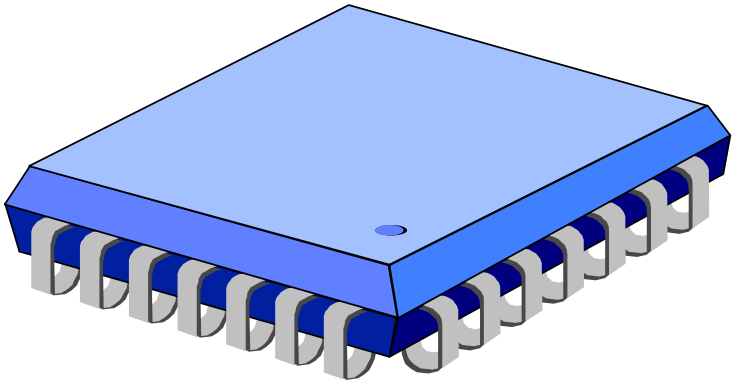
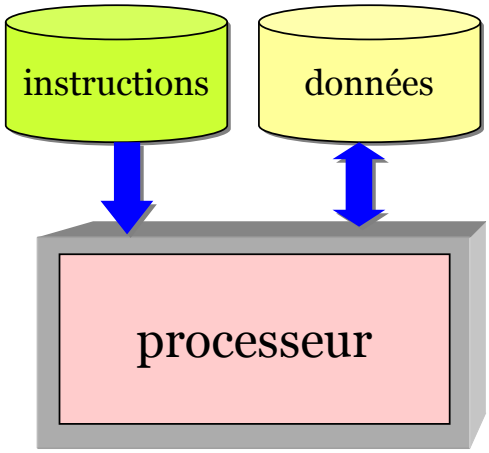
ELEC223

# Introduction aux architectures des microprocesseurs



# Traitement logiciel




*Intérêts :*

- **Flexibilité**
- **Temps de développement**

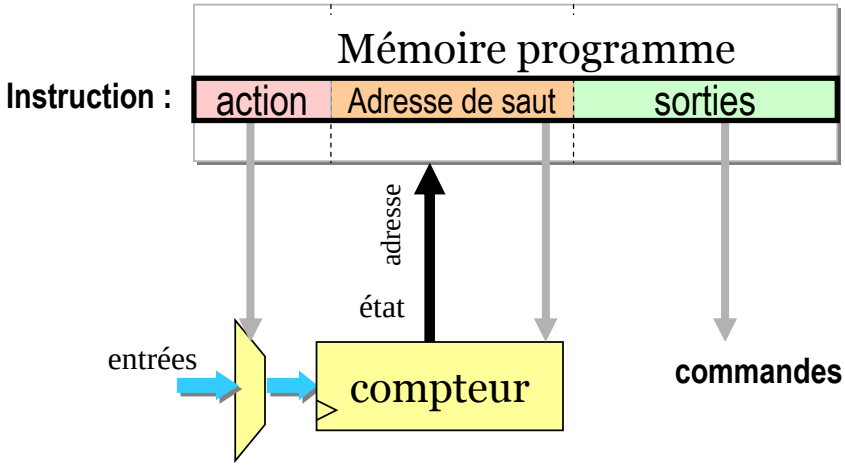


Les 2 types de traitement reposent avant tout sur des technologies **électroniques**.
Un traitement **logiciel** utilise un processeur alors qu'un traitement **matériel** utilise un contrôleur spécialisé.

Le traitement **logiciel** est par essence très flexible car il est facile de changer les instructions et leur séquence. Le génie logiciel permet des temps de développement très courts.

# Automate programmable

## Contrôle flexible des processus





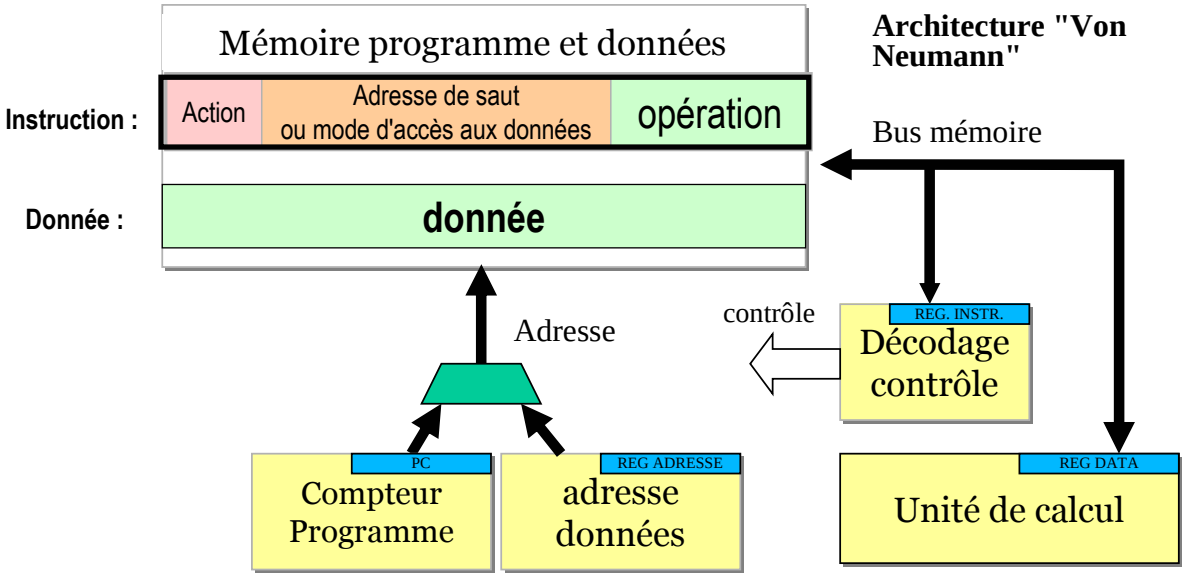

L'automate programmable ou le séquenceur est une machine à états de Moore générique dont le registre d'état est un compteur qui peut être incrémenté ou chargé par une nouvelle valeur. Ce compteur pointe dans une mémoire sur une « instruction » qui contient l'action à effectuer (+1 ou saut) en fonction des entrées et l'adresse de saut dans le cas d'un saut. L'instruction a également un champ pour les sorties servant à piloter des actionneurs extérieurs.

L'ensemble des instructions s'appelle un programme et le compteur s'appelle le compteur de programme ou PC. Cette architecture s'inspire de la fameuse machine de Turing où le ruban est remplacé par une mémoire électronique.

Cette architecture de séquenceur s'appelle aussi automate programmable. Elle est très utilisée en électromécanique pour piloter des robots industriels.

# Processeur

## Dédié au traitement de données

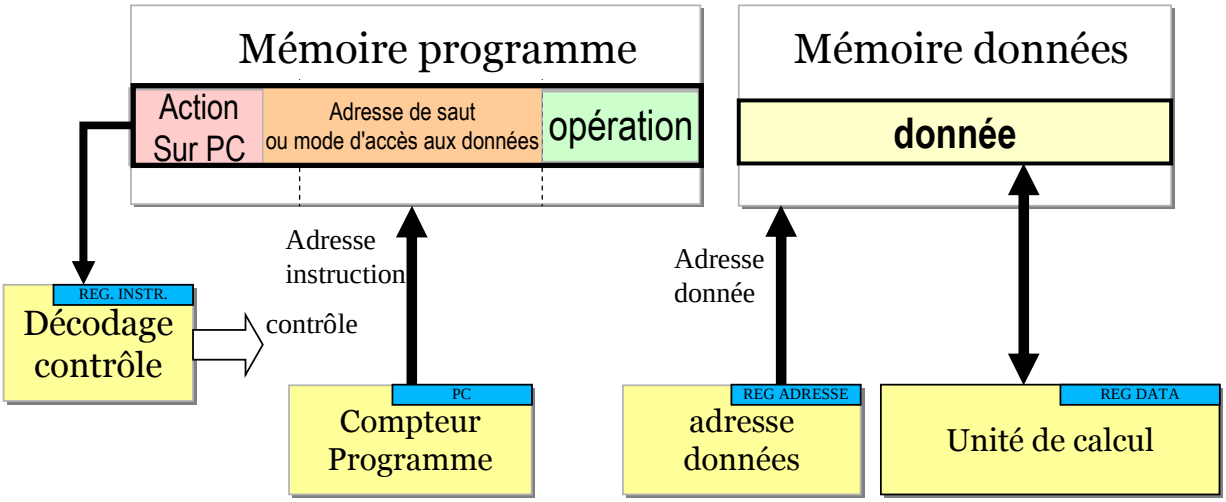





Le « processeur » est un automate dédié au traitement des données. Le champ des sorties (ou champ «opérations) vient alors commander l'unité opérative ou s'effectue les calculs. Dans l'architecture dite « Von Neuman » du nom d'un des pionniers ayant réalisé le premier ordinateur (l'ENIAC en 1945), les données sont stockées dans la même mémoire que les instructions. Dans le champ instruction, se trouve l'adresse de la donnée à traiter ou à stocker, dans le cas d'un opération, ou l'adresse de saut, dans le cas d'un branchement. Une unité de traitement spécifique à l'adresse est nécessaire pour que la mémoire soit adressée par le compteur de programme ou par l'adresse de la donnée.

# Architecture "Harvard"

Les mémoires programme et données sont séparées => accès et débits x2






L'architecture Von Neuman oblige à aller chercher successivement en mémoire l'instruction puis la donnée. En disposant de 2 mémoires respectives pour les instructions et les données, il devient possible de paralléliser les accès et donc d'augmenter les débits de calcul. C'est le but de l'architecture Harvard qui est utilisée dans tous les processeurs performants.

# Types de processeurs

- **Généralistes**
  - Calculs irréguliers
  - Souplesse
  - Puissance de calcul sous utilisée
- **Spécialistes**
  - Optimisés pour la puissance de calcul
  - Calculs réguliers
  - Unités d'execution dédiées
- **MicroContrôleurs**
  - Optimisés pour la faible complexité
  - Associés à des périphériques

Peuvent être embarqués dans les SoC

# Types d'instructions

+ Mode d'adressage des données

- **Traitement**
  - opérations arithmétiques et logiques

| OP | OP1 | OP2 |
|---|---|---|

- **Transferts des données avec la mémoire**
  - load, store

| LD | ADRESSE |
|---|---|

- **Contrôle**
  - Branchements
  - branchements aux sous programmes
    - Sauvegarde automatique de l'adresse de retour en « **PILE** »

| BR | ADRESSE |
|---|---|

- **Système**
  - Interruptions logicielles : Appel de fonctions de l '«operating system »

| SWI | |
|---|---|

- **Coprocesseur**
  - Instructions spécifiques à un opérateur extérieur « coprocesseur »

| COP | N° COP |
|---|---|



Les instructions servent à effectuer des opérations sur les données et à effectuer des sauts de programme conformément aux algorithmes de calcul. De façon à effectuer les calculs en interne et ne pas à aller chercher systématiquement les données en mémoire, les instructions LOAD et STORE servent respectivement à précharger des registres données et à sauvegarder les résultats en mémoire.

Certains processeurs disposent d'instructions facilitant la mise au point ou l'utilisation de système d'exploitation multi-tâches "OS" . C'est la cas des exceptions logicielles permettant au processeur de changer de mode. Par exemple les appels systèmes pour effectuer des processus du noyau de l'OS peuvent utiliser ce type d'instruction.

D'autre part les processeurs peuvent être aidés de coprocesseurs qui sont des circuits de calculs externes permettant d'accélérer le traitement. Pour éviter de faire de écritures/lectures avec le coprocesseur , le processeur communique avec lui par un canal de commandes dédié permettant au coprocesseur de se synchroniser et saisir à la volée les données à traiter ou à stocker

# Modes d’adressage

**exemples courants :**

| Mode | Accès aux données dans l'instruction |
|---|---|
| Immédiat | donnée elle même |
| Absolu | adresse de la donnée |
| Registre | numéro de registre |
| Registre Indirect | numéro de registre contenant l’adresse donnée (registre d'adresse) |
| Indirect | adresse d’une adresse donnée |



La donnée peut être directement indiquée dans l’instruction, ce qui correspond à l’adressage immédiat. Ce mode d’adressage présente les inconvénients suivants :

- Mot d’instruction long pour rajouter le champ donnée
- Un même programme ne peut pas être utiliser pour plusieurs listes de données.
- Une même liste de donnée ne peut pas être traitée par différents programmes

Pour les 2 derniers points, il est préférable d’indiquer l’adresse de la donnée dans l’instruction (mode d’adressage absolu). Pour réduire la taille de l’instruction, certains microprocesseurs considèrent la différence d’adresse qui les sépare avec le compteur programme de l’instruction (adressage relatif).

Si ce sont des registres internes qui contiennent les données, le mot instruction est encore réduit car il suffit de donner l’indice du registre.

Si la donnée est en mémoire, un registre peut servir de pointeur ou de registre d’adresse (mode d’adressage indirect). Dans ce cas l’instruction ne contient que l’indice de ce registre d’adresse et le processeur va effectuer un cycle supplémentaire pour accéder à la donnée en mémoire adressée par ce registre d’adresse.

# Modes de fonctionnement

- **Chaque mode correspond à des ressources propres**
  - permet la mise en place de protection pour les OS
  - Il existe au moins 2 modes : Superviseur et Utilisateur
  - Les ressources propres sont généralement des registres : PC, pointeur de pile
  - Le passage d'un mode à un autres s'effectue par les exceptions (interruption logicielle ou matérielle)



De façon à pouvoir exécuter des programmes de supervision ou systèmes d'exploitation multi-tâches et multi-utilisateurs, certains processeurs ont différents modes de fonctionnement.

Ces modes permettent de délimiter les ressources de façon à ce qu'un utilisateur ne puisse bloquer le système. Le passage d'un mode à l'autre s'effectue par un mécanisme d'exceptions qui est généré soit par une instruction, soit par une entrée du microprocesseur (interruption, reset, alerte,…)

# Exceptions

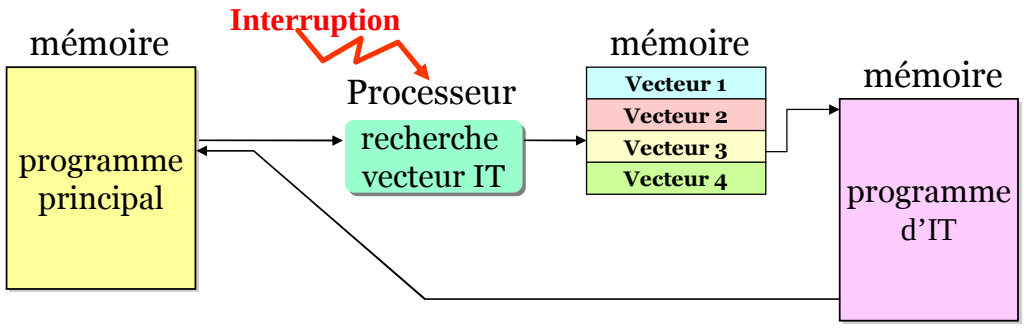
- **Permettent de signaler un événement particulier au processeur pour qu'il effectue un traitement spécifique.**
- **Elles peuvent être générées par :**
  - interruption logicielle (instruction d'appel système)
  - interruption matérielle (un signal d'interruption extérieur devient actif)
  - dysfonctionnement du système (échec d'un accès mémoire, instruction non reconnue,…)



Les exceptions permettent d'interagir aux évènement extérieurs de façon à dérouter le programme vers un traitement plus propice. Par exemple , les programmes d'entré/sortie peuvent nécessiter le passage  en mode "superviseur" par le biais d'un appel système qui effectuer une fonction sécurisée par le noyau du système. De même un système de surveillance de cuve peut réagir s'il reçoit une "**interruption**" de l'extérieur.

# Traitement des exceptions

- Une exception donne lieu à l ’exécution d’un programme de traitement pointé par un vecteur.






Lors de la réception d'un exception (une interruption dans cet exemple), l'exécution du programme principal est interrompu par le processeur qui va sauvegarder la valeur du PC (adresse de retour) de façon à revenir au programme principal.

Le processeur doit alors déterminer le générateur de l'exception et scruter en conséquence la table des vecteurs d'exception de façon à connaître l'adresse du programme de traitement En début de ce programme, une sauvegarde des registres peut être faite comme lors d'un appel à un sous–programme. La sauvegarde est effectuée dans une zone mémoire particulière : la "pile" de façon à ne pas écraser le contexte des registres du programme principal. Ce contexte est ensuite "dépilé" avant le retour au programme principal qui s'effectue en rechargeant la valeur du PC avec l'adresse de retour.

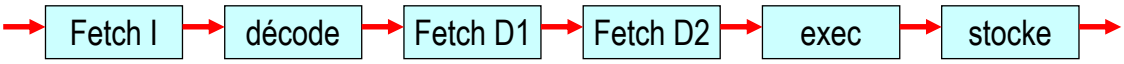
Les interruptions pouvant être nombreuses, un processeur est  associé à un contrôleur d' interruptions qui permet d'affecter des priorités en cas de simultanéité et de savoir quel élément est le générateur.

# Traitement des exceptions

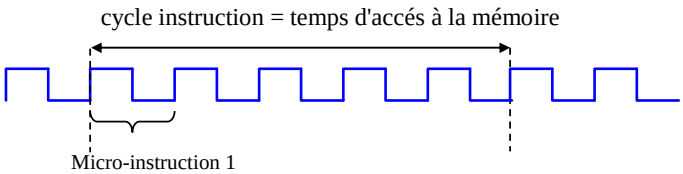
- **Les vecteurs d'exception sont regroupés dans une table située à un endroit spécifique de la mémoire.**
- **Dans le cas d'une interruption ils peuvent être communiqués par l'élément générateur**
- **Un contrôleur d'interruption peut être utilisé pour**
  - Gérer les priorités en cas de simultanéité
  - Masquer certaines sources

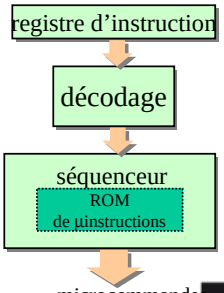
# Le passé : Processeur CISC

- L'accès à la mémoire extérieure est relativement lent
- L'exécution d'une instruction nécessite plusieurs étapes




▪ Idée : Utiliser le temps d'attente mémoire pour exécuter des instructions compliquées avec des modes d'adressage variés




Partie contrôle 70% (sous utilisée)
Partie opérative 30%



Dans les années 1970, après la naissance des premiers microprocesseurs, les accès mémoires étaient relativement lents et le processeur disposait de plusieurs cycles d'horloge (au moins 8 pour le 68000) pour aller chercher, décoder et exécuter l'instruction. Ce nombre de cycles permettait d'effectuer des calculs micro-séquencés qui donnait lieu à des instructions très compliquées mais peu souvent utilisées, ce qui justifie le nom de ce type de processeur : "Complex Instruction Set computer".

# Processeur RISC

- **Utiliser des registres, de la mémoire locale rapide ou « cache » pour éviter d'attendre**
- **Paralléliser les instructions : techniques pipeline**
- **Instructions simples au format fixe**
- **2 instructions pour l'accès à la mémoire : LOAD STORE**

Dans les années 80, cette approche devint possible grâce aux progrès technologiques . Le gain en performance doit tendre vers « une instruction en 1 cycle d'horloge ».



Dans la fin des années 1980, il est devenu plus facile d'intégrer de la mémoire rapide sous forme de bancs de registre, mémoire locale ou mémoire cache au sein même du processeur. Le micro-séquencement des instructions ne se justifiait plus mais les instructions devait être assez simples pour être exécutées le plus vite possible, c'est à dire en 1 cycle. Le concept "Reduced Instruction Set Computer" était né.

Il n'en demeure pas moins qu'il faut toujours les mêmes étapes pour exécuter une instruction RISC :

• Aller la chercher en mémoire

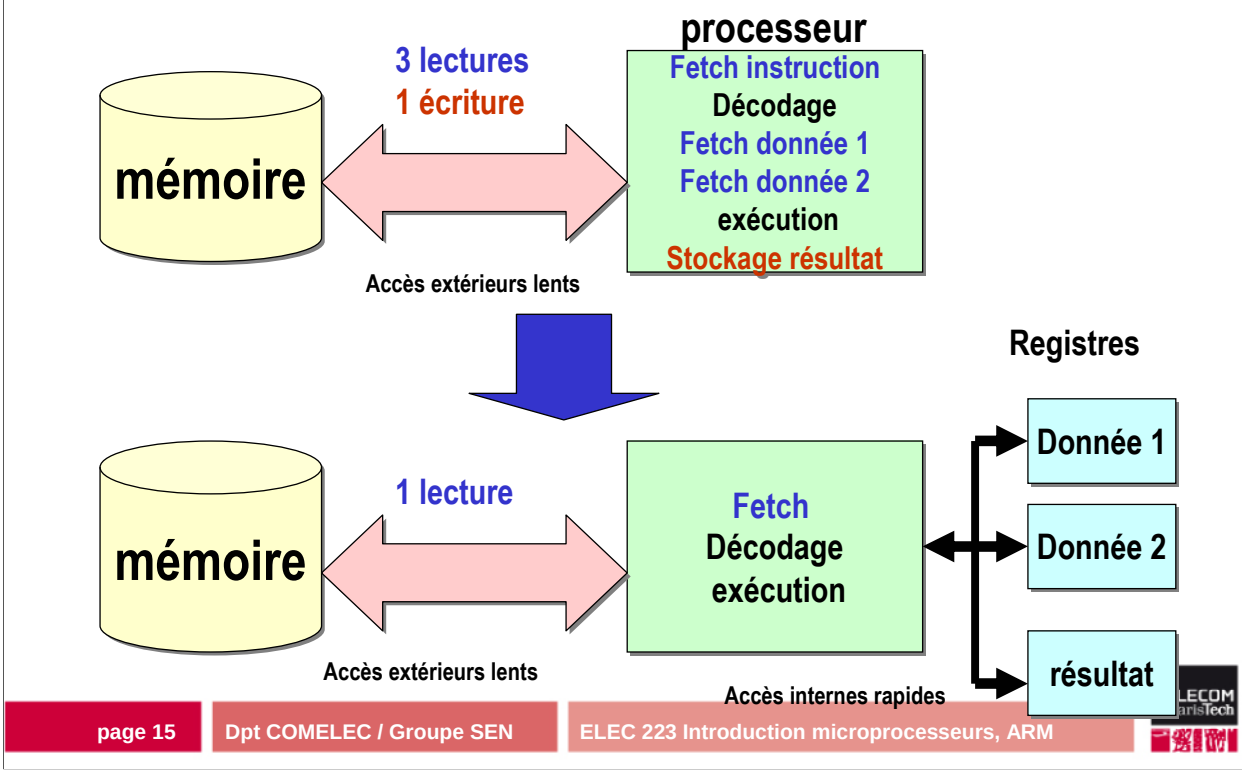
• La décoder

• L'exécuter

Un façon simple de "paralléliser" ces étapes est de les mettre en pipeline comme il va être vu par la suite.

D'autre part les calculs effectués sur des registres permettent de simplifier l'architecture et de faciliter le respect des performances. Pour charger ces registres ou sauver leur valeurs en mémoire, il suffit d'avoir respectivement 2 instructions simples LOAD et STORE.

# Utilisation de registres locaux



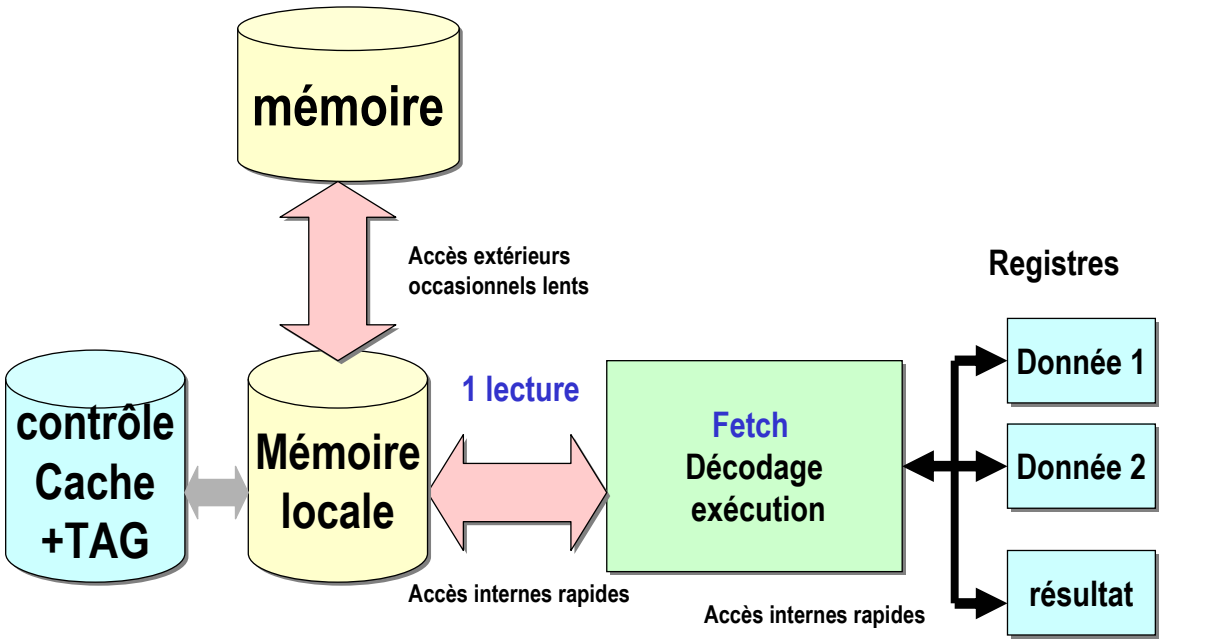



Dans l'exemple du haut, les données sont stockées en mémoire extérieure avec le programme (Von Neuman). Il faut 3 cycles de lecture (1 instruction et 2 données) et 1 cycle d'écriture (résultat).

Dans l'exemple du bas , les données sont préalablement stockées dans les registres. Dans ce cas il suffit d'une lecture de l'instruction.

Bien sûr les registres doivent être préalablement chargés ou lus avec des LOAD et des STORE, mais si on considère des opérations chaînées et un grand nombre de registres, il y a une perte de temps faible pour préparer le calcul.

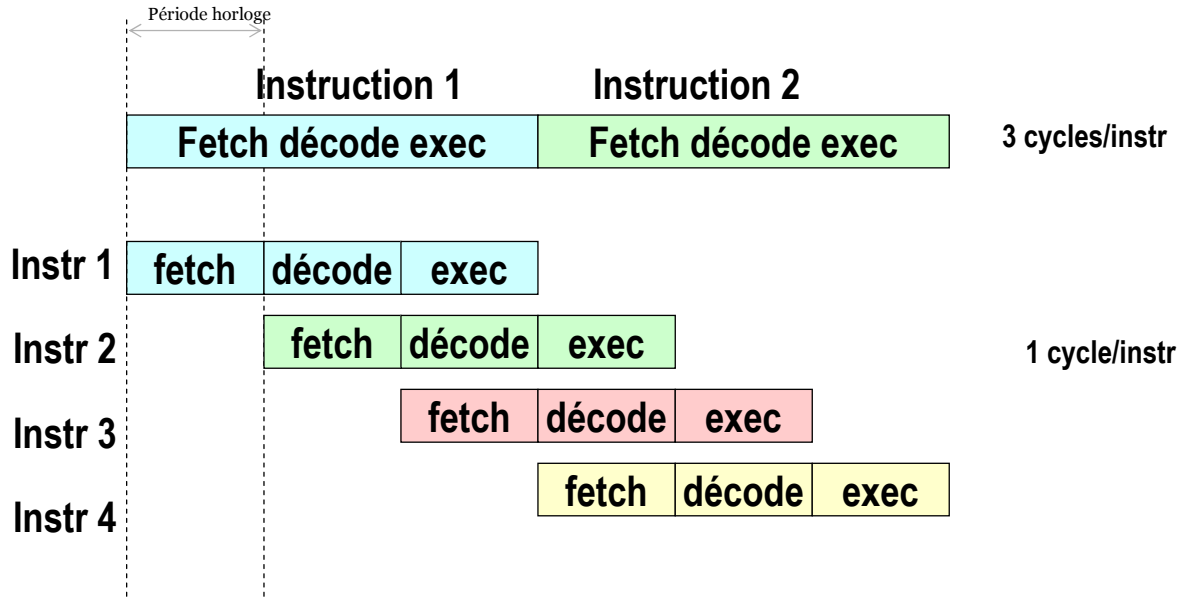
# Utilisation de mémoire locale






Les accès à la mémoire extérieure peuvent être accélérés avec l'utilisation d'une mémoire cache qui est par essence rapide et n'introduit pas de cycles d'attente.

# Utilisation du pipeline






Le pipeline d'instruction permet d'exécuter une nouvelle instruction à chaque cycle d'horloge.

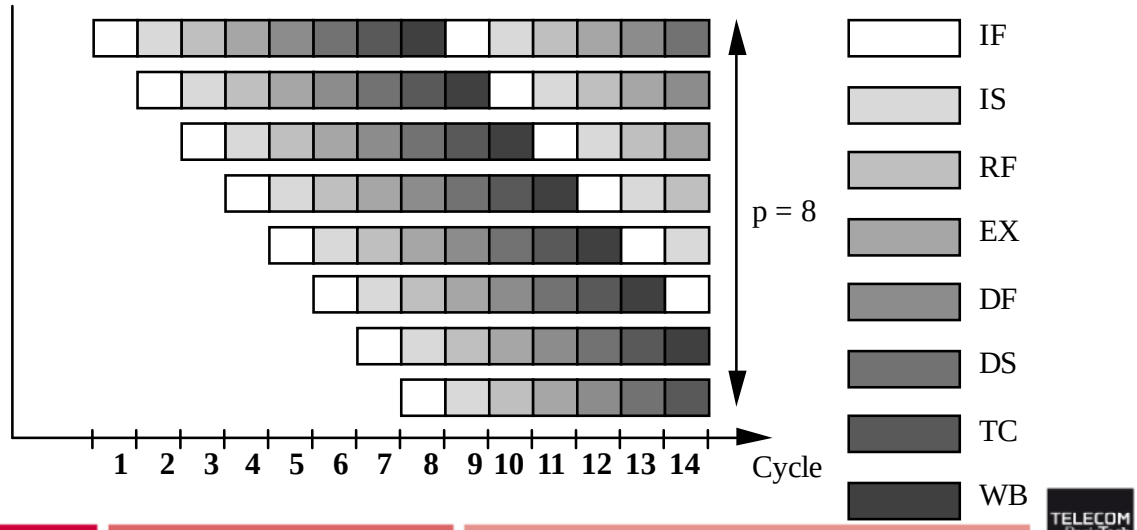
Dans cet exemple, il y a 3 étapes : **Fetch**, **Decode** et **Execute**.

Lors de l'étape 3, l'instruction 3 est dans la phase Fetch, alors que l'instruction 2 est décodée et l'instruction 1 est exécutée. Au cycle d'après c'est l'instruction 2 qui sera exécutée.

Les instruction ont toujours besoin de 3 étapes mais le débit d'instruction est au rythme de l'horloge.

# Architectures Super-pipeline

• Très grandes fréquences d'horloge



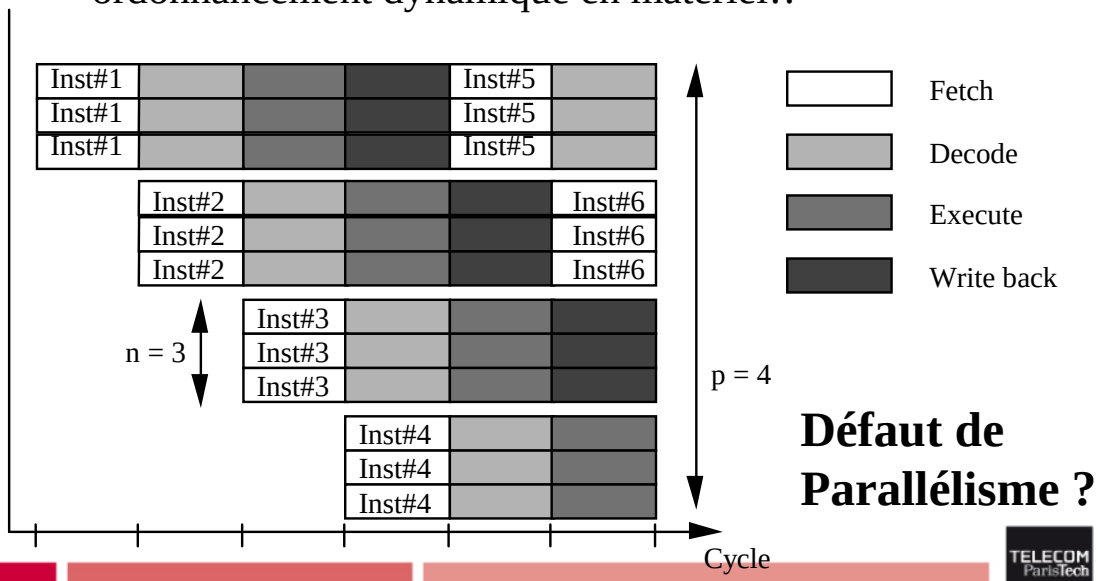



Certains processeurs comme le pentium ont de nombreux niveaux de pipeline (>20). Cette technique "superpipeline" permet de diminuer les temps critiques et donc d'augmenter la fréquence de fonctionnement.

En revanche comme nous allons le voir par la suite, la rupture du pipeline est pénalisante.

# Architectures Super-scalaires

- n unités d'exécution
- n instructions délivrées par cycle
- ordonnancement dynamique en matériel!!



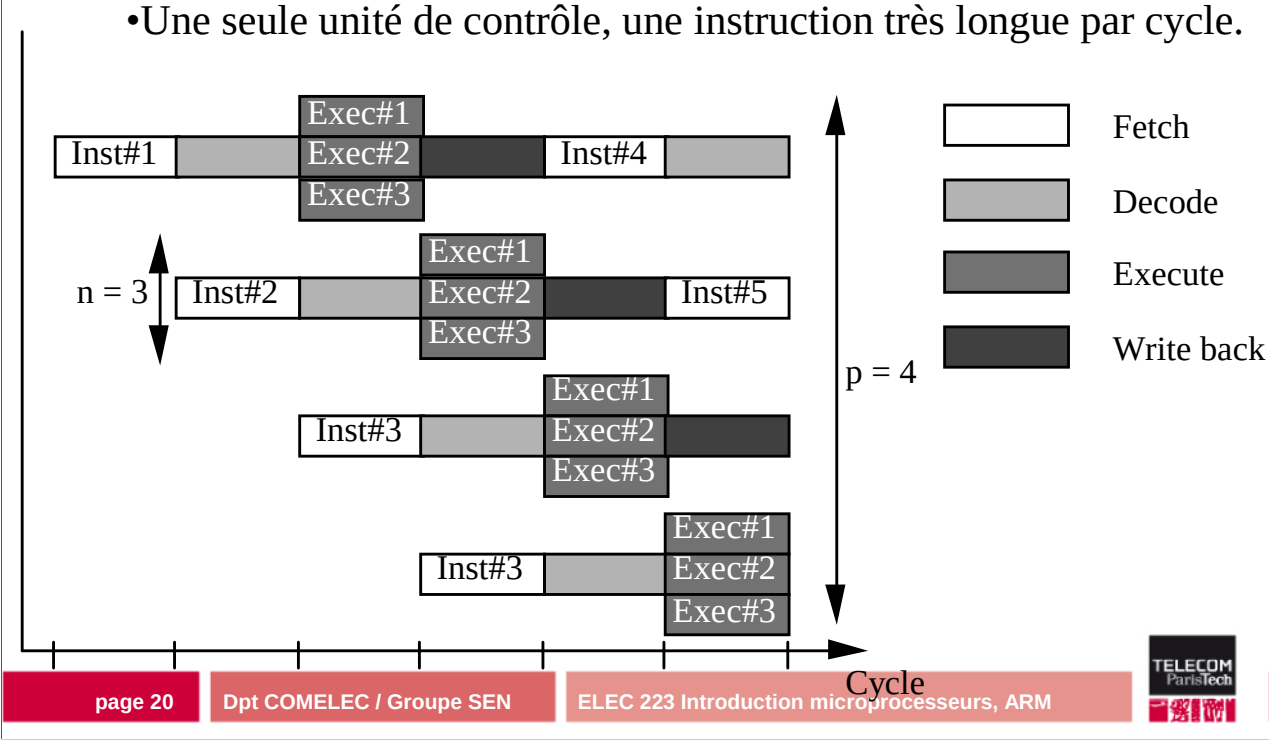

**Défaut de Parallélisme ?**



Cette technique vise à mettre en parallèle plusieurs processeurs en parallèle . Cette technique "superscalaire" repose sur une unité interne d'ordonnancement dynamique des instructions de façon à profiter pleinement du parallélisme.

## VLIW

•Une seule unité de contrôle, une instruction très longue par cycle.






Une autre technique consiste à disposer  d'unités d'exécution en parallèle mais d'agencer les instructions respectives au niveau du compilateur de telle sorte qu'une longue instruction correspondant à plusieurs instructions concaténées soit présentée au processeur. Cet agencement statique à la compilation permet en théorie d'atteindre des performances élevées par rapport aux techniques "superscalaires". Cependant l'écriture de tels compilateurs demeure très difficile et les résultats restent perfectibles, à moins d'utiliser des fonctions optimisées écrites en langage machine.

# Architecture VLIW

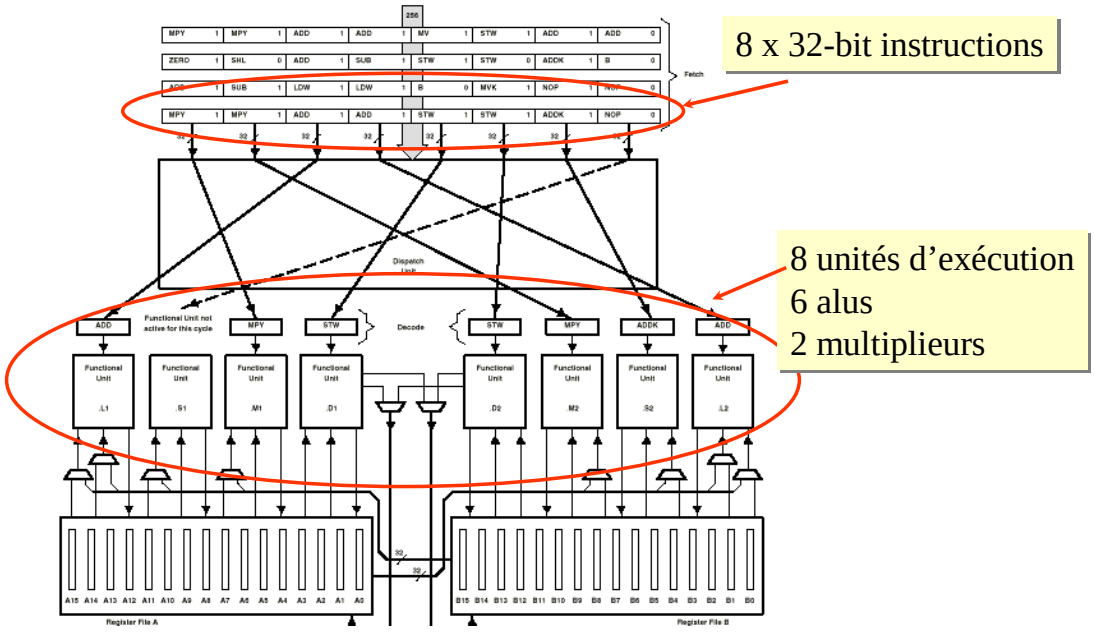
- **Formats fixes de l'instruction (champs indépendants)**
- **Ordonnancement statique à la compilation**
- **Les + :**
  - densité d'intégration accrue : surface de silicium difficile à exploiter avec le superpipeline/scalaire
  - complexité "exponentielle" du contrôle dynamique de grands pipeline et/ou d'un parallélisme instruction important
  - croissance des applications de type DSP.



La technique VLIW est très utilisées dans les processeurs de traitement du signal car elle permet de réduire la surface silicium, donc le coût, et offre des performances très rapides si on dispose de bibliothèques écrites en assembleur pour pouvoir profiter du parallélisme.

Les processeurs généralistes (pentium, AMD) sont soumis à des fortes contraintes de compatibilité et de généricité permettant difficilement ce type d'approche.

# Exemple : TMS320C6201



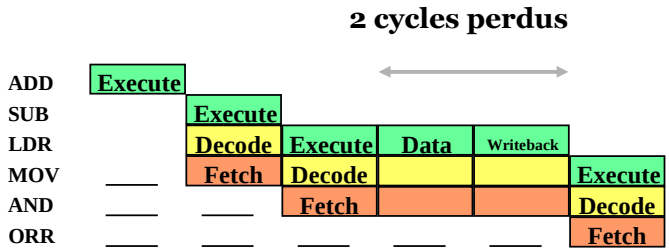



Le DSP C6X de Texas Instruments sont les précurseurs des architectures VLIW. Ils présentent 8 unités de calculs en parallèle.

# Problème : rupture du pipeline

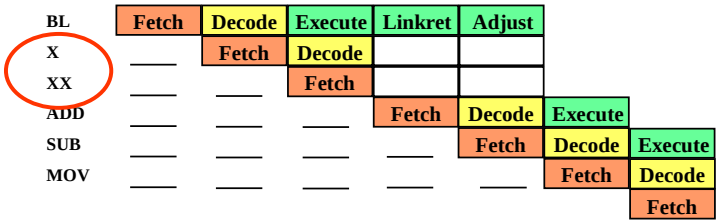
**2 causes principales :**

Accès mémoire extérieure




branchements

**2 instructions perdues**




**Solutions :** branchement retardé
prédiction de branchement



Le pipeline peut rompre pour différentes raisons. Voici quelques cas importants :

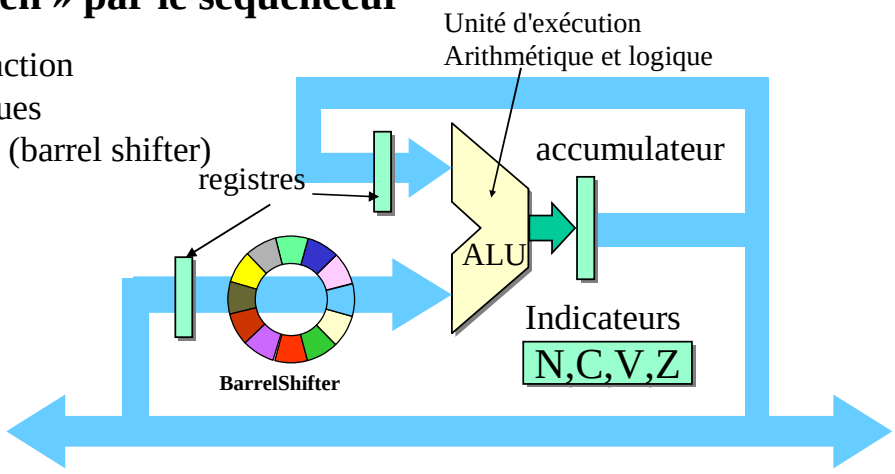
L'exécution d'un LOAD ou STORE : Dans une architecture Von Neuman, il faut un cycle pour présenter l'adresse de la donnée de façon à ce que celle-ci soit transférée du processeur vers la mémoire. D'autre part il faut 1 cycle en plus car le temps entre la mémoire et le registre est trop long. Plutôt que de réduire la fréquence d'horloge, il est préférable de rajouter un registre tampon et donc 1 cycle .

L'exécution d'un branchement : Le branchement s'effectue à l'issue du cycle d'exécution. Les N-1 instructions (N étant le nombre d'étages de pipeline), qui suivent le branchement ne sont pas exécutés si le branchement a lieu. Certains processeurs utilisent l'unité d'exécution pour effectuer des opérations de sauvegarde de l'adresse de retour dans un registre, dans le cas d'un appel à un sous programme.

# Partie opérative

## « Control Driven » par le séquenceur

- Addition-soustraction
- opérateurs logiques
- Décalages de bit (barrel shifter)




**opérateurs plus spécifiques :**

- Multiplication-accumulation (DSP)
- ALU + Multiplication en réel
- Calculs vectoriels



L'architecture classique de l'Unité arithmétique et logique UAL contient un registre accumulateur qui sert lui-même d'entrée à l'UAL pour la prochaine opération. Les microprocesseurs RISC ont la possibilité d'utiliser un registre quelconque comme accumulateur. D'autre part un des opérandes peut passer préalablement dans un "barrelshifter" avant de rentrer sur l'UAL. Cet opérateur effectue une rotation sur les bits de l'opérande, ce qui permet des multiplications ou divisions par des puissances de 2.

En sortie de l'UAL des indicateurs indiquent une propriété du résultat. Les indicateurs les plus courants sont :

- Z = résultat à 0
- N= 1 résultat négatif
- V = débordement
- C = retenu sortante

Les indicateurs sont utilisés par le processeur pour conditionner les instructions de saut.

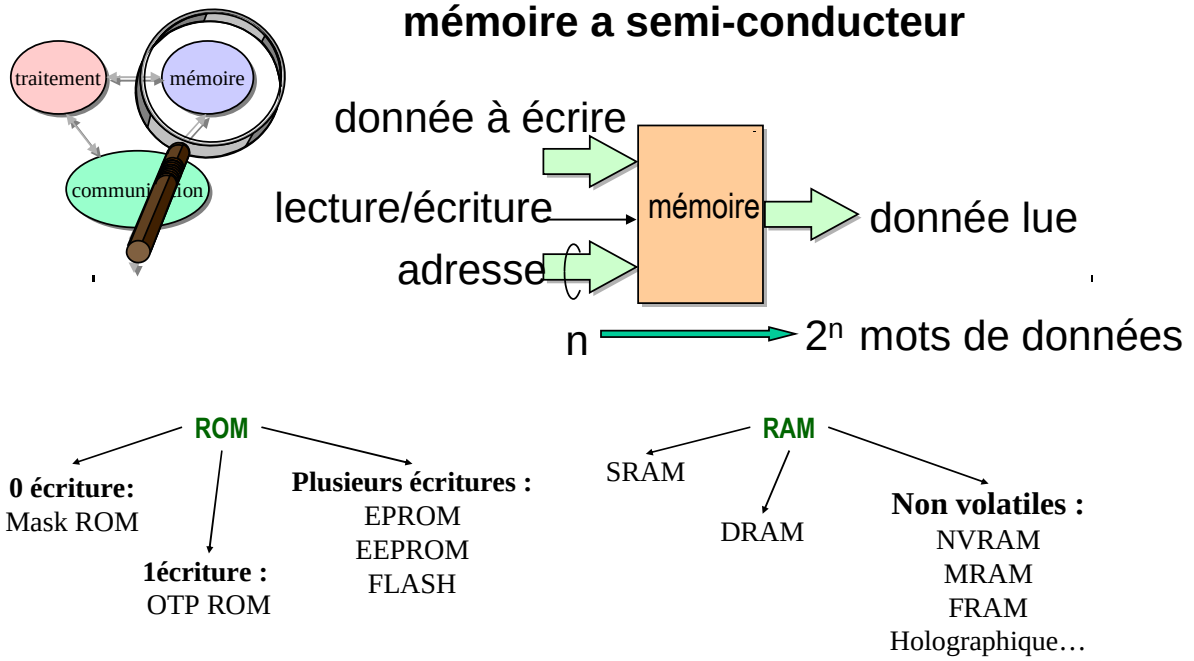
# Coprocesseurs

- **Adaptation à l'OS**
  - Gestion de la mémoire virtuelle (MMU) ayant 2 fonctions :
    - Translation adresse virtuelle en adresse physique
    - Contrôle des permissions d'accès
- **Augmentation des performances d'accès**
  - Contrôleur Cache
- **Calcul en représentation flottante**
- **Interface externe pour la personnalisation du coprocesseur**

# Performances des processeurs

- **Très nombreux facteurs**
  - Choix du langage (assembleur,C,java,…)
  - Style de codage
  - Compilation
  - Jeu d'instruction (8/16/32/64 bits)
  - Opérateurs disponibles
  - Nombre de période d'horloge /instruction (CPI)
  - Fréquence d'horloge
  - Cache
  - Vitesse de la mémoire centrale

# Mémoire





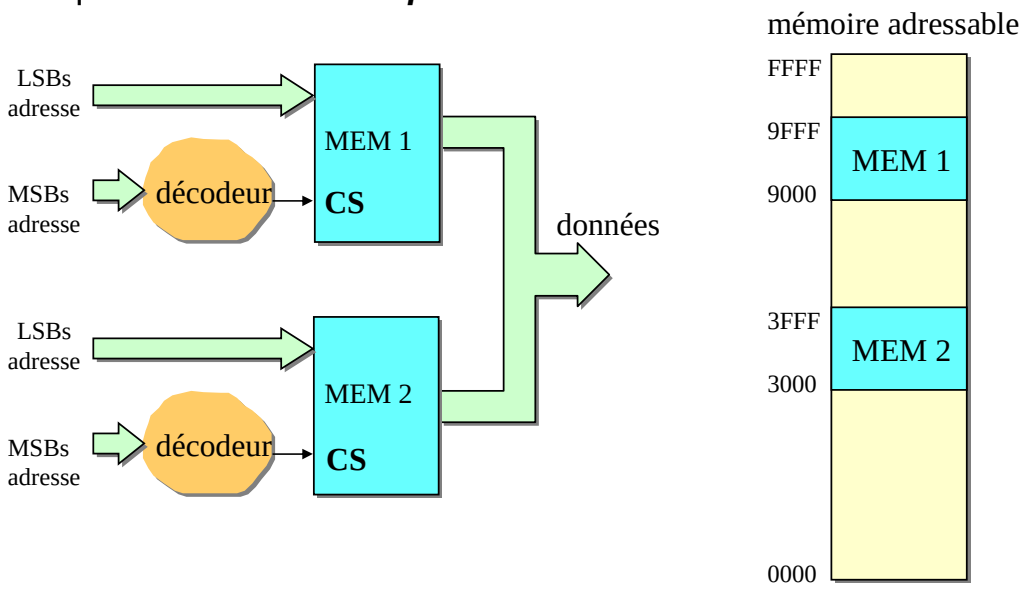

La mémoire électronique stocke 2^n mots de m bits. Un seul mot est accédé à la fois. Chaque mot est donc associé à une adresse. La mémoire agit comme une fonction combinatoire sur les bits d'adresse en lecture. En écriture il faut de plus activer le signal de commande d'écriture et lui fournir la donnée à écrire.

Les mémoires récentes sont synchrones et ont une horloge en entrée. Cela simplifie beaucoup le contrôle qui doit juste satisfaire la contrainte du chemin critique par rapport à la période d'horloge : Tcrit < Th

# Mappe mémoire

indique les champs d'adresse mémoire dans un système

Chaque boîtier a son ***"Chip Select"***





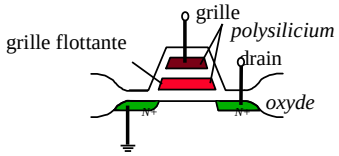

Un système électronique utilisant un champ mémoire, comme un microprocesseur, doit avoir une mappe mémoire correspondant aux zones d'adresse des différents éléments.

Une façon de dissocier 2 éléments, comme par exemple des boîtiers mémoire, est de piloter différemment un signal d'entrée souvent appelé "chip select" CS. Ce CS fait l'objet d'un décodage sur les bits de fort poids MSBs de l'adresse alors que le boîtier reçoit les bits de faible poids.

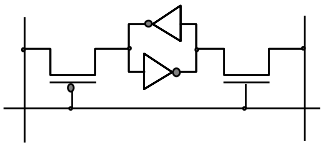
Si on considère un système disposant de 16 bits d'adresse, la mappe mémoire se situe entre 0x0000 et OxFFFF. Si on veut qu'une mémoire de 4K octets (donc 12 bits d'adresse) soit située dans la zone 0x9000 à 0x9FFF, Le CS doit correspondre au décodage des 4 MSBs à la valeur 9.

# Principales technologies silicium

**EEPROM, FLASH :** basée sur les transistors à grille flottante




**SRAM : CMOS**



**DRAM :** capacité grille d'un transistor MOS
=> nécessité de rafraîchissement mais capacité x4



Les principales technologies sont la mémoire **ROM** ou plus récemment **FLASH** , la RAM statique **SRAM** et la ram dynamique **DRAM**.

La mémoire FLASH utilise des transistors double grille avec grille flottante. Les charges injectées dans la grille flottante sont capturées et peuvent rester quelques années. La mémoire est dite non volatile.

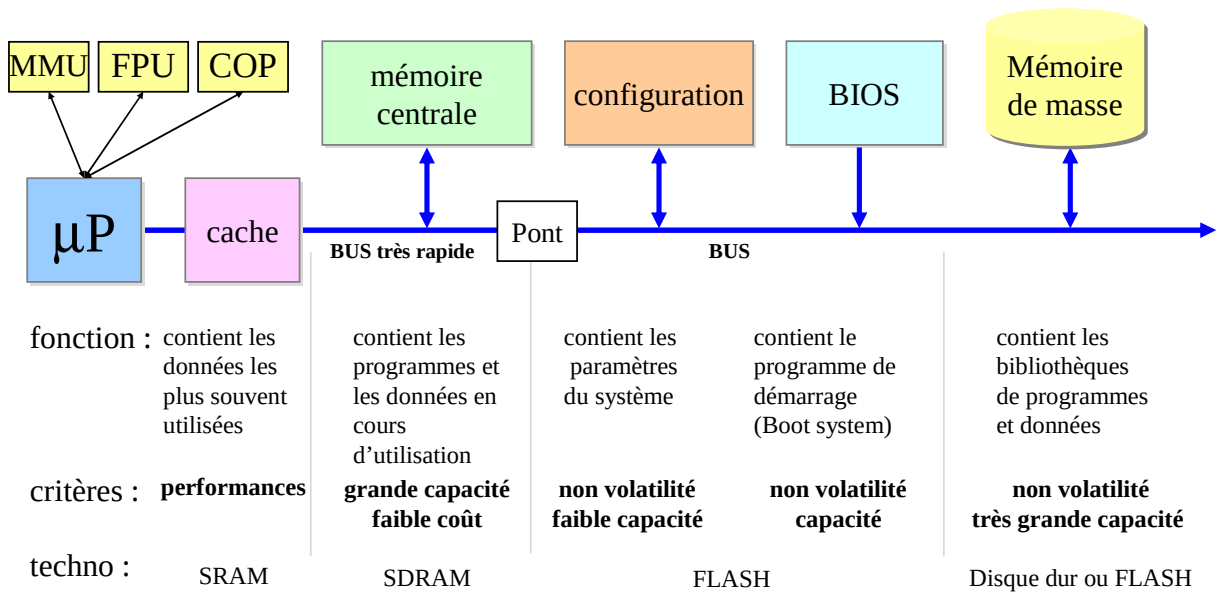
La mémoire SRAM repose sur un système stable assuré avec 2 inverseurs rebouclés sur eux-mêmes.  2 transistors de commande de part et d'autre permettent l'écriture et la lecture.

La mémoire DRAM est la capacité grille d'un transistor MOS. Cette petite capacité se décharge assez vite et un système de rafraîchissement est nécessaire.

Les mémoires SRAM et DRAM sont volatiles. La mémoire DRAM est optimale en terme de taille (1 point=1 transistor) mais est plus lente.

De gros progrès ont été fait sur les interfaces des mémoires de façon à accélérer les débits. Par exemple la mémoire FLASH est écrite par secteurs de données. De même la mémoire DRAM, assez lente en accès aléatoire devient rapide si toute une rangée est préchargée dans une mémoire rapide et accédée en rafale.

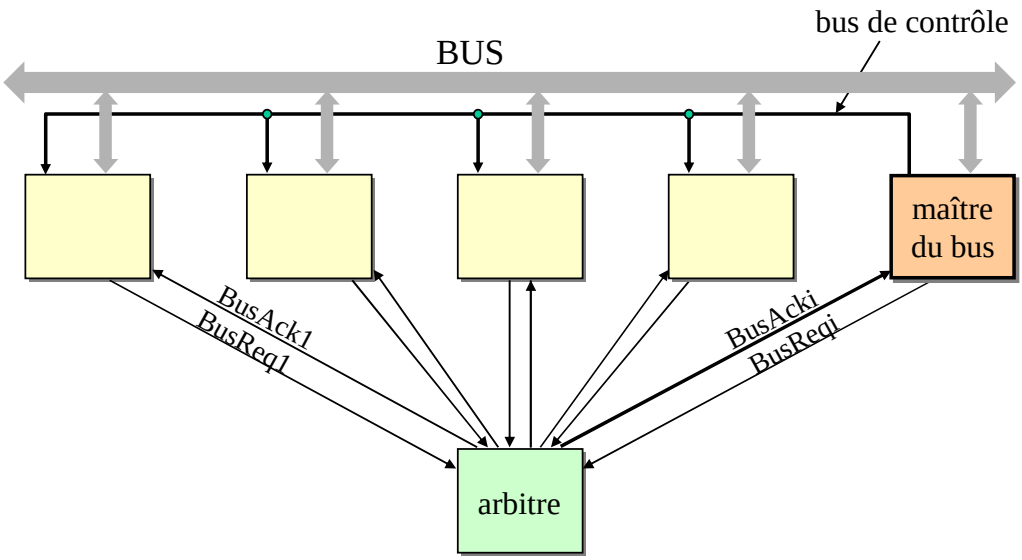
# Architecture d'un système à microprocesseur






Un système électronique peut utiliser différentes variétés de mémoires en fonction des besoins. Par exemple dans un ordinateur la SRAM rapide est utilisée comme cache et est gérée par un contrôleur de cache. La DRAM, de taille plus importante, est utilisée comme mémoire de programmes, la mémoire FLASH contient le BIOS pour le démarrage. La mémoire de masse utilise des technologies magnétiques , optiques ou électroniques avec l'augmentation en capacité des mémoires FLASH de type NAND FLASH

# Arbitrage de bus

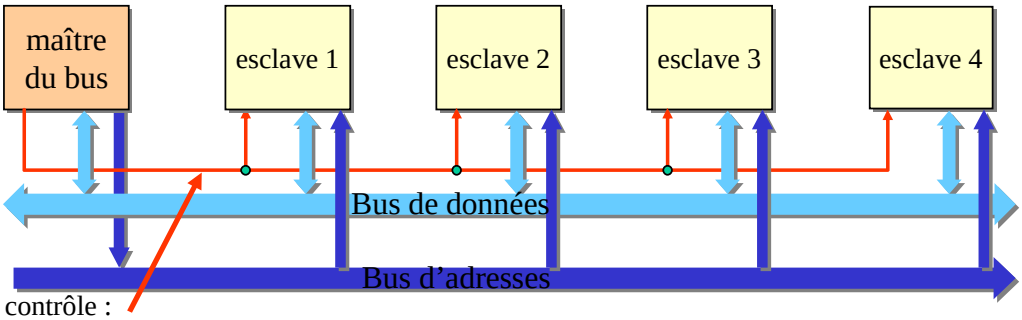





L'arbitre reçoit des requêtes des maîtres potentiels et leur renvoie une autorisation s'il leur accorde le bus. Une seule autorisation ou "jeton" est donné. De nombreuses stratégies existent pour avoir des accès équitables et éviter les "famines". L'arbitre peut faire tourner le jeton, change le niveau de priorité si un maître accapare trop souvent le bus,…

# Transfert sur un bus

- **Nécessité d'associer un champ d'adresse à un élément.**
  - chaque élément a son propre décodeur d'adresse générant un "CS"
- **Espace adressable**
  - soit dans la mappe mémoire
  - soit dans la mappe E/S (n'existe pas toujours)




bus de contrôle :

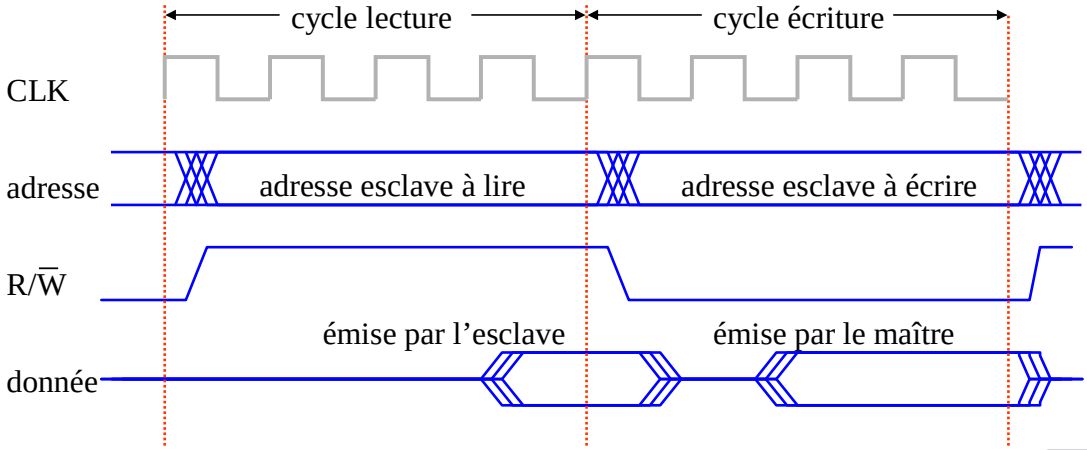
- horloge, activité du cycle de transfert, direction des transferts (lecture/écriture), gestion du protocole d'échange



Les éléments sur un bus ont leur propre champ d'adresse correspondant à l'activation de leur "Chip Sélect" . Quand un maître dispose du bus, il doit présenter l'adresse de l'"esclave" visé sur le bus d'adresse, le type de transaction sur le bus de commandes, et les données en écriture sur le bus de données ( ou attendre les données s'il s'agit d'une lecture).

## Protocole de communication synchrone

L'horloge rythme les échanges entre maître et esclave.
Le nombre de cycle est constant



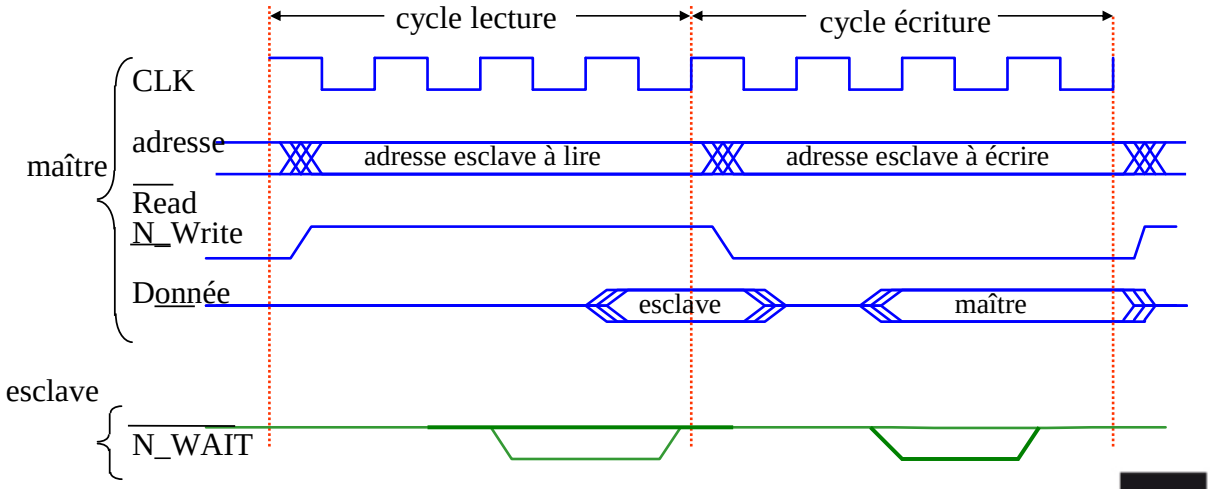



Le protocole utilisé sur un bus est celui qu'on retrouve dans tous les systèmes de communication. Il est soit direct , c'est à dire sans préoccupation d'acceptation de l'élément adressé, ou **asynchrone** c'est-à-dire avec un échange "poignée de main" ou "Handshake" pour s'assurer de la fiabilité de la transaction.

Dans le cas de l'échange direct, on parle plutôt d'échange **synchrone** en électronique car l'accès dure un nombre constant de cycles d'horloge. Les 2 éléments en communication se sont mis d'accord à l'avance sur le nombre de cycles d'horloge nécessaire à la transaction.

## Protocole de communication asynchrone

L'échange est assuré par un processus de ''Handshake'', l'esclave renvoie au maître soit un signal pour acquiescer ou ralentir l'échange.





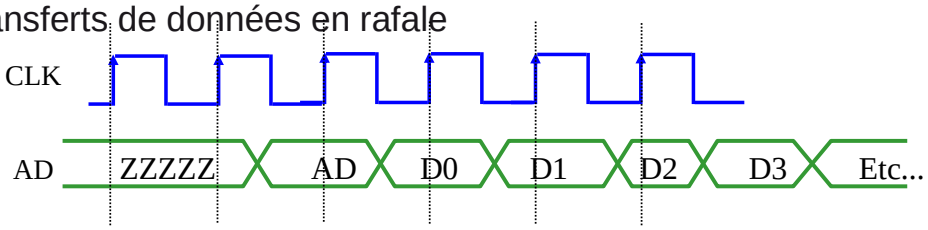

Le protocole asynchrone nécessite un signal "accusé de réception" ou "feu vert" de l'élément adressé vers le maître de façon à s'assurer que la transaction va bien se passer. Il permet à certains éléments lents de faire attendre le maître. Il existe 2 philosophies; Soit ce signal en retour dit : "Je suis prêt" , auquel cas il faut toujours que l'esclave génère ce signal à chaque transaction. Ou bien le signal en retour signifie "attendez !" auquel cas il n'est pas nécessaire que l'esclave génère le signal s'il est assez rapide.

Attention les protocole Asynchrone ne veut pas dire qu'il n'y a pas d'horloge. Dans l'exemple ci-dessus, il existe toujours une horloge car la logique est Synchrone mais la communication est Asynchrone.

# Caractéristiques de bus

- **Protocoles**
- **Débits max**
  - Transferts de données en rafale

**Exemple : PCI**



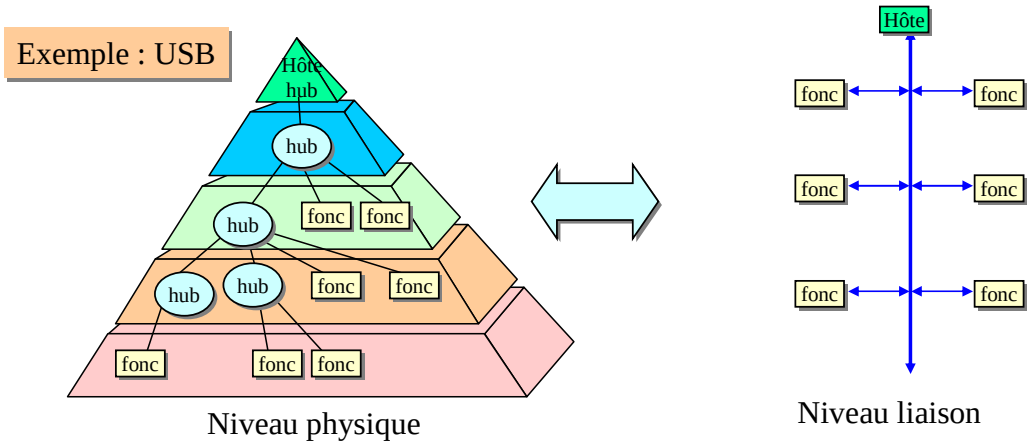

- **Liaison Série ou Parallèle**
- **Caractéristiques électriques (tension, courant, impédance)**
- **Configuration automatique au démarrage ou "à chaud"**
- **Isochronisme**
- **Interruptions**



Les bus se différencient par leur débits, leurs caractéristiques électriques, mécaniques, protocolaires,… Pour augmenter le débit, le mode rafale ou "burst" est communément employé. Il offre la possibilité de transmettre des paquets de données à raison d'une donnée par cycle d'horloge. Le bus peut respecter des contraintes d'isochronisme", c'est-à-dire que l'arbitre peut s'engager à maintenir un débit de  communication. Il peut aussi y avoir des signaux d'interruption pour que certains éléments critiques signalent un évènement important. Ceci évite au maître du bus de scruter périodiquement cet élément (polling).

# Topologie de bus

- **Certains bus peuvent avoir une topologie point à point au niveau physique et une topologie de type Bus au niveau "liaison"**






Le bus USB a une topologie virtuelle de bus au niveau "liaison" de la communication mais sa couche physique est composée de liaisons point à point. Il n'existe qu'un seul maître qui est le processeur.

# Quelques standards d'interface

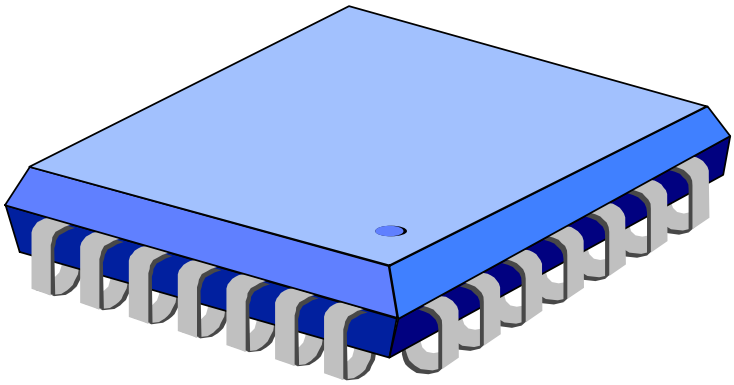
| INTERFACE | FONCTION | DEBIT | DISTANCE | STANDARDS |
|---|---|---|---|---|
| Homme -Machine | rentrée de données | $10^2$ b/s | $10^{-1}$ m | RS232, $I^2C$, USB, SPI |
| | lecture résultats | $10^8$ b/s | $10^0$m | DVI, IEEE1394 |
| Machine -Machine | stockage | $10^9$ b/s | $10^{-1}$ m | SATA, Fiber channel, IEEE1394 |
| | réseau local | $10^8$ b/s | $10^2$ m | WIFI, ethernet |
| | réseau distant | $10^7$ b/s | $10^4$ m | ADSL |



Les interfaces de communication tirent parti des avancées technologiques aussi bien dans le domaine des semi-conducteurs que ceux des algorithmes de traitement du signal et de communications.

Par exemple l'interface visuelle homme-machine bénéficie des algorithmes de compression d'image et de synthèse d'images avec des processeurs graphiques vidéo disposant d'une très grande puissance de calcul. Les communications distantes entre machines se font à très haut débit grâce aux technologies CDMA (téléphone portable, WIFI) OFDM (WIFI, TNT, ADSL) et codage canal (Turbo-codes, LDPC) .

# ARCHITECTURE de l'ARM7TDMI



## Qu'est ce qu'un ARM7TDMI?

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



Le processeur **ARM7** est un processeur RISC 32 bits très utilisé dans les téléphones portables. Il s'agit en fait d'un « cœur » de processeur initialement conçu par la société britannique ACORN. Un "cœur" signifie que ce processeur est vendu comme bloc à utiliser dans un circuit qui intègre d'autre blocs pour constituer un système sur puce "SoC". Son succès vient de sa petite taille et sa faible consommation. Il représente un très bon exemple pédagogique de par son architecture RISC simple et classique.

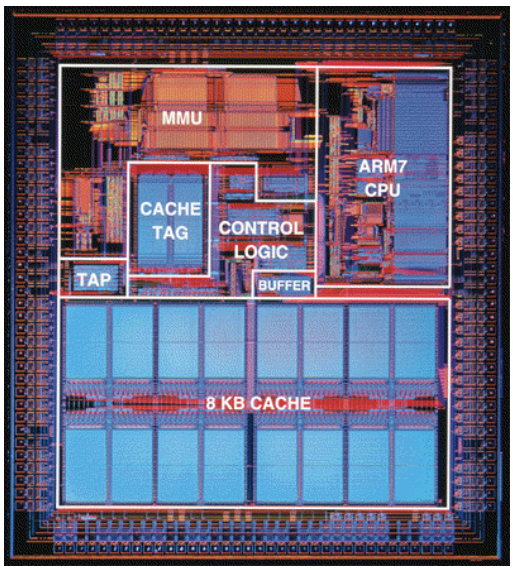
La lettre **T** symbolise le mode "Thumb" qui permet d'utiliser des instructions de 16 bits plutôt que 32 bits, de façon à diminuer la taille de la mémoire et donc le coût des équipements.

Le **D** signifie "Débugging" car le processeur dispose des facilités de débogage avec un "scan chain" autour du cœur de façon à pouvoir accéder aux bus données, adresse et contrôle du processeur.

Le **M** indique que le processeur dispose d'un multiplieur 32x8 lui permettant d'accélérer quelque peu les calculs de traitement du signal

Le **I** veut dire "In circuit Emulator" car le processeur a de la logique peu intrusive qui permet au concepteur d'analyser et piloter le déroulement de programme pour trouver les erreurs ou mettre au point l'environnement du processeur. L'émulation vient du fait qu'il est possible de piloter le processeur par l'extérieur et de lui adjoindre des ressources supplémentaires.

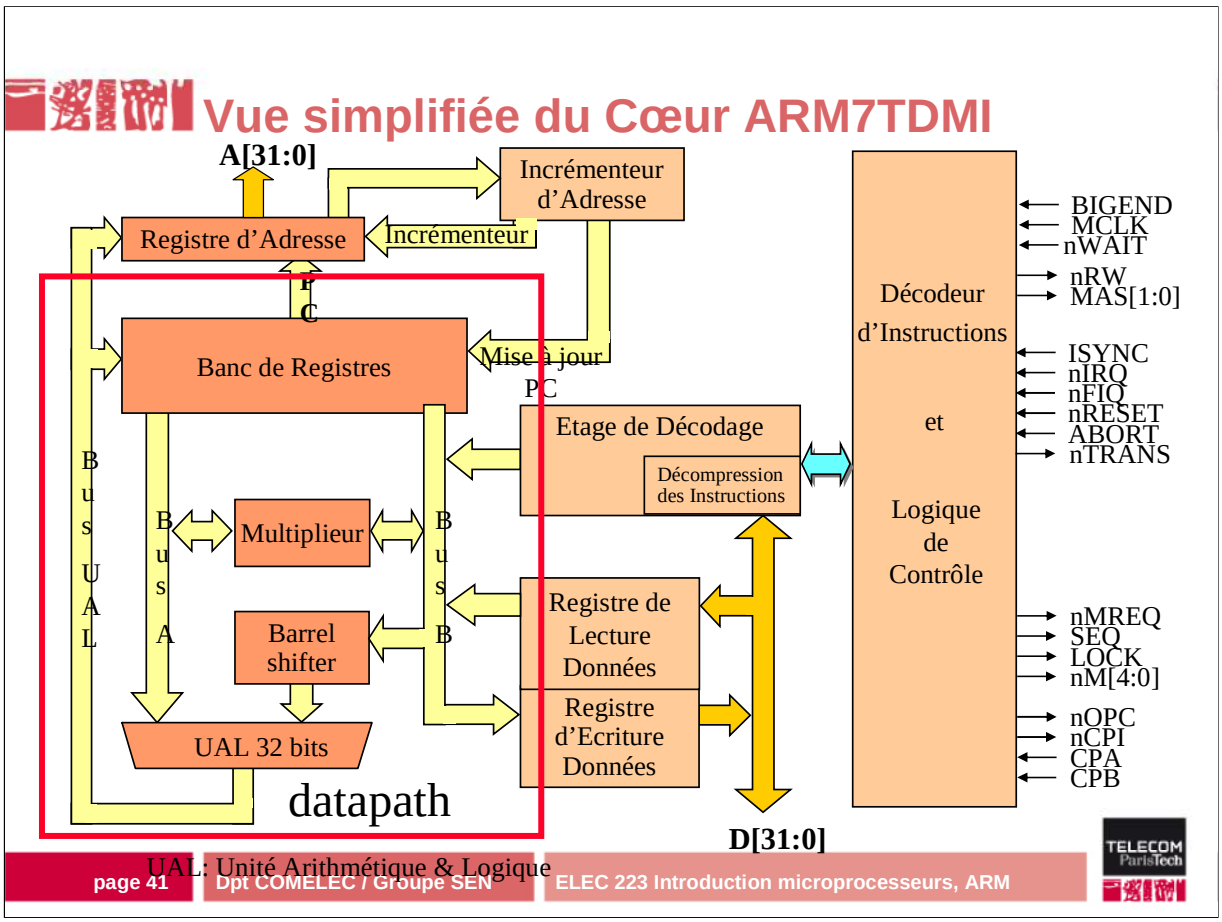
# Vue d'une puce utilisant un ARM7




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



Voici un exemple d'un circuit utilisant le cœur ARM7. La taille du processeur (0,5mm$^2$ en 180nm) est relativement petite par rapport à la mémoire cache et au circuit de gestion mémoire MMU.




L'architecture de l'ARM 7 est Von Neuman, c'est-à-dire qu'il n'y a qu'un bus pour véhiculer les instructions et les données. L'ARM dispose de 16 registres visibles sur lesquels les opérations sont effectuées. Les instructions LOAD/STORE servent respectivement à précharger/stocker les résultats de/vers la mémoire.

La partie opérative (le "datapath") est constitué de 2 bus A et B sur lesquels se font les opérations. Le flot de calcul est le suivant :

Banc de registres sur A et B  => Barrelshifter sur B => UAL => Banc de registres

Les opérations sur ce datapath durent 1 cycle. L' exécution de l'instruction

**ADD r0,r1,r2**

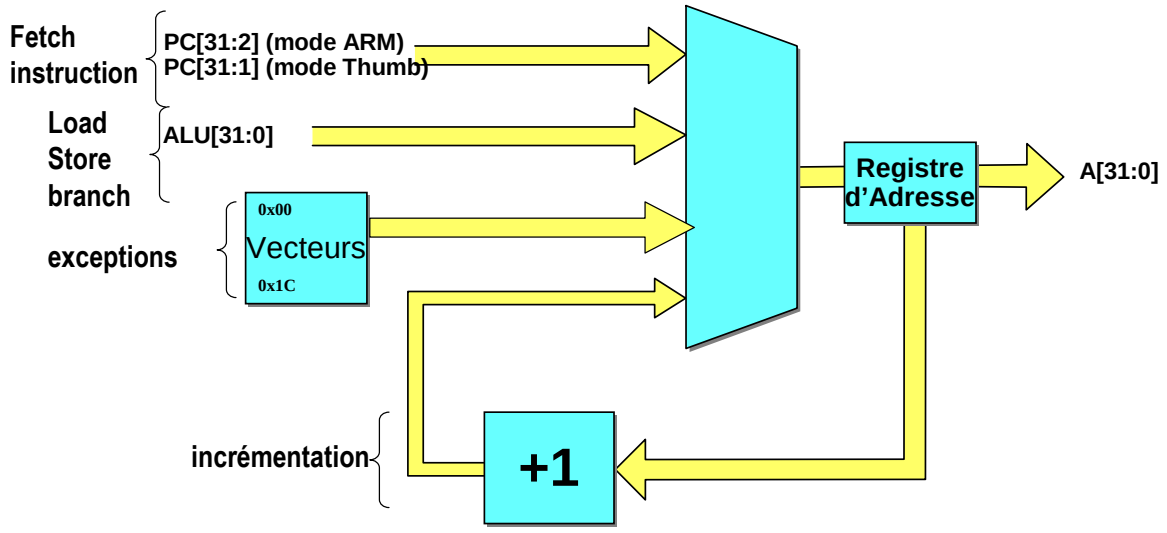
consiste à mettre r1 et r2 sur A et B, a effectuer l'addition dans l'UAL et écrire le résultat dans r0. Une opération de décalage aurait pu avoir lieu sur r2 dans le même cycle

**ADD r0,r1,r2,LSL#2**

Ici r2 est décalé de 2 bits sur la gauche avant d'être additionné

Le registre d'adresse fournit les adresses à la mémoire. Les blocs décompression et décodage des instructions servent au 2 premiers cycles du pipeline alors que le datapath sert au 3ème cycle. Les registres de Lecture et d'écriture données sont des tampons de données avec la mémoire externe pour les instructions LOAD/STORE. Le bloc de contrôle à droite est l'interface avec tous les signaux de commande externes, comme par exemple les interruptions, le contrôleur mémoire, les coprocesseurs,…

## Génération des Adresses



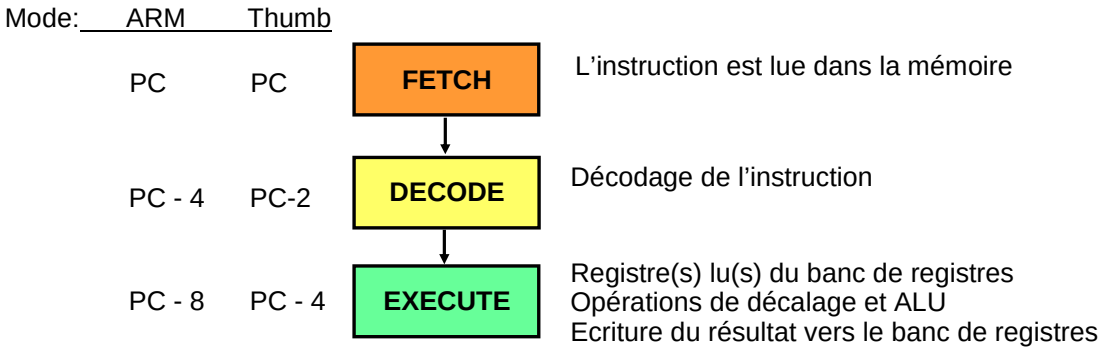



L'adresse A[31:0] fournie à la mémoire externe peut être générée par 4 sources différentes :

1. Le **registre d'adresse +1**. C'est le cas le plus fréquent correspondant au cas où les instructions se suivent. Il correspond à la valeur du compteur de programme **PC** incrémenté de 1 de façon à aller chercher l'instruction suivante. Comme le chemin d'incrémentation avec le vrai registre PC est critique, il est plus efficace d'effectuer l'incrémentation avec le registre d'adresse si celui-ci est préalablement chargé avec PC.
2. Le compteur de programme **PC**. En fait il ne sert qu'après les LOAD et STORE pour recharger le registre d'adresse avec la valeur de PC. Notez que les 2 bits de faible poids sont inutiles en mode 32 bits ARM car les instructions sont toujours alignées sur des mots de 32 bits. En mode Thumb, seul le bit de faible poids est inutile.
3. L'**UAL** lors de l'exécution d'un LOAD, STORE ou BRANCH, de façon à pointer sur une donnée en mémoire (LOAD ou STORE), ou effectuer un branchement.
4. Les **vecteurs** qui sont des adresses pour aller pointer vers les programmes des traitements d'exceptions.

# Le Pipeline d'Instructions

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




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



Les instructions sont "pipelinées" en 3 cycles d'horloge : FETCH, DECODE et EXECUTE. Ainsi le concept RISC (1 instruction par cycle) est respecté.

# Exemple: Pipeline Optimal

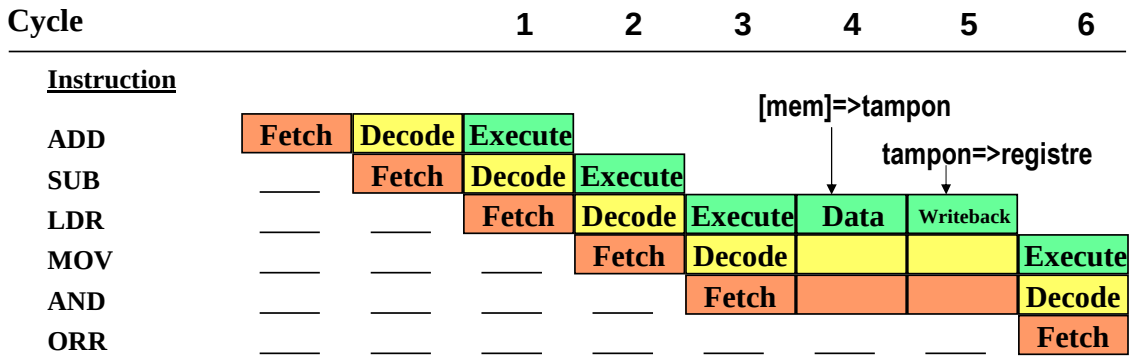
| Cycle | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **Instruction** | | | | | | |
| ADD | Fetch | Decode | Execute | | | |
| SUB | | Fetch | Decode | Execute | | |
| MOV | | | Fetch | Decode | Execute | |
| AND | | | | Fetch | Decode | Execute |
| ORR | | | | | Fetch | Decode | Execute |
| EOR | | | | | | Fetch | Decode | Execute |
| CMP | | | | | | | Fetch | Decode |
| RSB | | | | | | | | Fetch |

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



Voici le pipeline idéal avec un débit de calcul de 1 cycle par instruction (CPI=1)

# Exemple: Pipeline avec LDR



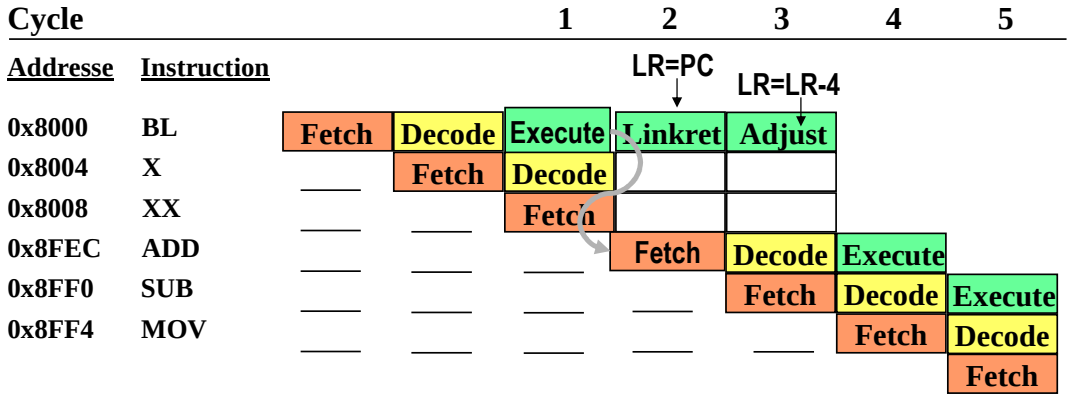

- L'instruction LDR lit une donnée en mémoire et la charge dans un registre
- il faut 6 cycles pour exécuter 4 instructions. CPI = 1,5



Le pipeline est rompu lors d'un LOAD ou STORE. Dans cet exemple le LDR (LOAD Register) nécessité d'aller chercher la donnée dans la phase d'exécution. 1 cycle (cycle 4) est nécessaire pour lire la mémoire et mettre la donnée dan un registre tampon. Un autre cycle (cycle 5) sert à transférer la donnée du registre tampon vers le banc de registre.

Les instructions LOAD et STORE prennent donc 3 cycles à la place de 1.

# Exemple: Pipeline avec Saut



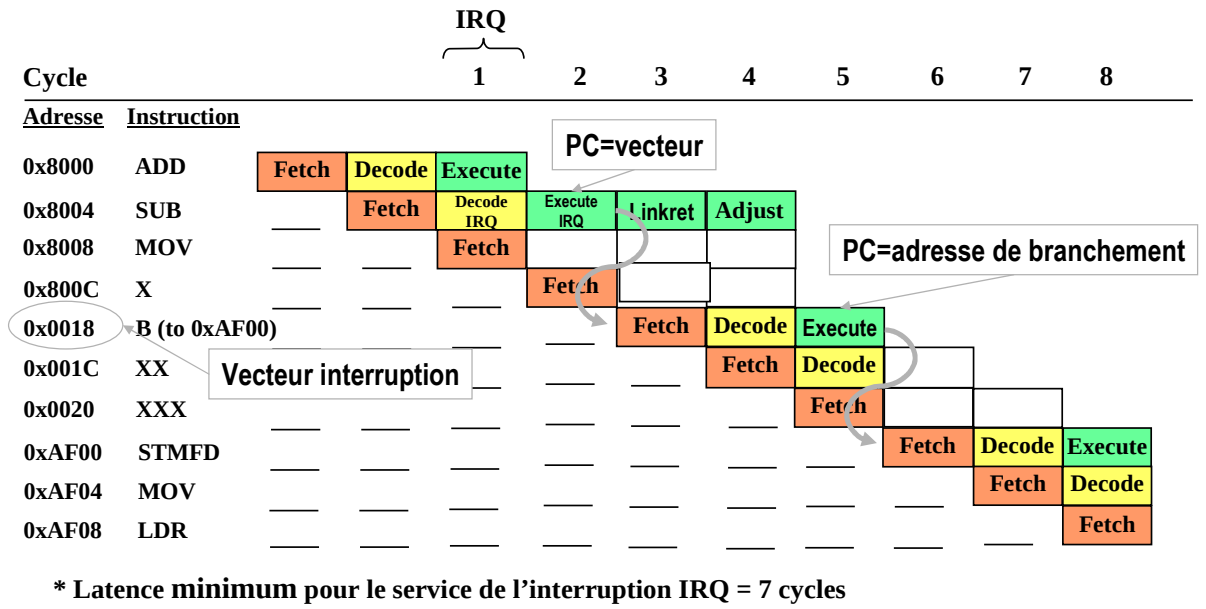

- L'instruction BL effectue un appel de sous-programme
- Le pipeline est cassé et 2 cycles sont perdus



L'instruction de branchement BRANCH est exécutée au cycle 1, c'est-à-dire que le PC est chargé à la nouvelle valeur. Dans cet exemple il s'agit plus précisément d'un branchement à un sous-programme **BL**, le L signifiant "Link"pour le retour au programme principal. Au cycle 2 la première instruction du sous-programme est lue (FETCH) et le datapath est utilisé pour sauver le PC dans le registre **LR** "Link Register". Au 3ème cycle la valeur du LR est ajustée de façon à ce que le sous-programme revienne précisément à l'instruction suivant le saut au sous-programme.

Le branchement prend donc 3 cycles à la place de 1 cycle.

# Exemple: Pipeline avec Interruption




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



Le déroulement est le suivant lorsque le processeur reçoit une interruption IRQ :

**Cycle1:** Le processeur est averti de l'interruption et vérifie si l'interruption est masquée. Si c'est le cas le processeur n'est pas interrompu, sinon il commence son exécution au prochain cycle.

**Cycle 2:** Exécution, c'est-à-dire récupération du vecteur, 0x18 pour IRQ, changement de mode, transfert CPSR vers SPSR (voir plus loin la signification de ces registres).

**Cycle 3:** Sauvegarde de PC-4 dans LR , lecture de l'instruction du branchement au programme de traitement BRANCH 0xAF00.

**Cycle 4 et 5:** cycles perdus à cause du branchement

**Cycle 6:** Lecture de la première instruction du programme de traitement de l'exception.

Le programme commence généralement par la sauvegarde du registre ne pile (empilage) et se termine par la récupération (dépilage). Le retour au programme principal s'effectue avec SUBS PC, LR ,#4 car le LR n'est pas exactement la valeur de retour au programme principal.

# Les Modes du Microprocesseur

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



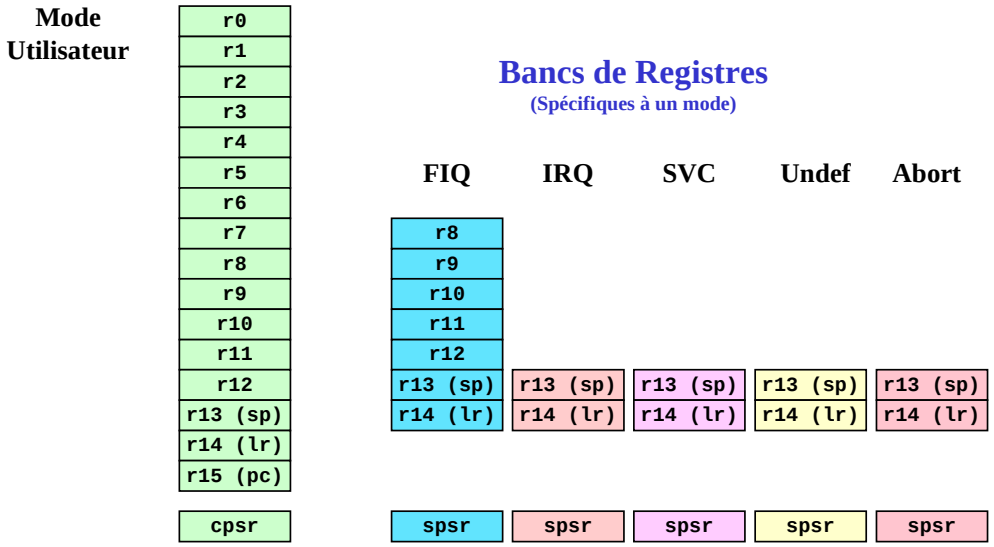
Le processeur ARM a plusieurs modes de fonctionnement différenciés par des ressources et des privilèges spécifiques. Ces modes simplifient le portage d'un système d'exploitation multi-utilisateurs.

Selon les modes, le coeur ARM active des signaux de commandes : nTRANS, HPROT qui peuvent être utilisés par des contrôleurs mémoire ou E/S externes pour autoriser ou non certaines zones mémoire.

D'autre part certaines opérations ne sont autorisées que dans certains modes.

# Les Registres

## Registres actifs



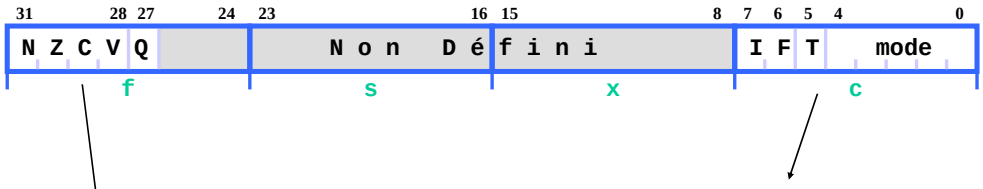



L'architecture ARM fournit 37 registres organisés en bancs associés aux modes.17 registres sont visibles

- 13 registres de données génériques (r0 - r12).
- Le registre spécifique r13 qui est réservé comme pointeur de pile et est propre à chaque mode. Il s'appelle aussi **SP**.
- Le registre spécifique r14 qui est aussi le registre de retour de sous-programme **LR**. Ce registre évite l'empilage du PC et est propre à chaque mode.
- Le registre spécifique r15 qui est aussi le compteur de programme **PC**.
- Le registre de statut **CPSR** (Current Program Status Register) qui contient des informations sur l'état du processeur
- Le registre **SPSR** qui est une copié du CPSR avant de changer de mode. Ce registre évite l'empilage du CPSR et est propre à chaque mode.

En mode "Fast Interrupt" FIQ, les registres de données sont nombreux de façon à ne pas avoir à sauvegarder le contexte des registres du programme principal et donc diminuer la latence pour accélérer le traitement.

# Les Registres d'État CPSR et SPSR




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

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



Les bits de fort poids représentent les indicateurs de l'UAL. Les bits 6 et 7 sont des bits de contrôle pour masquer les interruptions. Le bit 5 permet de passer en mode Thumb et les 5 bits de faible poids indiquent le mode.

# Accès à la Mémoire et aux E/S

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

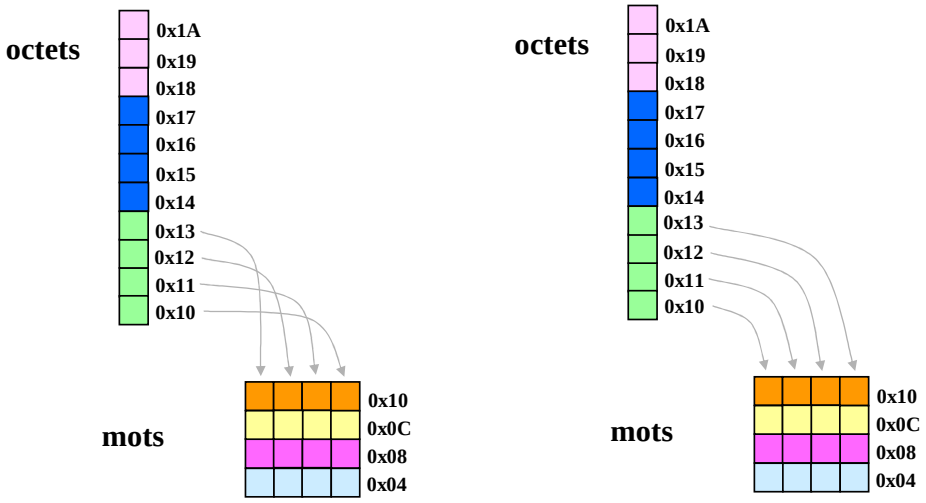


Le processeur ARM dispose de 32 bits d'adresse octets pour accéder aussi bien à la mémoire qu'aux E/S. Un mot a par définition une taille de 32 bits. Il est possible d'accéder des demi-mots sur 16 bits et des octets.

**Attention** les mots ont toujours des adresses multiples de 4 (ils sont forcément alignés) et les demi-mots ont des adresses paires.

# Organisation de la mémoire

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




« Little Endian » « Big Endian »



Au sein d'un mot et d'un demi-mot, il y 2 façon d'adresser les octets. Ces 2 types d'adressage existent depuis les premiers microprocesseurs 16 bits où Intel (8086) et Motorola (68000) avaient choisi chacun un type différent.

# Jeu d'Instructions ARM(1)

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

Positionnement des indicateurs

Execution si le résultat précédent est 0



Comme la majorité des langages "machine", la syntaxe du langage assembleur de l'ARM dispose de 5 champs pour décrire l'instruction :

| champ1 | champ2 | champ3 | champ4 | champ5 |
|---|---|---|---|---|
| Etiquette | Instruction | Résultat | Operande1 | Opérande2 |

L'**étiquette** correspond à l'adresse symbolique. Elle est optionnelle à part pour les débuts de sous-programme.

L'**instruction** peut être complétée de la taille des données (8,16 ou 32 bits) et d'une condition d'exécution.

Le **résultat** est toujours avant les opérandes car certaines instructions n'ont qu'un seul opérande

Le résultat et les opérandes sont des indices de registre contenant les données

Les registres opérandes pour LOAD et STORE sont des registres d'adresse (adressage indirect)

L'opérande 2 peut être complétée par un décalage effectué par le "barrelshifter"

# Jeu d'Instructions ARM(2)

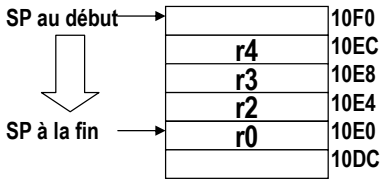
- Instructions spécifiques d'accès à la mémoire
  - `LDR r0,[r1],#4` ; *r0=mem(r1), r1= r1+4*
  - `STRNEB r2,[r3,r4]` ; *mem(r3+r4)=r2*
  - `LDRSH r5,[r6,#8]!` ; *r5=mem(r6+8), r6=r6+8*
  - `STMFD sp!,{r0,r2-r4}` ; *transferts multiples*
  - *;empilage : mem(sp+i)=liste de registres*

Si Z=0

En octet

En 16 bits
Avec extension
de signe sur les
16MSBs

**r6=r6+8 en fin d'execution**




Opération réciproque de dépilage :

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

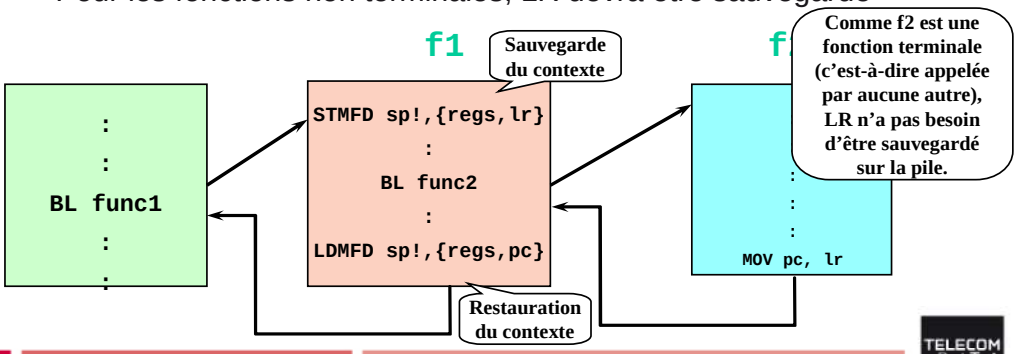


Dans les instructions de transfert, le registre entre crochet [r1] indique qu'il s'agit d'un registre d'adresse servant à pointer une donnée en mémoire

Il existe beaucoup de déclinaisons des LOAD et STORE. Les LOAD et STORE multiples sont très utiles pour l'empilage ou le dépilage en utilisant le premier registre comme pointeur de pile (normalement c'est r13=SP)

# Jeu d'Instructions ARM(3)

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






Le LR doit lui-même être empilé lorsqu'il y a des appels sous-programmes imbriqués. Remarquez l'instruction de dépilage à la fin du sous-programme f1 où un LOAD multiple permet de recharger le PC avec la valeur de LR.

# Executions conditionnelles

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

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

**+ petit et**
**+ rapide**

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

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



En mode 32 bits, les instructions exécutées conditionnellement permettent d'optimiser le code faisant appel à des branchements conditionnels.

# Exemple: Séquence d'Instructions

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

l'adresse [R8+0x10] dans R0

<< 2)

u de R1 à l'adresse [R8+0x14]

- Conditions initiales :

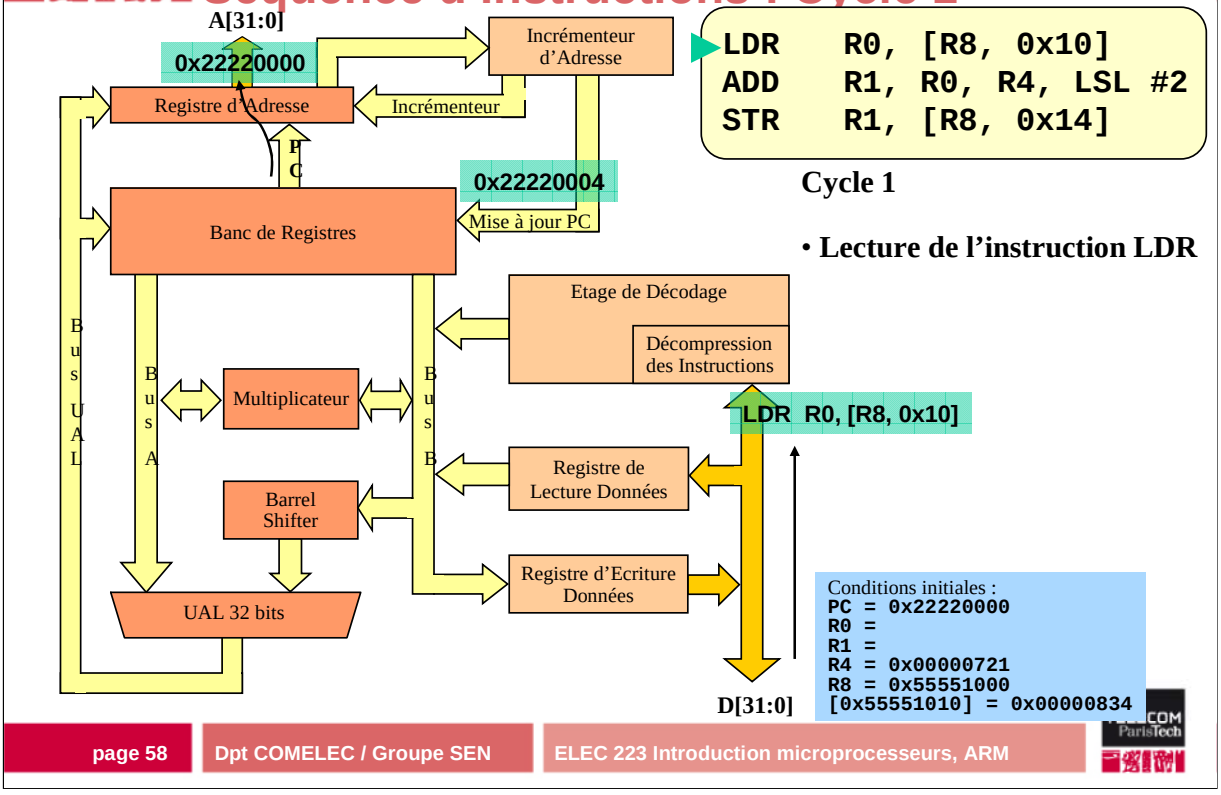
```
PC = 0x22220000
R4 = 0x00000721
R8 = 0x55551000
[0x55551010] = 0x00000834
```

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



De façons à comprendre le fonctionnement interne de l'ARM, considérons ces 3 instructions avec cet état des registres.

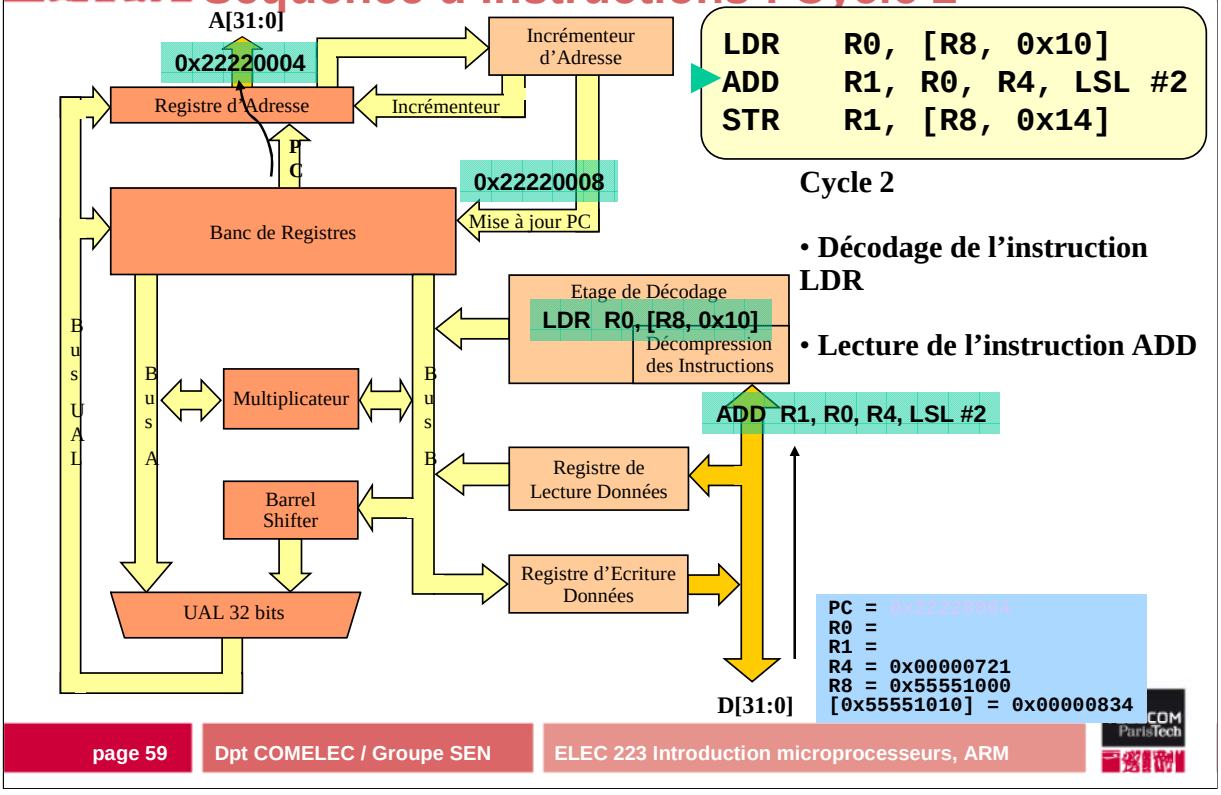
# Séquence d'Instructions : Cycle 1






L'instruction LDR est stockée dans un registre de l'étage de décompression.

# Séquence d'Instructions : Cycle 2




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

**Cycle 2**

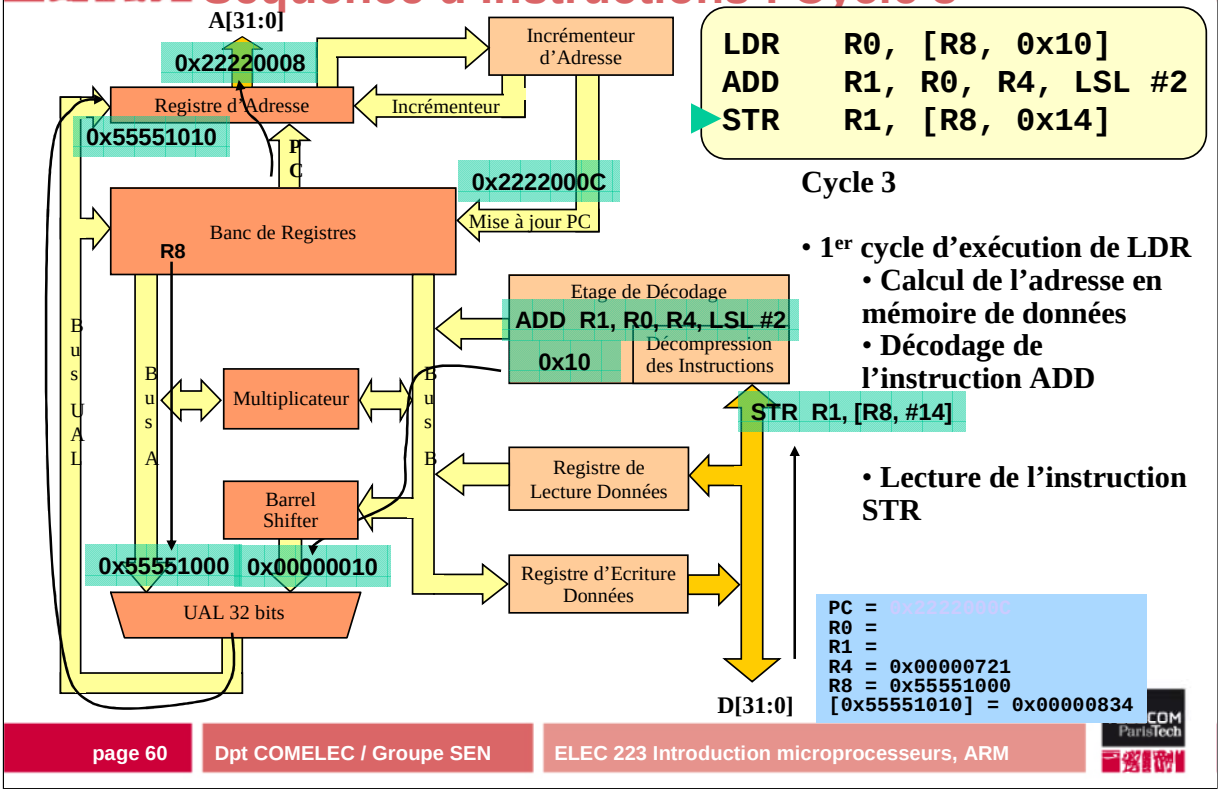
- **Décodage de l'instruction LDR**
- **Lecture de l'instruction ADD**



LDR est décodée

ADD est lue dans le tampon d'instruction

# Séquence d'Instructions : Cycle 3



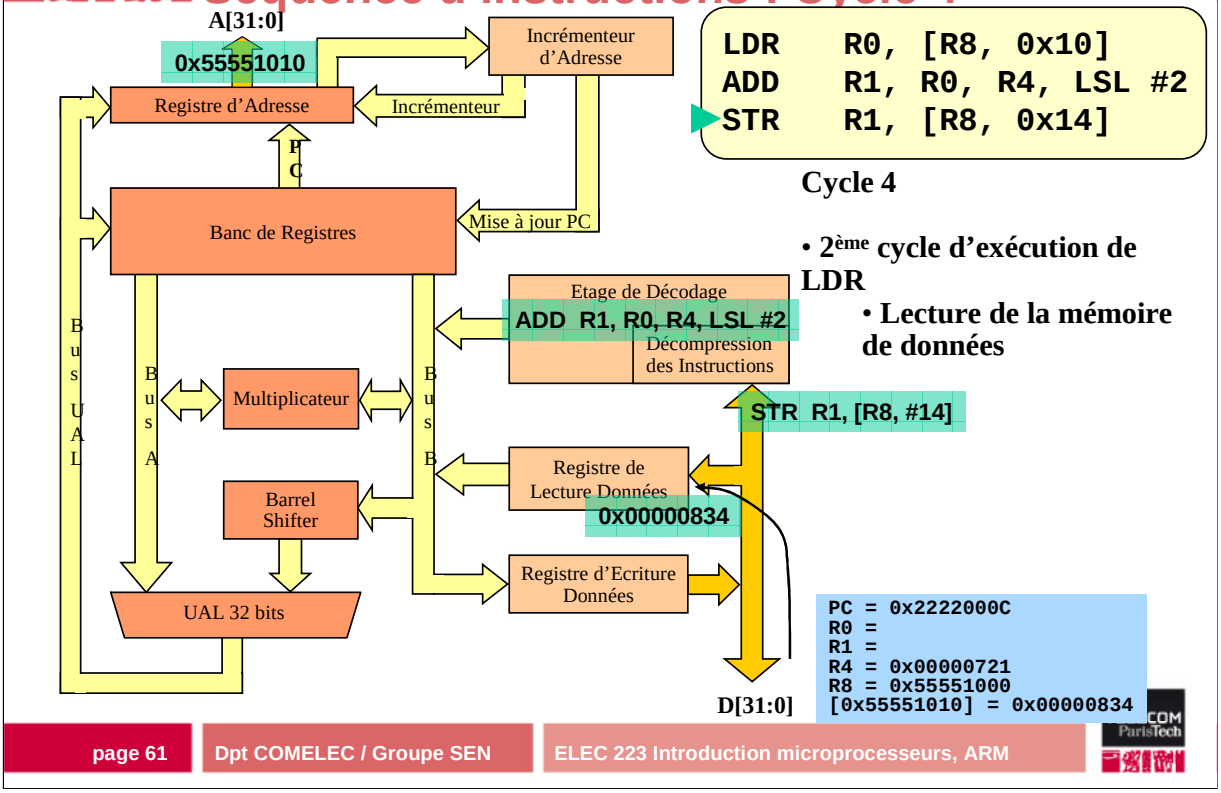



LDR est exécutée : Dans le datapath le calcul de R8 + 0x10 est effectué

ADD est décodée

STR est lue dans le tampon d'instructions

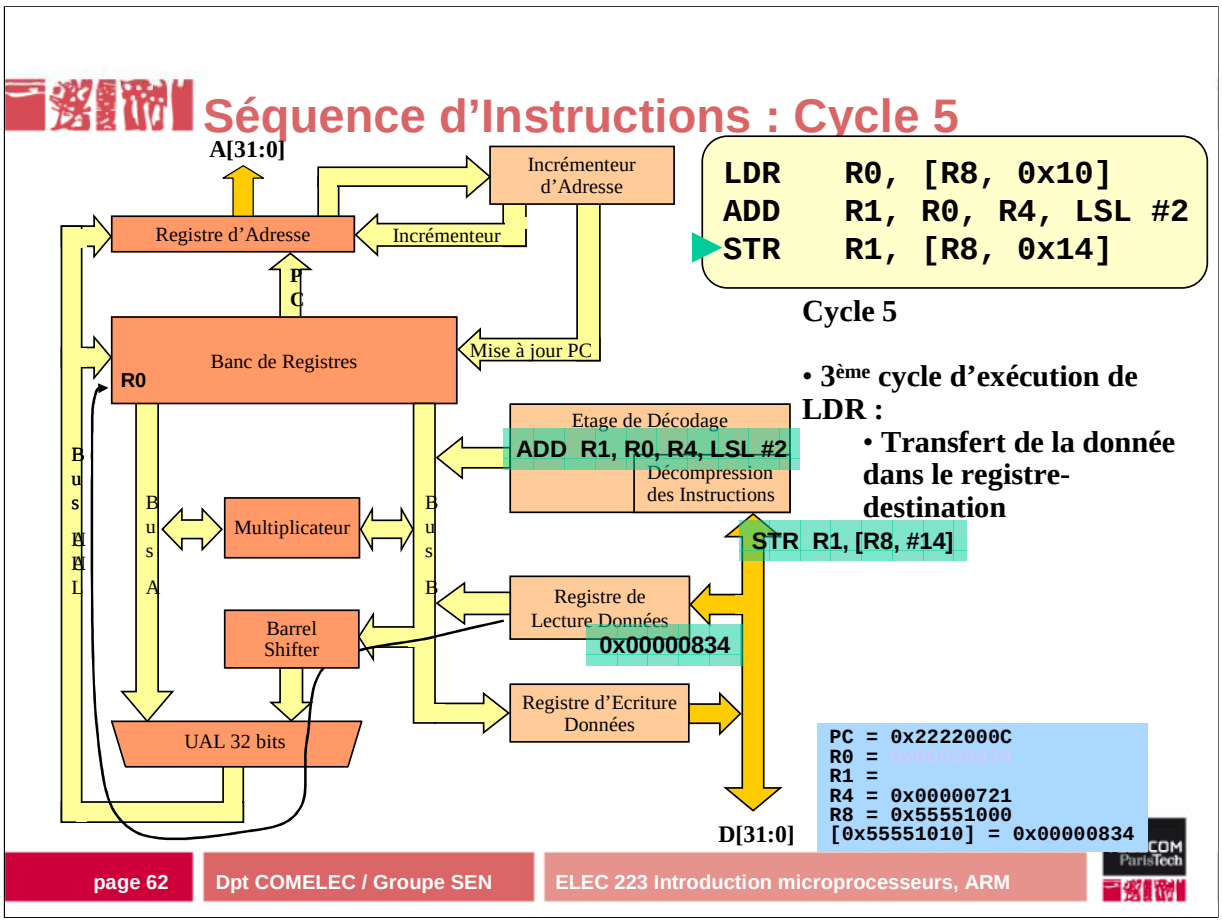
# Séquence d'Instructions : Cycle 4





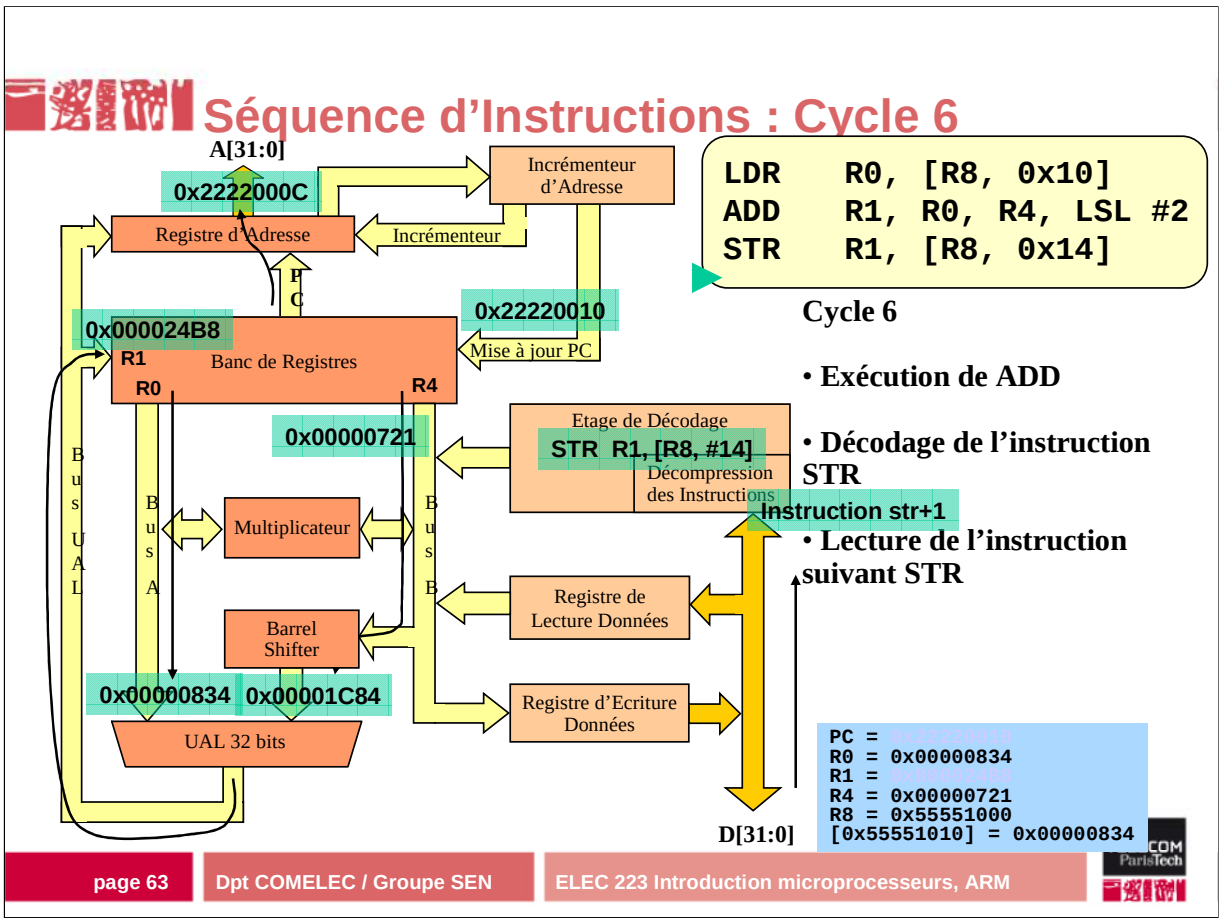

Le LDR est toujours exécuté car il faut aller lire une donnée en mémoire avec l'adresse R8+0x10. Cette donnée est stockée dans le registre tampon de lecture données

Les ADD et STR sont gelés respectivement dans l'étage de décodage et de tampon.




LDR n'est toujours pas terminé, il faut maintenant transférer la donnée lue du registre tampon vers le registre R0.

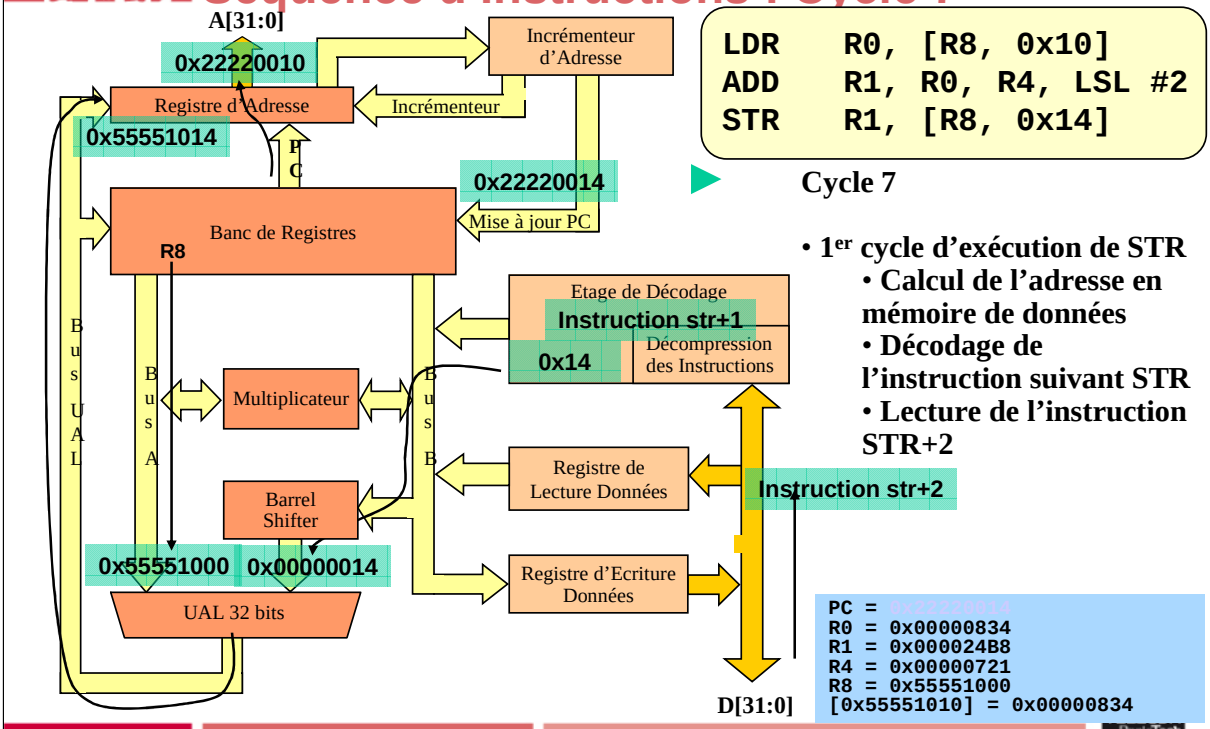
Les autres instructions sont toujours gelées.




Le ADD est maintenant exécuté. Le registre R4 qui est sur le bus B est décalé de 2 bits par le barrelshifter

Le STR est décodé

# Séquence d'Instructions : Cycle 7



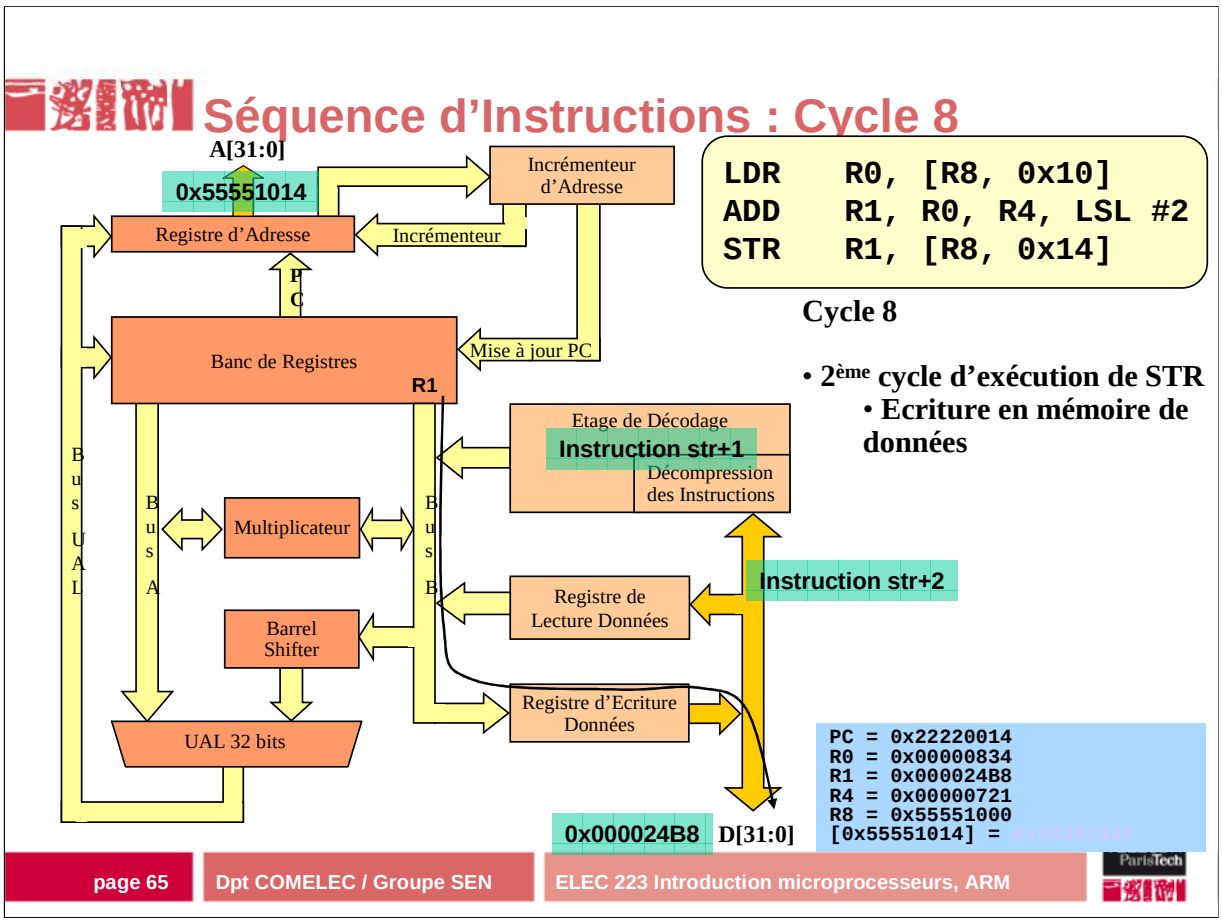

```
LDR   R0, [R8, 0x10]
ADD   R1, R0, R4, LSL #2
STR   R1, [R8, 0x14]
```

Cycle 7

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



Le STR est exécuté. Le calcul R8 + 0x14 est effectué dans le datapath




Le STR n'est pas terminé, le registre R1 est écrit dans le registre tampon avant d'être transférer à la mémoire ( 1 cycle de +)

# Exceptions

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

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

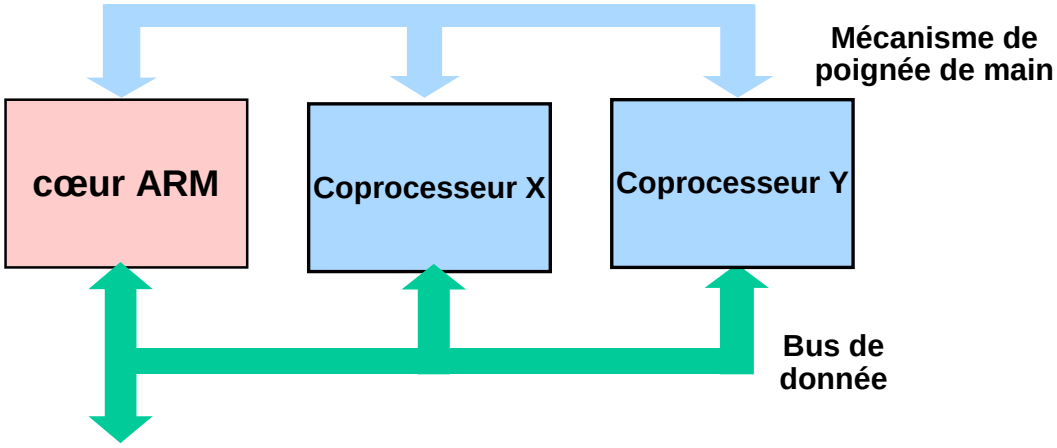
Le CSPR et le PC sont restaurés en même temps



Pour passer du mode USER vers un mode avec privilèges, il faut nécessairement une exception qui va forcer le processeur à aller effectuer un saut dans la table des vecteurs et effectuer un programme de traitement spécifique.

Pour le retour en mode USER, il faut que les registres LR et SPSR soient simultanément restaurés dans les registres PC et CPSR. C'est le but de l'indicateur S de l'instruction de retour du sous-programme d'exception.

## Coprocesseurs



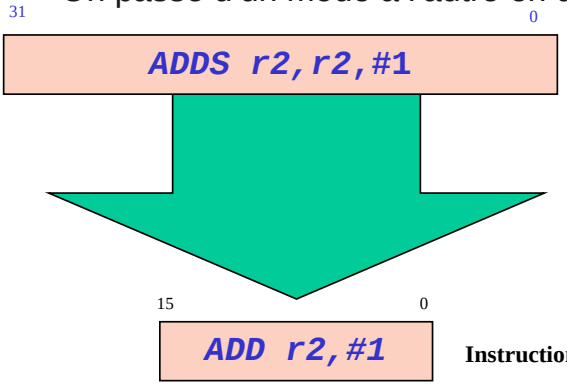

- 16 coprocesseurs peuvent être définis pour étendre le jeu d'instruction ARM
- Les coprocesseurs accèdent à la mémoire et aux registres directement par le biais d'instructions spécifiques
- Le coprocesseur 15 est dédié au contrôleur cache et MMU



Il est possible d'adjoindre des accélérateurs de calcul ou "coprocesseur" au processeur ARM. Les coprocesseur sont pilotés par l'ARM via une liaison spécifique. Lorsqu'une instruction coprocesseur est décodée, l'ARM déclenche le coprocesseur adressé qui va lire ou écrire à la volée les données transitant sur le bus. Si aucun coprocesseur n'est détecté, le processeur rentre en mode "undefined instruction" qui lui permet d'effectuer le calcul sans coprocesseur donc plus lentement.

# Mode Thumb

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




**Instruction en mode Thumb (16 bits)**

**Pour la plupart des instructions générées par le compilateur:**

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



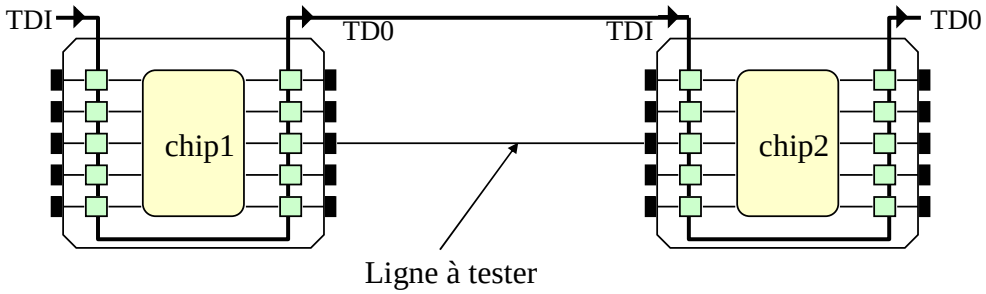
La taille du code en mode thumb correspond typiquement à 65-70% de la taille du code en mode ARM.

# Port JTAG

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

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

Boundary Scan Test :

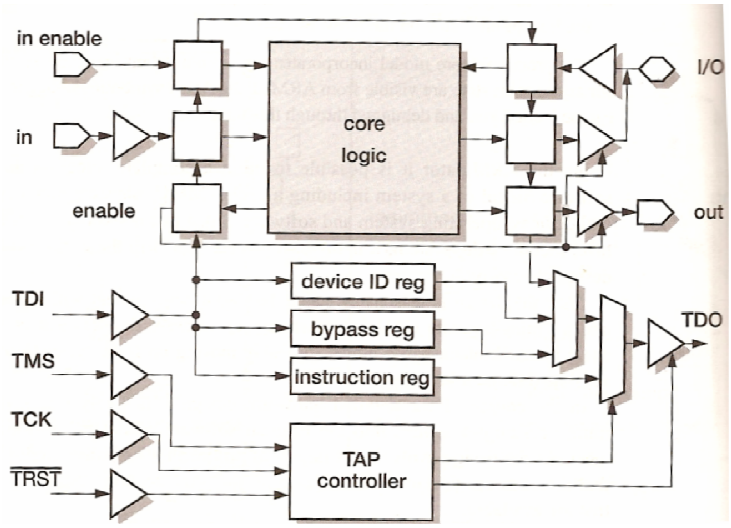




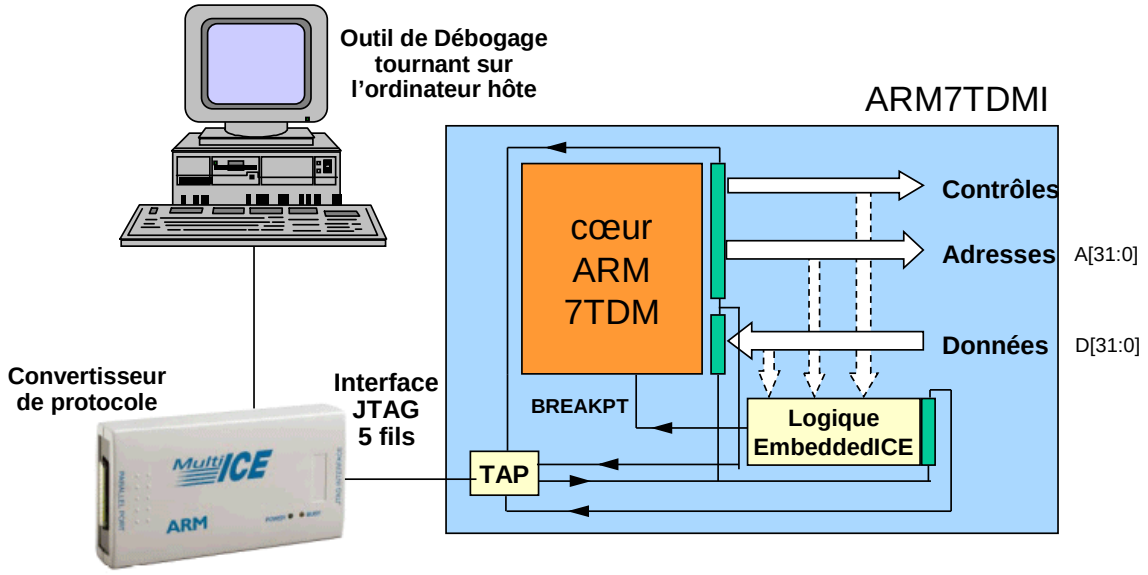

Le port JTAG "Joint Test Action Group" est une contrôleur gérant un flux de données série (TDI en entrée et TDO en sortie) permettant d'effectuer diverses opérations dont celle du test pour lequel il a été conçu à l'origine.

L'opération de test appelée "Boundary Scan test" permet de tester la connectivité des composants assemblés entre eux. Chaque composant a ses broches d'entrée/sortie reliées par un grand registre à décalage qui est lui même relié par les broches TDI et TDO à un autre circuit. Le tout constituant un énorme registre à décalage. Les contrôleurs JTAG ou "TAP controller" effectuent des opérations d'écriture ou de lecture dans ces registres. Il est donc possible de forcer un niveau logique sur une broche de sortie et de lire le niveau sur la branche d'entrée de la même équipotentielle d'un autre composant. Si les niveaux coïncident, l'équipotentielle est considérée intègre.

# JTAG architecture



# Débogage Embarqué






Il existe 3 chaînes de débogage sur le cœur ARM :

0 – E/S du cœur (incluant les données)

1 – données uniquement

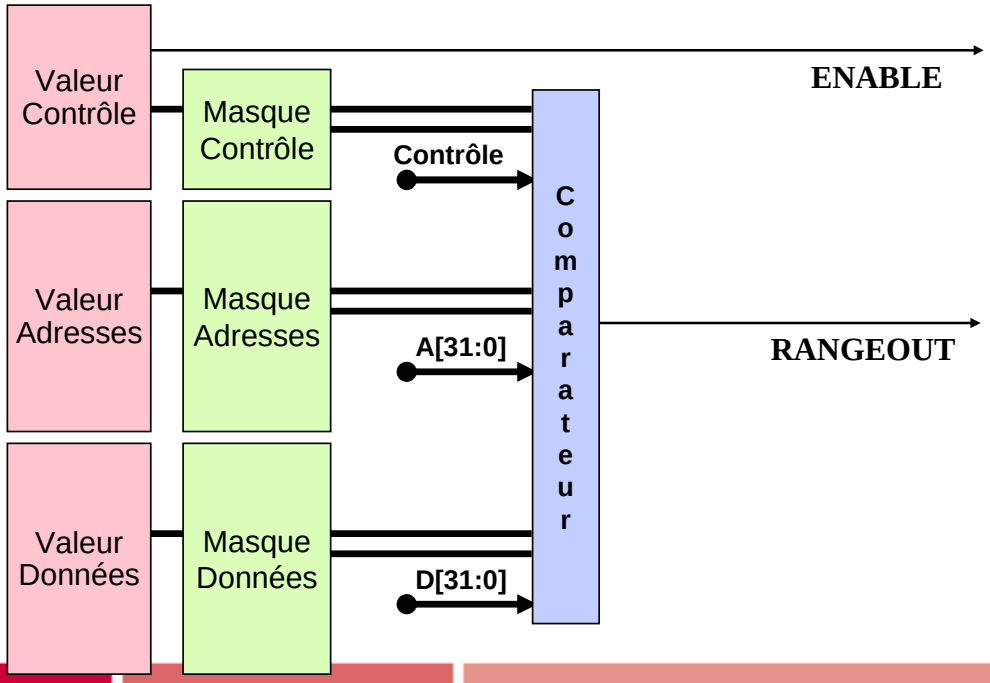
2 – emulateur embarqué

Le port JTAG permet d'accéder à ces 3 chaînes. Les données sont séparées pour aller plus vite dans la cas d'analyse des données seules.

L'émulateur embarqué "embedded ICE" analyse les évènements sur les bus et génère un point d'arrêt BREAKPT quand il y a comparaison avec un évènement préprogrammé. Dans ce cas le cœur est arrêté et islolé du reste du système. C'est au débogueur d'examiner et/ou changer les E/S par le biais des chaînes de débogage.

La communication avec l'émulateur embarqué s'effectue aussi par le JTAG en utilisant un convertisseur de protocole (MultiICE) .

## Unité de Gestion des Points d'Arrêt






Par le biais du JTAG, le concepteur peut accéder à la programmation de l'émulateur embarqué. Ceci consiste à définir la condition du point d'arrêt sur les bus de l'ARM.

L'émulateur consiste en une série de comparateur qui active le signal RANGEOUT qui va activé le point d'arrêt en quand la condition est remplie.

# ARM9TDMI

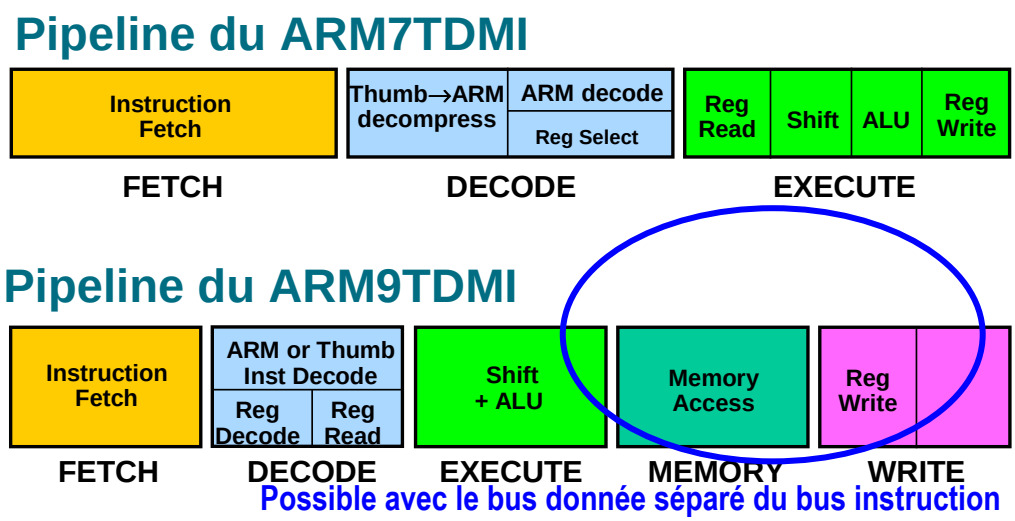
- **Implémentation à double bus (architecture Harvard)**
  - Augmente la bande passante entre le microprocesseur et la mémoire
    - Interface mémoire Instructions
    - Interface mémoire Données
  - Permet l'accès simultané aux mémoires Instructions et Données
  - => Modifications pour améliorer le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

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



La grosse différence consiste à passer d'une architecture Von Neuman à une architecture Harvard. Le fait de disposer des bus instructions/données séparés permet de modifier le pipeline pour avoir une phase d'accès à la mémoire donnée sans rupture de pipeline avec les instructions LOAD/STORE

## Modifications du Pipeline pour le ARM9TDMI

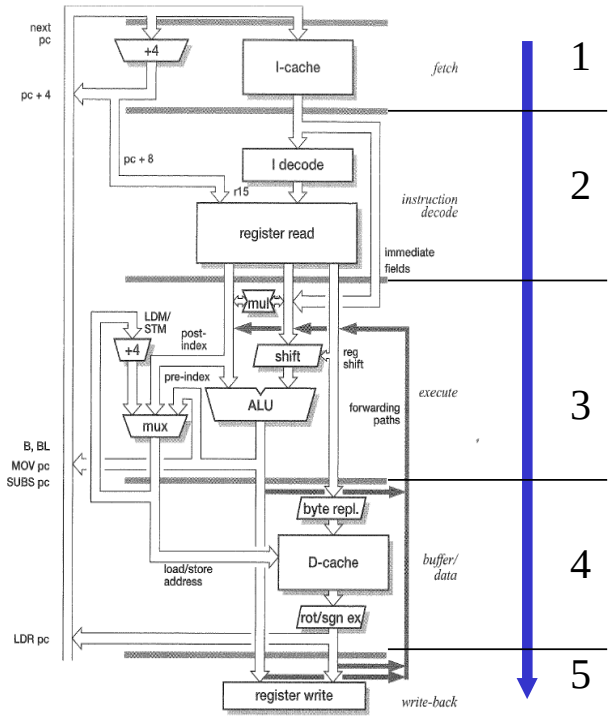





Dans l'ARM9, les 2 phases supplémentaire d'accès mémoire MEMORY et de transfert avec le registre tampon WRITE permettent d'effectuer des LOAD/STORE sans rompre le pipeline. Les 2 cycle perdus de l'ARM7 lors des LOAD/STORE ont maintenant disparus.

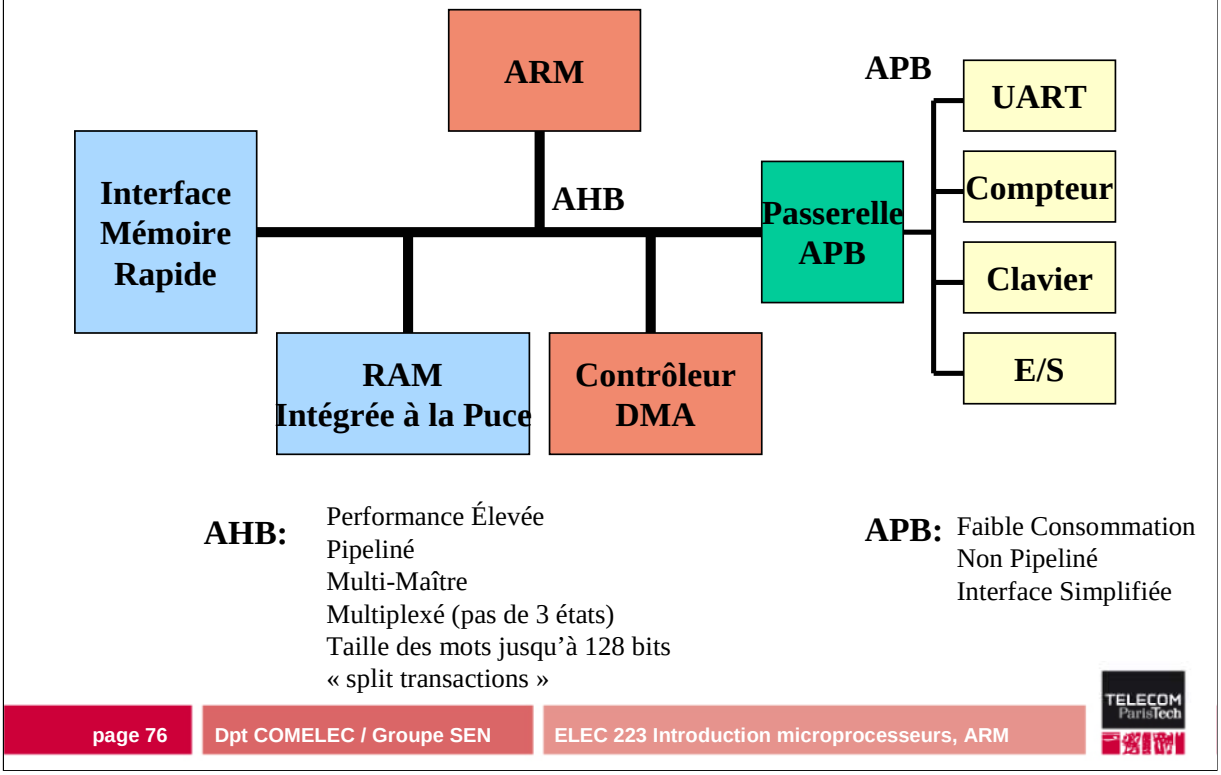
Comme l'ARM 9 dispose d'une architecture Harvard, le FETCH d'une nouvelle instruction peut se faire simultanément avec l'accès mémoire MEMORY.

# ARM9TDMI architecture




5 étages de pipeline

## le Bus AMBA



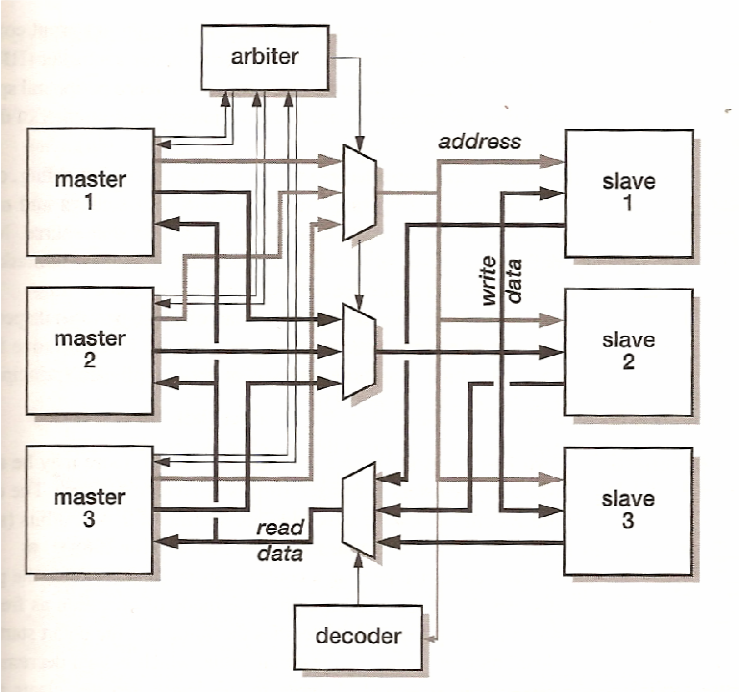



A côté du processeur, la société ARM propose le bus AMBA qui permet de concevoir une système complet. Ce bus AMBA est en fait composé de 2 bus :

- Un bus rapide Advanced High Bus
- Un bus lent Advanced Peripheral Bus.

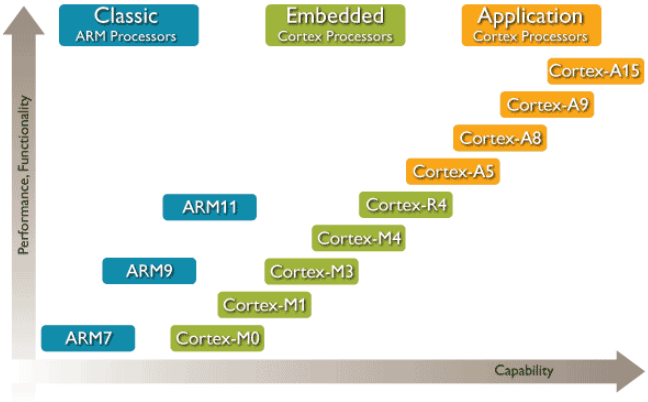
Le bus AHB permet de connecter des périphériques rapides comme les contrôleurs SDRAM, DMA, vidéo,…. Le bus APB permet de s'interfacer avec des périphériques lents comme un clavier, des E/S ne dépassant pas 1Mbit/s

# AMBA Multi-maître

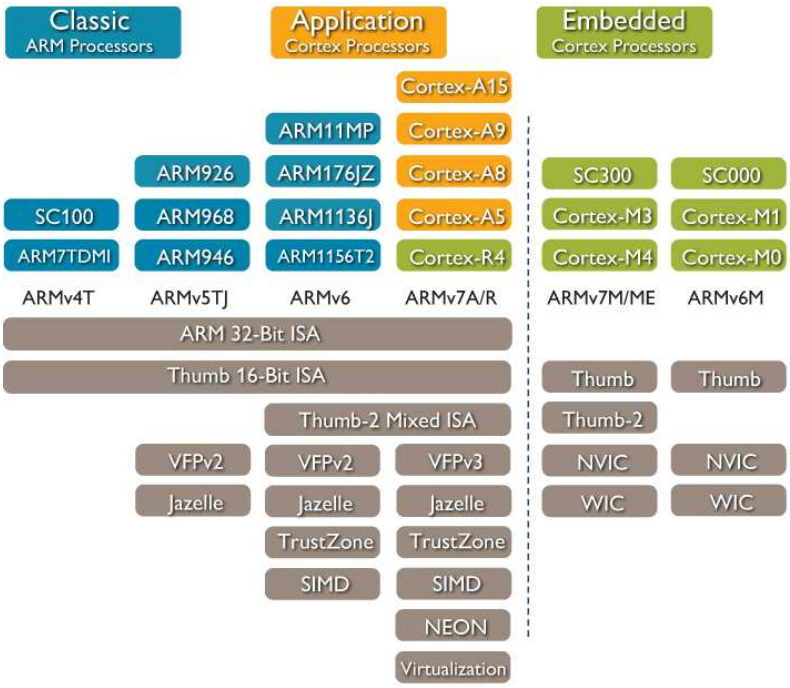


# Les nouveaux cœurs ARM

- **ARM cortex-A**
  - Application généraliste
- **ARM cortex-R**
  - Pour les applications temps réel
- **ARM cortex-M**
  - Microcontrôleur, orienté contrôle

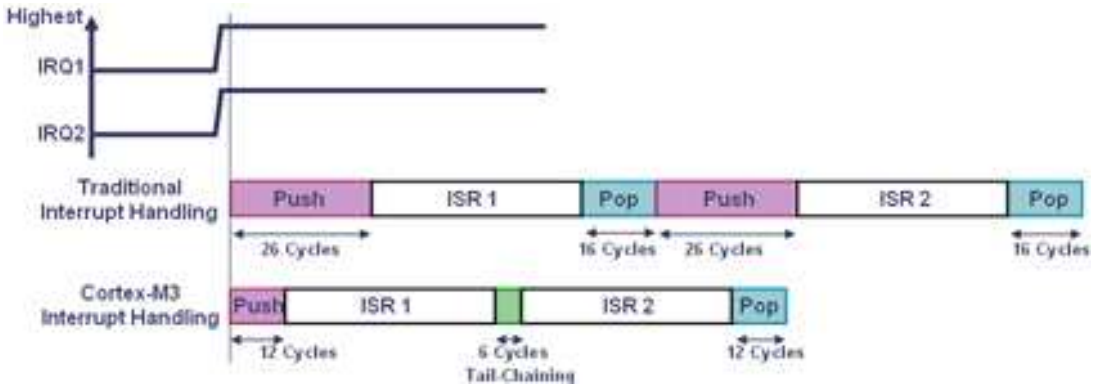


# Architectures des nouveaux cœurs ARM



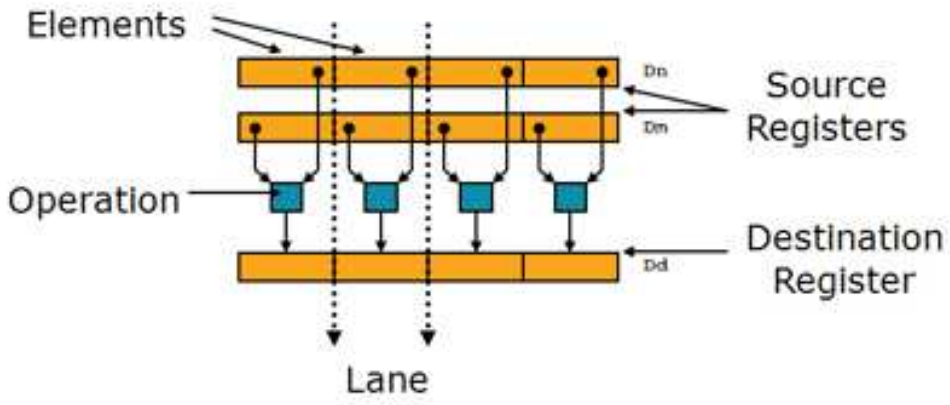
# Nested Vectored Interrupt Controller

- **Gestion des Push/Pop automatique**

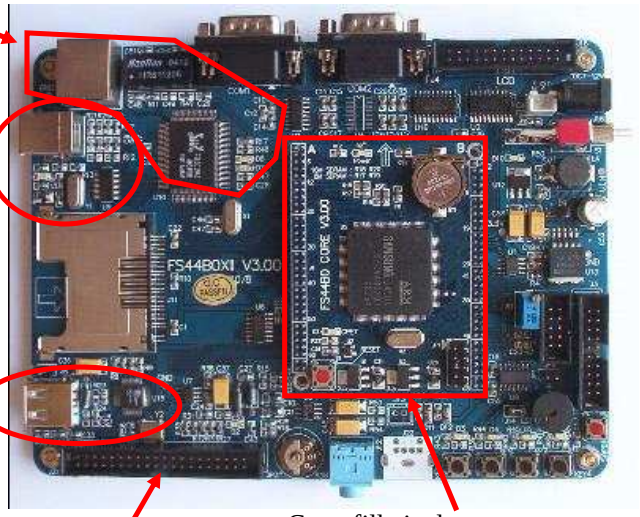


# NEON

- **Coprocesseur pour calcul vectoriel**



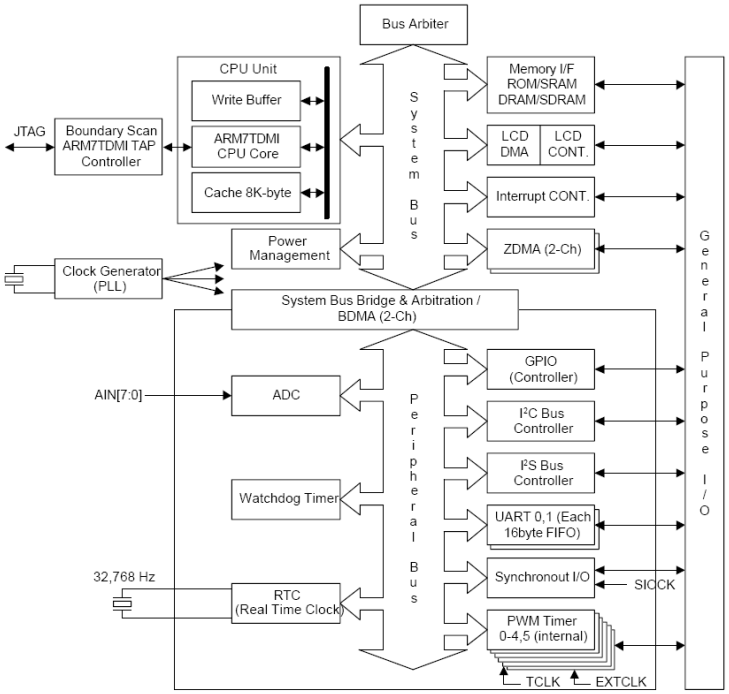
# La Carte FS44 BOX II






La carte dispose d'un processeur SAMSUNG S3C44BOX. Le système est articulé autour d'un noyau uClinux.

# Circuit SAMSUNG S3C44BOX






Le circuits SAMSUNG S3C44BOX dispose de nombreuses interfaces utiles à un système embarqué. C'est sur ce processeur que les TDs vont s'appuyer.