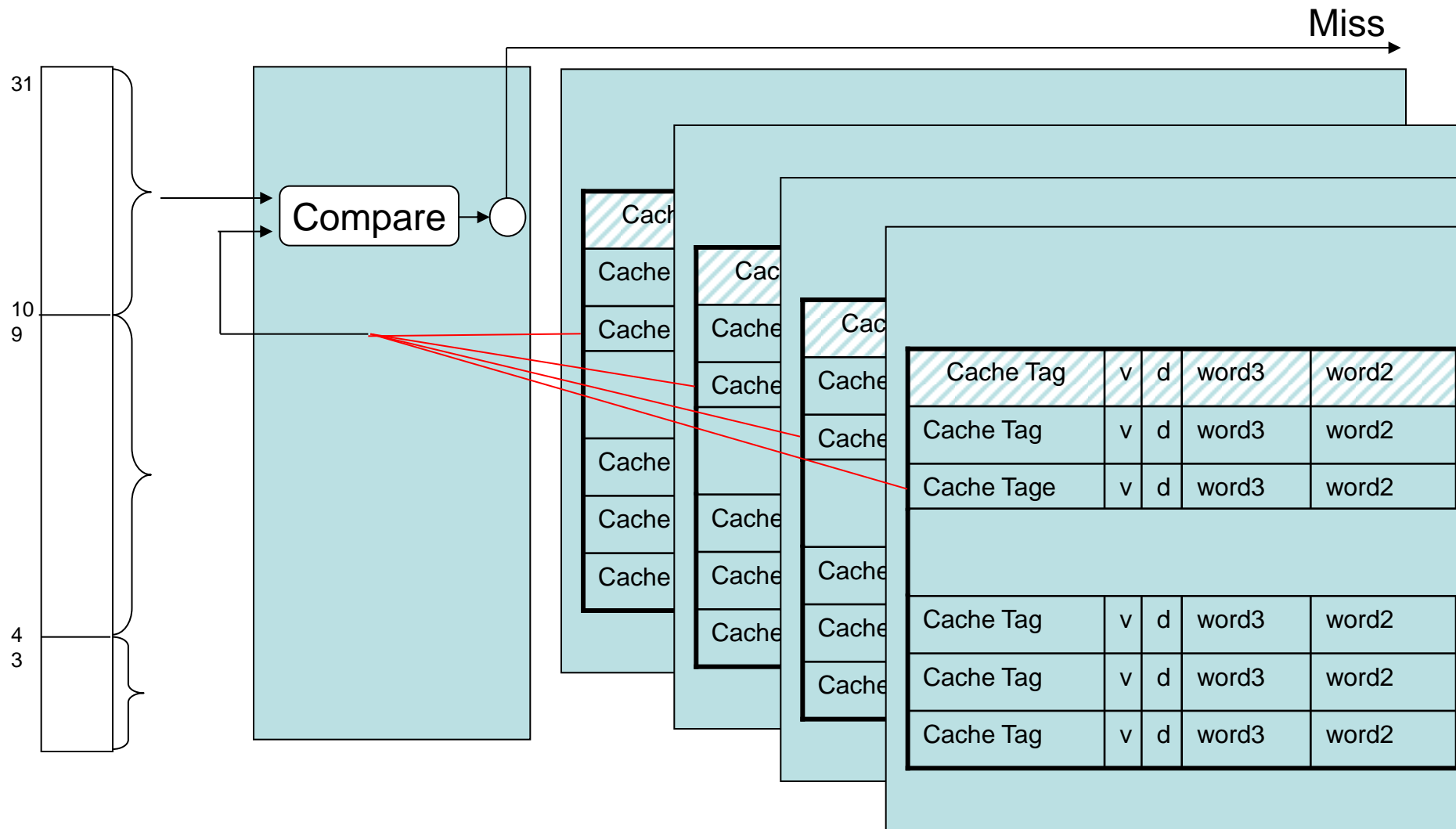
Dans cette situation, l'exécution des routines ne se fera jamais depuis la mémoire cache

# Mémoire associative

Pour palier à ce problème, on utilise des mémoires caches à accès indirect qui stocke les entrées Cache Tag dans une mémoire associative.

Une solution intermédiaire proposée pour les systèmes ARM, est de mettre à disposition quatre structures de mémoires cache pour chaque valeur de l'indice (Set Index). On divise le nombre d'entrée de la mémoire cache par 4, mais on contourne le problème mentionné précédemment. Cette solution limite la complexité du circuit par rapport à une mémoire cache associative.

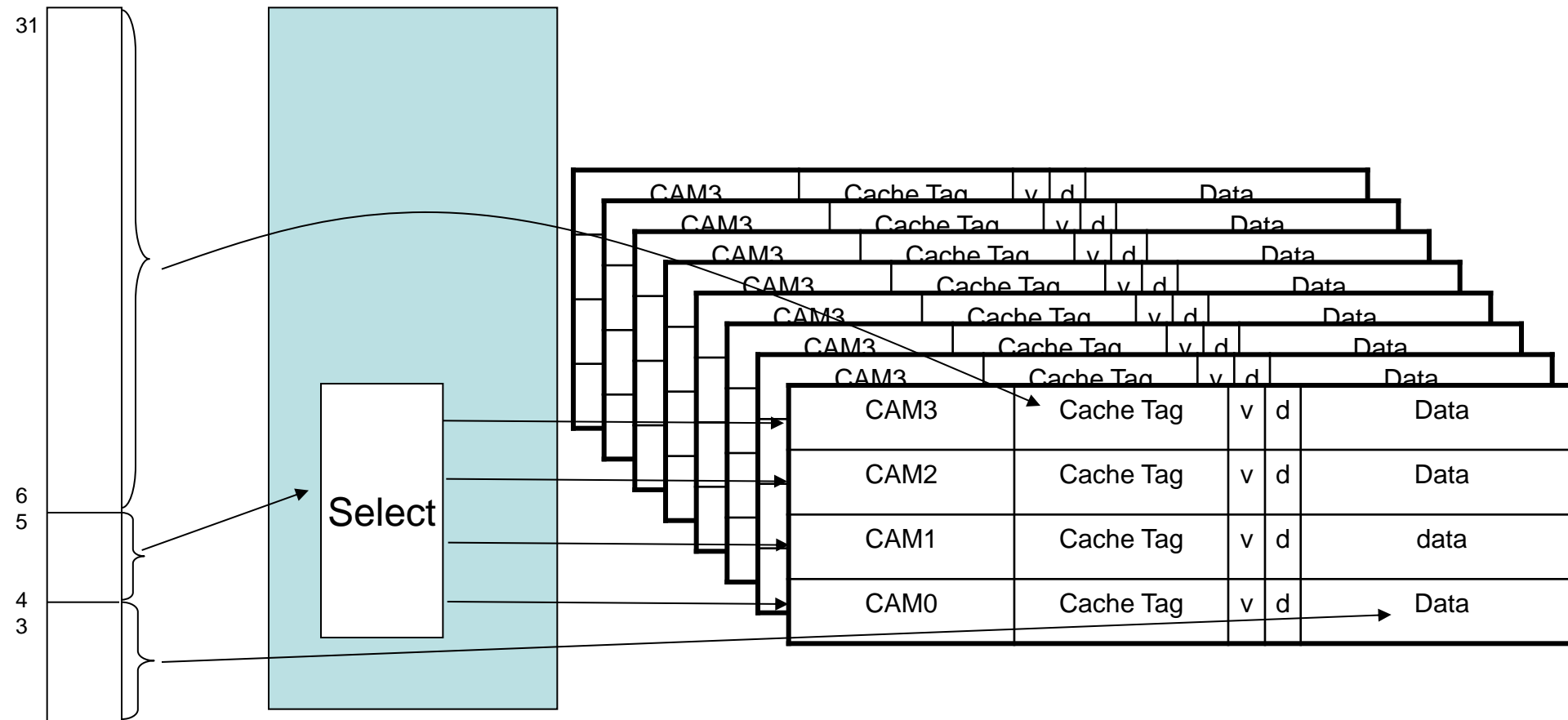




# ARM940T - cache

Ce processeur dispose d'une mémoire cache qui met en parallèle 64 structure de mémoire cache. Chaque structure de mémoire cache (way) dispose de 4 entrées, l'indice (Set/ Index) est codé sur 2 bits, et utilise une mémoire associative.

# ARM940T - cache



# Stratégie d'écriture

Les stratégies d'écriture des systèmes de mémoire caches ARM sont classiques:

**Writethrough:** Chaque écriture dans la mémoire cache est reproduite en mémoire centrale. La cohérence est assurée en permanence.

**Writeback:** Les écritures dans la mémoire cache ne sont pas nécessairement reproduites en mémoire centrale et l'état des mémoires peut-être incohérents.

Lorsque le processeur écrit dans le cache, il positionne le bit *dirty* de manière à indiquer l'incohérence. Lorsque la ligne sera supprimée du cache, l'écriture se fera.

Ce mécanisme est efficace, par exemple, pour les variables locales à une procédure qui ne doivent pas être sauveées en mémoire centrale.

# Stratégie de remplacement des lignes

Principalement deux stratégies:

**Round-robin:** La mémoire cache gère un pointeur qui est la prochaine ligne à remplacer. Le pointeur est incrémenté d'une unité à chaque remplacement (modulo le nombre de ligne...)

**Pseudorandom:** La ligne à remplacer est choisie aléatoirement.

# Stratégie d'allocation des lignes

Pour l'allocation des lignes, il y a aussi deux stratégies possibles:

1. (*read-allocate*) une ligne du cache est allouée chaque fois qu'une lecture est effectuée pour une donnée qui ne se trouve pas dans le cache (*read-miss*). Avec cette stratégie, une écriture d'une donnée qui ne se trouve pas en mémoire cache se fait simplement en mémoire centrale. Si la donnée se trouve en cache, le fonctionnement dépend de la stratégie writethrough ou writeback.
2. (*write-allocate*) même comportement qu'au dessus pour les lectures. Lorsqu'une écriture se produit, le cache alloue une ligne.



ARM720T	Writethough	Random	Read-miss
ARM740T	Writethough	Random	Read-miss
ARM920T	Writethough, writeback	Random, round-robin	Read-miss
ARM940T	Writethough, writeback	Random	Read-miss
ARM926EJS	Writethough, writeback	Random , round-robin	Read-miss
ARM946E	Writethough, writeback	Random , round-robin	Read-miss
ARM10202E	Writethough, writeback	Random , round-robin	Read-miss
ARM1026EJS	Writethough, writeback	Random , round-robin	Read-miss
Intel StrongARM	Writeback	Round-robin	Read-miss
Intel XScale	Writethough, writeback	Round-robin	Read-miss,write- miss

# Fonctionnement par l'exemple

Le cœur ARM940T propose les deux stratégies de remplacement – aléatoire et round-robin.

La structure du cache est de proposer 64 structures de cache parallèles et d'indexer sur 4 mémoires associatives différentes.

On cherche à écrire un programme qui va rapidement remplir la mémoire cache.

Pour cela, on accède à des adresses mémoires qui correspondent à des entrées du cache différentes:

0x20000, 0x20040, 0x20080, 0x200C0, 0x20100, etc.

Après 64 lectures on a rempli les entrées du cache qui correspondent à l'index (Data Index) 0 et la mémoire associative 0.

# Example

```
int readSet(int times, int numset) {  
    int setcount, value;  
    volatile int *newstart;  
    volatile int *start = (int*)0x20000;  
  
    __asm {  
        timesloop:  
            MOV     newstart, start  
            MOV     setcount, numset  
        setloop:  
            LDR     value, [newstart, #0]  
            ADD     newstart, newstart, #0x40  
            SUBS    setcount, setcount, #1  
            BNE     setloop  
            SUBS    times, times, #1  
            BNE     timesloop  
    }  
    return value;  
}
```

# Résultats

Avec numset = 64 et Round-Robin : 0.51 secondes

Avec numset = 64 et Aléatoire: 0.51 secondes

Avec numset = 65 et Round-Robin: 2.56 secondes

Avec numset = 65 et Aléatoire: 0.58 secondes.

# Programmation efficace en C



# Introduction

La programmation 'bas-niveau' des systèmes se fait en assembleur et souvent en C. Programmer en C nécessite l'utilisation d'un compilateur qui doit tirer parti de la structure particulière du processeur et de son jeu d'instruction pour générer du code aussi adapté que possible, c'est-à-dire:

qui s'exécute le plus rapidement possible

qui est aussi compacte que possible

La structure du code source (en C) a un impact sur les performances du code assembleur généré.

# Allocation des registres

Le compilateur essaye d'allouer un registre par variable locale déclarée (utilisée) dans une fonction C. Si plusieurs variables ne sont pas utilisées simultanément, le compilateur utilise le même registre pour ces variables.

Si le nombre total de variables est supérieur au nombre de registres le compilateur utilise **la pile**. L'accès à ces variables est lent car il nécessite un accès à la mémoire externe.

Lors de l'écriture d'une fonction il faut:

1. **minimiser l'utilisation de la pile**
2. **s'assurer que les variables les plus fréquemment utilisées sont stockées dans des registres.**



# Types de données en C

Comment le compilateur utilise les ressources du processeur pour définir les types de données C en assembleur et quels types sont plus avantageux que les autres.

Les processeur ARM7 possèdent des registres de 32 bits et des instructions sur des données 32 bits. L'architecture est de type load/store, les instructions s'exécutent seulement sur les registres (pas en mémoire)

Néanmoins, le jeu d'instruction dispose d'instructions permettant de manipuler des données de 8, 16, 32 ou 64 bits

# Types de données en C

Type de donnée C	taille
char	8 bits non signé
short	16 bits signé
int	32 bits signé
long	32 bits signe
long long	64 bits signé

# Type de données en C

**ATTENTION:** les compilateurs C définissent habituellement le type *char* comme une donnée *signée* sur 8 bits alors que les compilateur gcc utilise le type *non signé* pour les caractères.

Ceci car les premières version du proc. ARM ne pouvait pas manipuler facilement les données 8 bits signée (portabilité du code....).

Il faut donc éviter d'utiliser un caractère (**char** i) comme variable de compteur de boucle avec une condition du type (i>0).

# Instruction load/store

Pre-ARMv4	LDRB	charge une valeur 8 bits non signée
	STRB	transfert une valeur 8 bits signée ou non
	LDR	32 bits signée ou non
	STR	32 bits signée ou non
ARMv4	LDRSB	8 bits signée
	LDRH	16 bits non signée
	LDRSH	16 bits signée
	STRH	16 bits signée ou non
ARMv5	LDRD	64 bits signée ou non
	STRD	64 bits signée ou non

# Instructions load/store

Les instructions load qui s'exécutent sur des données **signées de 8 ou 16 bits** doivent étendre le signe sur 32 bits, le registre est complètement utilisé pour manipuler la donnée (calcul interne sur 32 bits).

Charger une donnée de type *int* en C (depuis 8, 16 ou 32 bits en mémoire) nécessite aucune instruction supplémentaire l'extension du signe est automatiquement prise en charge.

Lors du transfert de la valeur d'un registre vers une position mémoire sur 8 ou 16 bits les bits de poids faible du registre sont considérés.

# Variables locales

Les instructions ARM portent généralement sur des données 32 bits. Les variables locales doivent de préférence être de type *int* ou *long*. Les types *char* ou *short* doivent être évités. Si on travail en utilisant une arithmétique modulo 256 ( $255+1=0$ ) alors ce sont les bons types a utiliser.

**Exemple:** un programme qui calcule une somme de contrôle (CRC).

# Exemple 1

```
int checksum_v1(int *data)
{
    char i; // hypothèse fausse: un char occupe moins de place
            //car les registres sont 32 bits ainsi que les entrées de la pile
    int sum=0;

    for(i=0; i<64; i++)
    {
        sum += data[i]; // comme c'est un char, il faut calculer modulo 256
    }
    return sum;
}
```

# Exemple 1

code assembleur (i est du type char)

checksum\_v1

```
MOV    r2, r0           ; r2 = data
MOV    r0, #0           ; sum = 0
MOV    r1, #0           ; i = 0
```

checksum\_loop

```
LDR    r3, [r2, r1, LSL #2] ; r3 = data[i]
ADD    r1, r1, #1         ; r1 = i + 1
AND    r1, r1, #0xff      ; i = (char) r1
CMP    r1, #0x40          ; le paquet contient 64 mots
ADD    r0, r3, r0         ; sum += r3
BCC    checksum_loop      ; if (i<64) loop
MOV    pc, r14
```



# Exemple 1

code assembleur (i est du type int)

checksum\_v2

```
MOV    r2, r0           ; r2 = data
MOV    r0, #0           ; sum = 0
MOV    r1, #0           ; i = 0
```

checksum\_loop

```
LDR    r3, [r2, r1, LSL #2] ; r3 = data[i]
ADD    r1, r1, #1         ; r1 = i + 1
CMP    r1, #0x40          ; le paquet contient 64
```

mots

```
ADD    r0, r3, r0         ; sum += r3
BCC    checksum_loop      ; if (i<64) loop
MOV    pc, r14
```

# Exemple 2

les données sont sur 16 bits

```
short checksum_v3(short *data)
{
    unsigned int i;
    short sum=0;

    for(i=0; i<64; i++)
    {
        sum = (short)(sum + data[i]); // x+y est du type int
    }
    return sum;
}
```

# Exemple 2

code assembleur (les données a additionner sont sur 16 bits)

checksum\_v3

```
MOV    r2,r0          ; r2 = data
MOV    r0,#0           ; sum = 0
MOV    r1,#0           ; i = 0
```

checksum\_loop

```
ADD     r3,r2,r1,LSL #1 ; r3 = &data[i]
LDRH    r3,[r3,#0]      ; r3=data[i] pas d'offset autorise
ADD     r1,r1,#1        ; i++
CMP     r1,#0x40        ; compare i, 64
ADD     r0,r3,r0        ; r0 = r0 + r3
MOV     r0,r0,LSL #16
MOV     r0,r0,ASR #16   ; sum = (short)r0 extension du signe
BCC     checksum_loop
MOV     pc,r14
```

# Exemple 2

L'instruction LDRH ne permet pas tous les modes d'adressages de l'instruction LDR (pré index, registre d'offset et décalage), la première instruction **ADD** calcule l'adresse de la donnée. (la solution consiste à incrémenter un pointeur sur le tableau, c.f. après)

Les données à additionner sont du type short (16 bits signées) et les additions de type 32 bits signées. Pour convertir 32 bits signés-> 16 bits signés le compilateur utilise deux instructions. Un premier décalage à gauche (16 bits) pour enlever l'information contenue dans les bits de poids fort du registre et un décalage (16 bits) vers la droite **pour étendre le signe** (en utilisant une variable de type int pour la somme on supprime ces instructions, la conversion s'effectue une seule fois lorsque l'on quitte la fonction)

# Extension du signe

Les instructions

```
MOV    r0,r0, LSL #16
```

```
MOV    r0,r0,ASR #16
```

effectuent l'extension du signe de la donnée (signée) sur 16 bits aux 32 bits du registre. En effet (exemple sur 8 bits), on considère une valeur -3 codée sur 4 bits et étendue à 8 bits.

$-3 = -(0011) = 1101$  la dernière égalité utilise le complément à deux

on effectue la même manipulation sur 8 bits

$-3 = -(0000\ 0011) = \underbrace{1111\ 1101}$

les bits de poids fort sont tous à 1, pour étendre un nombre négatif codé sur 4 bits à 8 bits, il faut ajouter des 1

# Finalement

```
short checksum_v4(short *data)
{
    unsigned int i;
    int sum = 0;
    for(i=1; i<64; i++)
    {
        sum+=*(data++);
    }
    return (short) sum;
}
```

# Finalemment

checksum\_v4

MOV r2, #0

MOV r1, #0

checksum\_loop

LDRSH r3,[r0],#2 ; post index immédiat r3=\*(data++)

ADD r1,r1,#1 , i++

CMP r1,#0x40 ; i<64?

ADD r2,r3,r2 ; sum+=data[i]

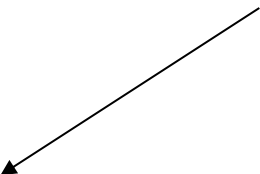
BCC checksum\_v4

MOV r0,r2, LSL #16

MOV r0,r0,ASR #16 ; r0 = (short)sum

MOV pc,r14 ; return

toujours mise à jour de r0 (voir la différence avec pré index et !)



# Variables locales – Arguments de fonctions

Convertir les variables locales de types *char* ou *short* en des variables de type *int* augmente les performances et diminue la taille du code.

Ce constat est aussi vrai pour les variables arguments de fonctions

## **exemple:**

```
short add_v1(short a, short b)
{
    return a+(b>>1);
}
```



# Arguments de fonctions

Les arguments a et b ainsi que la valeur de retour sont passés en arguments à la fonction en utilisant les registres (32 bits).

Le compilateur à deux options possibles:

1. il suppose que les données sont dans l'intervalle -32'768 - +32'767 les données ne sont pas modifiées (passage de paramètre **large, wide**)
2. il force les valeurs à être dans cet intervalle en étendant le signe des données sur 16 bits aux 32 bits des registres (passage de paramètre **étroit, narrow**)

Le compilateur doit aussi déterminer si c'est la fonction appelante (paramètre étroit) ou appelée (paramètre large) qui effectue le changement de type (cast)

# Valeur de retour

Si le compilateur retourne une valeur large alors la fonction appelante doit effectuer le changement de type (réduction des valeurs à l'intervalle).

Si le compilateur retourne les valeurs étroite alors c'est la fonction appelée qui effectue les conversions.

**Exemple de code assembleur:** paramètre étroits, valeur de retour étroit

add\_v1

```
ADD    r0,r0,r1,ASR #1    ; r0 =(int)r0+2*(int)r1
MOV     r0,r0,LSL #16
MOV     r0,r0,ASR #16     ; r0=(short)r0
MOV     pc,r14
```

# Exemple 2 compilateur gcc

add\_v1\_gcc

```
MOV    r0,r0,LSL #16
MOV    r1,r1,LSL #16
MOV    r1,r1,ASR #17          ; r1=(int)b>>2
ADD    r1,r1,r0,ASR #16      ; r1+=(int)a
MOV    r1,r1,LSL #16
MOV    r0,r1,ASR #17          ; r0 = (short)r1
MOV    pc,r14                 ; return r0
```

Les changements de types sont effectués sur les arguments et sur la valeur de retour.

# Type signé/non signé

faut-il choisir *signed int* ou *unsigned int* ?

Si le programme utilise des additions, soustractions et des multiplications, il n'y a pas de différences.

Il y a des différences si on utilise des **divisions**.

**exemple:** calcul de la moyenne de deux nombres

```
int average(int a, int b)
{
    return (a+b)/2;
}
```

# Type signé/non signé

average\_v1

```
ADD    r0,r0,r1          ; r0 = a + b
ADD    r0,r0,r0,LSR #31  ; if (r0<0) r0++
MOV    r0,r0,ASR #1      ; r0 = r0 >> 1
MOV    pc,r14
```

On doit tester si  $r0 < 0$  avant de décaler à droite (diviser par deux) car l'opération de décalage à droite ne correspond pas à une division par deux si le nombre est signé et négatif.

En effet:  $-3 = 1101$ ,  $-3 \gg 1 = 1110 = -2$  mais la division  $-3/2$  doit donner  $-1$  (on divise en arrondissant vers 0 comme on le fait pour les nombres positifs)

**Si c'est possible il faut préférer les entiers non signé dès que les divisions sont utilisées**

# Récapitulations

1. les variables locales qui se trouvent dans les registres doivent de préférences être de type int (signé ou non).
2. pour les variables globales, les tableaux qui se trouvent en mémoire il faut préférer le type qui utilise le moins de place possible. Les données doivent être parcourues en modifiant un pointeur sur la structure.
3. Utiliser des casts (int), (short) explicite dans les programmes lorsque l'on assigne une variable locale ou écrit la valeur d'une variable locale en mémoire (cast étroit explicite).
4. il faut éviter de laisser le compilateur effectuer des changements de types implicites. Utiliser des options de compilations pour signaler le casts implicites.
5. Eviter d'utiliser les types *char* et *short* pour les arguments de fonctions et les valeurs de retour.

# Les boucles

## **Boucles avec un nombre fixe d'itérations:**

retour sur le calcul de la somme de contrôle

code C optimal

```
int check_sum(int *data)
{ unsigned int i;
  int sum=0;
  for(i=0; i<64; i++)
  {
    sum+=*(data++);
  }
  return sum;
}
```

# Les boucles

compilation assembleur

checksum\_v5

MOV r2,r0 ; r2=data

MOV r0,#0 ; sum=0

MOV r1,#0 ; i=0

checksum\_v5\_loop

LDR r3,[r2],#4 ; r3=\*(data++)

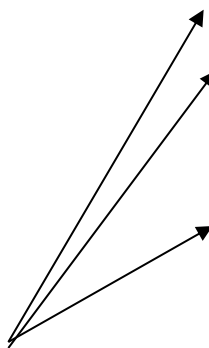
ADD r1,r1,#1 ; i++

CMP r1,#0x40 ; compare i, 64

ADD r0,r3,r0 ; sum += r3

BCC checksum\_v5\_loop

MOV pc,r14 ; return sum



trois instructions pour implémenter la boucle



# Les boucle en assembleur ARM

Deux instructions sont suffisantes pour implémenter une boucle:

1. Une soustraction pour décrémenter le compteur de boucle avec mise à jour du registre cpsr (en particulier le bit C)
2. Une instruction conditionnelle de branchement


Le compteur doit donc être initialisé avec le nombre d'itérations et être décrémenté pour éviter l'instruction CMP. Le compteur ne peut plus être utilisé comme index des données (data[i]) qui doivent être accédée par pointeur (\*(data++)).

# Exemple

code C

```
int checksum_v6(int *data)
{
    unsigned int i;
    int sum=0;
    for(i=64; i!=0; i--)
    {
        sum+=*(data++);
    }
    return sum;
}
```

i est non signé, alors la  
condition (i>0) est équivalente



# Exemple

code assembleur

checksum\_v6

MOV r2,r0 ; r2=data

MOV r0,#0 ; sum = 0

MOV r1,#0x40 ; i = 64

checksum\_loop

LDR r3,[r2],#4 ; r3 = \*(data++)

SUBS r1,r1,#1 ; i-- + mise-à-jour cpsr

ADD r0,r3,r0 ; sum+=r3

BNE checksum\_loop ; si (i!=0) boucle

MOV pc,r14 ; return sum



2 instructions pour la boucle

# Compteur signé/non-signé

Si le compteur de boucle est non signé, les conditions  $i \neq 0$  et  $i > 0$  sont équivalentes. Si le compteur est signé on est tenté d'utiliser la condition  $i > 0$ . On espère que le compilateur va générer le code suivant:

SUBS	r1,r1,#1	compare i avec 1, $i=i-1$
BGT	loop	branch if greater than

En fait le compilateur va générer le code suivant

SUB	r1,r1,#1	$r1 \leftarrow$
CMP	r1,#0	
BGT	loop	branch si $r1 > 0$

# Compteur signé/non signé

Le compilateur se méfie du cas où  $i = 0x80000000$

Le premier code compare  $0x80000000$  avec 1, comme  $0x80000000 < 1$   
La boucle se termine.

Le deuxième fragment de code commence par décrémenter le compteur pour obtenir  $0x7fffffff$ , qui est plus grand que zéro. La boucle continue.

# Boucles avec un nombre variable d'itérations

code C

```
int checksum_v7(int *data, unsigned int N)
{
    int sum = 0;
    for(; N!=0; N--)
    {
        sum +=*(data++);
    }
    return sum;
}
```

compteur non signé

le compteur est décrémenté vers 0  
(2 instructions assembleur pour la boucle)

on incrémente le pointeur

# Boucles avec un nombre variable d'itérations

code assembleur

checksum\_v7

```
MOV    r2,#0                ; sum = 0
CMP    r1,#0                ; compare N, 0
BEQ    checksum_v7_end
```

checksum\_v7\_loop

```
LDR    r3,[r0],#4           ; r3 = *(data++)
SUBS   r1,r1,#1             ; N- - + mise-à-jour de cpsr
ADD    r2,r3,r2             ; sum += r3
BNE    checksum_v7_loop
```

checksum\_v7\_end

```
MOV    r0,r2                ; r0 = sum
MOV    pc,r14
```

# Boucle avec un nombre variable d'itérations

Le compilateur test que le compteur N est plus grand que 0 avant d'effectuer une première boucle. Si le programmeur est certain que cette valeur est >0 alors il faut utiliser une structure **do-while**

```
int checksum_v8(int *data, unsigned int N)
{
    int sum = 0;
    do {
        sum += *(data++);
    }while (N--!=0)
    return sum;
}
```



# Boucle avec un nombre variable d'itérations

do-while, code assembleur:

checksum\_v8

MOV r2,#0

checksum\_v8\_loop

LDR r3,[r0],#4 ; r3 = \*(data ++)

Xcycles

SUBS r1,r1,#1 ; N- -

1 cycle

ADD r2,r3,r2 ; sum += r3

1 cycle

BNE checksum\_v8\_loop ; if (N!=0)

3 cycles

MOV r0,r2 ; r0 = sum

MOV pc,r14 ; return

# Technique de déroulement des boucles

Une boucle coûte deux instructions – une soustraction et un branchement conditionnel.

Pour le processeur ARM7, une soustraction s'exécute en 1 cycle et un branchement en 3 cycles, le coût de gestion d'une boucle est de 4 cycles par boucle.

Pour diminuer le coût de gestion de la boucle, on 'déroule' la boucle, c'est-à-dire que l'on recopie plusieurs fois le corps de la boucle (sans test ni soustraction)

Evidemment, c'est possible seulement pour certaines valeurs de N

# Déroulement des boucles

On suppose que le nombre de paquets N est un multiple de 4

```
int checksum_v9(int *data, unsigned int N)
{
    int sum = 0;
    do {
        sum += *(data++);
        sum += *(data++);
        sum += *(data++);
        sum += *(data++);
        N -=4;
    } while (N != 0);
}
```

# Déroulement des boucles

pour le code assembleur de checksum\_v8 le traitement du corps de la boucle s'exécute en  $X + 5$  cycles.

Si la boucle est déroulée, il faut exécuter 4 LDR  **$4 \times X$  cycles**, 4 ADD **4 cycles**, un SUBS **1 cycle**, un BNE **3 cycles** soit  $8 + 4 \times X$  cycles au total.

Temps de gestion de la boucle 4 cycles pour 4 accumulations.

# Déroulement des boucles

- seules les boucles qui sont fréquemment utilisées sont susceptibles d'être déroulées, sinon on augmente la taille du code sans gain de performance significatif
- il faut considérer aussi la taille du corps de la boucle. Plus le corps de la boucle est grand moins les pertes dues à la gestion de la boucle (4 cycles) sont importants et plus la duplication du code augmente la taille totale du code. De plus, ça diminue les performances de la mémoire cache. Généralement, si dérouler une boucle augmente les performances de moins de 1%, la technique n'est plus efficace.
- si le nombre total d'itérations à exécuter n'est pas un multiple constant de 2, 4, etc. on peut dérouler une partie du code et faire le reste en ajoutant des instructions (sans boucles)

# Déroulement des boucles

code C

```
int checksum_v10(int *data, unsigned int N)
{
    unsigned int i;
    int sum = 0;
    for (i=N/4; i!=0; i--)
    {
        sum+=*(data++); sum+=*(data++);
        sum+=*(data++); sum+=*(data++);
    }
    for(i=N&3;i!=0;i--) // & opérateur et sur les bits, positionne tous les
        sum+=*(data++); // bits à 0 excepté le deux premiers
    return sum;
}
```

# Boucles - résumé

- les compteurs doivent être décrémentés vers 0 pour éviter que le compilateur utilise un registre pour mémoriser le nombre d'itérations et effectuer la comparaison avec zéro (gratuit...)
- les compteurs de boucles sont non signés, les conditions de terminaisons ( $i \neq 0$ ) et ( $i > 0$ ) sont équivalentes (il faut éviter la situation où le compteur démarre à 0x80000000 et passe à 0x7FFFFFFF)
- utiliser un do-while de préférence à un for pour éviter un test au début de la boucle
- dérouler les boucles si c'est efficace
- préférer les structures de données qui génèrent des tableaux multiples de 2, 4, etc. et qui permettent le déroulement des boucles

# Allocation des registres

Le compilateur essaye d'allouer un registre par variable locale déclarée (utilisée) dans une fonction C. Si plusieurs variables ne sont pas utilisées simultanément, le compilateur utilise le même registre pour ces variables.

Si le nombre total de variables est supérieur au nombre de registres le compilateur utilise **la pile**. L'accès à ces variables est lent car il nécessite un accès à la mémoire externe.

Lors de l'écriture d'une fonction il faut:

1. **minimiser l'utilisation de la pile**
2. **s'assurer que les variables les plus fréquemment utilisées sont stockées dans des registres.**



# Convention

Le passage des paramètres en C utilise les conventions suivantes concernant l'usage des registres

**r0-r3:** Registres pour le passage de paramètres ainsi que la valeur de retour

**r4-r8:** Registres d'usage général pour la fonction. Les valeurs de ces registres doivent être sauveées/restaurées pour contenir la même valeur au début et à la fin de l'exécution de la routine.

**r9-r11:** idem que les registres r4-r8 sauf si certaines options de compilations sont activées.

# Convention

**r12:** un « scratch register », la fonction peut l'utiliser sans se soucier de le restaurer.

**r13:** le registre de pile, stack register, sr, (full descending)

**r14:** le registre de lien, link register, lr

**r15:** le compteur de programme, program counter, pc

En règle général les compilateurs peuvent utiliser les registres r0-r11 pour les variables locales, r12 pour les calculs intermédiaires. **Une bonne habitude est de limiter le nombre de variables locales à 12.**

# Convention

Si le compilateur doit utiliser la pile, il le fait pour les variables le moins fréquemment utilisées.

Une variable utilisée à l'intérieur d'une boucle compte plusieurs fois. Comme il peut être difficile pour le compilateur d'estimer le nombre de fois qu'une boucle est exécutée, **le compilateur sélectionne les variables qui se trouvent dans le plus grand nombre de boucles imbriquées.**

Le langage C définit le mot clé ***register*** pour indiquer qu'une variable doit (si possible) être sauvée dans un registre.

# Appels de fonctions - arguments

ARM définit un standard pour l'appel des procédures: ARM Procedure Call Standard (APCS), qui définit comment les arguments sont passés aux fonctions et comment sont gérées les valeurs de retour.

Les quatre premiers arguments entiers (pointeurs) sont passés dans les registres r0, r1, r2, r3.

Les autres arguments entiers (pointeurs) sont placés sur la pile (full descending) de manière croissante avec la mémoire

sp+12	argument 7
sp+8	argument 6
sp+4	argument 5
sp	argument 4

r3	argument 3	
r2	argument 2	
r1	argument 1	
r0	argument 0	valeur retour

# Appels de fonctions - arguments

Les arguments codés sur deux mots tel que les double ou long long sont passés dans deux registres consécutifs et retourné dans r0, r1

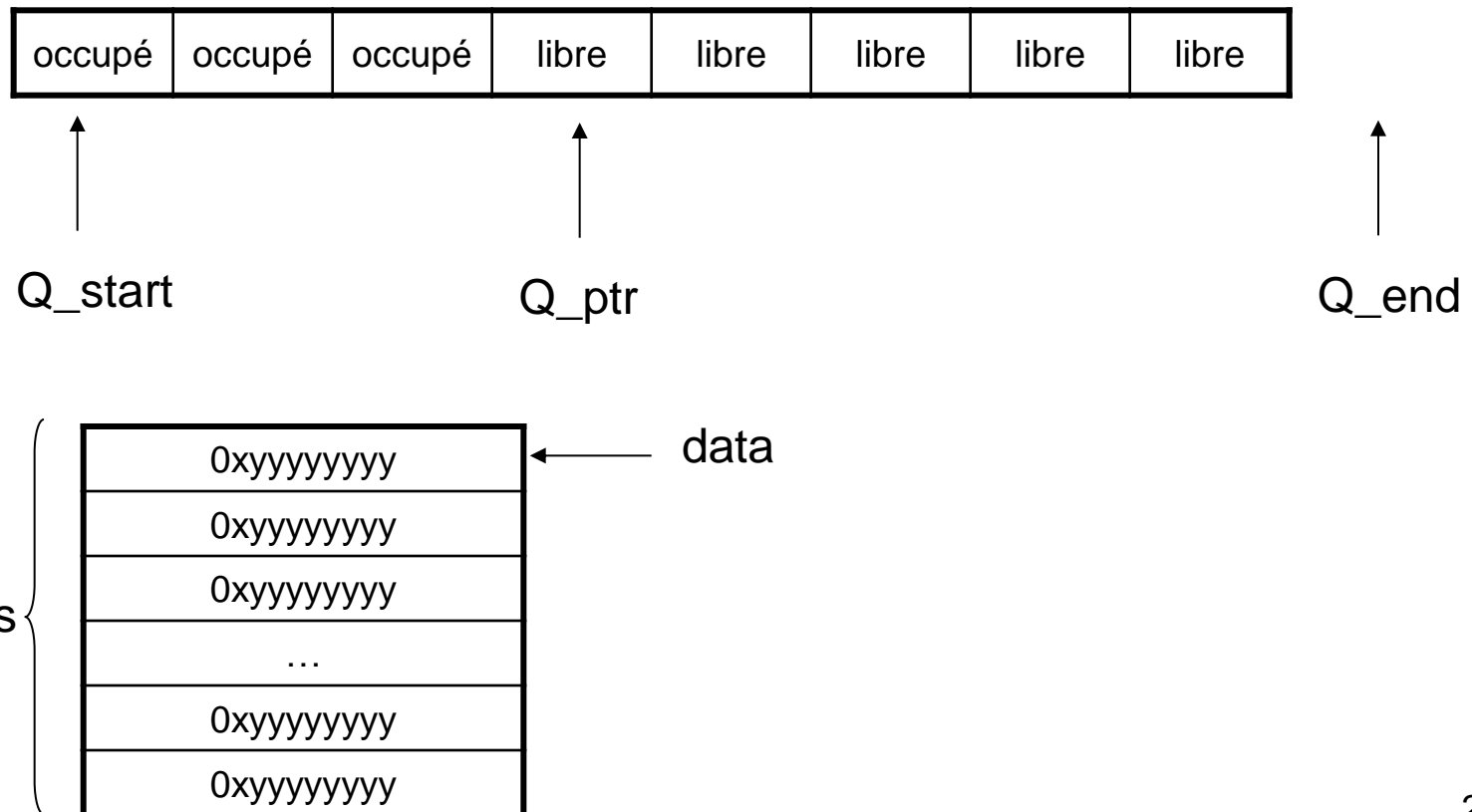
Les fonctions qui utilisent au maximum quatre arguments sont plus efficaces (l'utilisation de la pile nécessite des accès externes).

En C++ le premier argument passé lors de l'appel à la méthode d'un objet est le pointeur *this*. L'argument est implicite et supplémentaire aux autres arguments.

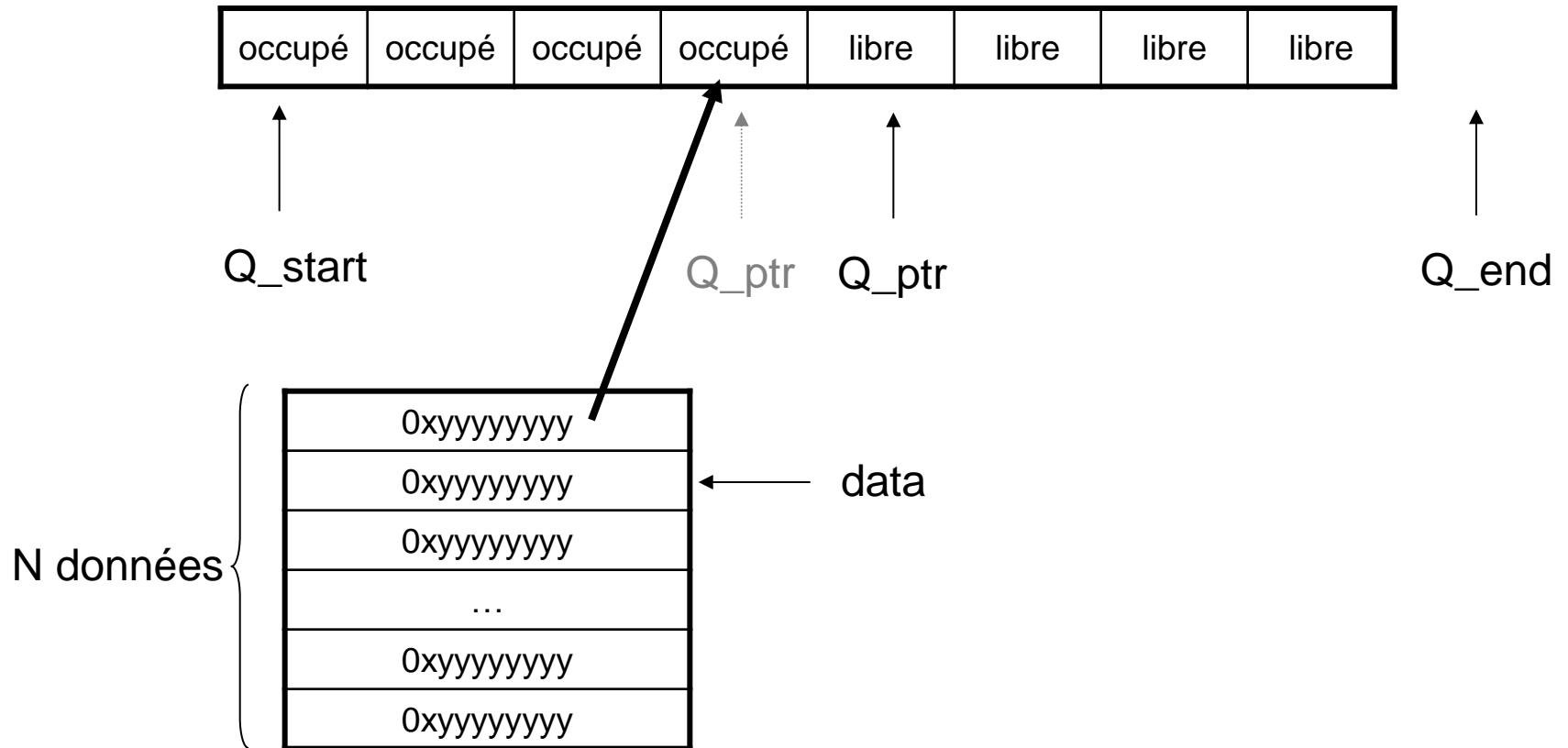
Si une fonction C nécessite plus de quatre arguments (trois en C++) alors *il faut les regrouper dans une structure* et passer en argument un pointeur sur la structure.

# Exemple

Une routine qui permet d'insérer des caractères dans une file d'attente (tampon circulaire).



# Exemple



# Code C

```
char *queue_bytes_v1(char *Q_start, char *Q_end, char *Q_ptr, char *data,
                    unsigned int N)
{
    do {
        *(Q_ptr++)=*(data++);
        if (Q_ptr==Q_end) Q_ptr=Q_start;
    } while (--N)
    return Q_ptr;
}
```

Q\_start: adresse du début de la queue (inclus)

Q\_end: adresse de la fin de la queue (non inclus)

Q\_ptr: adresse de la position courante dans la queue

data: adresse sur le début des données à transférer dans la queue

N : nombre de bytes a transférer



# Code assembleur

queue\_bytes\_v1

STR	r14, [r13,#-4]!	sauvegarde de lr sur la pile + mise à jour
LDR	r12, [r13,#4]	r12=N

queue\_v1\_loop

LDRB	r14, [r3], #1	r14 = *(data++), mise à jour de r3
STRB	r14, [r2], #1	*(Q_ptr++)=r14
CMP	r2, r1	if (Q_ptr == Q_end)
MOVEQ	r2, r0	Q_ptr = Q_start
SUBS	r12, r12, #1	N- - mise a jour du cpsr
BNE	queue_v1_loop	if (N!=0) goto loop
MOV	r0, r2	r0 = Q_ptr
LDR	pc, [r13], #4	return r0

# Utilisation d'une structure

```
typedef struct {  
    char *Q_start, *Q_end, *Q_ptr;  
} Queue;
```

```
void queue_bytes_v2(Queue *queue, char *data, unsigned int N) {  
    char *Q_ptr=queue->Q_ptr;  
    char *Q_end=queue->Q_end;  
    do {  
        *(Q_ptr++)=*(data++);  
        if (Q_ptr == Q_end) Q_ptr = queue->Q_start;  
    } while (--N)  
    queue->Q_ptr = Q_ptr;  
}
```

# utilisation d'une structure- code assembleur

queue\_bytes\_v2

STR	r14, [r13, #-4]	sauvegarde lr sur la pile
LDR	r3, [r0, #8]	r3 = queue->Q_ptr
LDR	r14, [r0, #4]	Q_end

queue\_v2\_loop

LDRB	r12, [r1], #1	r12 = *(data++)
STRB	r12, [r3], #1	*(Q_ptr++) = r12
CMP	r3, r14	if (Q_ptr == Q_end)
LDREQ	r3, [r0, #0]	Q_ptr = queue->Q_start
SUBS	r2, r2, #1	--N et mise a jour du cpsr
BNE	queue_v2_loop	if (N!=0) goto loop
STR	r3, [r0, #8]	queue->Q_ptr = r3
LDR	pc, [r13], #4	return

# comparaison v\_1 – v\_2

La deuxième version du programme qui utilise une structure contient une instruction de plus (au début du programme pour initialiser les pointeurs)

Chaque appel à la fonction nécessite l'initialisation de 3 registres (les trois arguments) pour la seconde version contre 4 registres et un paramètre à placer sur la pile (empiler et dépiler le paramètre N). Si on considère l'appel à la fonction la deuxième version comporte moins d'instructions (au moins 1 instructions pour modifier le registres supplémentaire et deux instructions pour empiler/dépiler).

Probablement que d'autres instructions sont 'économisées' en utilisant la structure dans d'autres parties du programme.

# Autres stratégies

Lorsqu'une fonction utilise peu de variables locales ou/et que le code généré par la fonction est petit le compilateur peut procéder à des optimisations **si le code de la fonction se trouve dans le même fichier que le code des fonctions appelantes**

1. Le compilateur utilise dans le corps du programme des registres différents que dans le corps de la fonction pour éviter des sauvegardes
2. Si possible le corps de la fonction est placé directement dans le corps du programme (inline) pour supprimer les instructions d'appels de la fonction (sauvegarde sur la pile, mise-à-jour des registres).

# Exemple 2

une fonction `uint_to_hex` converti un entier 32 bits en une chaîne de caractères de 8 caractères hexadécimaux.

```
unsigned int nybble_to_hex(unsigned int d) {  
    if (d<10) return d + '0';  
    return d - 10 + 'A';  
}
```

```
void uint_to_hex(char *out, unsigned int in) {  
    unsigned int i;  
    for(i=8;i!=0;i--){  
        in = (in<<4) | (in>>28);    // rotation à gauche de 4 bits  
        *(out++) = (char)nybble_to_hex(in & 15);  
    }  
}
```

# Exemple 2 – code assembleur

uint\_to\_hex

```
MOV    r3, #8           ; i=8
```

uint\_to\_hex\_loop

```
MOV    r1, r1, ROR #28   ; rotation 4 bits à gauche
AND     r2, r1, #0xf      ; r2 = in & 15
CMP     r2, 0xa           ; if (r2>=10)
ADDCS   r2, r2, #0x41      ; r2 += 'A' – 10 non signé
ADDCC   r2, r2, #0x30      ; r2 += '0' non signé
STRB    r2, [r0], #1       ; *(out++)=r2
SUBS    r3, r3, #1         ; i-- + mise-à-jour du cpsr
BNE     uint_to_hex_loop   ; if (i!=0) goto loop
MOV     pc, r14            ; return
```

# Exemple 2 – code assembleur

L'appel à la fonction `nybble_to_hex` à été supprimer et la fonction insérer dans le corps de la routine appelante (`inline`)

Stratégies pour une programmation efficace:

1. Utiliser des fonctions avec 4 arguments, utiliser des structures et passer en paramètres des pointeurs sur les structures.
2. Définir de petites fonctions, les inclure toutes dans le même fichier et définir les fonctions avant de les appeler pour permettre au compilateur de bien gérer les registres et les `inline`.
3. Les fonctions critiques doivent être précédées du mot clé `inline`.



# aliasing - pointeur

Le terme aliasing réfère à la situation où plusieurs pointeurs pointent sur la même adresse. Lorsque plusieurs pointeurs sont utilisés, le compilateur doit toujours suspecter que la valeur pointée par un pointeur a été modifiée même si ce pointeur n'a pas été utilisé.

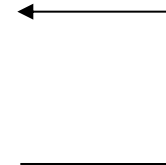
exemple:

```
void timer_v1(int *timer1, int *timer2, int *step)
{
    *timer1 += *step;
    *timer2 += *step;
}
```

# code assembleur

timer\_v1

```
LDR    r3, [r0,#0]      ; r3 = *timer1
LDR    r12, [r2,#0]     ; r12 = *step
ADD    r3, r3, r12      ; r3 += r12
STR    r3, [r0, #0]     ; *timer1 = r3
LDR    r0, [r1, #0]     ; r0 = *timer2
LDR    r2, [r2,#0]    ; r2 = *step
ADD    r0, r0, r2       ; r0 += r2
STR    r0, [r1, #0]     ; *timer2 = r0
MOV    pc, r14
```



l'écriture à pu modifier  
le contenu de [r2]  
aliasing possible

# structures

L'instruction LDR supplémentaire est un accès à la mémoire externe, à éviter. La solution consiste à utiliser une variable locale

```
typedef struct {int step;} State;  
typedef struct {int timer1, timer2;} Timers;  
  
void timers(State *state, Timers *timers)  
{  
    int step = state->step;  
    timers->timer1 += step;  
    timers->timer2 += step;  
}
```

# structures

ATTENTION, le code suivant génère aussi une instruction supplémentaire, même si des structures sont utilisées

```
void timers(State *state, Timers *timers)
{
    timers->timer1 += state->step;
    timers->timer2 += state->step;
}
```

*Un autre exemple est*

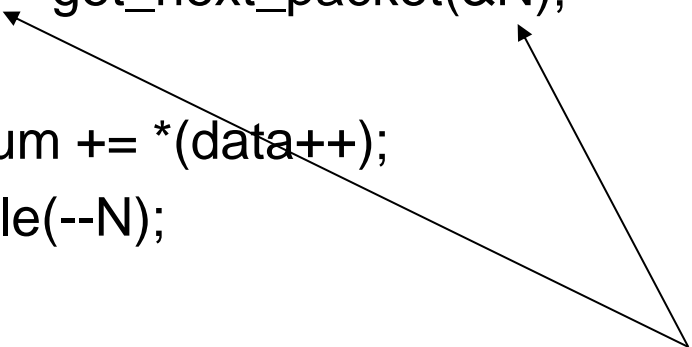
state->step

appel a une fonction

state->step ; réinitialisation de la variable à cause de l'appel à la fonction

# Exemple 2

```
int checksum(void)
{  int *data; int N, sum=0;
  data = get_next_packet(&N);
  do {
    sum += *(data++);
  } while(--N);
}
```



le compilateur doit prendre en compte  
que les deux pointeurs peuvent pointer  
sur la même donnée (aliasing)

# Exemple2 - assembleur

checksum

```
STMFD    r13!, {r4, r14}  
SUB       r13, r13, #8  
ADD       r0, r13, #4  
MOV       r4, #0  
BL        get_next_packet
```

sauvegarde r4, lr sur la pile  
crée deux variables sur pile  
**r0 = &N pour appel proc.**  
sum = 0

checksum\_loop

```
LDR       r1, [r0], #4  
ADD       r4, r1, r4  
LDR       r1, [r13, #4]  
SUBS     r1, r1, #1  
STR       r1, [r13, #4]  
BNE       checksum_loop  
MOV       r0, r4  
ADD       r13, r13, #8  
LDMFD     r13!, {r4, pc}
```

r1 = \*(data++)  
sum += r1  
r1 = N depuis la pile!  
r1– et mise à jour cpsr  
N=r1 écriture sur la pile !  
  
r0 = sum  
supprime les variables sur pile

## exemple 2 - remarque

Le compilateur réserve de la place pour deux variables sur la pile et en utilise une seule.....

L'assembleur ARMv5 met a disposition l'instruction LDRD (Load 64 bits) pour travailler sur 64 bits et les mots doivent être alignés sur des adresses divisibles par 8.

Le compilateur maintient les adresses de la pile alignées.

L'instruction LDRD n'est pas utilisée dans le corps de la routine mais pourrait l'être dans la routine `get_next_packet`

# aliasing recommendations

Il faut utiliser des variables locales pour éviter que le compilateur considère des données identiques comme différentes

Il faut éviter d'accéder une variable locale par son adresse pour éviter qu'elle se trouve sur la pile. Si c'est nécessaire il faut définir une nouvelle variable locale et copier la valeur utile.



# Alignement

L'organisation des structures composées peut avoir des conséquences sur les performances du code généré par le compilateur.

Les données accédées par les instructions load/store doivent être alignées.  
Jusqu'à la version ARMv5TE, on a les restrictions

1 byte	LDRB, LDRSB, STRB	Alignement quelconque
2 bytes	LDRH, LDRSH, STRH	Multiple de 2 bytes
4 bytes	LDR, STR	Multiple de 4 bytes
8 bytes	LDRD, STRD	Multiple de 8 bytes

# Alignement

Pour respecter ces contraintes, le compilateur ajoute des champs non utilisés (pad) dans les structure de données composées.

Exemple:

```
struct {  
    char a;  
    int b;  
    char c;  
    short d;  
}
```

	+3	+2	+1	+0
+0	pad	pad	pad	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	d[15,8]	d[7,0]	pad	c

(little-endian)

# Alignement

Pour optimiser l'occupation de la mémoire on doit réordonner la définition de la structure.

```
struct {  
    char a;  
    char c;  
    short d;  
    int b;  
}
```

(little endian)

	+3	+2	+1	+0
+0	d[15,8]	d[7,0]	c	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	libre	libre	libre	libre

# Alignement

Certain compilateur dispose d'option pour réduire la taille des données en mémoire en supprimant les bits de padding. Pour le compilateur *armcc*, c'est le mot clé *\_packed*, qui ne produit pas toujours une organisation optimale

```
_packed struct{  
    char a;  
    int b;  
    char c;  
    short d;  
}
```


	+3	+2	+1	+0
+0	b[23,16]	b[15,8]	b[7,0]	a
+4	D[16,8]	d[7,0]	c	b[31,24]
+8	libre	libre	libre	libre

Pour s'assurer d'obtenir le résultat voulu, il est recommandé d'introduire les champs de remplissages manuellement.

# Champs de bits

En C on peut définir des booléens en utilisant un seul bit de donnée en mémoire.

```
typedef struct {  
    unsigned int StageA : 1;  
    unsigned int StageB : 1;  
    unsigned int StageC : 1;  
} Stages_v1;
```



3 bits en mémoire

Stages\_v1 exemple;

exemple.StageA = 1;

# Champs de bits

L'ordre des bits en mémoire change selon le compilateur.  
Souvent le code généré n'est pas optimisé.

Exemple:

```
void dostagesA(void);  
void dostagesB(void);  
void dostagesC(void);
```

# Champs de bits

```
Void dostages_v1(Stages_v1 *stages) {  
    if (stages->StageA)  
        dostageA();  
    if (stages->StageB)  
        dostageB();  
    if (stages->StageC)  
        dostageC();  
}
```

# Champs de bits

dostages\_v1

STMFD r13!,{r4,r14}

MOV r4,r0

LDR r0,[r0,#0]

; r0 contient le champ de bits

TST r0,#1

; calcule r0 & #1, test le premier bit

BLNE dostageA

LDR r0,[r4,#0]

; r0 contient le champ de bits

MOV r0,r0,LSL #30

CMP r0, #0

BLLT dostageB

LDR r0,[r4,#0]

; nécessaire

MOV r0,r0, LSL #29

CMP r0,#0

LDMLTFD r13!,{r4,r14}

; return

BLT dostageC

LDMFD r13!,{r4,pc}

La fonction dostageA peut modifier le champ de bits!

sans link.



# Champs de bits

Le compilateur accède trois fois le champ de bits en mémoire pour prévenir les problèmes d'aliasing. Les champs de bits étant de taille quelconques, le compilateur utilise souvent des pointeurs.

Pour améliorer les performances, il est recommandé d'utiliser des opérations logiques.

```
typedef unsigned long Stages_v2;
```

```
#define STAGEA (1ul << 0)
```

```
#define STAGEB (1ul << 1)
```

```
#define STAGEC (1ul << 2)
```

# Champs de bits

```
void dostages_v2(Stages_v2 *stages_v2) {  
    Stages_v2 stages = *stages_v2;  
  
    if (stages & STAGEA) dostageA();  
    if (stages & STAGEB) dostageB();  
    if (stages & STAGEC) dostageC();  
}
```

Variable locale, plus d'aliasing



# Champs de bits

dostages\_v2

```
STMFD r13!,{r4,r14}  
LDR    r4,[r0,#0]  
TST    r4,#1  
BLNE   dostageA  
TST    r4,#2  
BLNE   dostageB  
TST    r4,#4  
LDMNEFD r13!,{r4,r14}  
BNE    dostageC  
LDMFD r13!,{r4,pc}
```

# Champs de bits

En conclusion, pour manipuler des champs de bits il est préférable d'utiliser des masques.

Exemple: *stages |= STAGEA; (enable)* *stages &= ~STAGEB; (disable)*  
*stages ^= STAGEc; (toggle)*

# Division

Le processeur ARM ne dispose pas d'instruction pour le calcul des divisions. Le temps d'exécution des routines des bibliothèques C standards varie entre 20 et 100 cycles en fonction de l'implémentation ou de la taille des opérandes.

Dans certaines situations il est possible de supprimer les opérations de divisions. Par exemple, pour la gestion d'un tampon circulaire. Dans ce cas, pour accéder à l'élément suivant du tampon, on écrit

*offset = (offset + increment) % buffer\_size;*

# Tampon circulaire

Il est plus efficace d'écrire

```
offset += increment;  
if (offset >= buffer_size)  
    offset -= buffer_size;
```

Le calcul du reste de la division peut prendre 50 cycles, la deuxième implémentation 3 (on suppose que  $\text{increment} < \text{buffer\_size}$ ).

# Division signée/non signée

Les routines de division de nombres signés sont implémentée en trois étapes:

1. On calcule la valeur absolue des opérandes
2. On calcule la division non signée
3. On calcule le signe

Il faut donc de préférence utiliser des nombres non signés.

# Division / reste

Les routines des bibliothèques C retournent généralement le résultat de la division et le reste. On utilise cette propriété pour calculer plus efficacement.

Exemple:

```
typedef struct { int x; int y; } point;
```

```
point getxy_v1(unsigned int offset, unsigned int bytes_per_line) {  
    point p;  
    p.y = offset / bytes_per_line;  
    p.x = offset - p.y * bytes_per_line;  
    return p;  
}
```



# Division / reste

Pour le calcul de  $p.x$ , on a voulu éviter de calculer une division. En fait, écrire  $p.x = \text{offset} \% \text{bytes\_per\_line}$ ; est meilleur, car le calcul de la première division retourne aussi le reste.

getxy\_v2

```
STMFD r13!, {r4,r14}
```

```
MOV    r4, r0
```

```
MOV    r0, r2
```

```
BL      _rt_udiv          valeurs de retour dans r0, r1
```

```
STR     r0, [r4, #4]
```

```
STR     r1, [r4, #0]
```

```
LDMFD r13!, {r4,pc}
```

# Division / Multiplication

Si un programme effectue souvent la division par un même dénominateur,  $z$ , il devient efficace de calculer  $1/z$  et de remplacer les divisions par des multiplications.

Dans tous les cas, on préfère calculer avec des valeurs entières et on évite les nombres en virgule flottante.

Une première idée est d'utiliser la transformation

$$n \div d = (n \cdot (2^{32} \div d)) \div 2^{32} = n \cdot s \div 2^{32}$$


Pour calculer la multiplication, on utilise des nombres entiers sur 64 bits

# Division / Multiplication

Cette approche nous oblige a effectuer une division avec des entiers sur 64 bits pour le calcul de s, ce qui est long.

D'autre part, si  $d=1$ , la valeur à mémoriser est aussi sur 64 bits.

En pratique, on utilise  $s = (2^{32} \div d) = \frac{2^{32}}{d} \div e_1; 0 < e_1 < 1$



exact

arrondis

Pour calculer la division, on utilise

$q = (\text{unsigned int})( ((\text{unsigned long long})n * s) >> 32);$

La division par  $2^{32}$  introduit une erreur

$$q = ns2^{32} \div e_2; 0 < e_2 < 1$$

# Division / Multiplication

D'où:

$$q = \left\lfloor \frac{n}{d} \right\rfloor \text{ ; } ne_1 2^{i-32} \leq e_2; \quad 0 \leq ne_1 2^{i-32} + e_2 < 2$$

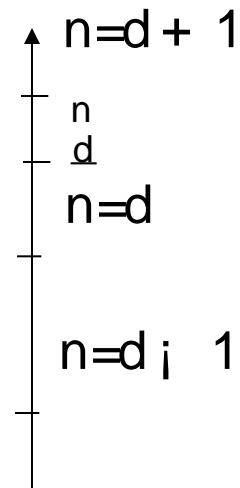
On a donc l'estimation  $\left( \left\lfloor \frac{n}{d} \right\rfloor + 2 < q \cdot \frac{n}{d} \right)$

$$q = \left\lfloor \frac{n}{d} \right\rfloor; \text{ ou } q = \left\lfloor \frac{n}{d} \right\rfloor + 1$$

Pour corriger le résultat, on calcule  $r = n - qd$ , en on vérifie que  $0 \leq r < d$

$r = n - q \cdot d;$

if ( $r \geq d$ ) {  $r -= d$ ;  $q++$ }



# Division / Multiplication

Exemple: On doit diviser des valeurs entières non signées contenues dans un tableau.

```
void scale( unsigned int *dest, unsigned int *src, unsigned int d,  
unsigned int N) {  
    unsigned int s = 0xFFFFFFFFu / d;  
    do { unsigned int n, q, r;  
        n = *(src++);  
        q=(unsigned int)(((unsigned long long)n * s) >> 32);  
        r = n - q*d;  
        if (r>=d) q++;  
        *(dest++)=q;  
    } while (--N);  
}
```

# Remarques

La multiplication 64 bits est implémentée en utilisant UMUL (maximum 4 cycles).

L'algorithme est le même si les nombres sont codés sur 16 bits, on utilise une multiplication sur 32 bits et

$$s = (2^{16} - 1) \cdot a$$

# Division constante

Si dans un programme une division par une constante  $d$  apparaît souvent, on peut optimiser l'exécution en cherchant une valeur de  $s$  qui donne le bon résultat. On cherche une approximation rationnelle

$$\frac{1}{d} \approx \frac{m}{2^{N+k}} = s$$

telle que  $n=d \cdot m \approx n=2^{N+k}; 0 \leq n < 2^N$

On a le résultat suivant

si  $2^{N+k} \leq ds \leq 2^{N+k} + 2^k$ ; alors  $n=d \cdot s$

si  $2^{N+k} - 2^k \leq ds < 2^{N+k}$ ; alors  $n=d \cdot s + s$

$0 \leq n < 2^N$

# Division constante

$$(n = (n \div d)d + r; \quad 0 \leq r < d)$$

En effet,

$$n \cdot 2^{N+k} = \frac{n \div d}{d} 2^{N+k} = \frac{(n \div d) 2^{N+k}}{d} + \frac{r 2^{N+k}}{d} < 2^{N+k} \frac{r+1}{d} < 2^{N+k}$$

$< 2^N \quad 0 \leq \dots \leq 2^k$

On procède de la même manière pour l'autre équation



# Division constante

$$(n+1)s_i - (n=d)2^{N+k} = (n+1) \frac{ds_i 2^{N+k}}{d} + \frac{(r+1)2^{N+k}}{d}$$

$< 2^N + 1 - 2^k \cdot \dots < 0$

# Division constante

En pratique, on a  $N=32$ , on choisit  $k$  tel que  $2^k < d \leq 2^{k+1}$  et

$$s = (2^{N+k} + 2^k) \cdot d = \frac{2^{N+k} + 2^k}{d} \cdot d; \quad 0 \leq e < 1$$

On a

$$ds = \frac{2^{N+k} + 2^k}{d} \cdot d; \quad ed < 2^{N+k} + 2^k$$

$$ds = \frac{2^{N+k} + 2^k}{d} \cdot d; \quad ed, \quad 2^{N+k} + 2^k \leq d, \quad 2^{N+k} + 2^k \leq \frac{2^{2k+1}}{2^k}$$

# Division constante

Avec ces choix de  $s$  et  $k$ , un des deux résultats s'applique toujours.  
On calcule donc  $ds$  et on implémente la multiplication selon

$$ds \begin{cases} < 2^N + k \\ < 2^N + k \end{cases}$$

# Implémentation de la division

On suppose donnés  $n$  et  $d$  et on cherche à calculer  $q=n/d$  la division entière et le reste  $r=n\%d$ .

Supposons qu'on sache que le quotient  $q$  peut être codé sur  $N$  bits. Une implémentation de la division consiste à calculer les  $N$  bits de  $q$  successivement.

En C: (division non signée)

```
unsigned udiv_simple( unsigned d, unsigned n, unsigned N) {  
    unsigned q=0, r=n;
```

# Implémentation de la division

```
do
{
  N--; // on test le bit suivant
  if ( (r>>N) >= d) {    // si r >= d * (1<<N)
    r -= (d<<N);
    d += (1<<N);
  } while(N);

  return q;
}
```

En fait, on décompose

$$n = c_0d + c_12d + c_24d + c_48d + \dots$$

# Implémentation de la division

Pour montrer que l'algorithme est correcte, on montre que l'expression ci-dessous est un invariant dans la boucle

$$n = qd + r, \quad 0 \leq r < d^{2N}$$

# En assembleur

## 7.3.1.1 Unsigned 32-Bit/32-Bit Divide by Trial Subtraction

This is the operation required by C compilers. It is called when the expression  $n/d$  or  $n\%d$  occurs in C and  $d$  is not a power of 2. The routine returns a **two-element** structure consisting of the quotient and remainder.

```
d      RN 0    ; input denominator d, output quotient
r      RN 1    ; input numerator n, output remainder
t      RN 2    ; scratch register
q      RN 3    ; current quotient
```

```
        ; __value_in_regs struct { unsigned q, r; }
        ; udiv_32by32_arm7m(unsigned d, unsigned n)
udiv_32by32_arm7m
```

```

MOV      q, #0                ; zero quotient
RSBS     t, d, r, LSR#3       ; if ((r>>3)>=d) C=1; else C=0;
BCC      div_3bits            ; quotient fits in 3 bits
RSBS     t, d, r, LSR#8       ; if ((r>>8)>=d) C=1; else C=0;
BCC      div_8bits            ; quotient fits in 8 bits
MOV      d, d, LSL#8          ; d = d*256
ORR      q, q, #0xFF000000    ; make div_loop iterate twice
RSBS     t, d, r, LSR#4       ; if ((r>>4)>=d) C=1; else C=0;
BCC      div_4bits            ; quotient fits in 12 bits
RSBS     t, d, r, LSR#8       ; if ((r>>8)>=d) C=1; else C=0;
BCC      div_8bits            ; quotient fits in 16 bits
MOV      d, d, LSL#8          ; d = d*256
ORR      q, q, #0x00FF0000    ; make div_loop iterate 3 times
RSBS     t, d, r, LSR#8       ; if ((r>>8)>=d)
MOVCS    d, d, LSL#8          ; { d = d*256;
ORRCS    q, q, #0x0000FF00    ; make div_loop iterate 4 times}
RSBS     t, d, r, LSR#4       ; if ((r>>4)<d)
BCC      div_4bits            ; r/d quotient fits in 4 bits
RSBS     t, d, #0             ; if (0 >= d)
BCS      div_by_0             ; goto divide by zero trap
; fall through to the loop with C=0
div_loop

```



```

; ... through to the loop with C=0
div_loop
    MOVCS    d, d, LSR#8        ; if (next loop) d = d/256
div_8bits
    RSBS     t, d, r, LSR#7      ; calculate 8 quotient bits
    SUBCS    r, r, d, LSL#7      ; if ((r>>7)>=d) C=1; else C=0;
    ADC      q, q, q             ; if (C) r -= d<<7;
    RSBS     t, d, r, LSR#6      ; q=(q<<1)+C;
    SUBCS    r, r, d, LSL#6      ; if ((r>>6)>=d) C=1; else C=0;
    ADC      q, q, q             ; if (C) r -= d<<6;
    RSBS     t, d, r, LSR#5      ; q=(q<<1)+C;
    SUBCS    r, r, d, LSL#5      ; if ((r>>5)>=d) C=1; else C=0;
    ADC      q, q, q             ; if (C) r -= d<<5;
    RSBS     t, d, r, LSR#4      ; q=(q<<1)+C;
    SUBCS    r, r, d, LSL#4      ; if ((r>>4)>=d) C=1; else C=0;
    ADC      q, q, q             ; if (C) r -= d<<4;
    RSBS     t, d, r, LSR#3      ; q=(q<<1)+C;
    SUBCS    r, r, d, LSL#3      ; calculate 4 quotient bits
    ADC      q, q, q             ; if ((r>>3)>=d) C=1; else C=0;
div_4bits
    RSBS     t, d, r, LSR#2      ; if (C) r -= d<<3;
    SUBCS    r, r, d, LSL#2      ; q=(q<<1)+C;
    ADC      q, q, q             ; calculate 3 quotient bits
div_3bits
    RSBS     t, d, r, LSR#1      ; if ((r>>2)>=d) C=1; else C=0;
    SUBCS    r, r, d, LSL#1      ; if (C) r -= d<<2;

```

```

        ADC      q, q, q           ; q=(q<<1)+C;
        RSBS     t, d, r, LSR#1    ; if ((r>>1)>=d) C=1; else C=0;
        SUBCS    r, r, d, LSL#1    ; if (C) r -= d<<1;
        ADC      q, q, q           ; q=(q<<1)+C;
        RSBS     t, d, r           ; if (r>=d) C=1; else C=0;
        SUBCS    r, r, d           ; if (C) r -= d;
        ADCS     q, q, q           ; q=(q<<1)+C; C=old q bit 31;
div_next
        BCS      div_loop          ; loop if more quotient bits
        MOV      r0, q             ; r0 = quotient; r1=remainder;
        MOV      pc, lr            ; return { r0, r1 } structure;
div_by_0
        MOV      r0, #-1
        MOV      r1, #-1
        MOV      pc, lr            ; return { -1, -1 } structure;

```

# Newton-Raphson

Une autre technique qui permet de calculer la division est d'utiliser la méthode de Newton\_Raphson.

Cette méthode permet de calculer itérativement la racine d'une équation  $f(x) = 0$

On suppose qu'on connaît une approximation  $x_n$  et on calcule une meilleure approximation avec la formule

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

# Newton-Raphson

On cherche à calculer  $n/d$ . On va calculer une approximation entière de  $n/d$  et on calculera

La fonction  $f(x) = d \cdot \frac{2^N}{x}$  possède l'approximation cherchée pour racine. L'itération du schéma de Newton-Raphson donne

$$x_{n+1} = x_n \cdot \frac{d \cdot 2^N = x_n}{2 \cdot x_n^2} = 2x_n \cdot \frac{dx^2}{2x_n^2}$$

# Racines carrées

La méthode de Newton-Raphson est très générale et permet de calculer toutes les fonctions classiques, racine carrée, exponentielle, sinus, cosinus, ....

Pour le calcul de la racine carrée, on peut aussi utiliser une stratégie d'essai et soustraction. Cette méthode est particulièrement efficace lorsque la précision demandée n'est pas trop grande (typiquement moins de 16 bits).

Si  $d$  est un nombre entier, le résultat du calcul de la racine carrée est deux entiers  $q$  et  $r$  tels que

$$d = q^2 + r; \quad 0 \leq r < 2q$$

# Racines carrées

Remarquez que si  $r > 2q$  alors  $q$  peut-être incrémenté.  
 $(q+1)^2 = q^2 + 2q + 1$

Supposons que  $d$  soit un entier sur 32 bits. Le résultat  $q$  est codé sur 16 bits et le reste  $r$  sur 17 bits.

On teste successivement les bits  $n = 15, 14, 13$ , etc de  $q$ . Le nouveau reste se calcule comme suit (initialement  $q = 0, r = d$ )

$$r_{\text{new}} = d - (q + 2^n)^2 = (d - q^2) - 2^{n+1}q - 2^{2n} = r_{\text{old}} - 2^n(2q + 2^n)$$

Donc, si  $r_{\text{new}} < 0$  on effectue la modification  
 $r \leftarrow r + 2^n(2q + 2^n)$   
 $q \leftarrow q + 2^n$

# Racines carrées

En C,

```
unsigned t, q=0, r=d;
do {
    N--;
    t=2*q+(1<<N);
    if ((r>>N) >= t) {
        r -= (t<<N);
        q += (1<<N);
    }
} while(N);
```

q et r contiennent le résultat

# Saturation

Pour prévenir un dépassement de capacité (overflow) pendant les calculs on fixe les valeurs de opérandes dans des limites déterminées, c'est la saturation.

saturation sur 16bits: la valeur de x est contenue dans l'intervalle  
-0x00008000 <-> +0x00007FFF

saturation sur 32 bits: -0x80000000 <-> +07FFFFFFF

Lorsqu'un calcul provoque un dépassement de capacité, le résultat est fixé à la valeur maximum (QADD, QDADD, QSUB, QDSUB)



# Saturation

Par exemple:

Arithmétique non saturée

$$0x7fffffff + 1 = 0x80000000 \text{ (résultat négatif)}$$

Arithmétique saturée

$$0x7fffffff + 1 = 0x7fffffff$$

# Microcontrôleur

Quelques éléments intégrés dans un microcontrôleur (en plus du processeur qui forme le cœur du circuit)

## **Périphériques pour la communication:**

- UART, Universal Asynchronous Receiver Transmitter (transmetteur/récepteur asynchrone universel)
- Ports série
- ..

## **Mesure et activateurs:**

- Des timers (horloges) pour les contraintes temps réel
- Des actuateurs pour commander des moteurs électriques, typiquement PWM Pulse Width Modulation (modulation de la largeur d'impulsion)

## **Ressources logiciel adaptées:**

- Système d'interruption performant, plusieurs niveaux de priorités
- Instructions conditionnelles
- Des instructions de manipulation des bits pour accéder aux périphériques

# Microcontrôleur

## **Convertisseur analogique/digital**

- Réalisation de systèmes de mesure

## **Jeu d'instruction spécialisé pour le traitement des signaux**

- Fonctions DSP (Digital Signal Processing)

En revanche, les systèmes dispose généralement de

- Peu de mémoire
- Pas de disque, etc.
- Performances modestes (pas toujours!) en MIPS (millions d'instructions par secondes)

# Systèmes embarqués

Un domaine d'application important des microcontrôleurs est le domaine de **systèmes embarqués**.

- Systèmes de freinage dans les véhicules
- Les téléphones portables
- Les assistants personnels
- Systèmes de mesure
- Caméras numériques
- Connexion Internet sans fils
- ...

Ces systèmes sont souvent des systèmes **Temps Réel**, c'est-à-dire des systèmes pour lesquels les contraintes temporelles sont importantes.

1. Les données sont traitées immédiatement et non en différé
2. Le temps de réponse du système à une sollicitation doit être fixé (au moins borné) et déterministe

<b>GBA</b>	<b>Function</b>
00h	SoftReset
01h	RegisterRamReset
02h	Halt
03h	Stop/Sleep
04h	IntrWait
05h	VBlankIntrWait
06h	Div
07h	DivArm
08h	Sqrt
09h	Arctan
0Ah	Arctan2
0Bh	CpuSet
0Ch	CpuFastSet
0Dh	GetBiosChecksum
0Eh	BgAffineSet
0Fh	ObjAffineSet
10h	BitUnPack
11h	LZ77UnCompWram
12h	LZ77UnCompVram
13h	HuffUnComp
14h	RLUnCompWram
15h	RLUnCompVram
16h	DifF8bitUnFilterWram
17h	DifF8bitUnFilterVram
18h	DifF16bitUnFilter
19h	SoundBias
1Ah	SoundDriverInit
1Bh	SoundDriverMode
1Ch	SoundDriverMain
1Dh	SoundDriverVSync
1Eh	SoundChannelClear
1Fh	MidiKey2Freq
20h	SoundWhatever0
21h	SoundWhatever1
22h	SoundWhatever2
23h	SoundWhatever3
24h	SoundWhatever4
25h	MultiBoot
26h	HardReset
27h	CustomHalt
28h	SoundDriverVSyncOff
29h	SoundDriverVSyncOn
2Ah	SoundGetJumpList

# Systèmes embarqués

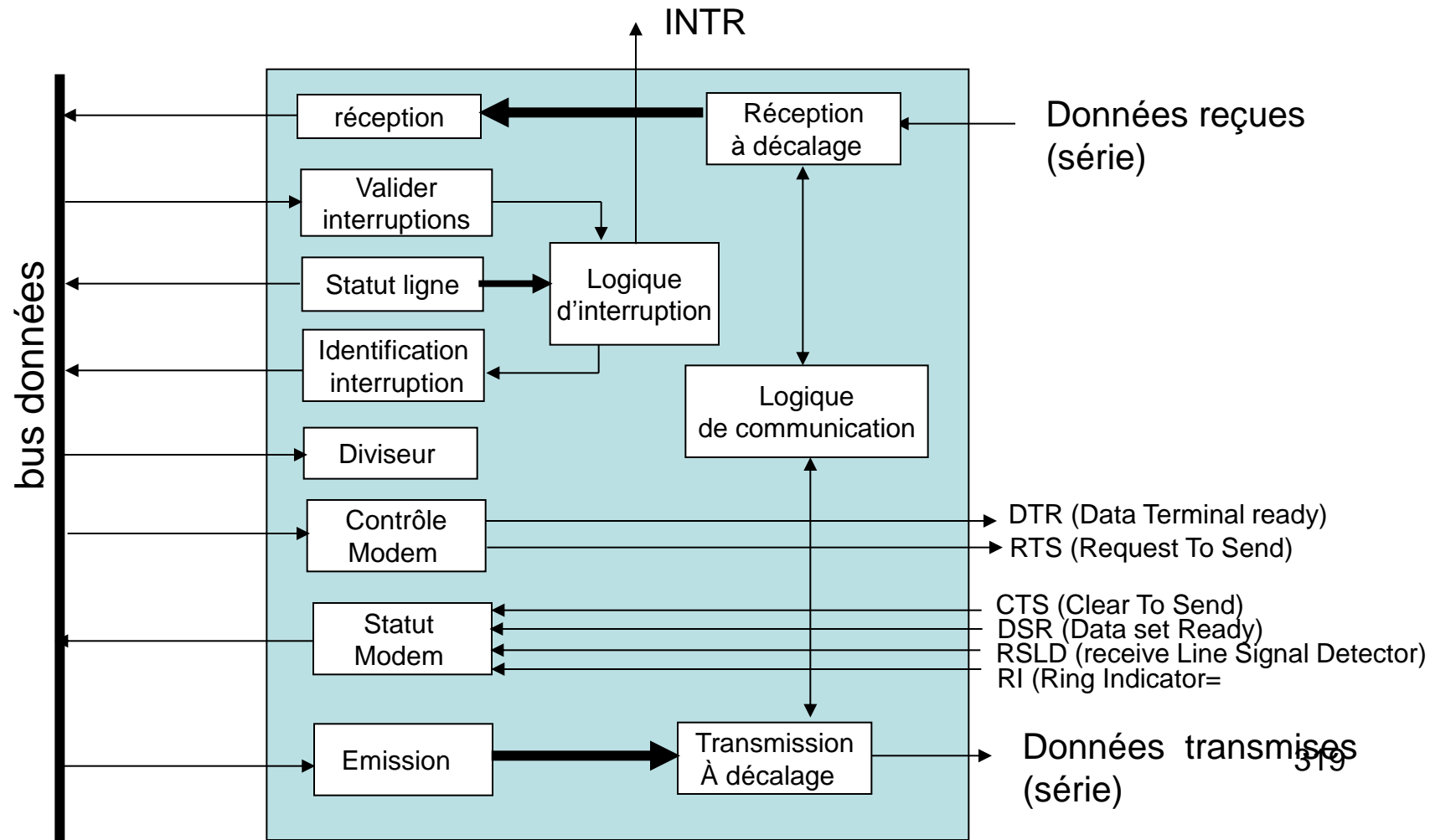
On peut caractériser ces systèmes par les contraintes qu'il doit satisfaire. En général, la puissance de calcul n'est pas le critère le plus important d'un tel système. Des critères classiques sont

- L'encombrement physique, d'où un besoin d'intégrer le plus de fonctions possible sur un même circuit
- La consommation d'énergie
- La tolérance aux pannes, développement de système fiable.

# Un périphérique classique

## UART (port série)

UART : Universal Asynchronous Receiver Transmitter



# UART

- Registre de réception: contient le dernier caractère reçu
- Le registre de validation des interruptions: permet d'indiquer si l'on fonctionne en mode interruption ou pas
- Le registre qui indique le statut (l'état) de la ligne : libre, occupée
- Le registre d'identification de l'interruption: contient un numéro d'interruption qui est fourni au processeur pour spécifier l'interruption à traiter (la routine à exécuter)
- Le registre diviseur: permet de définir la vitesse de transmission
- Le registre de contrôle du modem
- Le registre de statut du modem: positionné par le modem, permet au processeur de connaître l'état du périphérique
- Le registre d'émission: contient le prochain caractère à émettre

**Chaque registre est accédé par le processeur via le bus d'adresse (sélection) et via le bus de données (pour écrire/lire sa valeur)**



# Traitement d'une interruptions

Le traitement des interruptions dépend du processeur, on présente ici un exemple inspiré du processeur **Motorola 68'000**. Toujours utilisé dans certaine microcontrôleur 68376, 68332

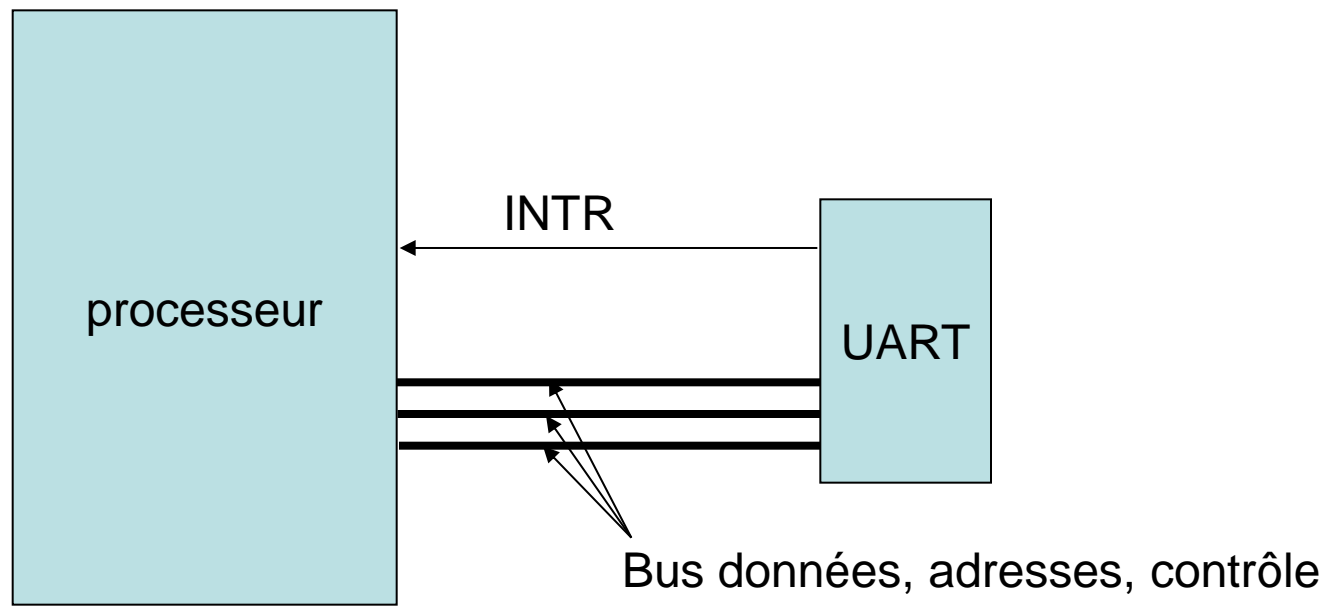
1. L'UART active les signaux INTR, en fait 3 signaux qui correspondent au niveau de priorité de l'interruption
2. Si le niveau courant du processeur est inférieur (et pas égal!) au niveau de l'interruption alors on traite l'interruption

## **C'est-à-dire.**

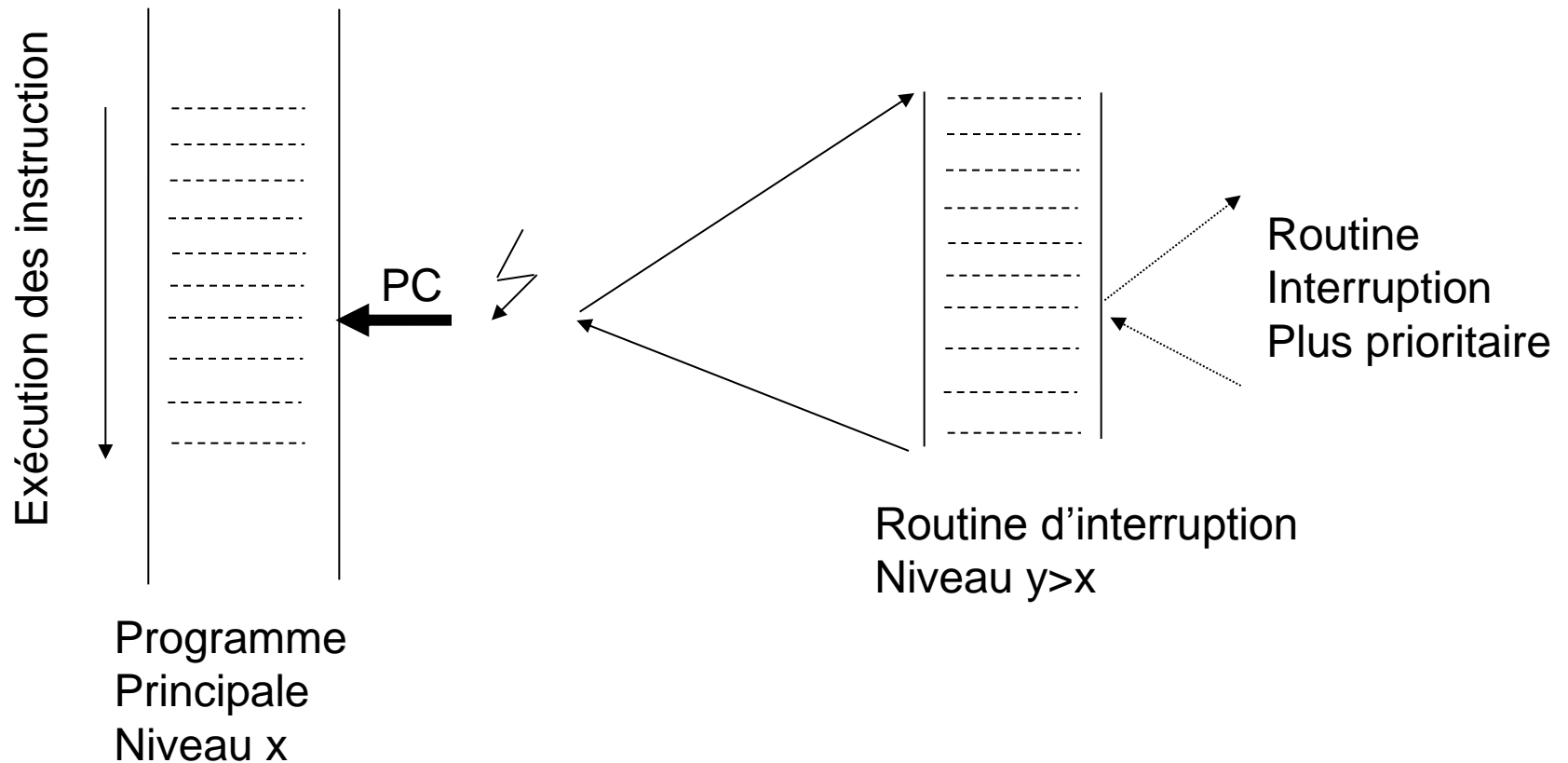
1. On termine l'exécution de l'instruction courante
2. On effectue un cycle de lecture pour 'lire' le **numéro** de l'interruption à traiter. Le numéro est fourni par l'UART et est contenu dans le registre d'identification des interruptions (préalablement définit par le processeur)
3. Le processeur exécute la routine qui correspond au numéro d'interruption (en utilisant **la table des interruptions**)
4. Reprend l'exécution du programme interrompu

# Interruptions

Le signal INTR est activé par l'UART pour indiquer au processeur qu'une interruption (matériel) doit être traitée. Par exemple, à la réception d'un caractère.



# Traitement d'une interruption



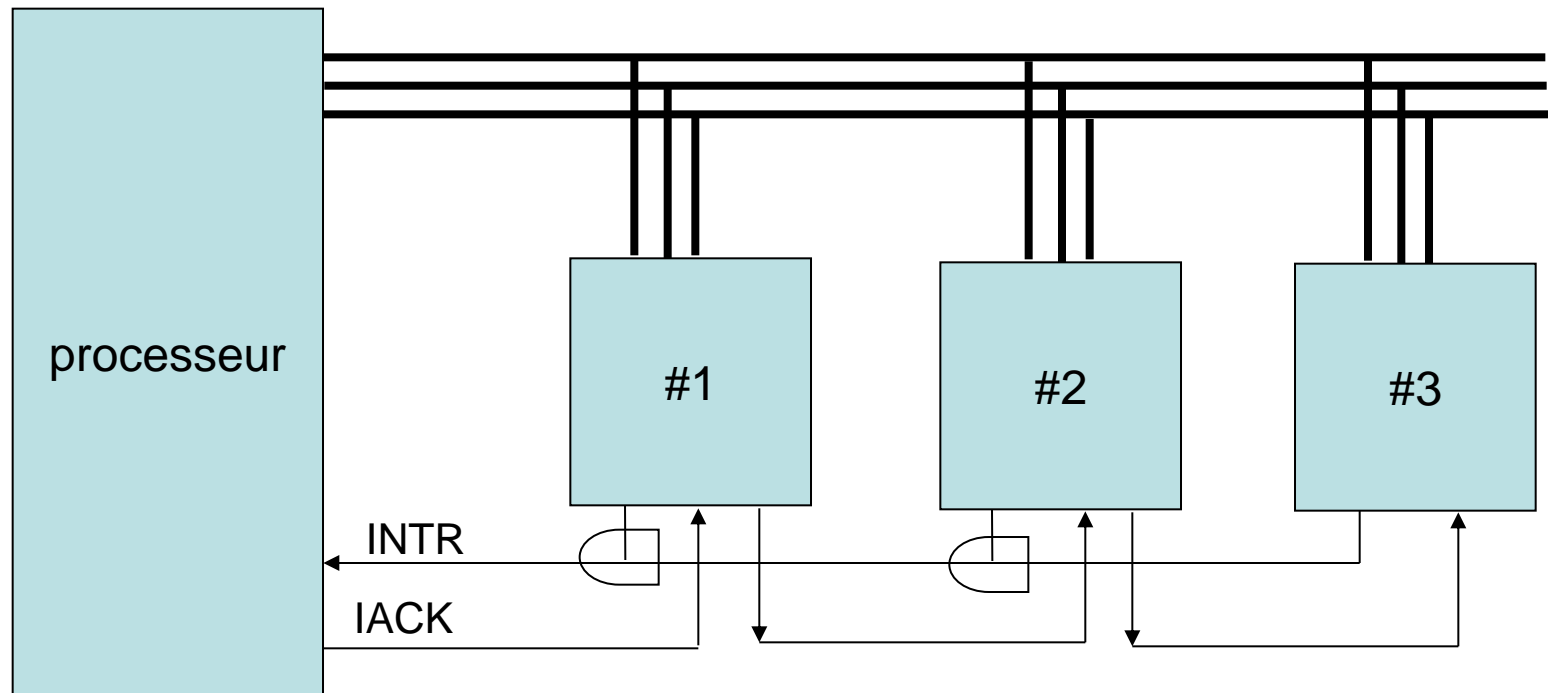
# Table (vecteur) d'interruption CPU32 (68'000)

Numéro de l'interruption	Adresse	Description
0	000	Reset (Stack Pointer)
1	004	Reset (PC)
2	008	Erreur bus
3	00C	Erreur adresse
4	010	Instruction illégale
5	014	Division par zéro
6	018	Instruction CHK
...	...	...
32	080	Instruction TRAP
...		
47		Instruction TRAP
48-63	0C0 – 0FF	Réservé
64	100	Interruption utilisateur
...	...	...
255	3FC	Interruption utilisateur

De la routine  
(le handler)  
d'interruption

# Daisy Chain

On utilise le Daisy Chain ou cascading pour cascader plusieurs circuits qui utilisent le même niveau d'interruption



IACK = Interrupt Acknowledge

# Daisy Chain

Pour un même niveau de priorité, les circuits sont ordonnés selon leur position dans la chaîne. Les circuits qui reçoivent le signal IACK du processeur peuvent décider

1. de fournir leur numéro d'interruption
2. De passer le signal IACK au prochain périphérique qui pourra fournir son propre numéro

# PIC

Un **PIC** (Programmable Interrupt Controller) ou **GIC** (Generic Interrupt Controller) est un circuit spécialisé qui permet d'interfacer le processeur avec les périphériques.

Le PIC implémente toutes les fonctionnalités qui permettent le fonctionnement efficace du système d'interruption: gère les priorités, fournit le numéro d'interruption, masquage des interruptions, acquittement des interruptions, virtualisation.

Souvent le circuit est accédé pour configuration par le processeur comme un périphérique.

# Interruptions

- Les sources d'interruption peuvent être soit matériels soit logicielles (exceptions)
- Les interruptions matériels sont **asynchrones** et permettent une réponse rapide du système à un événement
- Les interruptions logicielles sont **synchrones** et permettent par exemple de changer la priorité d'exécution du processus. Ce qui est utile pour exécuter certaines opérations en mode superviseur (OS). Les programme utilisateurs n'ayant pas la priorité suffisante pour les exécuter
- Une alternative au Daisy Chain est la **scrutation**, le processeur testant tous les registres d'états des périphériques pour trouver celui qui a initié l'interruption
- Une alternative au système d'interruption est la scrutation (polling) où le processeur passe son temps à tester si un événement a eu lieu.
- Le système d'interruption du 68'000 est **vectorisé**



# Les interruptions ARM7TDMI

Le système permet d'utiliser 7 interruptions différentes.

Interrupt/exception		Adresse	Adresse #2
Reset	RESET	0x00000000	0xFFFF0000
Undefined instruction	UNDEF	0x00000004	0xFFFF0004
Software Interrupt	SWI	0x00000008	0xFFFF0008
Prefetch abort	PABT	0x0000000C	0xFFFF000C
Data abort	Dabt	0x00000010	0xFFFF0010
Reserved	----	0x00000014	0xFFFF0014
Interrupt request	IRQ	0x00000018	0xFFFF0018
Fast interrupt request	FIQ	0x0000001C	0xFFFF001C

# Les interruptions ARM7TDMI

A l'adresse correspondant à une interruption se trouve une *instruction*.

Lorsqu'une interruption est levée, le processeur suspend l'exécution en cours, charge l'instruction qui se trouve à l'adresse correspondante et l'exécute.

Généralement, la table d'interruption contient une instruction de branchement qui permet d'exécuter une routine.

**Reset** : contient la première instruction exécutée après une initialisation (reset) du processeur, par exemple après mise sous tension.

**Undefined Instruction** : le processeur ne peut pas décoder une instruction

# Les interruptions ARM7TDMI

**Software interrupt** : le processeur a exécuté une instruction SWI, par exemple pour effectuer un appel à une routine du système d'exploitation.

**Prefetch abort** : le processeur essaye de charger une instruction à une adresse non permise.

**Data abort** : idem mais pour une donnée

**Interrupt resquet (fast)** : une interruption est levée par un composant matériel externe. Ces interruptions peuvent être masquée (voir le registre *cpsr*)

# Un programme pour illustrer

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

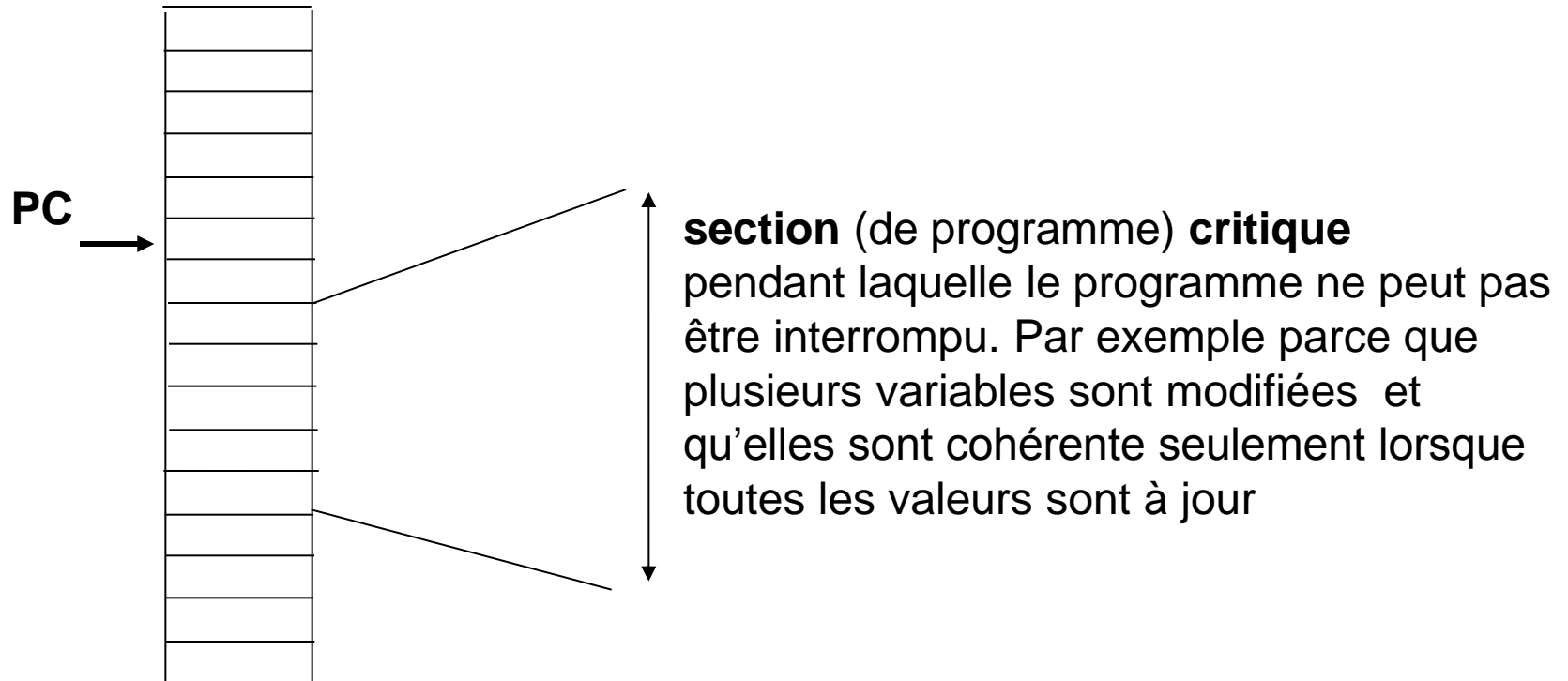
struct two_words { int a, b; } memory;

void handler(int signum) // la routine est appelée périodiquement toutes les secondes
{
    printf ("%d,%d\n", memory.a, memory.b);
    alarm (1);
}

int main (void)
{
    static struct two_words zeros = { 0, 0 }, ones = { 1, 1 };
    signal (SIGALRM, handler);
    memory = zeros;
    alarm (1);
    while (1)
    {
        memory = zeros;
        memory = ones;
    }
}
```

# Masquage des interruptions

Dans certaines situations il est important de masquer les interruptions, c'est-à-dire d'empêcher leur prise en compte par le processeur.



Un programme en mémoire

# Masquage des interruptions

- Pour les processeurs du type 68'000 on a la possibilité d'utiliser les différents niveaux de priorités pour masquer les interruptions. Le niveau 7 est non masquable. On peut se placer en niveau 7 pour masquer toutes les interruptions
- Certaines processeurs (Intel 80'186,...) possède un bit particulier dans le registre d'état qui permet d'activer/désactiver la prise en compte des interruptions par programme. Les instructions sont STI (Set Interrupt Flag) pour masquer les interruptions et CLI (Clear Interrupt Flag) pour les activer.

