Architecture des Systèmes Intégrés

Ce document regroupe les notes prises lors des cours magistraux de l'unité d'enseignement Architecture des Systèmes Intégrés. Cette UE était enseignée au premier semestre de l'année universitaire 2004-2005 par Pirouz Barzagan Sabet au sein de l'université Paris VI - Pierre et Marie Curie.

Ce document a été rédigé par Jean-Baptiste Voron (alors étudiant en master d'informatique) durant l'année universitaire 2004-2005. Ce document est mis à la disposition des étudiants en informatique qui suivent le même module grâce à l'A.E.I.P.6

Version 06/11/06.(e)

Table des matières

Introduction

Présentation de l'Unité d'Enseignement

L'architecture matérielle systèmes informatiques a connu une évolution foudroyante au cours des 20 dernières années. Cette évolution est principalement dûe à l'augmentation des capacités d'intégration sur silicium, puisque le nombre de transistors intégrables sur une puce double tous les deux ans. Les principes architecturaux présentés dans ce cours s'appliquent donc aussi bien aux ordinateurs "classiques" qu'aux systèmes intégrés sur puce qu'on trouve dans les téléphones portables, dans les machines à laver, ou dans les voitures... On analyse comment et pourquoi on est passé des processeurs CISC microprogrammés des années 70 aux processeurs superscalaires actuels, en passant par les architectures RISC des années 80/90. On présente également les principaux mécanismes matériels dont dépendent les performances des systèmes informatiques : hiérarchies de mémoire, mécanismes de cache, mémoire virtuelle, explotation du parallélisme gros grain grâce aux architectures multi-processeurs.

En se basant sur l'exemple du processeur Mips R3000, on analyse les causes de l'apparition des architectures RISC pipelinées et leurs différences par rapport aux architectures CISC microprogrammées. On détaille le principe du pipeline synchrone et l'impact de cette technique sur le compilateur. On présente ensuite le principe des processeurs superscalaires. Une seconde partie du cours porte sur l'organisation de la hiérarchie de mémoires et les mémoire cache. Les principaux mécanismes des bus système sont illustrés par l'exemple du PiBus, dans le contexte des systèmes intégrés sur puce. Puis, le cours aborde le fonctionnement des machines multiprocesseurs à mémoire partagée et les problèmes liés à la synchronisation et à la cohérence de la mémoire.

- Cours de Monsieur Pirouz Barzagan-Sabet

Chapitre 1

Quelques rappels…

Avant de se « plonger » dans les profondeurs du fonctionnement d'un système intégré, quelques rappels concernant le processeur étudié en licence sont nécessaires. Au cours de ce chapître seront développées les notions d'architecture et de réalisation d'un processeur, ainsi que tous les principes et mécanismes de base ayant trait au processeur MIPS R3000.

1. Réflexion préliminaire

Il faut faire une première distinction entre les termes « **architecture** » et « **réalisation** ». L'architecture, aussi appelée « **vue externe** », est le point de vue de l'utilisateur (du programmeur), alors que la réalisation concerne le concepteur du microprocesseur. Cette dernière est aussi appelée « **vue interne** »

L'architecture du processeur regroupe donc :

- Le jeu d'instruction
- La gestion mémoire
- Les registres
- Reset / Interruption / Exception

Ces quatre points définissent **l'architecture d'une machine**

2. Rappel sur le MIPS R3000

Développé par le professeur **J.Hennesy** pour l'université de Stanford, le processeur MIPS peut être classé dans la famille des **processeurs RISC**

A. *Les registres visibles du logiciel*

Dans cette machine 32 bits, on compte 32 registres *(Integer Register)*. Le contenu de ces registres est donc un **nombre entier**. Ils sont désignés de la façon suivante : `R0 ... R31`

On distingue parmi ces registres :

- R0 : Registre poubelle (Trash Reg) : car R0 = 0 (en permanence)
- R31 : Registre de lien (Link Register) : ce registre contient l'adresse de retour lors de l'appel à un sous-programme.

On ajoute, à ces 32 registres, **Lo** et **Hi** pour effectuer des multiplications et des divisions
Dans le cas d'une multiplication, le registre **Lo** contient les 32 bits de poids faible du résultat alors que le registre **Hi** contient les 32 bits de poids fort. Dans le cas d'une division, les registres Lo et Hi se partagent le reste et le quotient de la division.

On ajoute encore **4 registres spéciaux** :

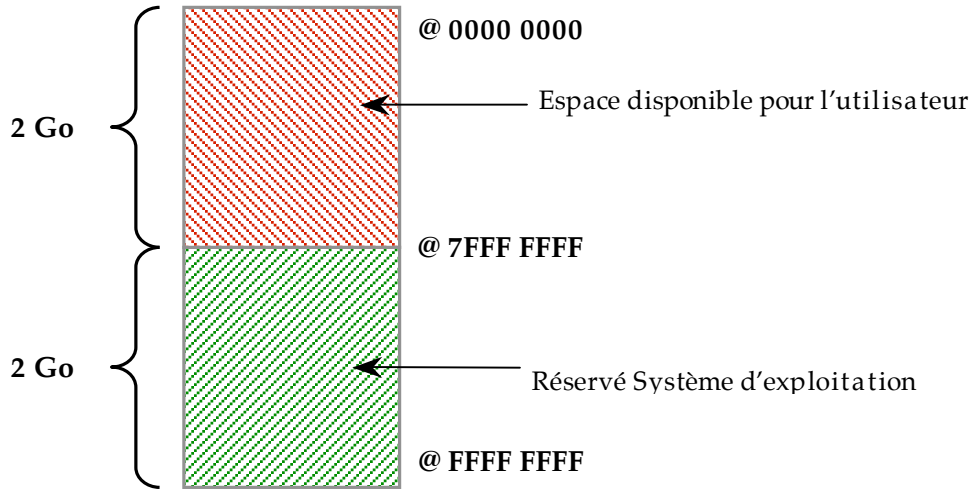
- SR : Status Register : position entre mode user et mode système
- CAUSE : Cause Register : information sur la cause de l'exception / interruption
- EPC : Exception Progam Controler : adresse de retour de l'interruption
- BAR : Bad Adresse Register

Notons qu'aucun registre PC (compteur ordinal) n'est présent !

B. *Gestion de la mémoire*

Les adresses sont contenues dans un registre mémoire et disposent donc de 32 bits.
On peut donc adresser 2^{32} adresses différentes, donc 2^{32} cases mémoires différentes.
Comme une case mémoire contient 1 octet, ces 2^{32} cases représente 4Go de mémoire

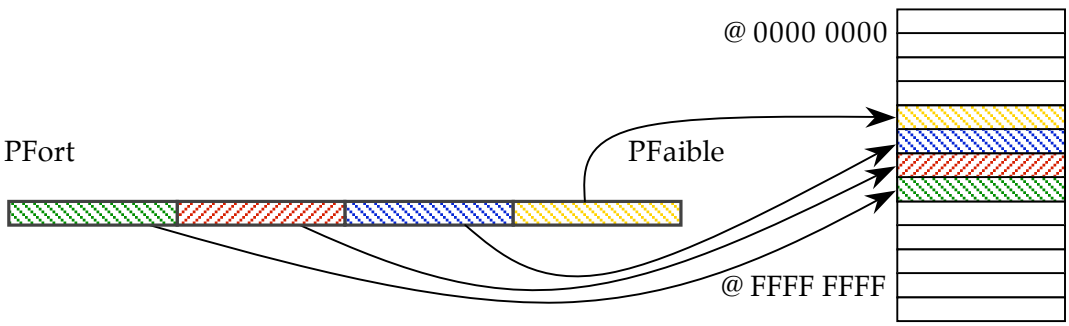
Cette mémoire est construite de la façon suivante :



Le mode de lecture du MIPS permet de lire :

- Octet par octet
- ½ mot par ½ mot (*2 octets*)
- Mot par mot (*4 octets*)

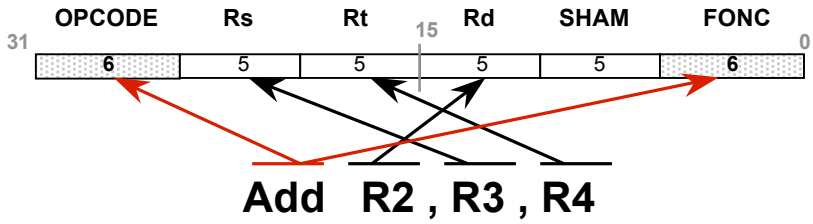
Il faut de plus faire attention à la **convention d' « endianess »** utilisée.
Pour le MIPS on utilise la convention **« little endian »**



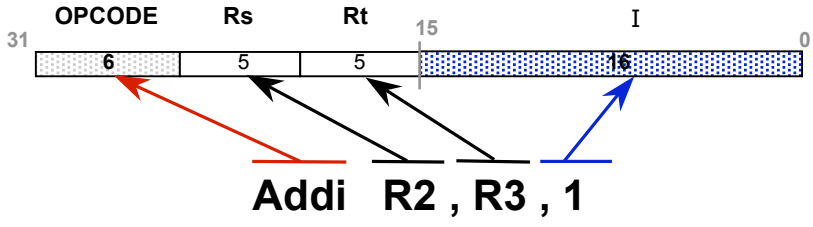
C. Le jeu d'instructions

On distingue 3 instructions : **R, I, J**

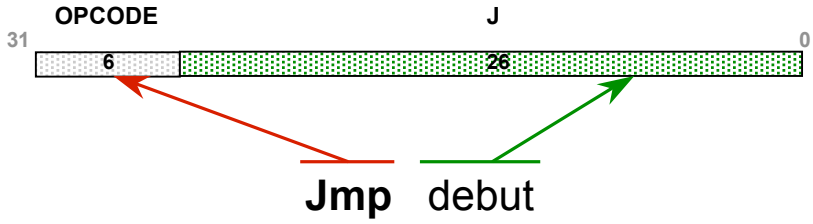
- L'instruction R



- L'instruction I



- L'instruction J



➥ **Les instructions arithmétiques :**

R

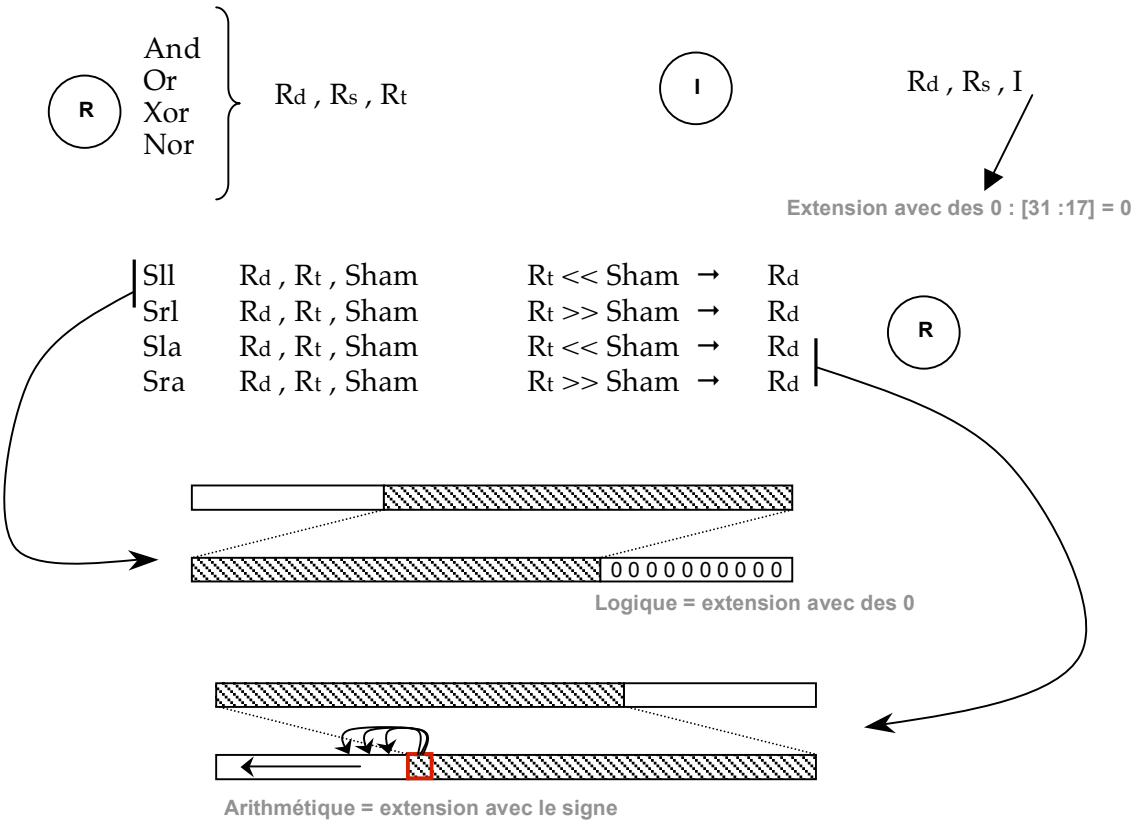
Add	R_d , R_s , R_t	$R_s + R_t \rightarrow R_d$	Détection de l'overflow
Sub	R_d , R_s , R_t	$R_s - R_t \rightarrow R_d$	Détection de l'overflow
Addu	R_d , R_s , R_t	$R_s + R_t \rightarrow R_d$	Pas de détection de l'overflow
Subu	R_d , R_s , R_t	$R_s - R_t \rightarrow R_d$	Pas de détection de l'overflow

I

Addi	R_d , R_s , I	$R_s + I \rightarrow R_d$	
Addiu	R_d , R_s , I	$R_s + I \rightarrow R_d$	→ Pas de detection d'overflow

32 bits — 16 bits donc nécessité de reporter le bit 16 sur les bits [31 :17]

➥ Les instructions logiques :

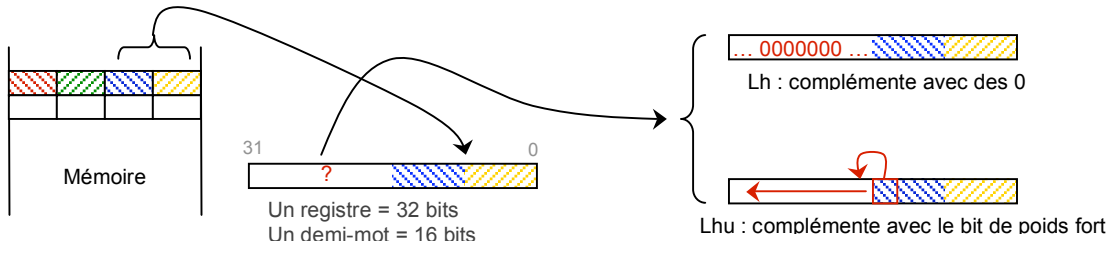


➥ Les accès mémoire:

Comme précisé ci-dessus, les accès mémoires (écritures ou lectures), peuvent se faire par octet (byte), demi-mot (half-word), ou mot entier (word). MIPS implémente donc plusieurs instructions d'accès mémoire.

I

Lw Rd, I (Rs)	M (Rs + I) → Rd	Load Word
Sw Rt , I (Rs)		Store Word
Lh Rd , I (Rs)		Load Half Word
Lhu Rd , I (Rs)		Load Half Word Unsigned
Sh Rt , I (Rs)		Store Half Word

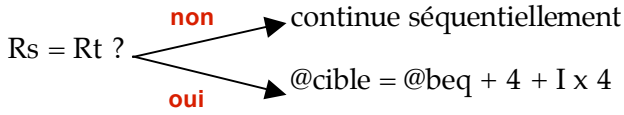


I

Lb Rd , I (Rs)	Load Byte
Lbu Rd , I (Rs)	Load Byte Unsigned
Sb Rt, I (Rs)	Store Byte

➥ Contrôle

- Branchement conditionnel



BEQ R_s , R_t , Label	Branch if EQual	
BNE R_s , R_t , Label	Branch if Not Equal	
BLEZ R_s , Label	Branch if Less or Equal to Zero	
BLTZ R_s , Label	Branch if Less Than Zero	
BGTZ R_s , Label	Branch if Greater Than Zero	
BGEZ R_s , Label	Branch if Greater or Equal to Zero	
BLTZAL R_s, Label	Branch if Less Than Zero And Link	
BGEZAL R_s , Label	Branch if Greater or Equal to Zero And Link	

- Branchement inconditionnel

J Label	Jump
Jr R_s	Jump Register
Jal Label	Jump And Link
Jalr R_s	Jump And Link Register

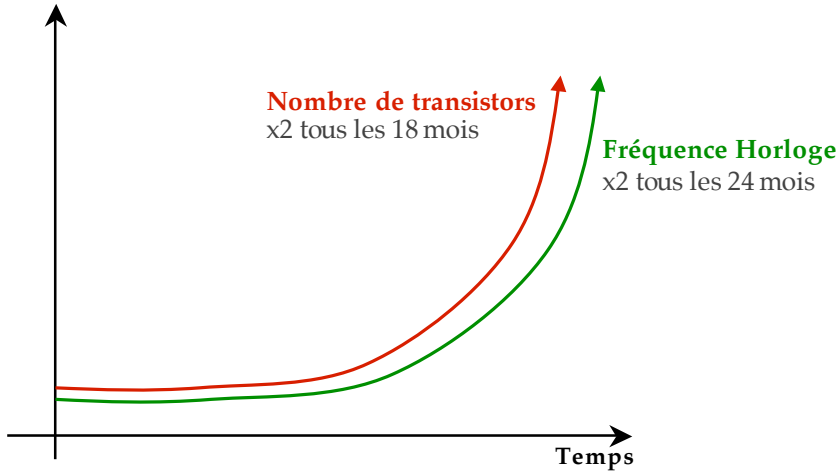
Chapitre 2

Processeurs RISC

Ces rappels concernant précisemment le processeur étudié doivent nous permettre de classer le MIPS R3000 dans une famille qui lui confère certaines propriétés. L'étude de ces dernières nous donnera l'occasion de suivre plus en détail le cheminement de l'information et son traitement au sein du processeur.

1. Familles de processeurs

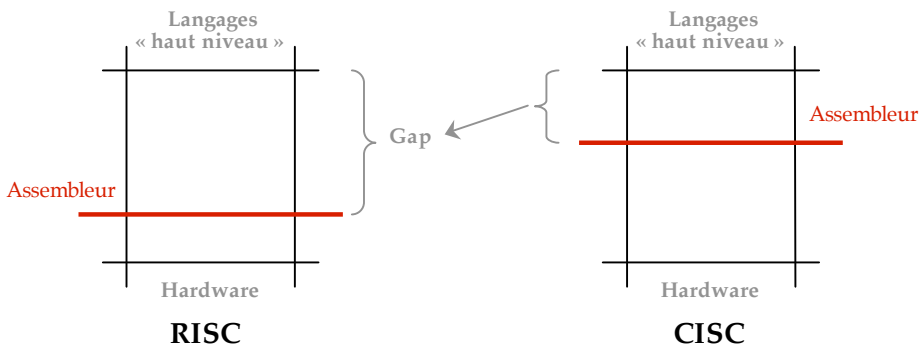
Il faut considérer dans un premier temps la séparation qu'il existe entre les deux grandes familles de processeurs : CISC et RISC. Avant tout, prenons en compte la **loi de Moore**



Cette analyse valable depuis les années 50 permet de considérer le problème des processeurs de manière différente. Nous allons y revenir dans quelques instants.

La particularité qui permet de distinguer un processeur RISC d'un processeur CISC est l'implémentation et la relation qu'il existe entre le matériel et le langage dit « d'assembleur »

On constate en effet, qu'un processeur CISC va tendre à accepter des instructions complètes et complexes, de plus en plus proches d'un langage de programmation dit de « haut niveau » alors qu'un processeur RISC n'acceptera, quant à lui, des instructions très simples et plutôt éloignée des langages de haut niveau.



Considérons l'instruction suivante : **A = B + C**

Cette instruction est bien évidemment utilisable sur un processeur RISC comme sur un processeur CICS, la seule différence venant de sa véritable implémentation et traduction.

Pour un processeur **CISC**, en **VAX** on aura : `Add @A, @B, @C`

Pour un processeur **RISC**, en **MIPS** on aura :
```
Lw R4 , @B
Lw R5 , @C
Add R6 , R4 , R5
Sw R6 , @A
```

Du point de vue du programmeur, le critère qui va permettre de juger si le processeur est performant est la **rapidité d'exécution** du programme. Ainsi, on attend principalement d'un processeur qu'il dispose d'une **fréquence horloge élevée**, mais aussi qu'il soit capable de **traiter le plus d'instruction possible en un cycle**.

Il apparaît clairement que le processeur RISC demande bien plus d'instruction assembleur que le processeur CISC. En effet, dans cet exemple, il faudra 4 instructions en assembleur MIPS contre une seul pour l'assembleur VAX.

Si l'on considère maintenant le diagramme de la loi de Moore, on se rend compte que la puissance des processeurs est devenue telle, que ce facteur 4 est bien rapidement absorbé.

Le principal inconvénient du processeur RISC est donc qu'il demande une **mémoire plus importante** afin de stocker les diverses instructions. Dans cet exemple, le processeur CISC aura besoin de 8 octets alors que le processeur MIPS en demandera 16.

Cependant, si l'on considère la situation actuelle du marché des composants informatiques, la mémoire ne pose plus aucun problème. La considération précédente perd donc tout son poids et ne permet plus de faire pencher la balance d'un coté ou de l'autre.

Les processeurs RISC ont donc largement leur place dans la grande famille des processeurs.

On note la **performance d'un processeur** :

$$\textbf{Perf} = \frac{\textbf{F}}{\textbf{CPI}}$$

= #cycles nécessaire pour traiter une instruction

Les considérations précédentes, nous montrent que pour que le processeur soit efficace (perf = 1) , il faut que la réalisation permette de tendre vers la limite où **CPI = 1** c'est à dire ou **chaque instruction est exécutée en un cycle.**

Le processeur doit être conçu avec le matériel permettant de satisfaire cette condition !

2. Le langage « machine »

Détaillons les opérations nécessaires pour l'exécution des instructions suivantes :

- **Add :**
  - Lire l'instruction
  - Décoder l'OPCODE
  - Lire les opérandes (R_s, R_t)
  - Opération (+)
  - Ranger le résultat
  - Calculer l'adresse de l'instruction suivante

Schéma d'éxecution

- **Lw :**
  - Lire l'instruction
  - Décoder l'OPCODE
  - Lire les opérandes (R_s, I)
  - Calcul de l'adresse (R_s + I)
  - Accès mémoire pour lire la donnée
  - Range le résultat (R_d)
  - Calculer l'adresse de l'instruction suivante

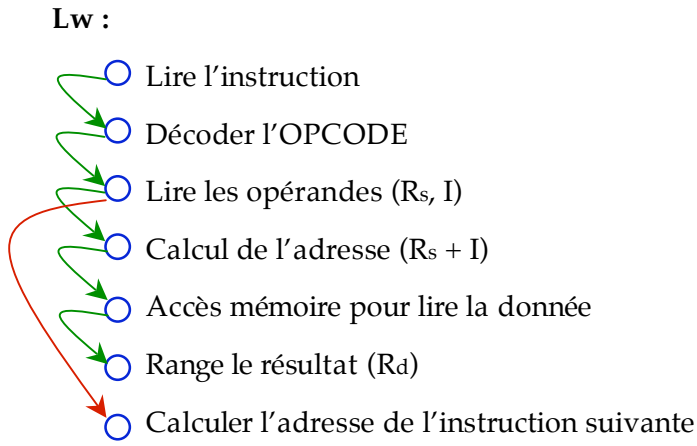
- **Sw :**
  - Lire l'instruction
  - Décoder l'OPCODE
  - Lire les opérandes (R_s, R_t, I)
  - Calcul de l'adresse (R_s + I)
  - Accès mémoire pour écrire la donnée
  - Calculer l'adresse de l'instruction suivante

- **Jr :**
  - Lire l'instruction
  - Décoder l'OPCODE
  - Lire les opérandes (R_s)
  - Calculer l'adresse de l'instruction suivante

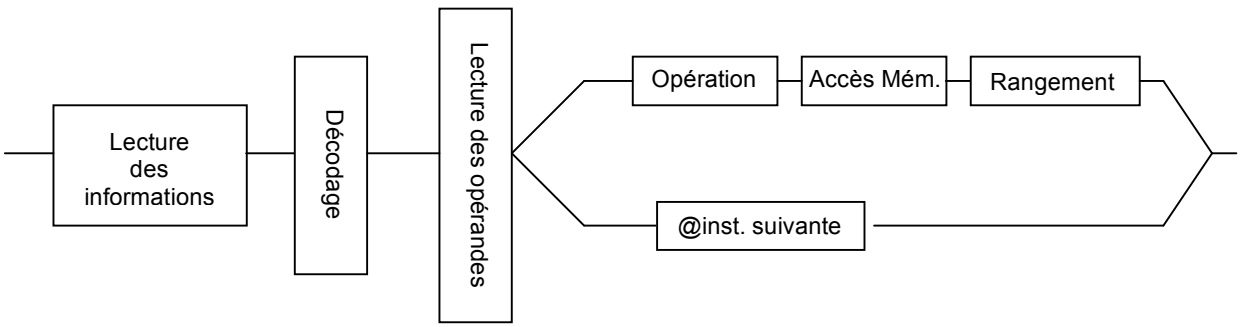
N'importe quelle instruction est donc clairement traductible en une **suite finie d'opérations**. Et la principale caractéristique d'un processeur RISC est que chacune de ces instructions rentre dans **le même schéma d'exécution**.

Voilà donc ce que l'on peut appeler le « **Set d'Instructions** » du processeur.

Si l'on choisit maintenant de déterminer si **l'ordre des opérations élémentaires** est important, nous sommes amenés à construire un **graphe de dépendances** :



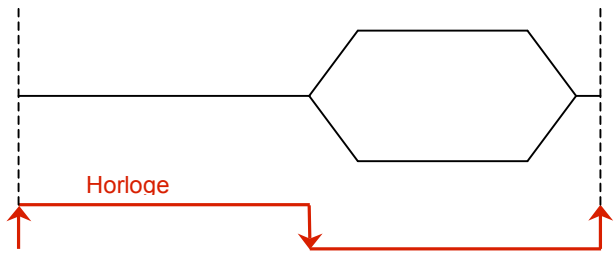
On distingue clairement deux graphes distincts.
Ces graphes vont tout simplement définir les **deux chaînes d'exécution** réalisée dans le MIPS :



Il est bon de rappeler que l'objectif est de faire en sorte que toutes les instructions suivent le même schéma d'exécution et trouve par conséquent un chemin dans le schéma présenté ci-dessus.

Remarquons qu'une instruction n'est pas obligée de suivre toutes les étapes. L'instruction Add par exemple ne fait pas d'accès mémoire, mais on considère qu'elle respecte quand même le schéma d'exécution.

En ne perdant pas de vue notre objectif, nous pouvons maintenant adapter un signal d'horloge sur le schéma précédent.



Le CPI est effectivement d'un cycle par instruction. La performance est donc de

Perf = F / CPI = F / 1 = F instruction / sec

Même si le CPI est à 1, le nombre d'opérations à effectuer lors du traitement d'une instruction est conséquent et bride la fréquence du processeur. Notre nouvel objectif est donc d'augmenter cette fréquence

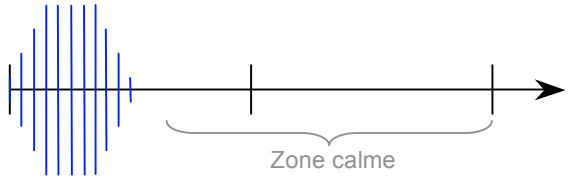
Chapitre 3

Pipelining 1ère Partie

L'étude de la progression d'une instruction dans un processeur amène logiquement une question d'optimisation. Comment atteindre les objectifs de permformances fixés dans le chapitre précédent ? Quelles méthodes ou stratégies pouvons-nous mettre en place pour améliorer et accélerer l'accès des instructions au processeur.

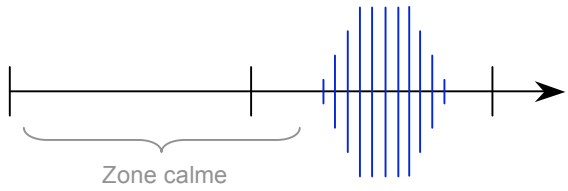
1. Notions de vagues et de propagation

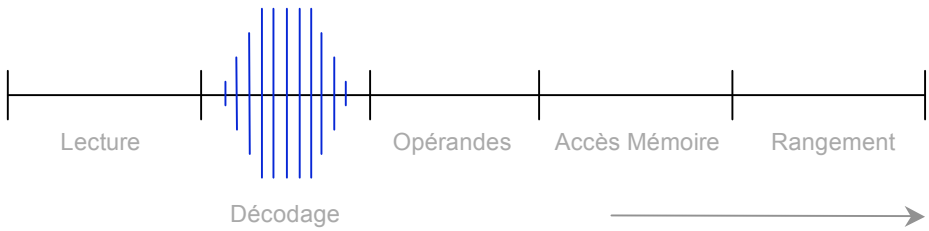
Dans l'exemple proposé ci-contre, on peut constater qu'à un instant «`t`» donné, la vague implique de l'agitation à un endroit bien précis.



Les zones suivantes et précédentes sont calmes.

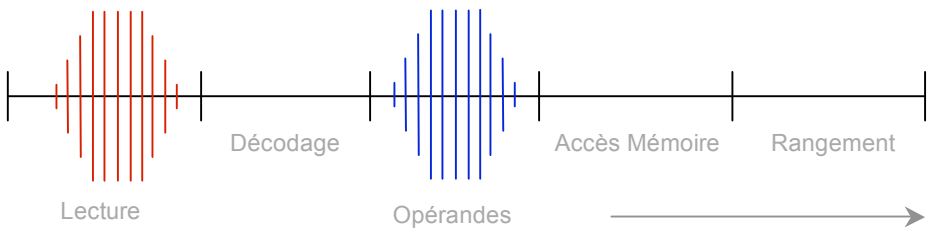
Appliquons maintenant cet exemple de vague à la propagation d'une instruction dans le schéma d'exécution du processeur.





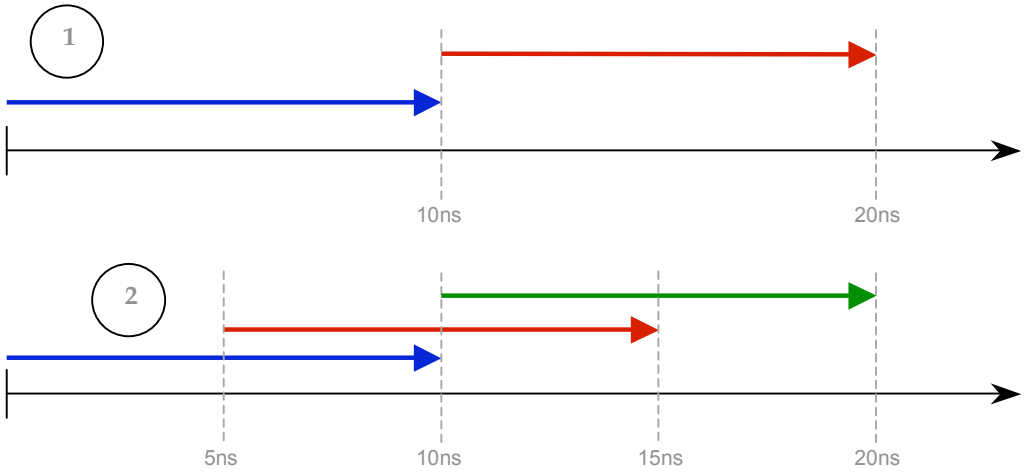
Il apparaît alors clairement, qu'à un instant « `t` », seule une petite partie du matériel est utilisée. En général, on considère que si le processeur utilise ce principe, 90% du matériel reste au repos.

Pour optimiser l'utilisation du matériel, on peut détecter l'avancement de la première vague (instruction) et injecter une nouvelle vague (instruction) dans le schéma d'exécution.



Notre objectif est donc d'utiliser le même matériel pour traiter des flots d'instructions.

En supposant que nous sachions « fabriquer » de tels flots d'instructions, nous pouvons déjà calculer le gain engendré :



Le processeur traite plus d'instructions dans le même laps de temps :

- **Dans le cas n°1** : Chaque instruction nécessite **10ns.**
  Le débit du matériel est de **1 inst/10ns**

- **Dans le cas n°2** : Chaque instruction nécessite **10ns**
  Le débit du matériel est de **1 inst/5ns**

On arrive alors a exécuter 3 instructions en 20ns au lieu de 2 ce qui représente un gain de **50%**

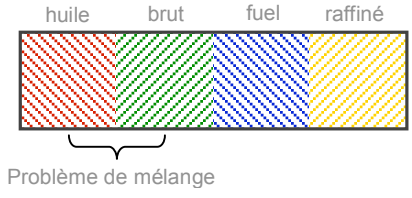
Notons qu'a aucun moment, les deux instructions ne doivent se rencontrer. Elles doivent en permanence utiliser des ressources distinctes. Il faut donc trouver un moyen d'isoler chacune de ces instructions.

2. Le Pipeline

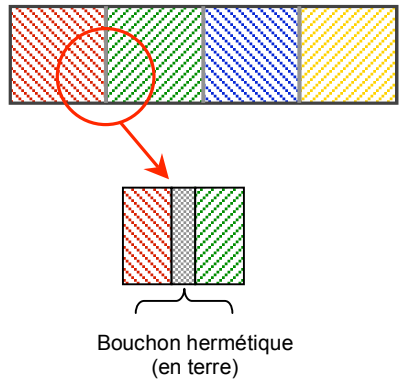
A. *La technique*

On utilisera donc la **technique du pipeline** inventée par IBM en 1975.
Cette technique a été empruntée au domaine pétrolier qui utilise des matériels de même nom.
Toute la problématique d'un tel dispositif est de faire circuler un maximum d'éléments (dans ce cas-là : hydrocarbures, huiles, pétroles etc…) dans un seul et même conteneur.

On pourrait alors imaginer envoyer successivement chacun de ces éléments de la façon suivante :



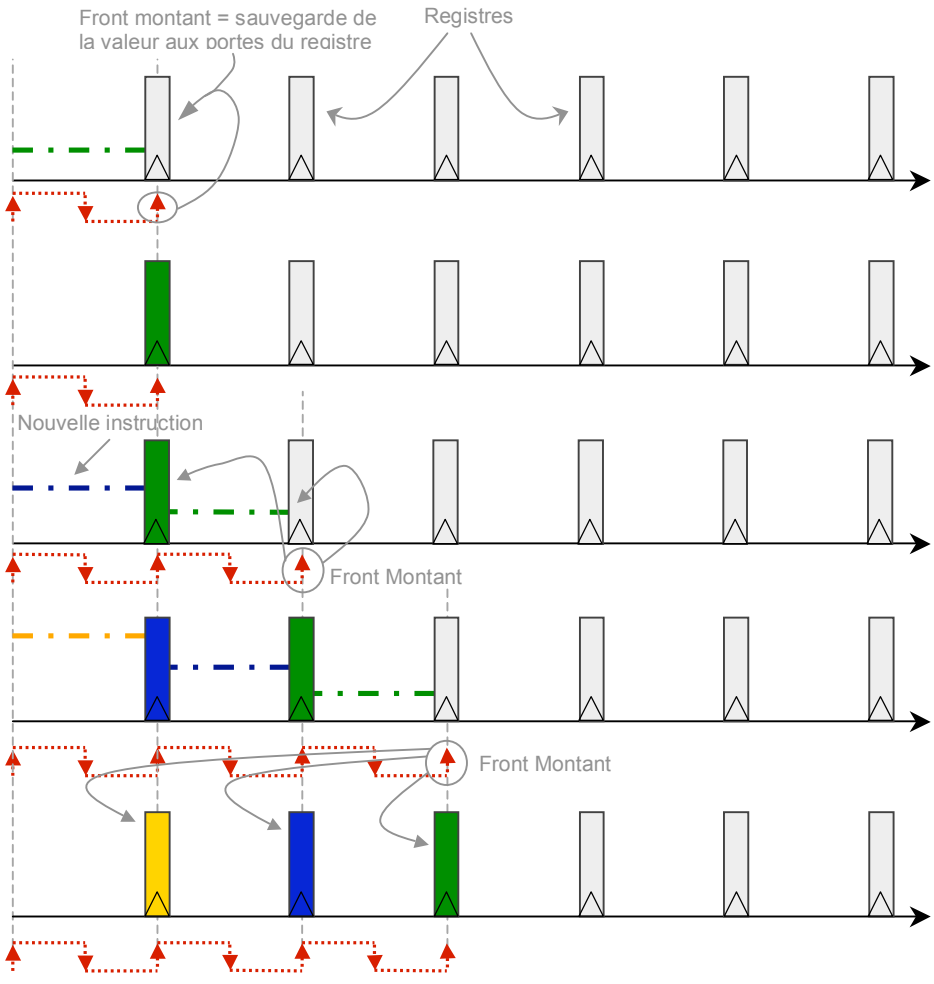
Mais il apparaît clairement un problème aux interfaces. En effet, rien empêche les différents éléments de se mélanger, ce qui demanderait au final un effort conséquent pour les séparer.
La technique finalement retenue par les industriels consiste à injecter un bouchon hermétique entre chaque élément.



B. *L'implémentation*

En informatique, on utilise la même technique , en remplaçant, cependant, les bouchons de terre par des registres.

Ce sont ces derniers, qui feront office de séparation hermétique entre les diverses vagues de données (instructions)



Un même processeur, entrecoupé de registres, peut donc accueillir plusieurs instructions en même temps ; chaque partie du processeur continuant à traiter une seule instruction à la fois.
Cette technique garantie « l'isolation » de chacune des instructions.

Notons que le temps de traitement d'une instruction reste toujours 10ns.
Ce temps est appelé **temps de latence**.

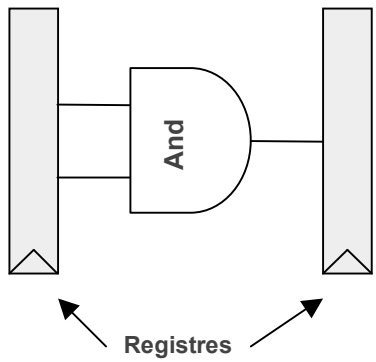
La seule différence avec le modèle classique du processeur, est qu'une instruction peut commencer à être traitée **alors que la précédente n'a pas fini, elle, d'être traitée**…

- Plus on rajoute de registres
- Plus on découpe le processeur
- Plus le processeur peut traiter un grand nombre d'instructions différentes
- Plus la fréquence de traitement augmente.

On en déduit donc que le pipeline augmente le débit de traitement des instructions mais en aucun cas le temps de latence

Nous avons vu précédemment qu'un découpage fin du processeur impliquait une augmentation de la fréquence de ce dernier. Ainsi, on peut découper notre processeur MIPS en 2, 4, 6, 10 ou 100 parties.

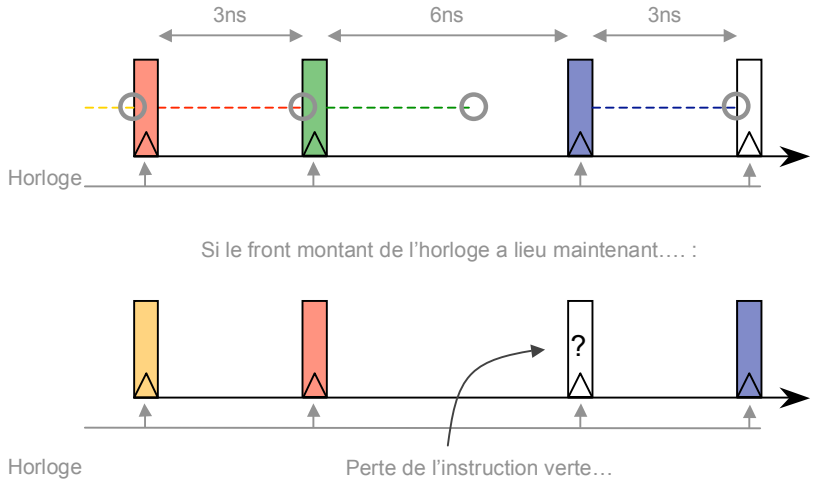
La seule limite physique que l'on puisse rencontrer, se trouve au niveau des portes logiques.
En effet, on ne pourra pas insérer un registre à l'intérieur même d'une porte logique.



Il faut toutefois faire attention au découpage que l'on réalise. Quelques règles doivent être satisfaites pour éviter tous les problèmes.

Échantillonnage

Nous avons vu précédemment, que les registres utilisaient le front montant de l'horloge pour « photographier » les valeurs qui se trouvent à leur entrée à ce moment-là. Tous les registres réagissant au même front d'horloge, les instructions doivent toutes être arrivées aux portes des registres au moment du front montant de l'horloge. Les données seraient perdues le cas échéant.



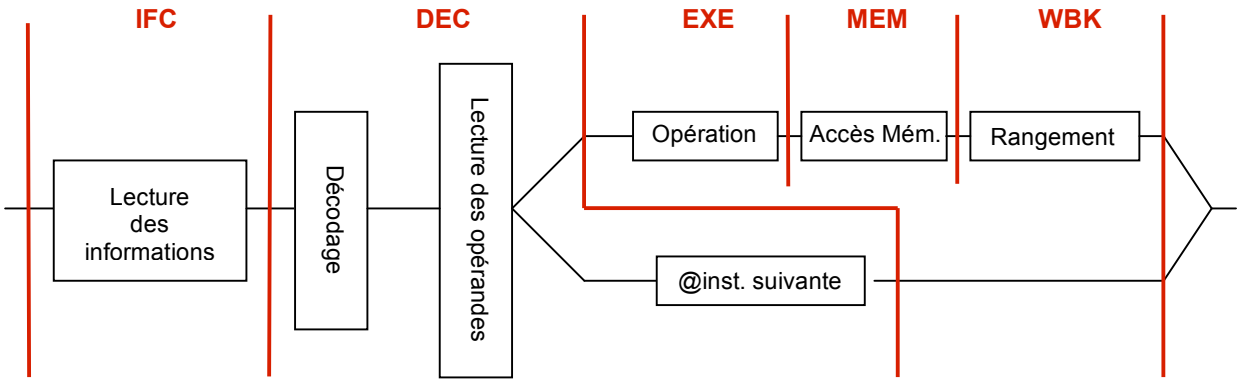
Cette petite « démonstration » nous permet donc de conclure que l'horloge doit être calquée sur le plus long des parcours entre deux registres.

3. Les règles d'un pipeline

Des remarques précédentes, on peut déduire les deux premières règles d'or de la conception d'un processeur utilisant la technique pipeline

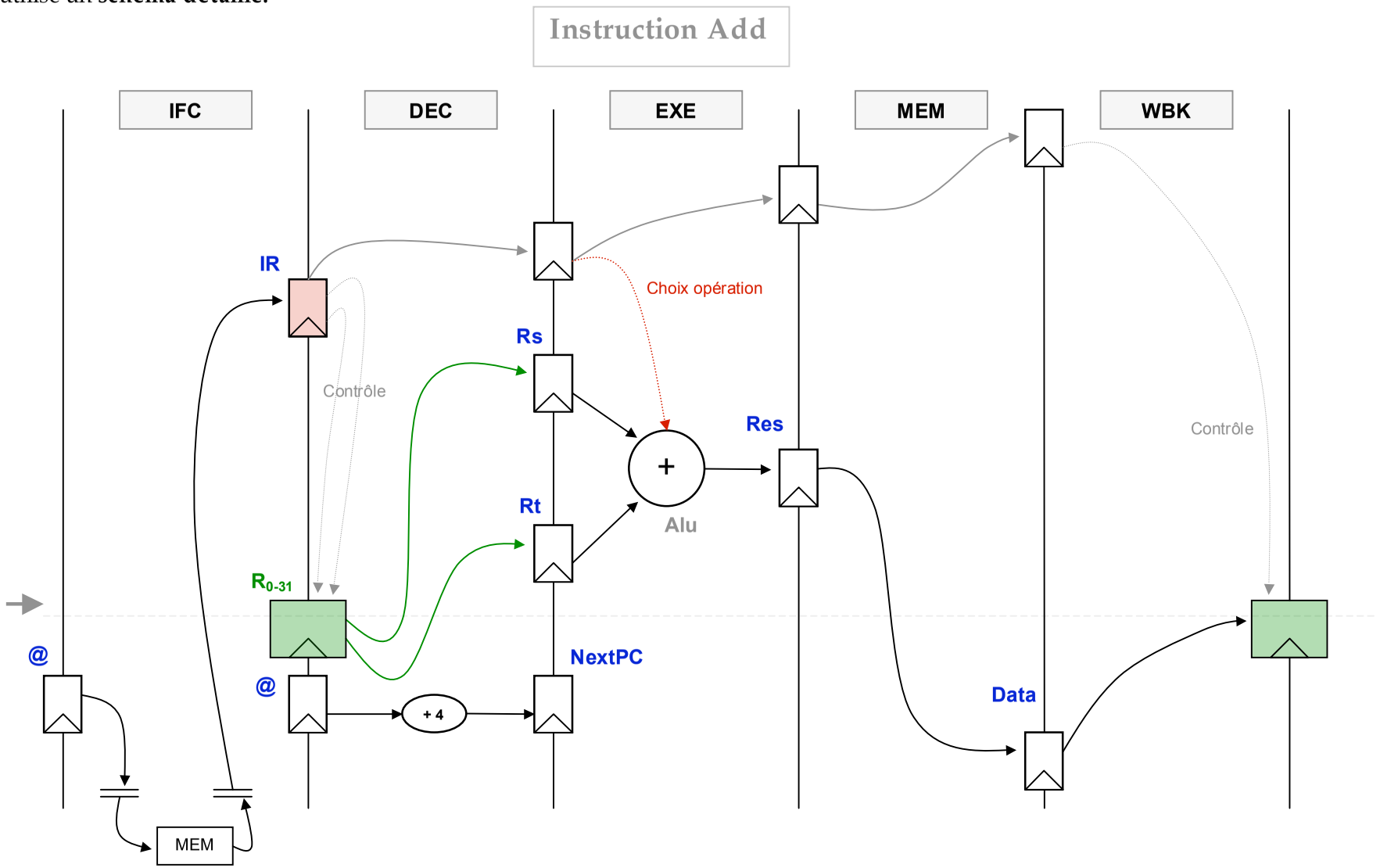
★ **1° Le pipeline doit être équilibré. (Les chemins entre registres doivent être équivalents)**
★ **2° Chaque étape et opération doit être séparée par un registre**

Reprenons désormais le schéma d'exécution vu précédemment, et tentons d'étudier le découpage proposé initialement par les équipes de concepteurs.



IFC : Instruction Fetch
DEC : Insctruction Decode
EXE : Execute
MEM : Memory Access
WBK : Write Back

Pour suivre plus facilement le cheminement d'une instruction dans un processeur utilisant la technique du pipeline, on utilise un **schéma détaillé.**



Une lecture verticale de ce schéma nous donne le déroulement de l'exécution dans un « compartiment » du processeur. Une lecture horizontale, nous donnera au contraire, l'utilisation d'un matériel.

Si l'on considère la flèche grisée (en bas à gauche), nous pouvons constater que la ligne du banc de registre n'est occupée que par ce matériel. De même pour tous les autres registres représentés sur ce schéma.

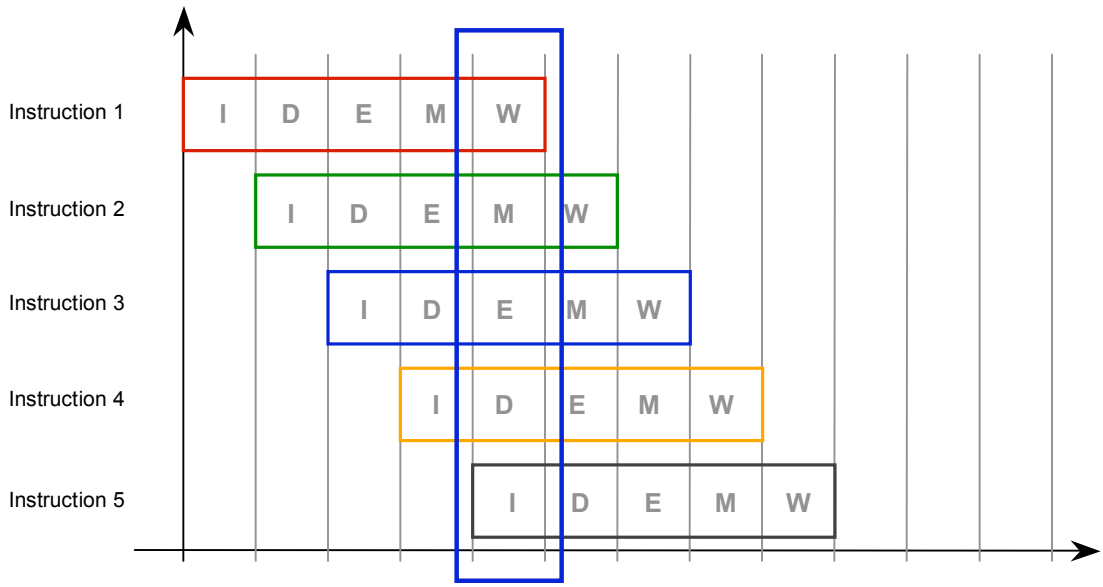
Cette particularité, nous permet de garantir le respect de la 3° règle d'or :

★ 3° Chaque matériel appartient à un cycle unique.

Sur le schéma précédent, on retrouve les 5 « compartiments » d'exécution que l'on a définis précédemment. L'instruction passe donc au fil des fronts montants de l'horloge de compartiment en compartiment.

L'optimisation vient du fait que chaque compartiment *(à un moment donné)* traite le bout d'une instruction différente. Cette caractéristique peut se représenter à l'aide du schéma suivant :

On distingue alors la vision du processeur à un moment donné ainsi que le déroulement d'une instruction donnée dans le temps. Notons aussi que la troisième règle d'or des processeurs pipeline est bien vérifiée.

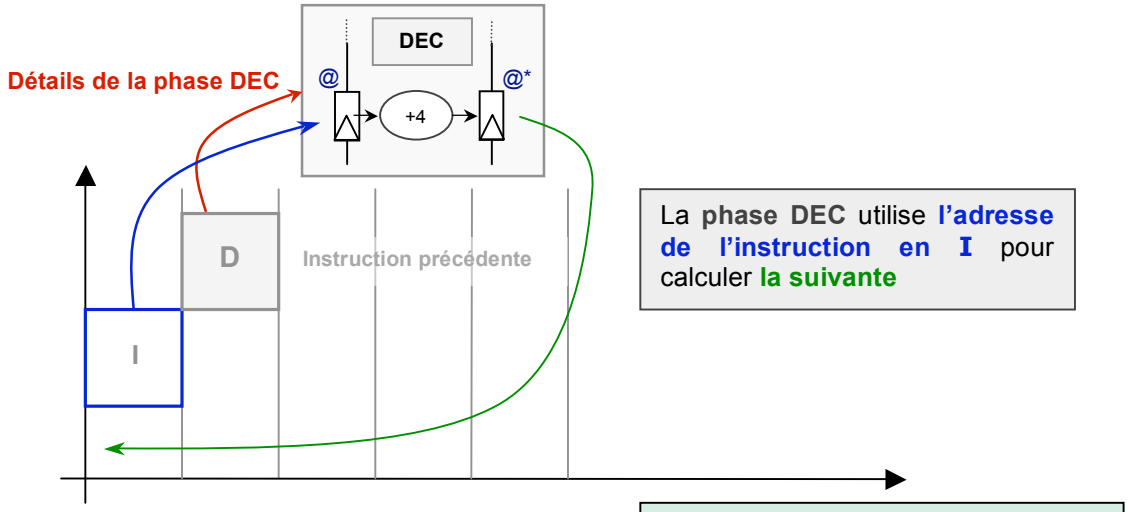


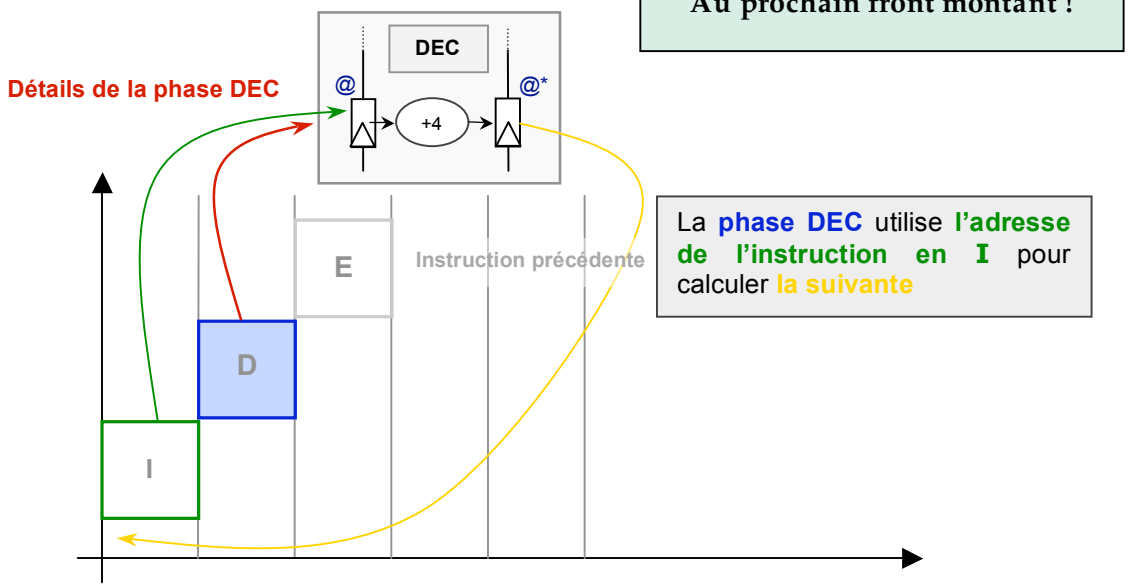
Vision Processeur

4. Détails de l'implémentation

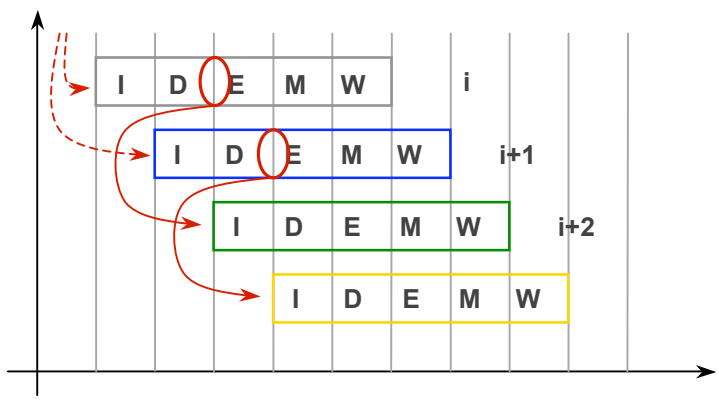
A. *Calcul des adresses*

Avant de poursuivre l'étude de cette implémentation, faisons un point sur le calcul de l'adresse de l'instruction suivante.





En résumé, voici les interactions entre les phases DEC et les chargements d'instruction :

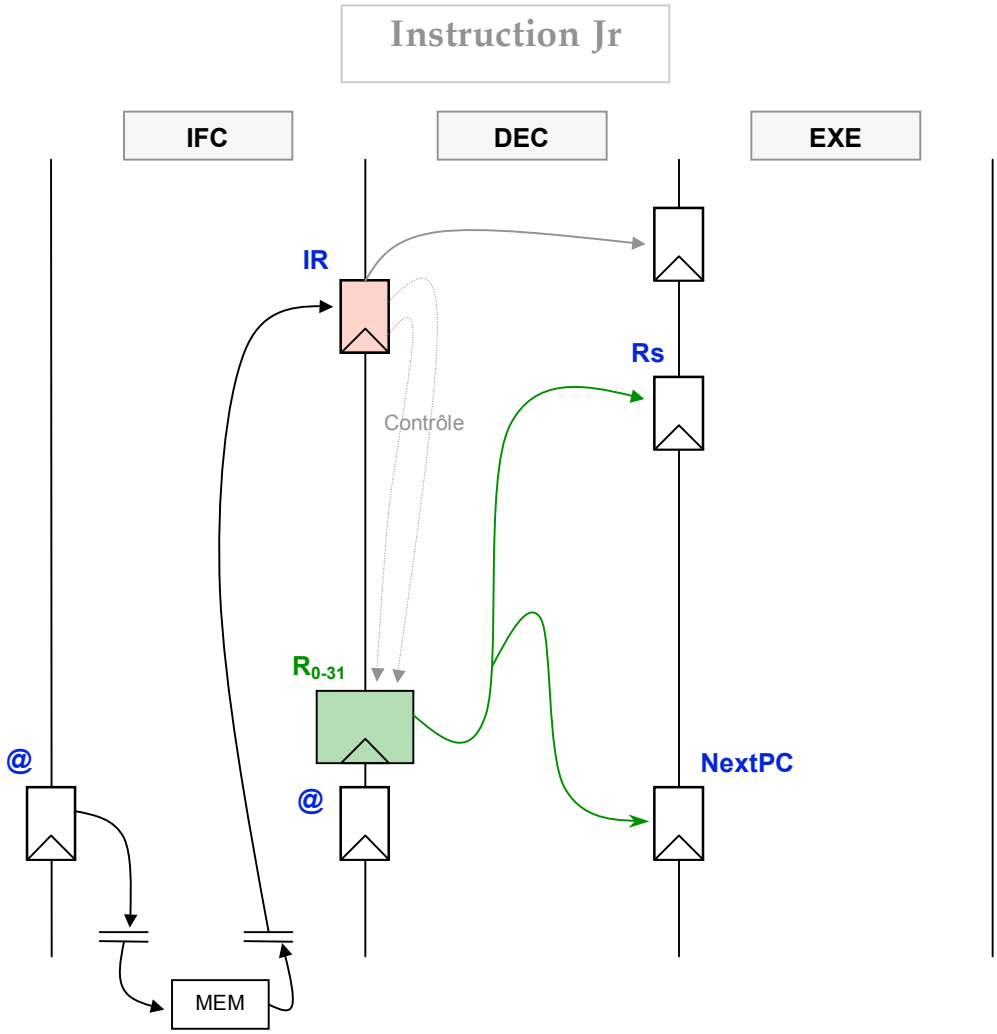


L'adresse est disponible à la fin de la phase DEC de l'instruction **i**. Cette adresse est celle de l'instruction **i+2**

À noter ! L'adresse est toujours disponible à la fin de la phase DEC

B. *Les instructions de branchement*

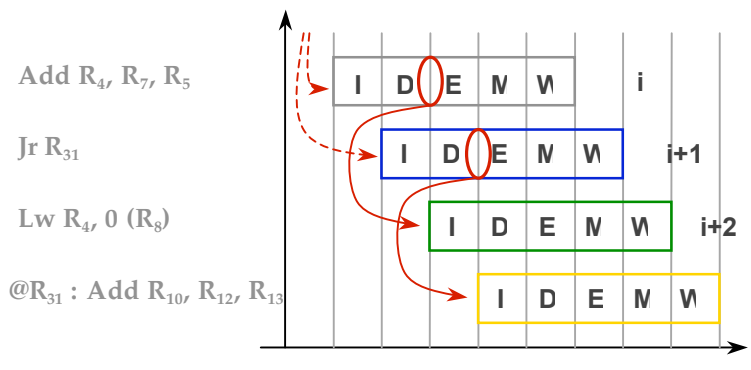
Le schéma proposé est intéressant sur beaucoup de points, mais **révèle une faiblesse** lorsqu'il s'agit de traiter **les instructions de branchement**.
Représentons dans un premier temps le **schéma détaillé** d'une instruction de branchement.



Nous remarquons encore une fois que l'adresse de l'instruction suivante **(i+2)** est disponible à la fin de la phase de décodage.

Reprenons maintenant le schéma précédent (page 8) avec la suite d'instructions suivante :

```
debut:   Lw R2, 0 (R6)
         Add R4, R7, R5
         Add R8, R3, R4
         Jr R31
boucle:  Lw R4, 0 (R8)
         Sub R5, R7, R8
         Sll R5, R5, 2
         ...
@R31:    Add R10, R12, R13
         Sw R10, 0 (R9)
```



L'exécution de ce programme **devrait** produire la séquence :

```
Lw R2, 0 (R6)
Add R4, R7, R5
Add R8, R3, R4
Jr R31
Add R10, R12, R13
```

Or le schéma d'exécution nous montre l'apparition de l'instruction **Lw R4, 0 (R5)** qui se trouve être celle qui suit le branchement « `Jr` ». En effet, notre système de calcul, ne nous permet pas de **détecter le branchement**, et le branchement ne sera donc effectif qu'après la phase DEC de l'instruction « `Jr` »

▶ **Que faire ?**
Il y a très clairement, dans le pipeline, une instruction qui n'a rien à y faire !

Deux solutions sont envisageables :

1° Nous pouvons « tuer » l'instruction dans le pipe.

CISC

☑ Aucun problème pour le programmeur

☒ Implémentation matérielle (hardware) assez complexe

2° On l'exécute quand même

RISC

☑ Rien à faire au niveau matériel

☒ Le programmeur (compilateur) doit anticiper le branchement.
Agencement des instructions pour que « l'intruse » ne pose pas de problème.

Nous choisissons évidemment la deuxième solution. ;o)

Le compilateur devra donc veiller à ce que l'instruction suivant le branchement ne touche pas aux registres utilisés et ne provoque pas d'autres réactions inattendues.
Il existe deux méthodes pour gérer un tel travail :

Insertion d'un Nop | Réagencement du code

Dans le cas du réagencement, l'instruction choisie est une instruction non dépendante des suivantes et inversement.
Dans le cas présenté ci-dessus, l'instruction **Lw R4, 0 (R8)** aurait pu être remplacé par l'instruction **Lw R2, 0 (R6)**

Chapitre 4

Pipelining 2ème Partie

La technique du pipeline mise en avant dans le chapitre précédent constitue très certainement une optimisation interessante pour le processeur étudié. Cependant, un certain nombre de problèmes reste non abordé et les objectifs proposés en début de cours ne sont toujours pas atteint.

1. Analyse de l'étage DEC

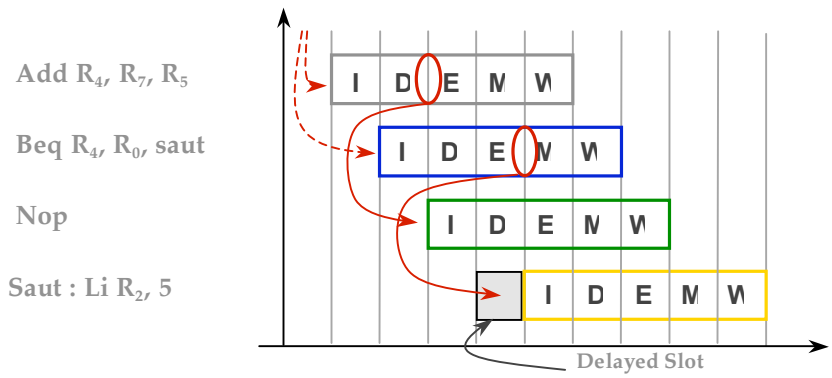
A. *Le problème*

Nous avons vu dans le cours précédent que la nouvelle adresse était toujours disponible à la fin de l'étage de décodage de l'instruction (DEC). Dans certains processeurs pipeline, le calcul d'adresse se fait dans une « tranche » décalée :EXE .

Nous avons vu, dans le cas de branchement conditionnel, que l'expression d'égalité ou d'inégalité devait être vérifiée dans l'étage DEC pour que l'adresse soit disponible à la fin de la « tranche » comme le préconise et l'oblige la règle de ce découpage.

Décaler le calcul de cette adresse permettrait ainsi de libérer du matériel (comparateur, additionneur etc…) dans la couche DEC .

A contrario, l'adresse n'étant disponible qu'en EXE, l'instruction suivante ne pourra pas se charger aussi rapidement. En d'autres termes, le processeur sera obligé d'attendre cette adresse pour charger la suivante.



Ces slots libres sont optimisables par le processeur.
Nous avons vu dans le cours précédent, qu'il existe une technique de réagencement de code qui permet d'utiliser des instructions antérieures à la boucle, sans aucun rapport avec cette dernière, à la place de «°`nop`°».

Cette technique est là encore applicable, mais il faut néanmoins tenir compte des capacités des compilateurs à mettre en œuvre efficacement cette méthode.

Nombre de « delayed slots » successifs	Taux de réussite du réagencement
1	75%
2	5%
3	0%
4	0%

Très vite, l'insertion d'instruction « `nop` » devient nécessaire, ce qui provoque alors une baisse du CPI utile. En guise d'application, on pourrait se demander qu'elle serait l'implémentation de l'instruction de branchement conditionnel **« `Beq Rs, Rt, Label` »**

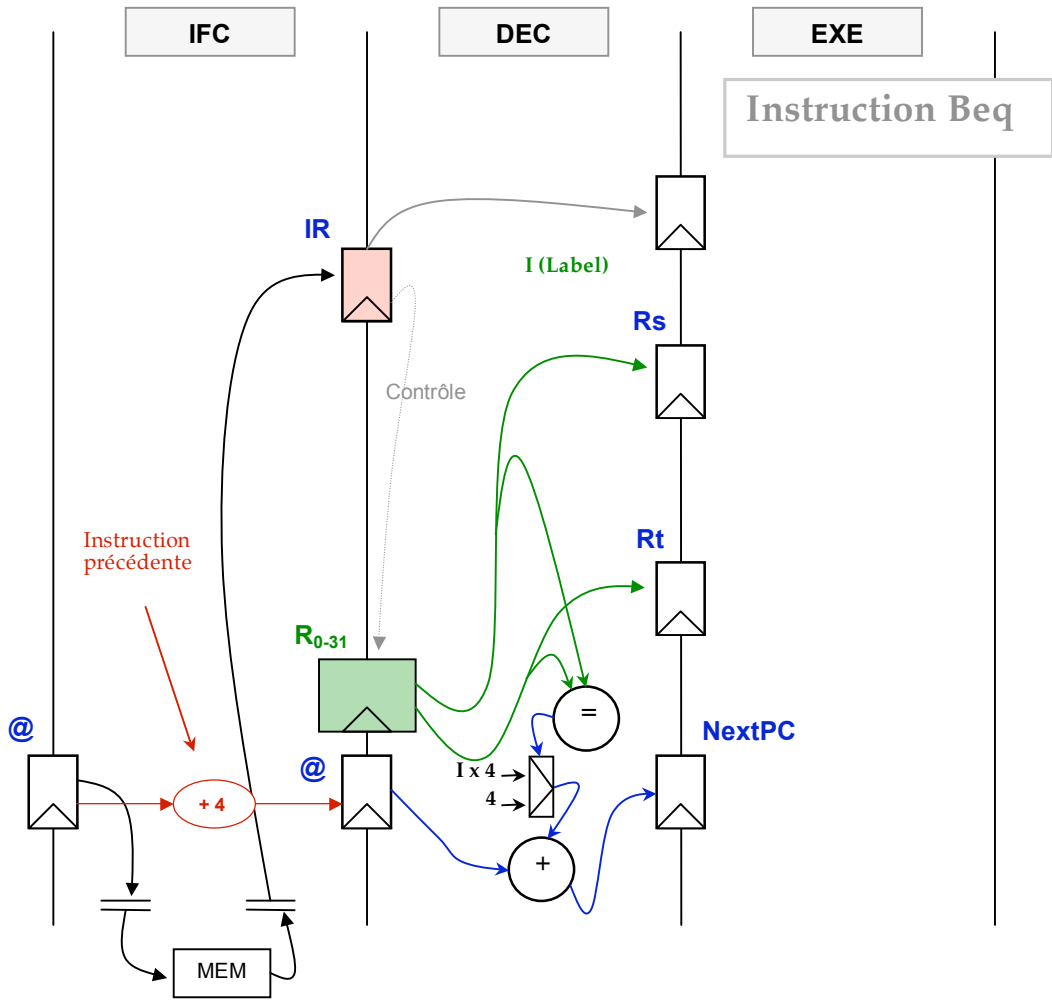
Tout le problème vient du calcul de l'adresse de branchement.
Si le branchement réussi alors : • **PC = (@beq+4) + I x 4**
Sinon • **PC = (@beq+4) + 4**

La première partie du résultat **(@beq+4)** est obtenue par le passage de l'instruction précédente dans la couche DEC du pipeline. En d'autres termes, lorsque l'instruction « `beq` » arrive dans la couche DEC du pipeline, le registre d'adresse contient déjà **@beq+4**

Cette particularité nous oblige à positionner une instruction séquentielle juste avant le branchement (et non une instruction de saut par exemple…)

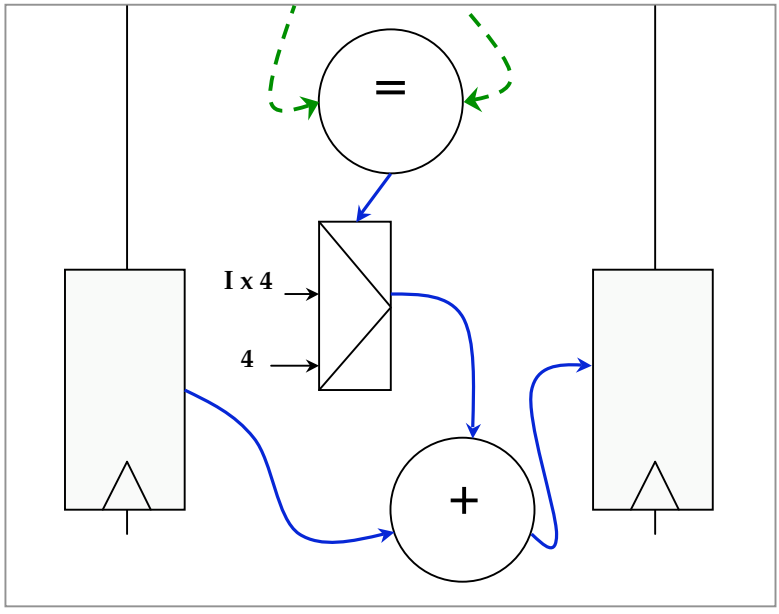
B. *Première solution*

En tenant compte de cette contrainte, proposons une première implémentation de l'instruction de



branchement

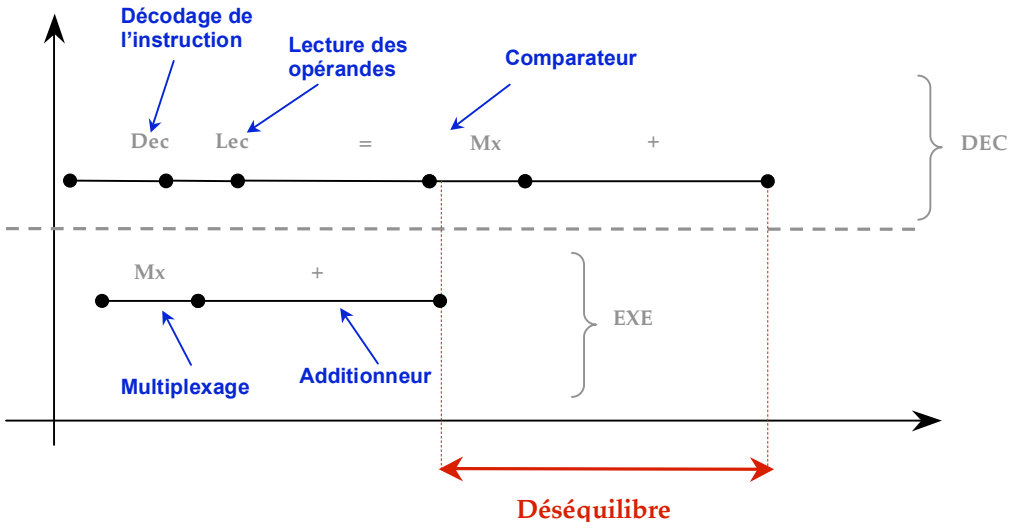
Nous voyons assez bien sûr cet exemple **la complexité** de l'étage DEC.



Dans le cours n°3 nous avons mis en avant une règle d'or (la première) :

★ **Le pipeline doit être équilibré**

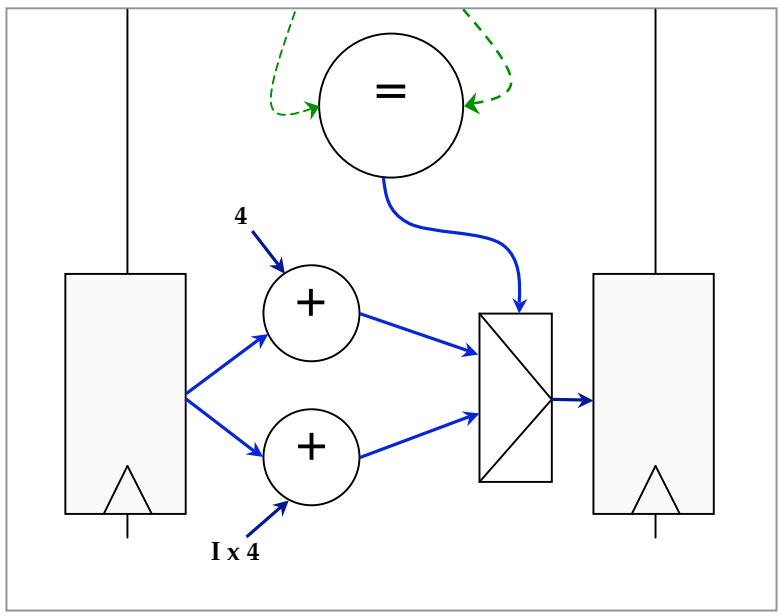
Si on part du principe que l'étage EXE peut se résumer à un multiplexage et à une addition (au pire des cas) , on peut mettre « en concurrence » les deux étages sur le schéma suivant :



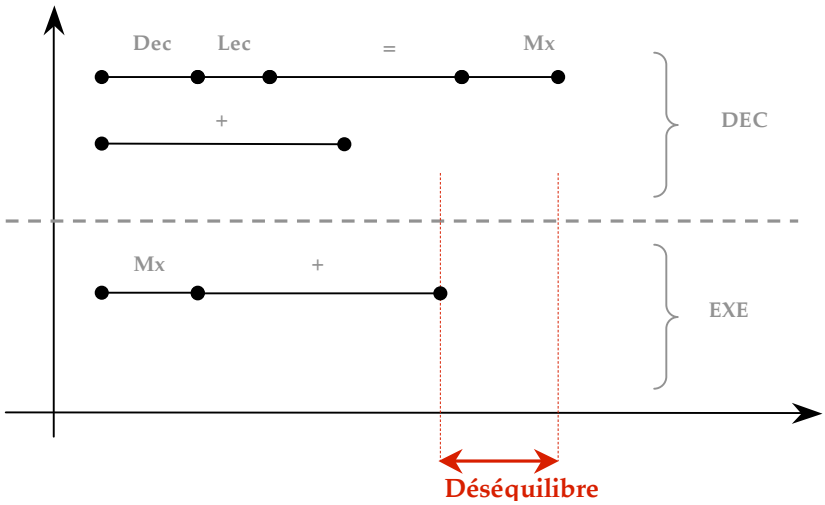
Il faut donc trouver un moyen pour **réduire le décalage entre les deux couches**.

Une des méthode proposée est de paralléliser le décodage, la lecture des opérandes ainsi que la comparaison avec le calcul des adresses. Ainsi on calculerait les deux adresses possibles et on choisirait au dernier moment seulement laquelle des deux on souhaite sauvegarder dans le registre.

C. *Deuxième solution*



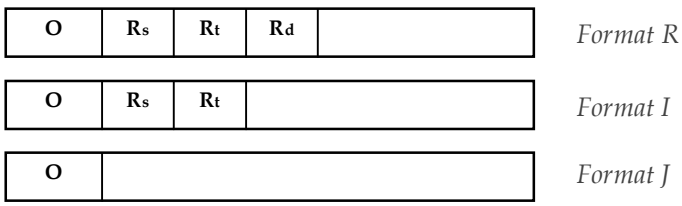
Reprenons la comparaison avec la couche EXE :



Le déséquilibre entre les deux couches est encore **trop important**…

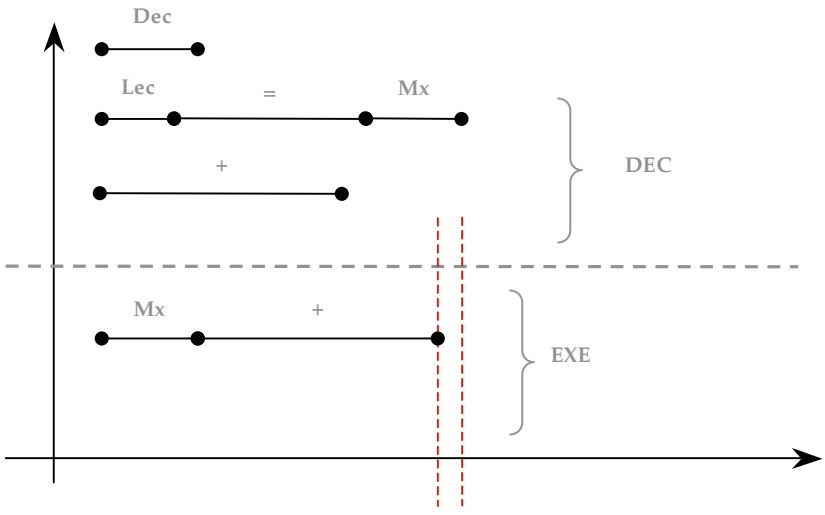
D. *Troisième solution*

Nous allons donc proposer une troisième implémentation de l'instruction « `beq` ».
Mais avant tout, il nous faut revenir sur le codage des instructions :



Ce codage particulier permet aux registres Rs et Rt d'être toujours placés aux mêmes endroits. Cette particularité permet ainsi de lire, à priori et sans décodage, cet emplacement pour chaque instruction.
Si l'instruction nécessite, après décodage, l'utilisation des valeur Rs et Rt, elle auront déjà été lues, et il ne restera donc plus qu'à les utiliser…

Cette technique permet d'insérer un **nouveau niveau de parallélisme** dans l'étage DEC :

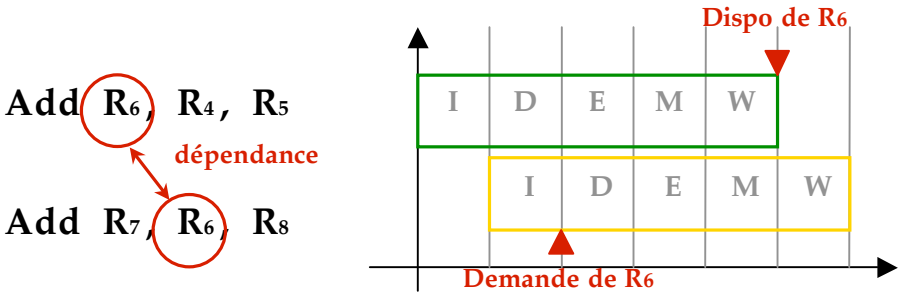


Le déséquilibre est cette fois-ci, tout à fait négligeable.

▶ **Nous pouvons alors dire que le pipeline est équilibré.**

2. Dépendances entre instructions (retour)

Considérons la suite d'instructions suivantes :



On peut constater sur cet exemple que les deux instructions ne pourront s'exécuter tel quel. Il va falloir changer quelque chose pour que les dépendances soient respectées.

Deux solutions sont envisageables :

1° Solution matérielle (hardware)
Il faut faire en sorte que les choses se passent bien

CISC

☑ Assembleur classique. Pas de problème pour le programmeur

☒ Matériel supplémentaire

2° Solution logicielle

RISC

☑ Conception matérielle simplifiée

☒ Le compilateur va souffrir.

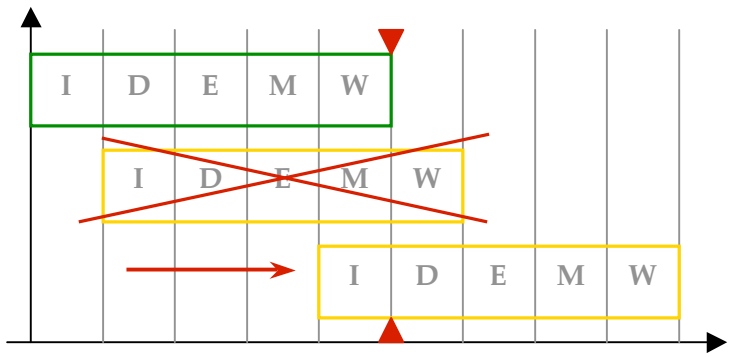
La seconde solution retient évidemment notre attention, s'inscrivant dans la logique RISC.

A. *Solution logicielle*

Nous allons donc faire en sorte que les dépendances soient respectées, et décaler la seconde instruction pour que la disponibilité et la demande du registre soit en correspondance :

Pour permettre ce décalage, nous devons dans un premier temps insérer des « nop » entre les deux « add ».

```
Add R6, R4, R5
Nop
Nop
Nop
Add R7, R6, R8
```



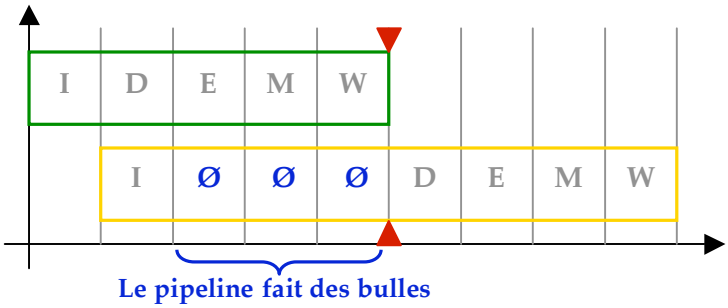
Puis dans un second temps, chercher à remplacer ces « nop » par des instructions indépendantes. Cependant, comme vu précédemment, si il est relativement facile de trouver une instruction pour remplacer un « nop », la manipulation se révèle plus ardue pour 2 ou 3 instructions…

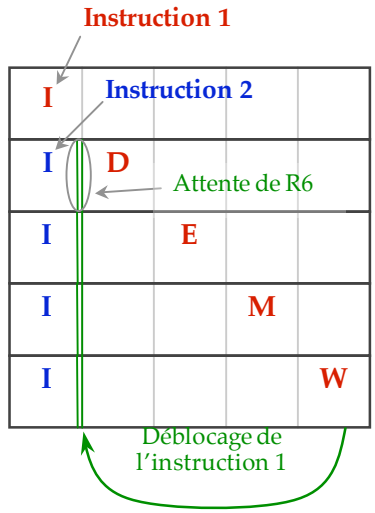
Les « nop » sont donc irremplaçables et le CPI de notre processeur s'effondre, puisqu'une seule instruction utile sur 4 instructions est exécutée par cycle…

Cette solution est donc irrecevable, et nous nous tournons donc vers la première solution : **la solution matérielle.**

B. *Solution matérielle*

Le principe consiste à retarder l'exécution d'une instruction dans le pipeline tant que les ressources dont elle a besoin ne sont pas disponibles.

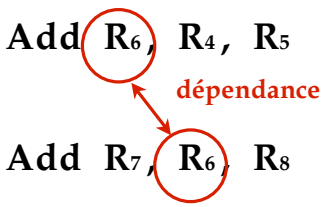


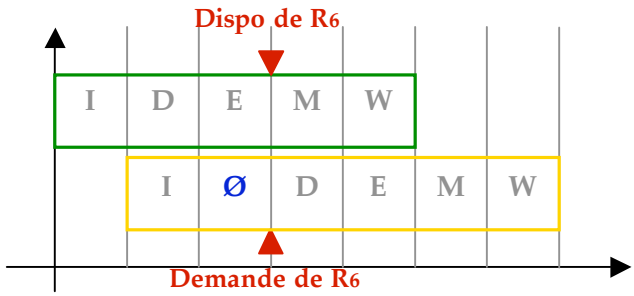


Si on regarde plus en détails les instructions, on peut remarquer que le registre R6 n'est pas vraiment nécessaire. Il ne sert dans cet exemple que de registre temporaire entre les deux « `Add` »

Le second « `Add` » n'a besoin que de la somme de R4 et de R5 qui est, elle, disponible à la fin de l'étage EXE de la première instruction.

Le schéma simplifié peut donc être modifié :

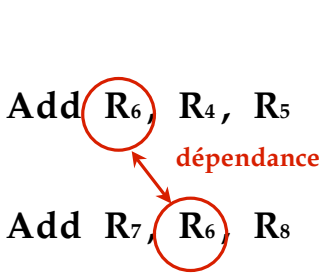


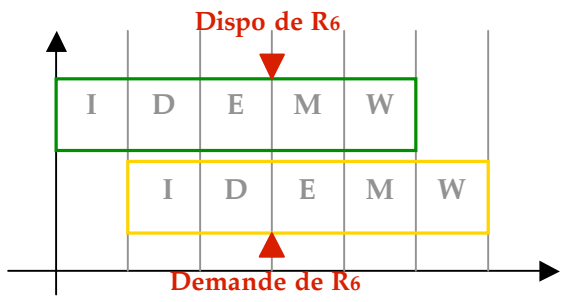


On remarque alors qu'une seule bulle est nécessaire (au lieu de 3 tout à l'heure)

Nous pouvons faire mieux !

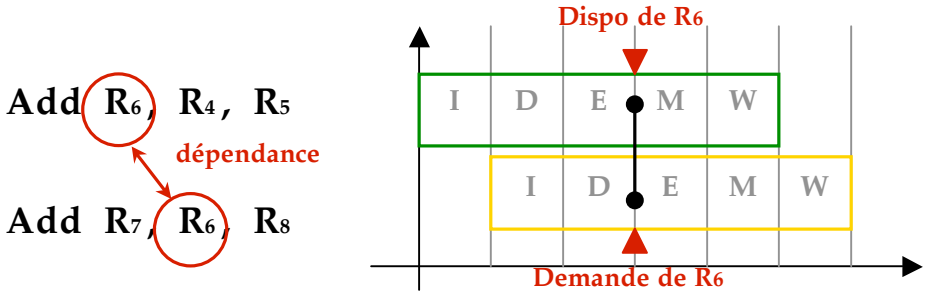
Si on regarde la seconde instruction, le « `Add` » n'a vraiment besoin des opérandes qu'au début de l'étage EXE. Cette remarque nous permet de re-proposer un schéma simplifié :



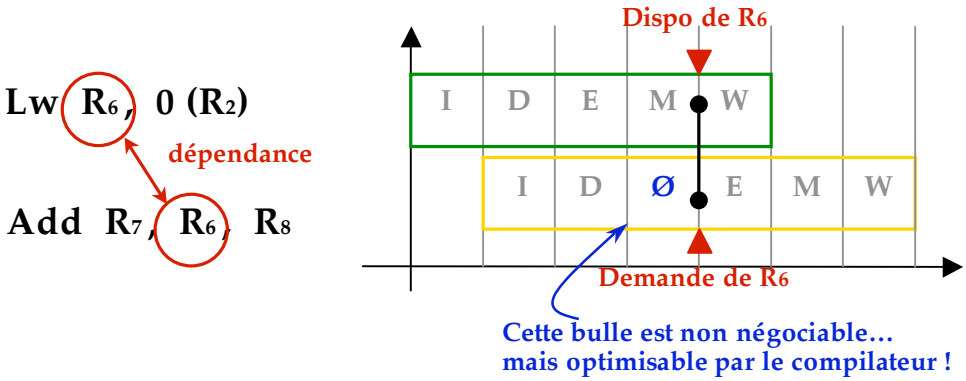


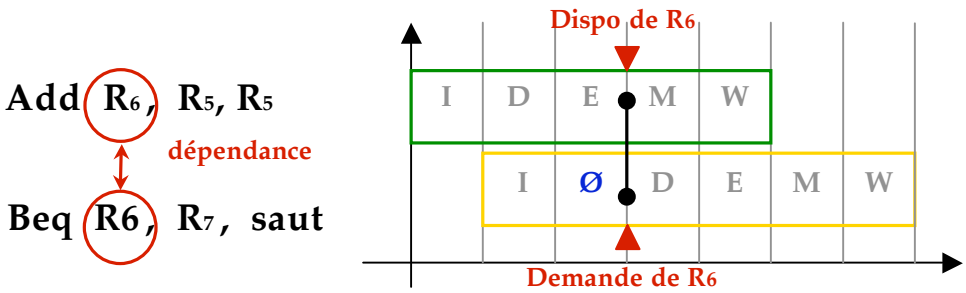
C. Le Bypass

Cette implémentation est optimale et nous pouvons donc proposer un matériel reliant directement ces deux étages. Ce matériel est appelé : **bypass** et se représente de la façon suivante sur le schéma simplifié :



Cette technique du bypass est applicable à toutes les instructions.
Prenons, par exemple, le cas des instructions suivantes :





3. Un exemple

Considérons une simple fonction C :

```
f(int *v, int size)
{
  for (i=0; i>size; i++)
    v[i] = 2 * v[i];
}
```

A. *Une solution*

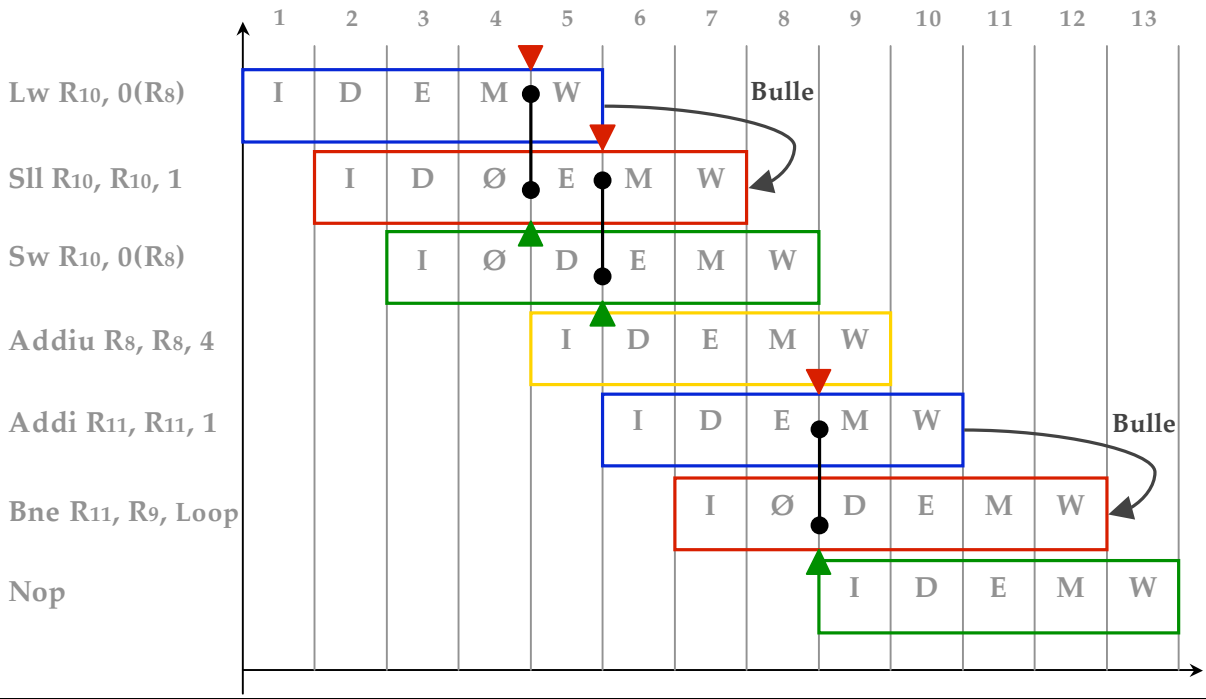
Celle-ci peut s'écrire en assembleur sous la forme :

```
R8 ← V
R9 ← sizeof(V)
R11 ← 0
```

```
1     Loop :  Lw    R10, 0(R8)
2             Sll   R10, R10, 1
3             Sw    R10, 0(R8)
4             Addiu R8, R8, 4
5             Addi  R11, R11, 1
6             Bne   R11, R9, Loop
7             Nop
```

La fin du cours n°3 nous a prouvé qu'il était nécessaire de faire suivre toute instruction de branchement par une autre instruction. Ainsi, nous rajoutons ligne 7 une instruction « nop ».

Le schéma simplifié de cette série d'instruction est le suivant :



Nombre de cycles = **9**
Nombre d'instructions = **7**
Nombre d'instructions utiles = **6**

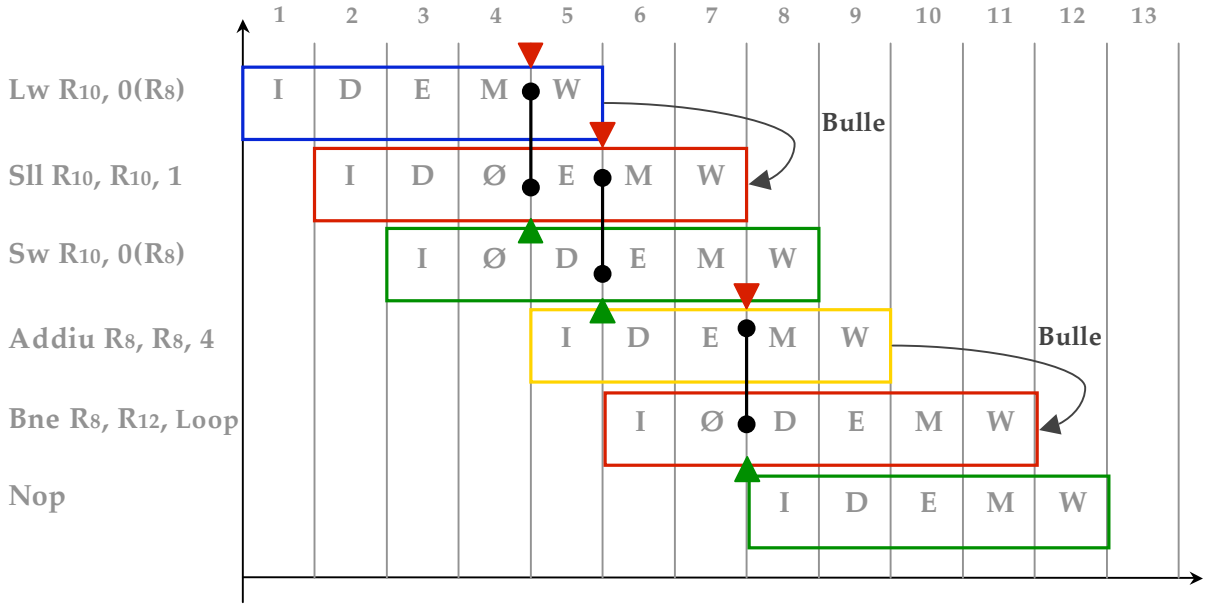
CPI = 9/7 = 1,28
CPI utile = 9/6 = 1,5

L'objectif depuis le début de ce cours est d'obtenir un CPI utile de 1
Il nous faut donc optimiser le code que nous avons proposé ci-dessus.

B. *Une optimisation : registre spécialisé*

Supposons que nous rangeons au préalable `R12 ← v + 4 * sizeof(v)`
Cette manipulation nous permet de supprimer l'instruction 5 et d'obtenir le code suivant :

```
1     Loop :  Lw      R10, 0(R8)
2             Sll     R10, R10, 1
3             Sw      R10, 0(R8)
4             Addiu   R8, R8, 4
5             Bne     R8, R12, Loop
6             Nop
```



Nombre de cycles = **8**
Nombre d'instructions = **6**
Nombre d'instructions utiles = **5**

CPI = 8/6 = 1,33
CPI utile = 8/5 = 1,6

C. *Une optimisation : le réagencement*

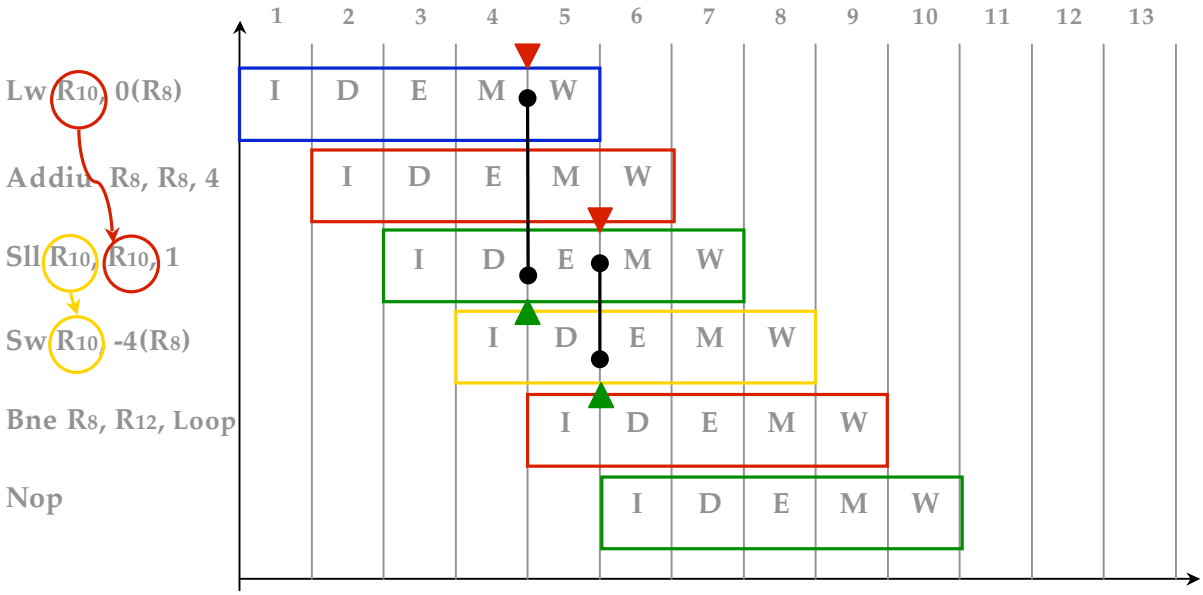
L'objectif est maintenant de faire **claquer les bulles.** Pour ce faire, ré-agençons les instructions :

```
1     Loop :  Lw      R10, 0(R8)
2             Addiu   R8, R8, 4
3             Sll     R10, R10, 1
4             Sw      R10, -4(R8)
5             Bne     R11, R9, Loop
6             Nop
```

Nous avons, dans cet exemple, disposé les instructions de façon à limiter les dépendances.

Les instructions 1, 3 ont été écartées (dépendance sur R10). De même pour les instructions 2 et 4 (dépendance sur R8). **Attention** cependant à rester cohérent ! Le réagencement des instructions **peut provoquer quelques modifications** dans le code (4).

Nous avons donc le nouveau schéma simplifié :



Cette réorganisation du code nous a permis de supprimer toutes les bulles.

Nombre de cycles = **6**	**CPI = 6/6 = 1**
Nombre d'instructions = **6**	**CPI utile = 6/5 = 1,2**
Nombre d'instructions utiles = **5**	

D. *Une optimisation : Substitution d'instruction*

Mettons en oeuvre, enfin, la technique de substitution d'instruction pour remplacer l'instruction « `nop` » suivant le branchement. Pour ce faire, il faut trouver une instruction indépendante du branchement dans le code précédent le « `nop` ».

L'instruction « `Sw` » (ligne 4) semble parfaitement correspondre à cette description :

```
1     Loop :  Lw      R10, 0(R8)
2             Addiu   R8, R8, 4
3             Sll     R10, R10, 1
4             Bne     R11, R9, Loop
5             Sw      R10, -4(R8)
```

Nombre de cycles = **5**	**CPI = 5/5 = 1**
Nombre d'instructions = **5**	**CPI utile = 5/5 = 1**
Nombre d'instructions utiles = **5**	

Chapitre 5

Pipelining 3ème Partie

Après un bref retour sur un point particulier du principe de pipeline, nous nous attarderons, et malgré toutes les optimisations et nouvelles stratégies proposées, sur de nouvelles méthodes pouvant encore réduirele coût des programmes.

1. Retour sur les dépendances

Nous avons pu mettre en évidence dans le cours précédent l'importance des dépendances au sein d'un programme. Les bulles provoquées à l'intérieur du pipeline sont aisément représentables sur un schéma simplifié…Il en est tout autrement sur un schéma détaillé.

Sur le schéma suivant, vous pourrez constater la présence d'un cycle de gel (grisé) qui se traduit par le report du contenu des registres d'un étage à l'autre. Le processeur étant chargé de combler les registres qui auraient du être occupés.

Notons aussi qu'aucune incrémentation du registre d'adresse n'est faite lors de ce cycle

Le schéma détaillé suivant propose l'exécution des instructions :

```
1    Lw    R8, 0 (R6)
2    Add   R10, R8, R11
3    Nop
```

Voir page suivante

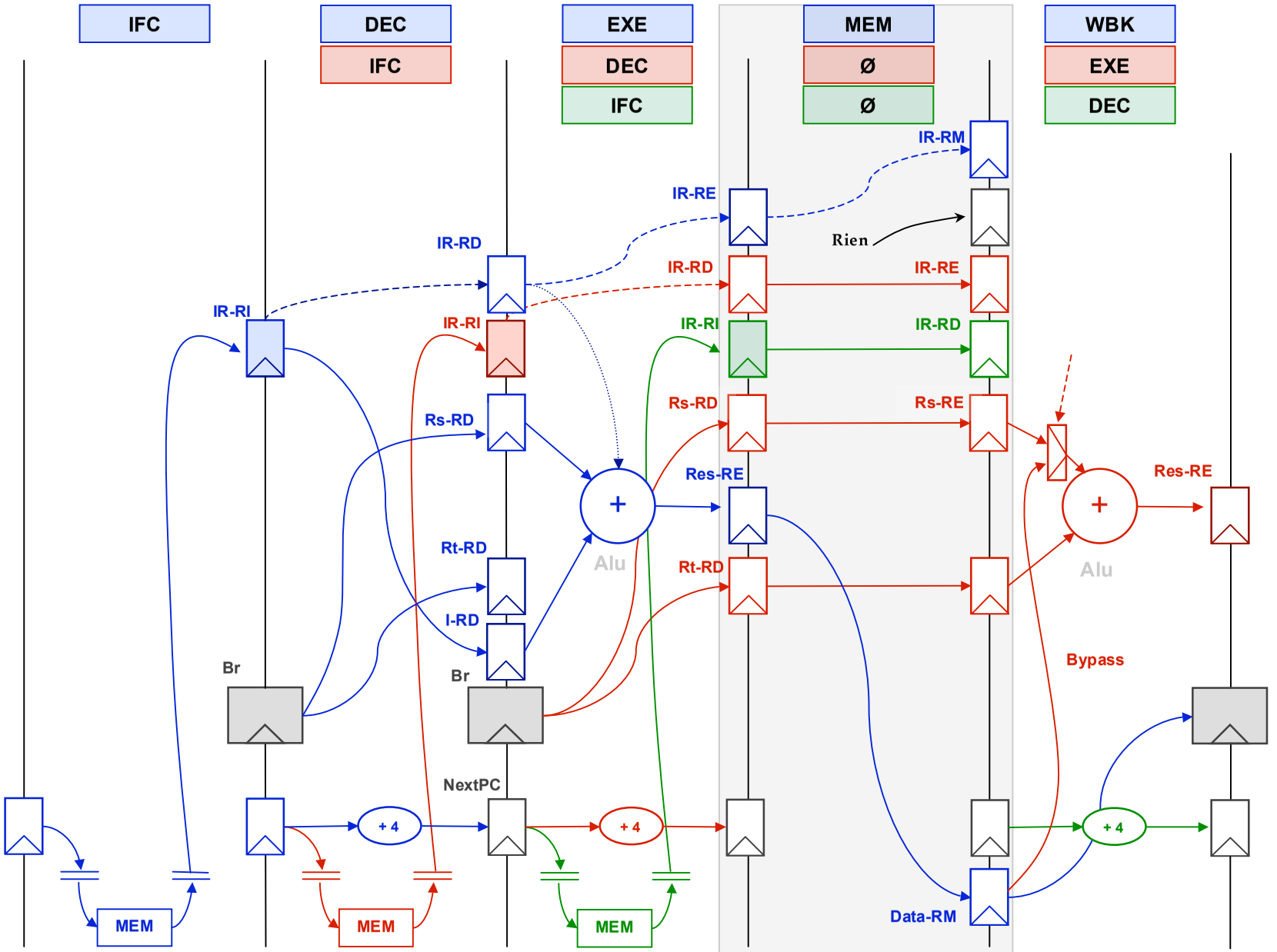
2. Retour sur les optimisations

Reprenons le programme en langage assembleur proposé au cours précédent :

```
1    Loop :   Lw      R10, 0(R8)        # récupère la valeur
2             Sll     R10, R10, 1       # mult 2
3             Sw      R10, 0(R8)        # sauvegarde
4             Addiu   R8, R8, 4         # change l'indice
5             Addi    R11, R11, 1       # incrémente le compteur
6             Bne     R11, R9, Loop     # test de sortie de boucle
7             Nop                       # delayed slot
```

La version optimisée de ce même programme était :

```
1    Loop :   Lw      R10, 0(R8)
2             Addiu   R8, R8, 4
3             Sll     R10, R10, 1
4             Bne     R8, R12, Loop
5             Sw      R10, -4(R8)
```


IFC
DEC
IFC
EXE
DEC
IFC
MEM
Ø
Ø
WBK
EXE
DEC
IR-RM
IR-RE
Rien
IR-RD
IR-RD
IR-RE
IR-RI
IR-RI
IR-RI
IR-RD
Rs-RD
Rs-RD
Rs-RE
Res-RE
Res-RE
+
+
Alu
Alu
Rt-RD
Rt-RD
I-RD
Br
Br
Bypass
NextPC
+ 4
+ 4
+ 4
MEM
MEM
MEM
Data-RM

A. *Optimisation :LES & Pipeline logiciel*

Au lieu de considérer la boucle dans sa totalité, nous pouvons choisir d'isoler le cœur de la boucle, c'est-à-dire, le groupe d'instruction qui modifie vraiment les éléments du tableau.

En langage C le cœur de la boucle est donné par : `v[i] = 2 * v[i]`

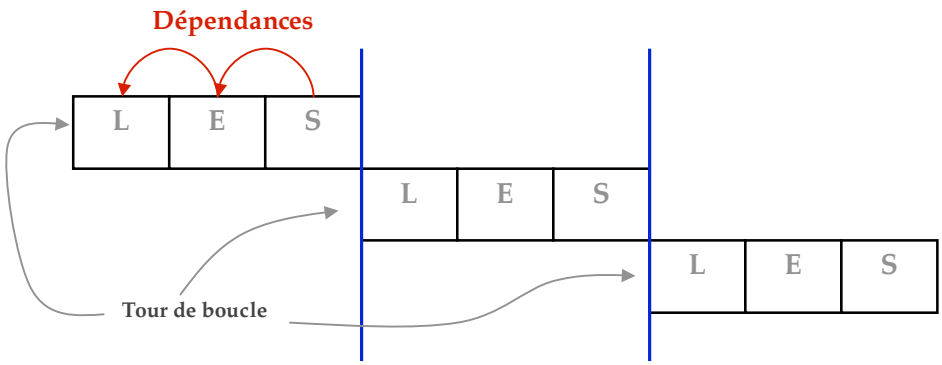
En langage assembleur, ce *cœur de boucle* est données par les instructions :

```
        Lw      R10, 0(R8)
        Sll     R10, R10, 1
        Sw      R10, -4(R8)
(ou     Sw      R10, 0(R8) dans la version non optimisée)
```

Cet agencement d'instruction peut être résumé en 3 lettres : L E S

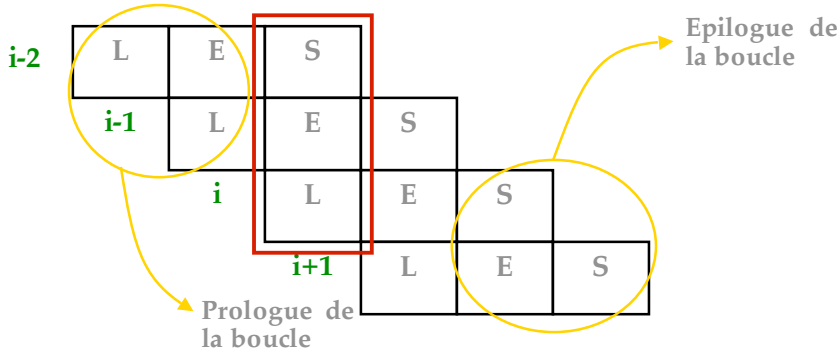
L pour **load** :	`Lw R10, 0(R8)`	Chargement de la donnée
E pour **execute** :	`Sll R10, R10, 1`	Traitement de la donnée
S pour **store** :	`Sw R10, 0(R8)`	Sauvegarde du résultat

On peut remarquer que le même traitement est appliqué quelle que soit la donnée.
On peut donc tenter de représenter l'exécution de la boucle par le schéma suivant :



Il s'agit maintenant, de trouver un moyen d'optimiser cet agencement.
Les dépendances concernent clairement les parties L E et S d'un même tour de boucle.
Rien ne nous empêche donc de *quasi-superposer* les tours de boucle à la manière d'un pipeline.

Nous obtenons ainsi un **pipeline logiciel** :



Si on développe le cœur de boucle pour les tours i, i+1, i+2 on obtient :

```
→ Lw    R10, 0(R8)      }
  Sll   R10, R10, 1     }  i
  Sw    R10, 0(R8)      }

  Lw    R15, 4(R8)      }
→ Sll   R15, R15, 1     }  i-1
  Sw    R15, -4(R8)     }

  Lw    R19, 8(R8)      }
  Sll   R19, R19, 1     }  i-2
→ Sw    R19, -8(R8)     }
```

Il reste à choisir les instructions en tenant compte des remarques précédentes et en s'arrangeant pour que les valeurs puissent passer de registre à registre. La liaison sera ici assurée par l'instruction Sll (execute)

```
1     Loop :    Sw      R19, -8(R8)
2               Sll     R19, R9, 1
3               Lw      R9, 0(R8)
4               Addiu   R8, R8, 4
5               Bne     R8, R12, Loop
6               Nop
```

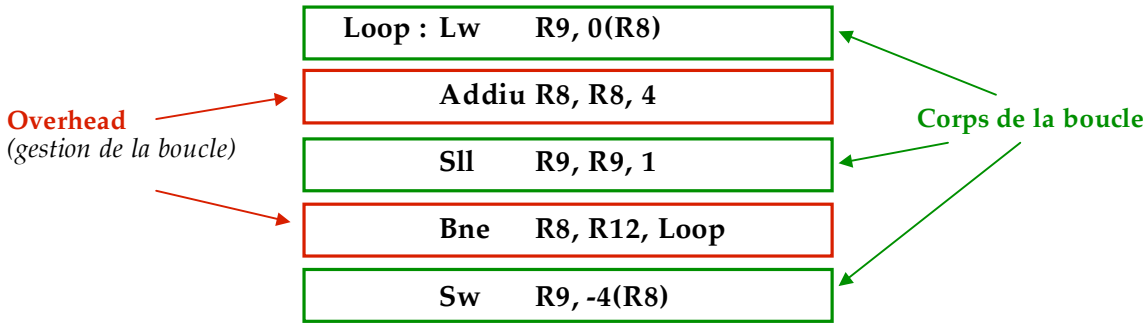
*L'instruction 1 provient du **S** de l'instruction i-2*
*L'instruction 2 provient du **E** de l'instruction i-1*
*L'instruction 3 provient du **L** de l'instruction i*

Une optimisation conduit à :

```
1     Loop :    Sw      R19, -8(R8)
2               Addiu   R8, R8, 4
3               Sll     R19, R9, 1
4               Bne     R8, R12, Loop
5               Lw      R9, -4(R8)
```

B. *Optimisation : Partage de l'overhead*

La seconde optimisation sur ce code nécessite de se pencher plus particulièrement sur le bien-fondé des instructions que nous exécutons. Ainsi, nous devons distinguer deux parties dans le code ci-avant :



Tout le principe est de partager le « poids » de l'overhead entre les corps de boucle.

Le principe est le même que le suivant :

- Lorsque vous vous rendez, en voiture, à votre travail vous déplacez votre poids (le corps de la boucle) ainsi que le poids de la voiture (l'overhead). Ce qui fait un déplacement de masse conséquent si 1000 personnes font la même chose.

L'intérêt réside dans le fait de partager le poids de votre voiture, en prenant par exemple d'autres personnes (4). Ainsi, en prenant 3 autres personnes avec vous, vous réalisez un déplacement de masse de : *4 * 70 Kg + 1000 Kg* = ***1280 Kg*** au lieu de *4*(70 + 1000)* = ***4280 Kg***

Il faut donc appliquer cette technique à notre boucle, en incluant 2 corps de boucle pour le même « overhead » :

```
Loop :      Lw        R9, 0(R8)
            Lw        R19, 4(R8)
            Sll       R9, R9, 1
            Sll       R19, R19, 1
            Sw        R9, 0(R8)
            Sw        R19, 4(R8)
            Addiu     R8, R8, 8
            Bne       R8, R12, Loop
            Nop
```

overhead

Cette technique, outre de diminuer la surcharge de la boucle permet une multiplication des possibilités de réordonnancer le code.

Ici, la version finale sera :

```
Loop :      Lw          R9, 0(R8)
            Lw          R19, 4(R8)
            Sll         R9, R9, 1
            Sll         R19, R19, 1
            Addiu       R8, R8, 8
            Sw          R9, 0(R8)
            Bne         R8, R12, Loop
            Sw          R19, 4(R8)
```

Chapitre 6

Processeurs Superscalaires

Après l'étude du principe du pipeline et donc des processeurs supportant cette méthode (processeur « pipelinés »), nous sommes en mesure de proposer une nouvelle stratégie de traitement des instructions… Pouvons-nous encore améliorer les performances obtenus par optimisations matérielles et logicielles ?

1. Aller plus loin ?

- Nous avons jusqu'à présent étudié différents types de réalisation de processeurs.
- Nous avons réussi à atteindre le $CPI_{utile} = 1$.
- La technique du pipeline nous a aussi permis d'augmenter la fréquence des processeurs.

On obtient donc un processeur relativement performant !

Peut-on faire plus ?
Comment augmenter encore les performances ?

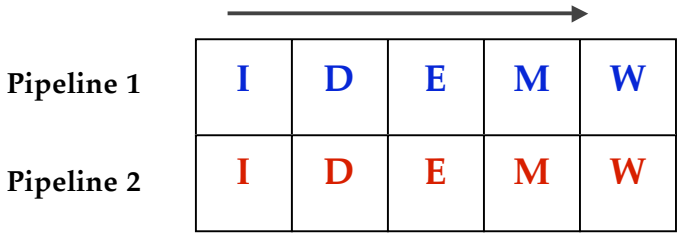
▶ **On peut augmenter la fréquence** en découpant plus finement le processeur. Les processeurs ainsi obtenus sont appelés processeur à *superpipeline* ou à *pipeline profond*

Mais les cours précédents nous ont aussi permis d'établir que plus les pipelines étaient profonds, plus les problèmes étaient nombreux :

- Augmentation du nombre de delayed slots
- Augmentation du nombre de registres nécessaires

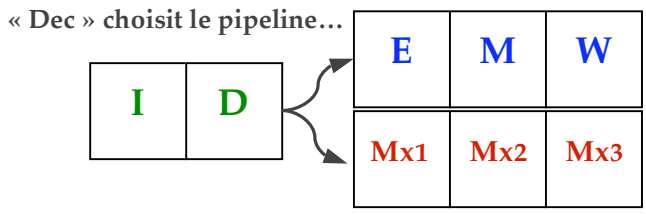
▶ **On peut intervenir sur le CPI**. Étant déjà à 1, nous souhaitons maintenant le faire descendre jusqu'à 0,5 cycle / instructions ou encore s'arranger pour exécuter 2 instructions en 1 seul cycle.

Cette technique est utilisée dans les **processeurs superscalaires**.
Les deux instructions ne s'exécutent pas sur des processeurs séparés mais dans des pipelines séparés :



Maintenant que le principe est posé, il nous reste à proposer une implémentation…

Le premier MIPS à pipeline disposait d'une version simplifiée d'un pipeline parallèle spécialisé. En effet, l'instruction de multiplication « `Mul` » est très vite apparue comme étant la plus longue de toutes. Elle prenait en fait 3 fois plus de temps qu'un simple « `Add` ».
Il a donc été décidé de concevoir un pipeline spécialement dédié aux instructions de multiplication :

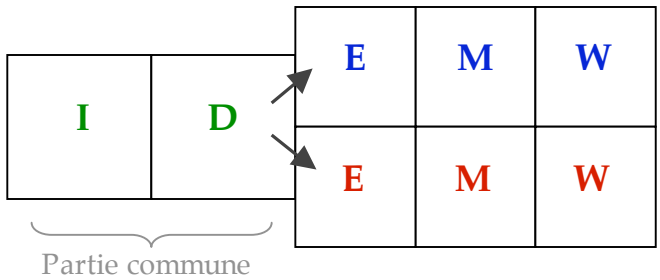


- **Les processeurs superscalaires généralisent ce double pipeline.**

2. Le double pipeline

A. *Implémentation*

Proposons donc maintenant une implémentation de notre nouveau mips pour qu'il supporte la technique du double pipeline :



Sur ce schéma on constate que :

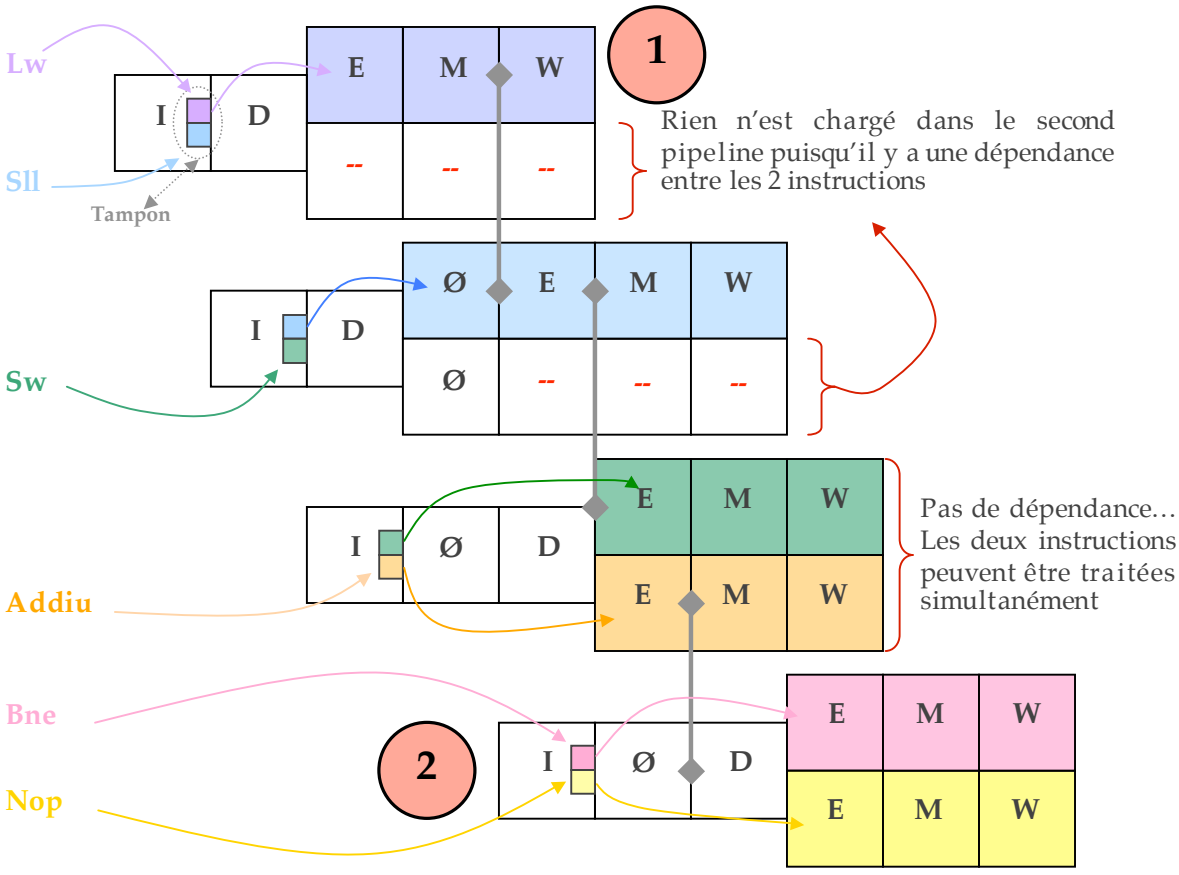
- **IFC** doit, à chaque cycle, lire 2 instructions
- **DEC** doit, à chaque cycle, décoder et aiguiller 2 instructions

Moins visible sur ce schéma, on peut tout de même supposer que le nombre de bypass sera modifié. En effet, alors que 8 bypass sont nécessaires sur la version simple du mips (1 seul pipeline), 16 bypass (+ bypass croisés) sont nécessaires pour un mips à 2 pipelines.

Si on reprend le code :

Lw R9, 0(R8)
Sll R9, R9, 1
Sw R9, 0(R8)
Addiu R8, R8, 4
Bne R8, R12,

On peut proposer un **schéma simplifié** de l'exécution de ce dernier sur un mips superscalaire.



Notons que le code ne demande plus que **6 cycles**…
La représentation présentée ci-dessus n'est qu'une simplification de la véritable représentation (proposée ci-dessous). Elle permet cependant de mettre en avant quelques particularités de cette implémentation.

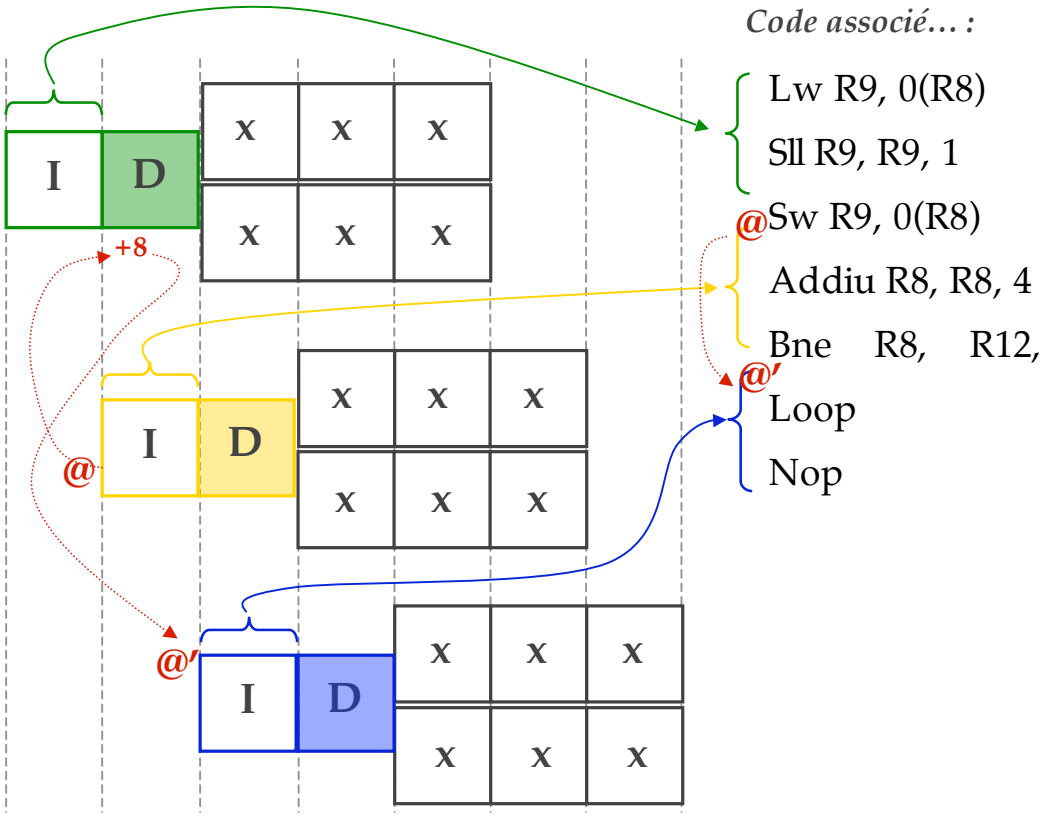
- On remarque immédiatement la présence d'un buffer, appelé « **buffer de prefetch** », qui permet à l'étage « I » de stocker les instructions qu'il récupère. Nous reviendrons sur le fonctionnement précis de ce buffer ci-après.

- On peut aussi se pencher sur le point **1.** En effet, on peut constater que seul un pipeline est utilisé lorsqu'une dépendance est détectée entre les deux instructions qui devraient être exécutées. Le processeur MIPS utilisé respecte le principe du « **In Order Execute** », qui préconise que les instructions doivent êtres exécutées dans l'ordre.

- Le point **2** est un peu plus complexe. Il concerne particulièrement les instructions de branchements. Avant de détailler le chargement et l'exécution de ces instructions, il faut s'assurer de bien avoir compris le calcul des adresses.

Le cours n°3 détaille le calcul des adresses pour un pipeline simple. La technique reste similaire pour un processeur superscalaire. On notera, par contre, qu'au lieu d'un simple « **@ instruction en cours de fetch + 4** », on réalisera un « **@ instruction en cours de fetch + 8** ».

On peut représenter cette caractéristique sur un schéma :

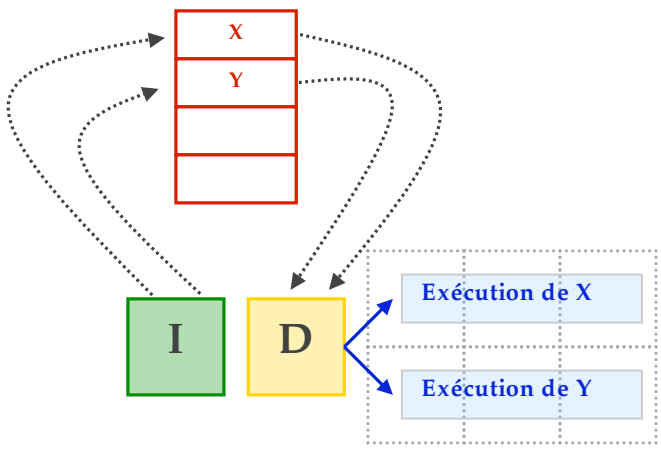


Une fois ce principe de calcul d'adresse acquis, il faut se préoccuper de la gestion du buffer de préfetch. Ce buffer est chargé de « sauvegarder » les instructions *fetchées* au cas où elles ne seraient pas immédiatement traitées (cas de dépendances par exemple)

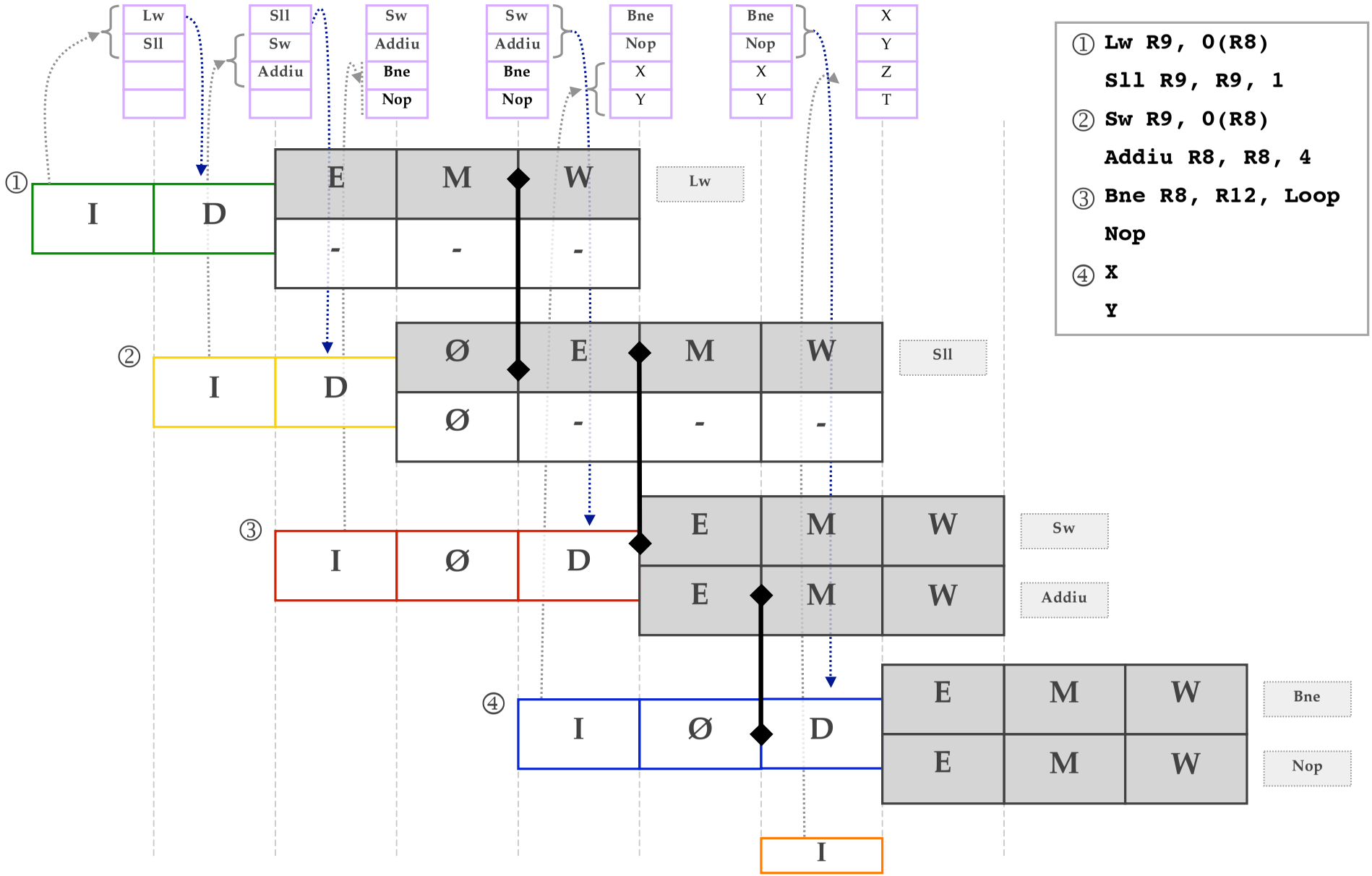
Voici un détail du fonctionnement de ce tampon :

La phase « I » permet le chargement des instructions dans le buffer, tandis que la phase « D » lit et décode la/les première(s) instruction(s) du buffer.

En cas de dépendance, seule la première instruction est lue. La seconde est conservée dans le buffer.



On peut donc maintenant proposer un schéma simplifié en détaillant principalement les états successifs du buffer de préfetch.


Lw
Sll
Sw
Addiu
Bne
Nop
X
Y
Z
T
① Lw R9, 0(R8)
Sll R9, R9, 1
② Sw R9, 0(R8)
Addiu R8, R8, 4
③ Bne R8, R12, Loop
Nop
④ X
Y
I
D
E
M
W
Ø
-
Lw
Sll
Sw
Addiu
Bne
Nop

Le déroulement de l'exécution ne pose aucun problème.
Le buffer joue parfaitement son rôle.

Cependant, le schéma s'arrête juste avant **un passage critique**…

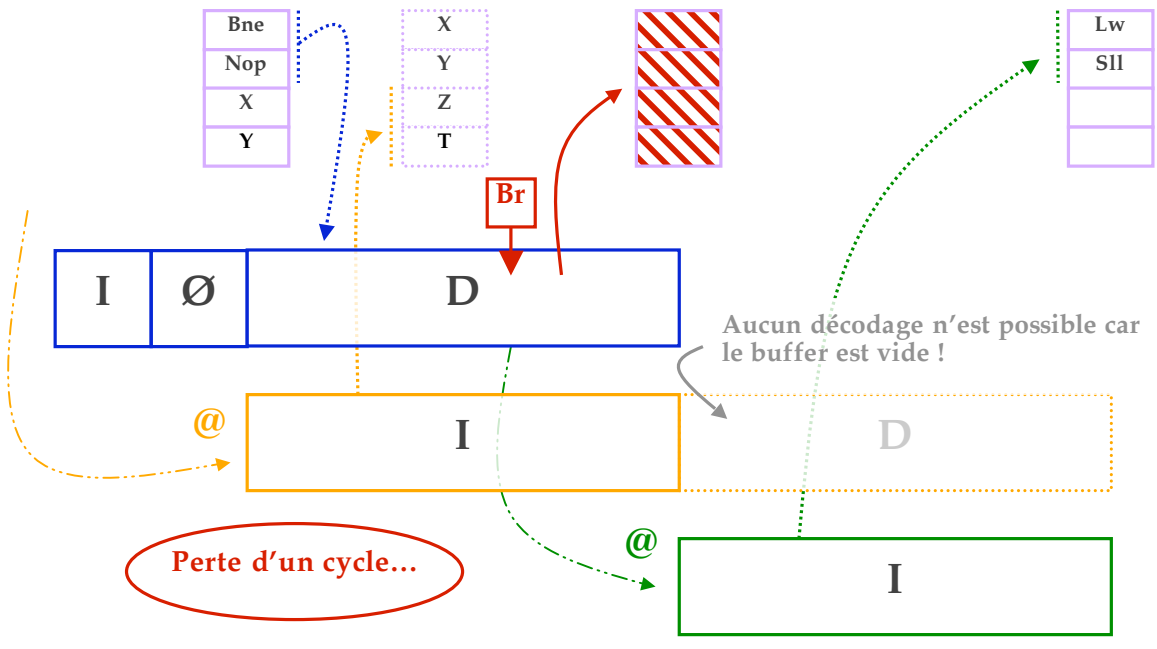
B. *Gestion des branchements*

Si on considère l'instruction bleue (la quatrième), on se rend compte que l'étage « I » charge deux instructions « folkloriques » dans le buffer (X et Y). L'étage « D » de l'instruction bleue va donc calculer l'adresse du branchement. Pendant ce temps l'étage « I » de l'instruction suivante (orange) va charger dans le buffer 2 nouvelles instructions (Z et T)…

Beaucoup trop d'instructions n'ayant rien a voir avec notre code risquent d'être exécutées. Il faut donc trouver une solution pour éviter la prise en compte de ces chargements dans le buffer.

Pour ce faire, on implémente un système qui permet dès la détection d'une instruction de branchement de vider (flush) le buffer de préfetch. Ainsi, dès la fin du décodage de l'instruction « `Bne` » l'étage « D » impose le vidage du buffer.

Ce vidage provoque donc la perte des instructions suivant le branchement ainsi que celles venant juste d'être récupérées par l'étage « I » suivant.



La perte d'un cycle à chaque appel d'une instruction de branchement est très claire sur ce schéma. Ne pourrait-on pas **prédire le résultat du branchement** pour, au lieu de vider le buffer, le remplir avec les instructions cibles ?

C. *Prédictions de branchements*

Plusieurs méthodes de prédictions existent :

- ***50% - 50%***

  Dans ce cas, le branchement est prédit « pris » une fois sur deux…

- ***Prédiction de branchement dynamique***

  - Première phase 50% - 50%
  - 2° phase souvenir du dernier tour de boucle
  - Cette information binaire (oui/non) est stockée avec l'instruction.
  - Il faut donc implémenter une mémoire locale : **le scory-board.**

- ***Prédiction de branchement statique***

  Le bit information est positionné par le compilateur.
  Il est aussi positionnable par l'utilisateur via l'utilisation de pragmas

Chapitre 7

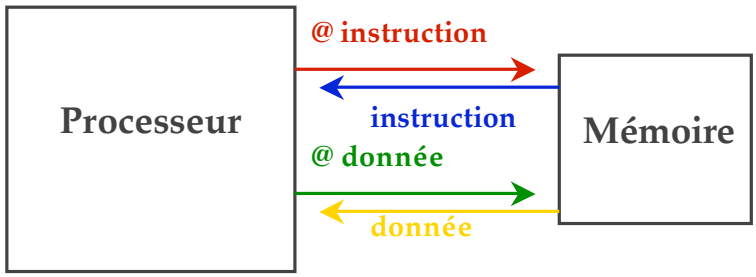
Hiérarchie des Mémoires

Nous avons jusu'a présent, étudié le processeur comme composant « solitaire ». Cependant, ce matériel s'inscrit dans un système complexe ou divers autres unités dialoguent entres-elle. Comment ces unités, les mémoires, s'organisent-elles autour du processeur ? Ont-elles une influence sur les performances du processeur et de l'ensemble ?

1. Présentation du concept

A. *Types de mémoires*

Le schéma général, mais néanmoins simplifié à l'extrême, de l'organisation « processeur–mémoire » peut être présenté de la façon suivante :



Il faut cependant respecter un certain nombre de contraintes pour que l'ensemble processeur-mémoire fonctionne correctement. Ainsi, la mémoire doit répondre en un cycle afin que le processeur n'ait pas besoin d'insérer des cycles de gel, ce qui réduirait fortement les performances de ce dernier.

Cette remarque s'applique évidemment à **l'interface instruction** (transferts d'instructions) comme à **l'interface data** (transferts de données), et implique donc :

- On doit pouvoir répondre à 2 requêtes mémoire à chaque cycle.
- La mémoire doit travailler à la même fréquence que le processeur.

Il faut aussi tenir compte de l'espace d'adressage demandé par le processeur.
En effet, celui-ci codant les adresses sur 32 bits, sous-entends un espace d'adressage de 2^{32} adresses soit : $4x10^9$ adresses. Notre solution doit donc être capable de stocker 4Go de données à un prix raisonnable.

On distingue plusieurs technologies de mémoires :

- **Mémoire statique**
  - Elle travaille à la même fréquence que le processeur.
  - Mais elle est limitée à quelques Ko
- **Mémoire dynamique**
  - Fréquence de travail : fréquence(proc) / 5.
  - Taille max : Quelques Mo
- **Disque dur**
  - Temps d'accès de quelques millisecondes.
  - Quelques Go d'espace disponible

Il existe aussi une dernière technologie :

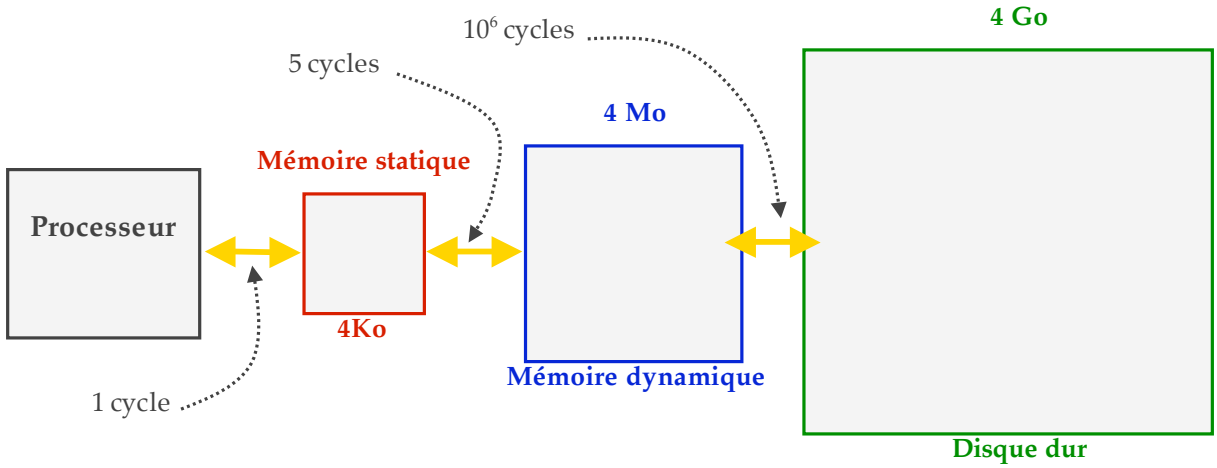
- **Bande magnétique**
  - Temps d'accès de quelques minutes.
  - Plusieurs To d'espace

Chaque technologie dispose clairement d'avantages (-) et d'inconvénients (-).

Il nous faut donc tirer le meilleur de chacune des solutions proposées en les « mixant » au sein d'une même implémentation.

B. *Première proposition*

On propose donc une première version de notre organisation :



Cette organisation répond-t-elle à la problématique ?

Pour répondre à cette question, il faut calculer le nombre de cycle moyen pour accéder à une donnée… :

La probabilité qu'une donnée (parmi 4Go) soit présente dans la mémoire statique est :

$$P_1 = P(cache) = \frac{4\times10^3}{4\times10^9} = 10^{-6}$$

La probabilité qu'une donnée soit présente dans la mémoire dynamique (et pas dans le cache) :

$$P_2 = P(dynamique) = \left(\frac{4\times10^6}{4\times10^9}\right) - P(cache)$$
$$= 10^{-3} - 10^{-6}$$

La probabilité qu'une donnée soit présente sur le disque dur (et pas dans les autres mémoires) :

$$P_3 = P(disque\ dur) = 1 - P(dynamique)$$
$$= 1 - 10^{-3}$$

Connaissant de plus le nombre de cycles nécessaires pour accéder à chaque type de mémoires, nous pouvons en déduire la formule suivante :

$$N = \sum_{i=1}^{3} c_i \times P_i$$
$$= \left(1 \times 10^{-6}\right) + \left(5 \times \left(10^{-3} - 10^{-6}\right)\right) + \left(10^{6} \times \left(1 - 10^{-3}\right)\right)$$
$$; \ 999\ 000 \text{ cycles}$$

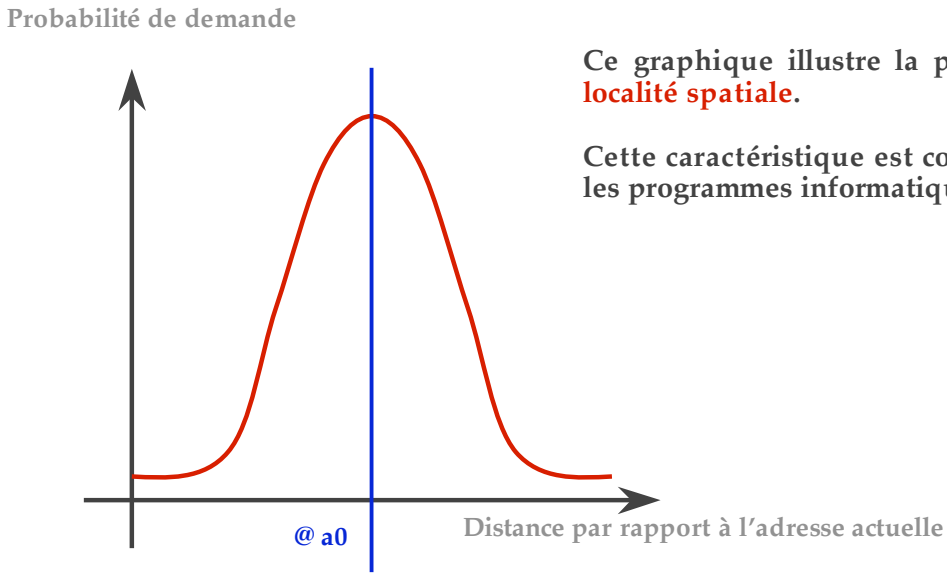
La traduction de ce résultat en « français » est : **Il faudra en moyenne 999 000 cycles pour accéder à une donnée en mémoire.** Cette solution n'est évidemment pas envisageable. Dans ce cas, le processeur passerait la plus grande partie de son temps à attendre la donnée qu'il a demandée.

Le nombre d'accès pour chaque type de mémoire est impossible à changer.
Nous ne pouvons donc influer que sur les probabilités de présence de la donnée dans les mémoires.

Autrement dit, il faut donc s'arranger pour que les données auxquelles on souhaite accéder se trouvent dans la mémoire de plus bas niveau (la plus proche du processeur).

On parle alors de **remplissage intelligent du cache**, ou de **prédiction des accès processeur.**

Cette technique s'appuie sur des statistiques et sur une assertion très simple : « La prochaine adresse demandée par le processeur a de grandes chances d'être une voisine de l'actuelle (a0). »



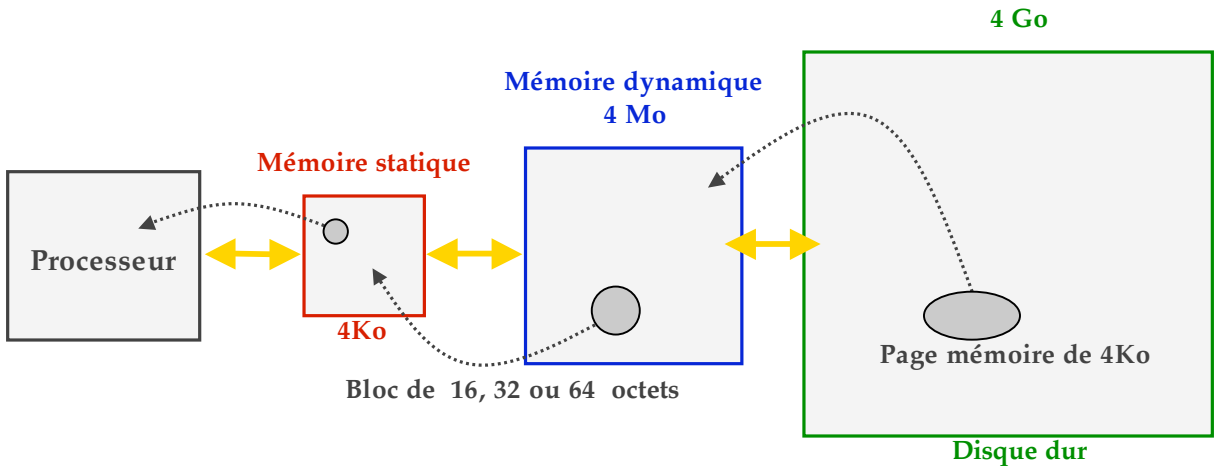
Ce graphique illustre la propriété de la localité spatiale.

Cette caractéristique est commune a tous les programmes informatiques…

La propriété que nous venons de mettre en avant provient **de la technique de réservation de la mémoire** (qui consiste à allouer des plages de mémoires contiguës) et de **la manière dont nous écrivons le code** (très souvent séquentielle)

2. Organisation des mémoires

Conscient que la proposition précédente ne convient pas, il faut donc trouver une nouvelle proposition… en mettant en application le principe de la localité spatiale par exemple. Pour ce faire, nous allons mettre en place une stratégie de déplacement de bloc ou de pages entre les différentes mémoires :



Ainsi, si je demande l'accès à une donnée « d » à l'adresse a0 :

- Si l'adresse est dans le cache… je la charge.
- Si l'adresse n'est pas dans le cache je vais chercher dans la mémoire dynamique.
    - Si l'adresse est dans la mémoire dynamique, je rapatrie un bloc de données encadrant la donnée à laquelle je souhaite accéder. Une fois ce bloc dans la mémoire statique (le cache), je peux envoyer ma donnée au processeur.
    - Si l'adresse n'est pas dans la mémoire dynamique, je vais la chercher sur le disque dur. Elle y est forcément, et je rapatrie donc une page de mémoire encadrant la donnée à laquelle je souhaite accéder. Puis je déplace un bloc de données vers le cache. Et enfin je charge la donnée dans le processeur.

Grâce à cette technique de chargement, je peux optimiser les accès futurs.
En effet, selon le principe de la localité spatiale, la prochaine donnée à laquelle je souhaite accéder a de grandes chances de se trouver à un emplacement très proche de la donnée que je traite actuellement (l'instruction par exemple).

On a donc de grande chance, lors de la demande d'accès à cette nouvelle donnée, de la trouver dans le cache (chargée dans le bloc contenant la précédente donnée).
Et si par malheur, elle ne se trouvait pas dans le cache, les chances sont quasi-certaines de la trouver dans la mémoire dynamique (dans la page chargée précédemment depuis le disque dur).

Ainsi le nombre de cycles nécessaire pour accéder à une donnée est donné par l'expression suivante :

$$N = \underbrace{(1 \times 0{,}95)}_{\text{Mémoire Statique}} + \underbrace{(5 \times 0{,}05 \times (1 - 10^{-5}))}_{\text{Mémoire Dynamique}} + \underbrace{(10^{6} \times 10^{-5})}_{\text{Disque Dur}}$$

1 : Nombre de cycles nécessaires pour accéder à la donnée dans la mémoire statique.

0,95 : Probabilité de trouver la donnée dans la mémoire statique

10^6 : Nombre de cycles nécessaires pour accéder à la donnée sur le disque dur.

10^{-5} : Probabilité d'avoir à chercher cette donnée sur le disque = Probabilité de ne pas trouver la donnée dans les deux mémoires précédentes

5 : Nombre de cycles nécessaires pour accéder à la donnée sur la mémoire dynamique.

0,05 : Probabilité de ne pas avoir trouvé la donnée sur la mémoire statique

$1\text{-}10^{-5}$: Probabilité de trouver la donnée dans la mémoire dynamique.

L'application numérique nous donne : $N = 1{,}7 \text{cycles}$

Ce résultat peut paraître tout à fait honorable à première vue.

Cependant, supposons que la donnée ne se trouve pas dans le cache (mémoire statique). Il faut donc, comme proposé précédemment, amener un paquet de 32 octets de données de la mémoire primaire (dynamique) vers le cache. Jusqu'à maintenant, nous supposions que le transfert de ces 32 octets se faisait en une seule fois.

Or les liaisons inter-mémoires sont de capacités réduites…

En supposant que la liaison entre ces deux mémoires ait une capacité de 4 octets, il faudra donc : $t = \frac{32}{4} = 8 \text{transferts}$ pour rapatrier la donnée de la mémoire primaire jusqu'au cache.

▶ **Il faut absolument tenir compte de ce paramètre dans nos calculs !**

On repose donc le calcul précédent en tenant compte des capacités des liaisons, ainsi que des transferts supplémentaires, afin de faire « remonter » la donnée jusqu'au processeur.

Voilà donc un calcul mis à jour de la proposition précédente :

$N = (1 \times 0{,}95)$

$+ ((5 \times 8) \times 0{,}05 \times (1-10^{-5})+1)$

$+ ((10^{6} \times 4000) \times 10^{-5} + (5 \times 8) + 1)$

1 : Nombre de cycles nécessaires pour accéder à la donnée dans la mémoire statique.

0,95 : Probabilité de trouver la donnée dans la mémoire statique.

5 x 8 : Le chargement d'un bloc nécessite 8 transferts demandant chacun 5 cycles.

0,05 : Probabilité de ne pas avoir trouvé la donnée sur la mémoire statique.

$1-10^{-5}$: Probabilité de trouver la donnée dans la mémoire dynamique.

1 : Une fois le bloc chargé en mémoire statique, il faut transférer la donnée vers le processeur, ce qui requiert 1 cycle.

10^{6} x 4000 : Le chargement d'une page de 4000 octets nécessite 1000 transferts demandant chacun 10^{6} cycles.

10^{-5} : Probabilité de ne pas trouver la donnée dans les deux mémoires précédentes.

5 x 8 : Une fois la page chargée en mémoire dynamique, il faut transférer le bloc (issu de la page) vers la mémoire statique, ce qui requiert 8 transferts de 5 cycles chacun.

1 : Une fois le bloc chargé en mémoire statique, il faut transférer la donnée vers le processeur, ce qui requiert 1 cycle.

On obtient finalement : **N = 40045 cycles**

Encore une fois, ce coût n'est pas acceptable, et risque de détériorer sérieusement les performances de notre processeur si il est laissé tel quel.
Il faut donc chercher de nouvelles solutions pour faire baisser ce nombre moyen de cycles.

Dans les deux expressions présentées précédemment, nous pouvons extraire une « composition » commune. En effet, chaque calcul repose sur l'expression suivante :

$$N = \sum p_i \times c_i$$

Probabilité de miss — Coût du transfert

La probabilité de miss est un paramètre très complexe, et par conséquent, nous aurons beaucoup de mal à le faire évoluer.
Le **coût du transfert** est donc notre seul moyen d'agir sur le nombre moyen de cycle. Ce coût est donné par la somme :

- du temps d'accès à la mémoire (paramètre matériel… Impossible à changer)
- du nombre d'accès à la mémoire

▶ **Ce nombre d'accès est donc le seul paramètre sur lequel nous puissions agir.**

Dans notre modèle actuel, la capacité de la liaison inter-mémoire (de 4 octets) impose un nombre élevé d'accès. L'objectif serait donc d'augmenter cette capacité afin de faire tendre le nombre d'accès vers 1.

Il faudrait pour cela, être capable de transférer 4Ko (entre le disque dur et la mémoire dynamique) d'un seul coup. Cette solution imposerait de « câbler » cette liaison avec 4000 fils (1 par octet) au lieu de 4 aujourd'hui.

Multiplier par 1000 le nombre de fils est totalement irréalisable.
Il faut trouver (encore une fois) une autre solution.

Chapitre 8

Notion de Bus Système

Nous avons démontré dans le chapitre précédent que la stratégie utilisée pour l'accès aux données dans les diverses mémoires se révélait déterminante dans les performances du système entier. Quelles nouvelles méthodes pouvons-nous mettre en place pour améliorer la simple organisation des mémoires proposée ?

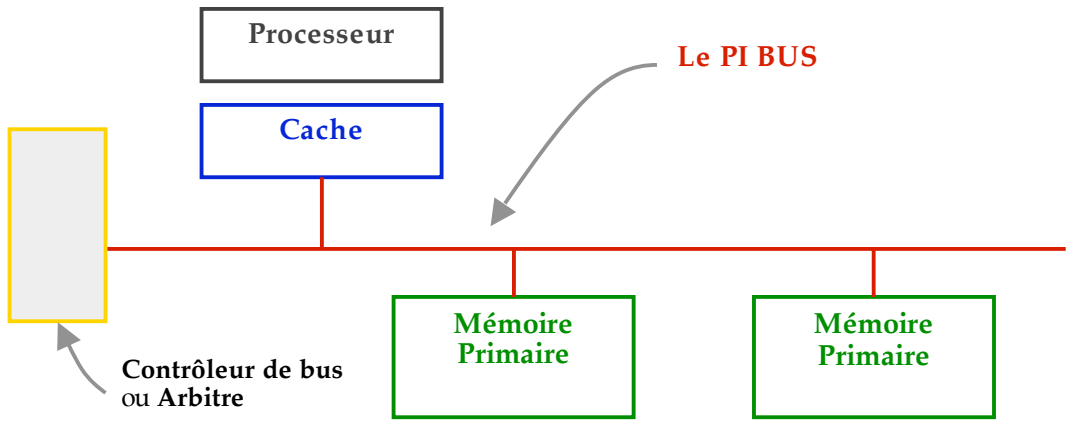
1. Echanges entre les diverses mémoires.

Le but de ce cours est d'étudier plus précisément les moyens d'accès aux données :

- entre cache et mémoire principales
- entre cache et disque

A. *Environnement du bus*

Nous considérerons dans un premier temps les échanges entre le cache et la mémoire primaire. Ces deux entités sont mises en relation par un bus : le PI BUS



Ce schéma nous permet de faire quelques remarques :

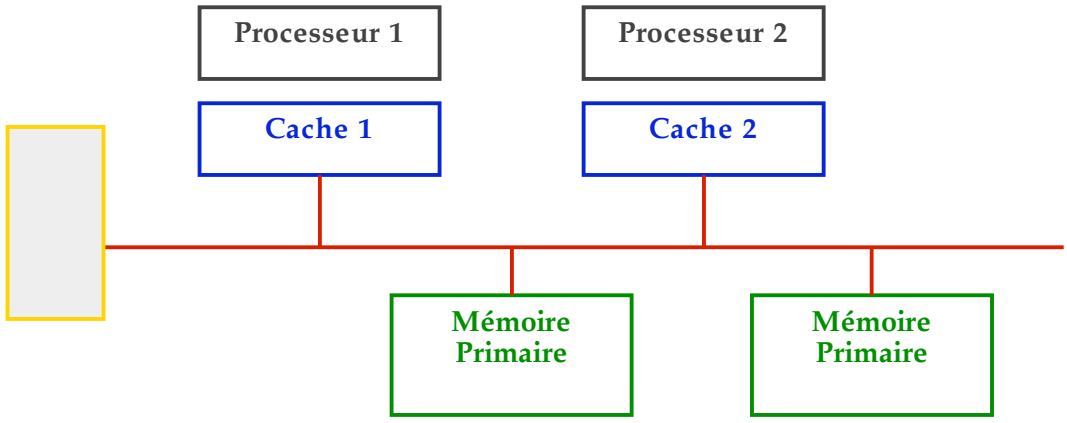
► La mémoire primaire (ou mémoire principale) n'est pas une seule entité mais se compose de plusieurs occurrence de mémoire.

► Certains composants se trouvent « au-dessus » du bus et d'autre « au-dessous ». On définit alors 2 catégories de composants : les maîtres et les esclaves.

► Un composant n'est ni au-dessus, ni au-dessous du bus. Il s'agir de l'arbitre qui « surveille » et « coordonne » les échanges sur le bus

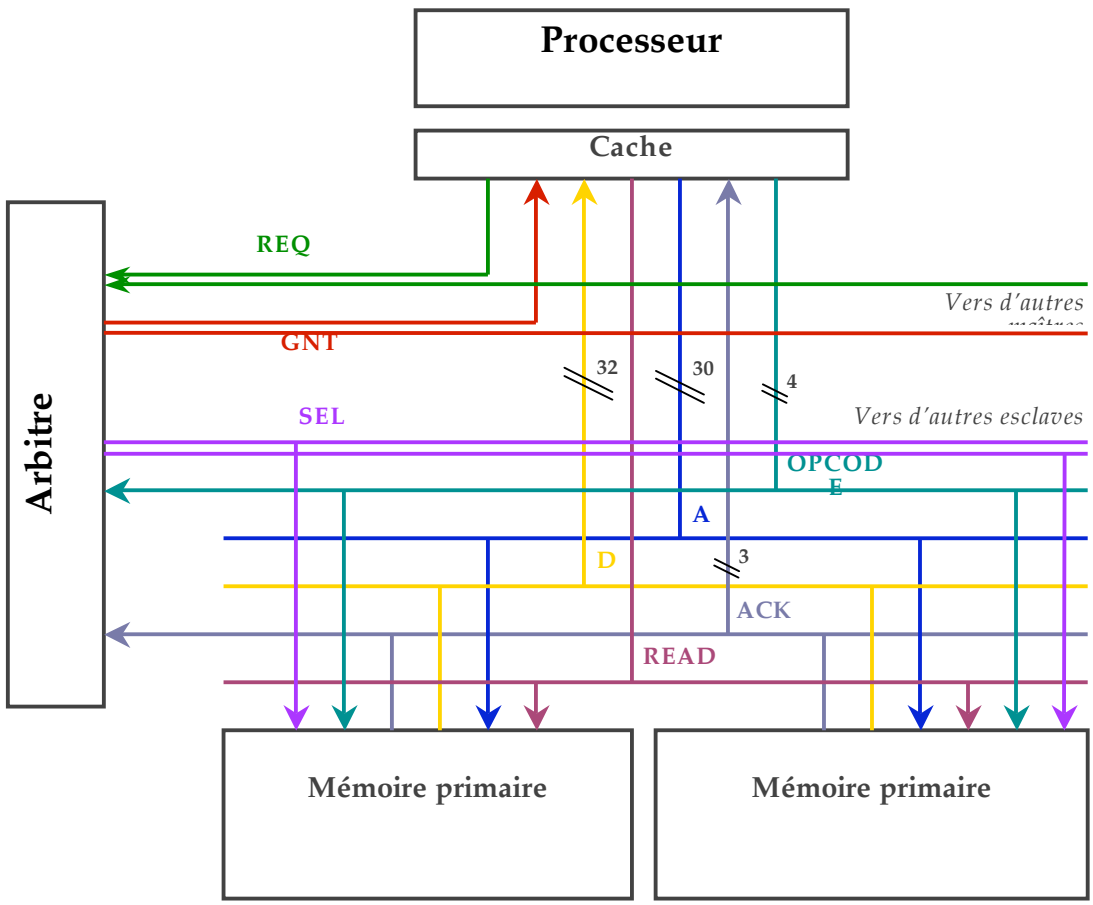
Le comportement des composants va nous permettre de les classer dans les catégories énoncées ci-avant. Ainsi, le cache, composant **capable de prendre l'initiative** en commandant un transfert, est défini comme **maître**.
Les mémoires sont, par conséquent, définies comme esclaves.

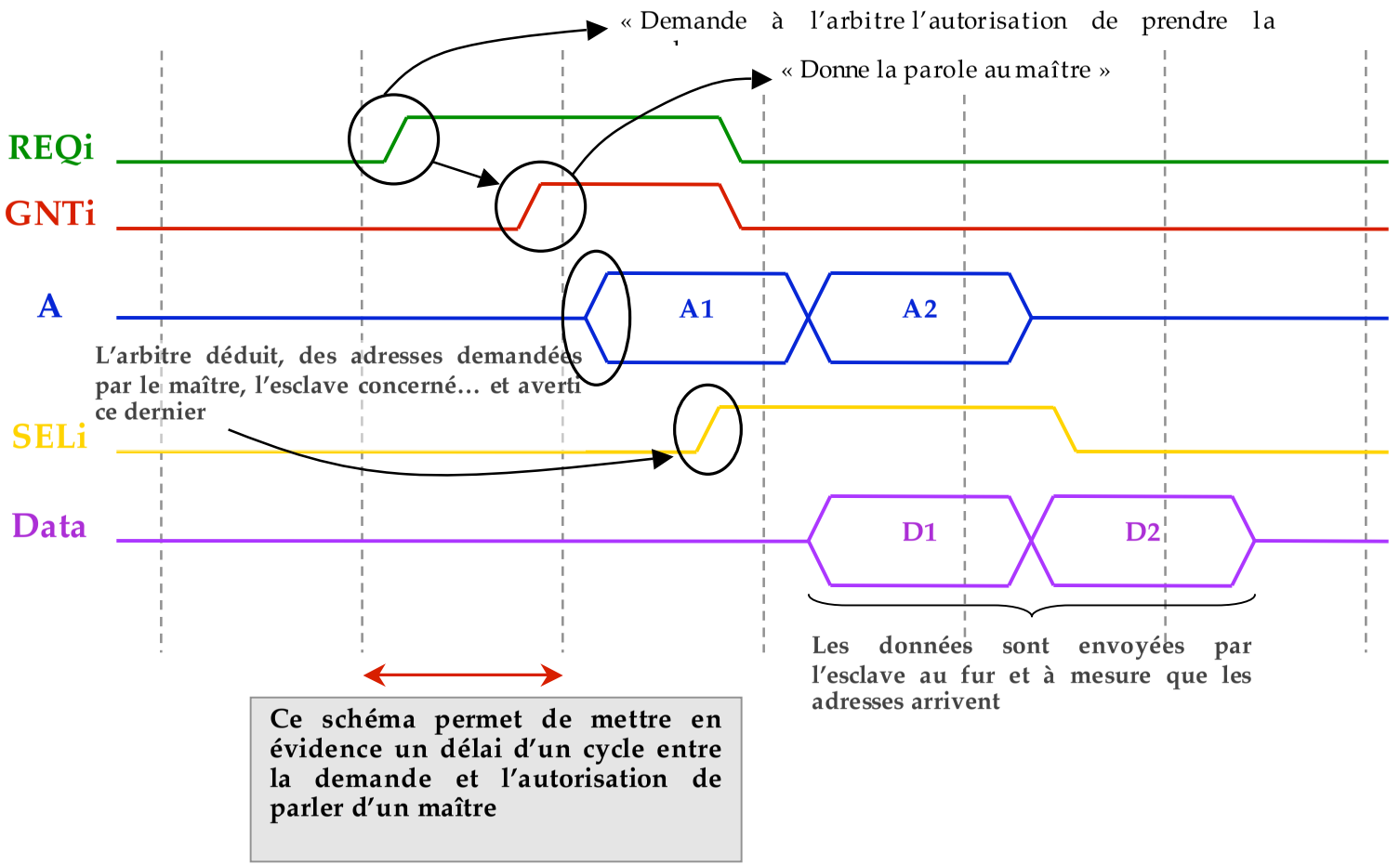
On peut, une fois cette hiérarchie définie, proposer une architecture plus complexe mettant en jeu plusieurs processeurs interagissant avec plusieurs occurrences de mémoires primaires.



B. *Détails du PI BUS*

Le bus (ici représenté en rouge) est en fait un ensemble de signaux (câbles) reliant les divers composants. On peut ainsi détailler le schéma de façon à faire apparaître les liaisons entre les maîtres, les esclaves et l'arbitre :





Désigner un « **maître par défaut** » permet d'annuler ce délai (dans le cas où le maître désirant parler est effectivement le défaut). Reste à déterminer la façon de choisir le maître par défaut.

Le cas présenté ci-avant se révèle plutôt simple. Il s'agit simplement, pour l'arbitre, de faire en sorte que **l'ordre des étapes de communication** soit respecté :

- Demande de prise de bus
- Autorisation d'utilisation du bus
- Annonce des adresses
- Détermination de l'esclave
- Échange de données

La situation devient beaucoup plus complexe si l'on considère des bus reliant **plusieurs maîtres** à plusieurs esclaves. Dans ce cas, l'arbitre devra non seulement gérer la communication et l'échange de données entre un maître et un esclave, mais aussi donner la priorité à un maître plutôt qu'à un autre et instaurer ainsi une **véritable politique de prise de bus**.

2. Stratégies d'arbitrage

Afin de gérer au mieux les conflits risquant d'apparaître entre les différents maîtres, l'arbitre doit mettre en place une stratégie d'arbitrage.

On en dénombre plusieurs dont les principales sont :

A. *Stratégie Statique*

Il s'agit de **numéroter les divers maîtres** connectés au bus.
Cette numérotation implique alors un ordre de prise de parole.

En considérant ainsi 5 maîtres numérotés : M1, M2, M3, M4, M5 tous connectés au PIBUS et souhaitant tous « s'exprimer », nous serons capable, en utilisant cette stratégie, de **déterminer l'ordre** dans lequel les maîtres devront prendre la parole :

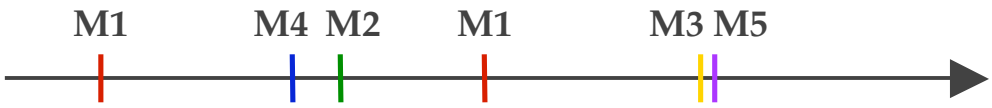
D'abord M1, puis M2, M3, M4 et enfin M5.

Cette stratégie très simple à implémenter se révèle toutefois dangereuse car entraînant un **risque évident de famine**.

En effet, considérons le même exemple que précédemment.
Ajoutons le fait que M2 est très « bavard ». L'arbitre n'aura que faire des maîtres M3, M4, et M5 et donnera systématiquement le bus à M2. Cela constitue donc très clairement une situation de famine qui empêchera tous les maîtres numérotés > 2 de réquisitionner le bus.

B. *Stratégie Dynamique*

On attribue le bus aux maîtres en tenant compte, cette fois-ci, de la date de la requête. Cette solution est aisément représentable sur une ligne du temps :



Sur cet exemple, la priorité sera donnée aux maîtres suivants (dans cet ordre) :

M1 → M4 → M2 → M1 → M3 → M5

Les risques de famine sont largement diminués en utilisant cette stratégie, mais un **problème d'implémentation** handicape cette technique. En effet, il faut à chaque demande (GNT) d'un maître enregistrer la date. Cet enregistrement peut se faire à l'aide d'un timestamp (**estampillage**), mais nécessite un « investissement » matériel plus important…

Ce problème de coût nous amène donc à proposer une nouvelle stratégie.

C. *Stratégie à Priorités Dynamiques*

Cette technique aussi appelée « stratégie à priorité tournante » reprend les points positifs de chacune des deux stratégies précédentes et les implémente de façon à contourner les problèmes évoqués ci-avant.

Cette stratégie se compose de 2 phases :

- La première partie concerne **l'initialisation d'une liste de priorité**. Cette liste utilise une numérotation (cf. stratégie statique). Chaque maître se voit donc attribuer un numéro d'ordre.

- La seconde partie de l'algorithme met en place la gestion de cette liste, en imposant à chaque maître ayant obtenu le bus de se positionne en fin de liste accompagné des maîtres qui le précédaient :

(priorité haute) **M1 → M2 → M3 → M4 → M5**

***M4** demande le bus… OK pour **M4**… Repositionnement des maîtres de priorités plus fortes.*

M5 → M1 → M2 → M3 → M4

***M2** et **M4** demandent le bus… **M2** plus prioritaire… OK pour **M2** : + repositionnement*

M3 → M4 → M5 → M1 → M2

***M5** et **M1** demandent le bus… **M5** plus prioritaire… OK pour **M5** : + repositionnement*

M1 → M2 → M3 → M4 → M5

Ce principe nous permet notamment d'attribuer le statut de « **maître par défaut** » au premier maître de la liste. En effet, le premier de la liste n'a pas pris le bus depuis plus longtemps que les autres et a donc de grande chance de le demander.

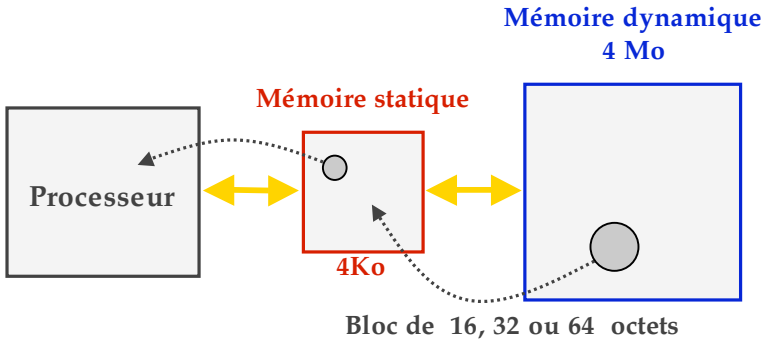
Chapitre 9

Caches & Cohérence

Les chapitres 7 et 8 nous ont permis de mettre en évidence la présence de matériel permettant une communication efficace entre le(les) processeur(s) et la (les) mémoire(s). Nous avons détaillé l'implémentation du PI BUS, entrons maintenant dans les détails de l'implémentation des caches…

1. Gestion du cache

Rappelons dans un premier temps le fonctionnement global du système des mémoires :



L'unité de transfert entre la mémoire dynamique et la mémoire statique est **le bloc**.
La mémoire statique doit donc être capable de gérer l'arrivée d'une telle quantité de données et, de préférence, minimiser le rechargement de données fréquemment utilisées.

Ces deux points constituent la principale préoccupation qui nous animera tout au long de cette étude.

On distingue plusieurs façons de gérer le cache :

- **Le Direct Mapping** *(Cache à correspondance directe)*

Ce type de cache associe un emplacement fixe à un bloc. Cet emplacement est définitif !

- **Le Set Associative** *(Cache associatif à ensemble de blocs)*

Chaque bloc peut être associé à un groupe d'emplacement.

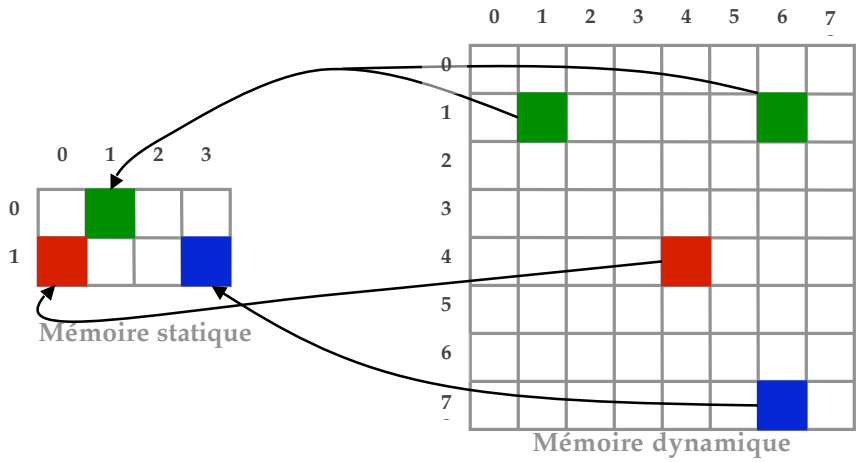
- **Le Full Associative** *(Cache totalement associatif)*

Chaque bloc peut être associé à n'importe quel emplacement.

A. *Le « Direct Mapping »* ***(Cache à correspondance directe)***

Comme énoncé précédemment, chaque bloc extrait de la mémoire principale doit trouver une correspondance unique dans la mémoire statique. Il convient donc de **partitionner**, c'est à dire, de découper la mémoire statique en fonction de la taille de la mémoire dynamique afin de constituer une relation entre les deux mémoires.

Considérons dans notre premier exemple que la mémoire dynamique propose 64 blocs de données alors que la mémoire statique ne dispose que de 8 blocs. Sachant que chaque bloc contient 16 octets de données, on en déduit que la mémoire dynamique contient 1024 octets de données (soit 1 Koctet) et le cache : 128 octets.
Une première stratégie consiste à associer un bloc de la mémoire statique pour tous les blocs d'une même ligne de la mémoire dynamique.



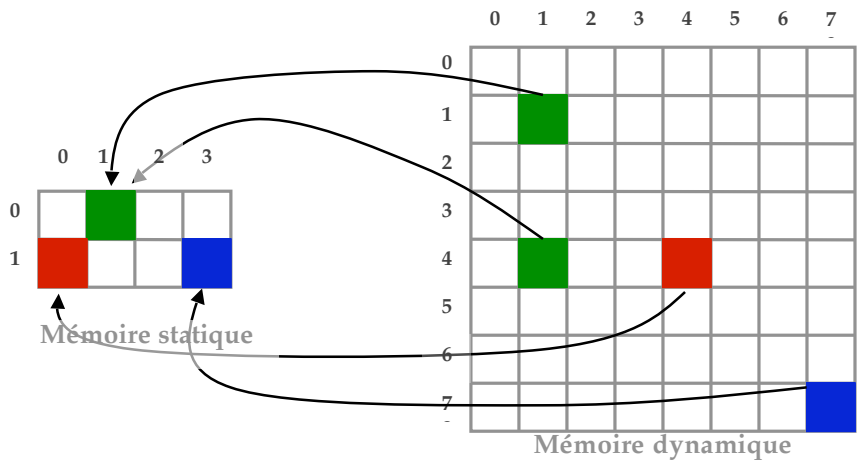
Cette disposition permet au cache de gérer une mémoire de 64 cases alors que lui-même n'en dispose que de 8. Cependant, un problème apparaît lorsqu'on évoque le principe de **la localité spatiale.**

En effet, nous avons pu constater dans les chapitres précédents que les données étaient souvent stockées de façon contiguë en mémoire. En imaginant, dans cet exemple, que l'on souhaite charger deux blocs contigus en mémoire dynamique, il se produira un effacement puis un rechargement de la case du cache concernée.

Cette stratégie, peu efficace dans la majorité des cas, nécessite donc quelques ajustements pour permettre une gestion efficace lors de l'exécution de programmes chargés de façon séquentielle en mémoire.

Nous proposons donc une stratégie associant une case du cache avec une colonne de la mémoire dynamique. Ainsi, de la même façon que précédemment, chaque bloc d'une colonne **j** de la mémoire dynamique est associé à la case **j** du cache.

Voici donc cette stratégie :



Le cas du bloc vert et rouge illustre le fait que deux blocs « proches » en mémoire dynamique se verront quand même distingués en mémoire statique.

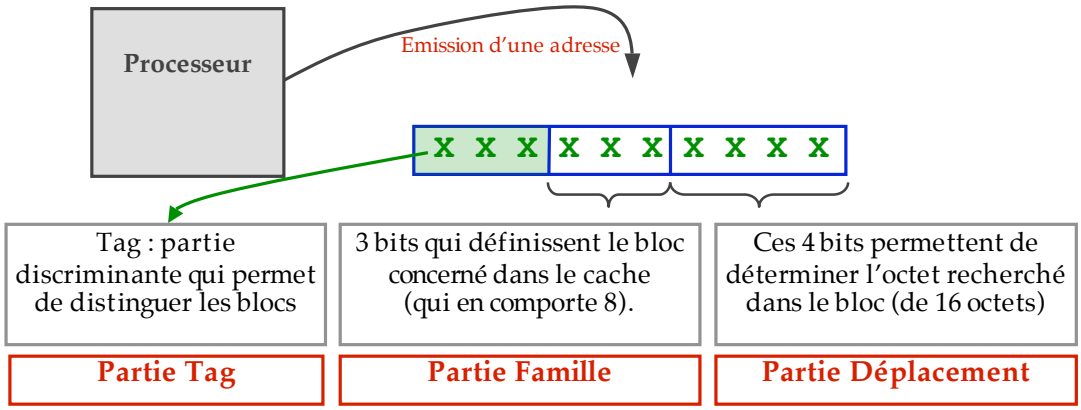
Cette stratégie utilise l'intéressante propriété mathématique de la répartition en colonne qui préserve la distance dans le cache de 2 blocs en mémoire dynamique (pourvu qu'elle soit « faible »)

B. *Le point de vue du processeur*

Il ne faut pas oublier que les blocs de données (ou d'instruction) sont demandés par le processeur. Ce dernier émet donc ses requêtes sous forme d'adresses de données à trouver dans le cache.

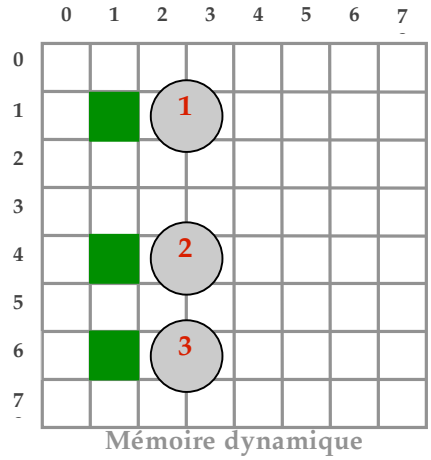
Nous avons vu précédemment que le processeur s'adresse à une mémoire disposant de 1024 octets de données. Le processeur peut donc accéder à l'une de ces 1024 données. Il doit donc pouvoir émettre 1024 adresses.

Par conséquent, les adresses seront sur 10 bits (2^{10}= 1024)



Certaines explications sont nécessaires pour la compréhension de l'utilité de la partie Tag dans l'adresse. Nous avons vu précédemment (dans l'explication des différentes stratégies d'organisation) que plusieurs blocs de la mémoire dynamique sont associés au même bloc de la mémoire statique.

En considérant la stratégie de la répartition en colonne (dernière stratégie étudiée), on peut représenter le problème de la façon suivante :



Les trois cases vertes présentées ci-dessus sont associées à la même case de la mémoire statique. Étudions quelles sont leurs adresses :

- Cas n°**1** : Adresse de la case : **0010010000**
- Cas n°**2** : Adresse de la case : **1000010000**
- Cas n°**3** : Adresse de la case : **1100010000**

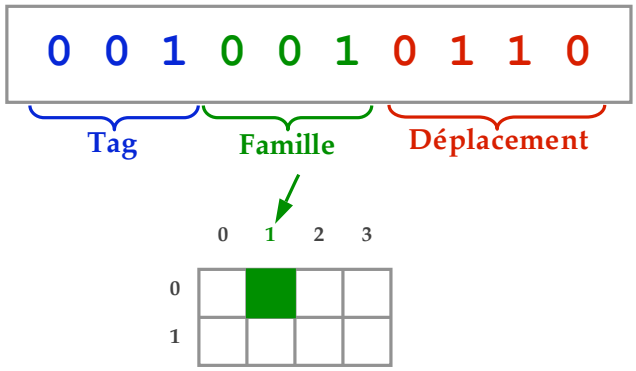
À noter les adresses des cases particulières :
Case 0-0 : `0000000000` *et Case 7-7 :* `1111110000`

↳ **On suppose maintenant que le cache (mémoire statique) est vide.**

↳ **Le processeur demande l'adresse : 0010010110 (case 1-1).**

Le cache étant vide, le bloc de données doit être rapatrié depuis la mémoire dynamique.
Une fois récupéré, il faut le stocker dans le cache (mémoire statique) avant d'envoyer la donnée vers le processeur.

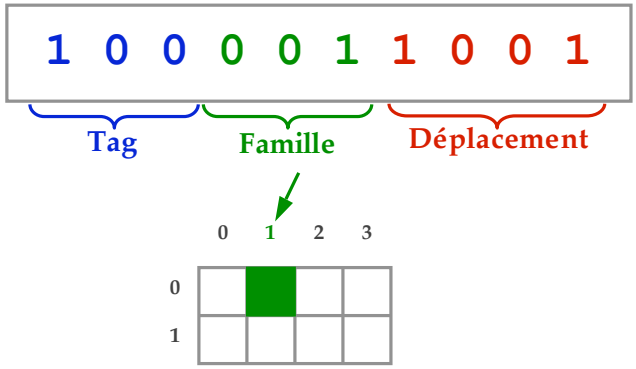
Le schéma de décomposition de l'adresse présenté ci-avant sera alors appliqué :



L'analyse de l'adresse désigne le bloc 1.
Cette décomposition nous permet donc de stocker au bon endroit (dans le cache) le bloc provenant de la mémoire dynamique.

↳ **Le processeur demande l'adresse : 1000011001 (case 1-4).**

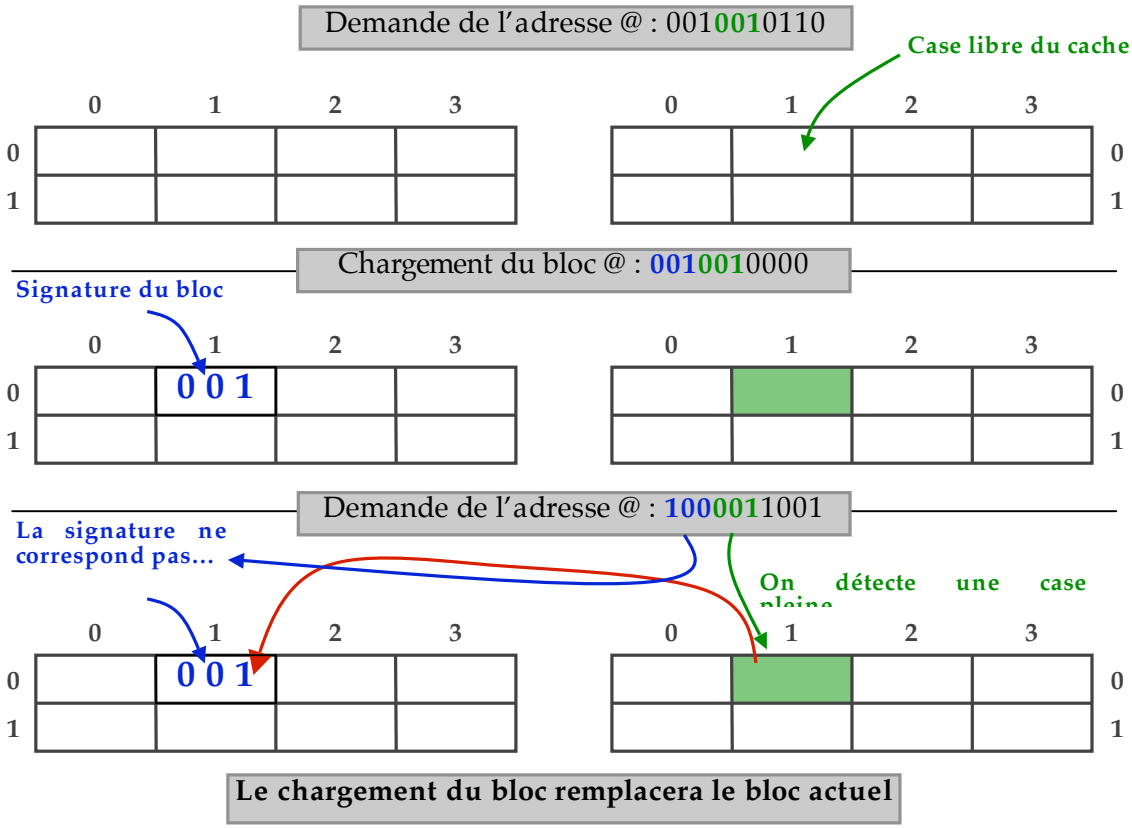
La première opération faite par le cache, est de vérifier si la donnée demandée n'est pas déjà présente dans le cache. Pour ce faire, l'adresse est analysée :



L'analyse de l'adresse indique le bloc 1 du cache.

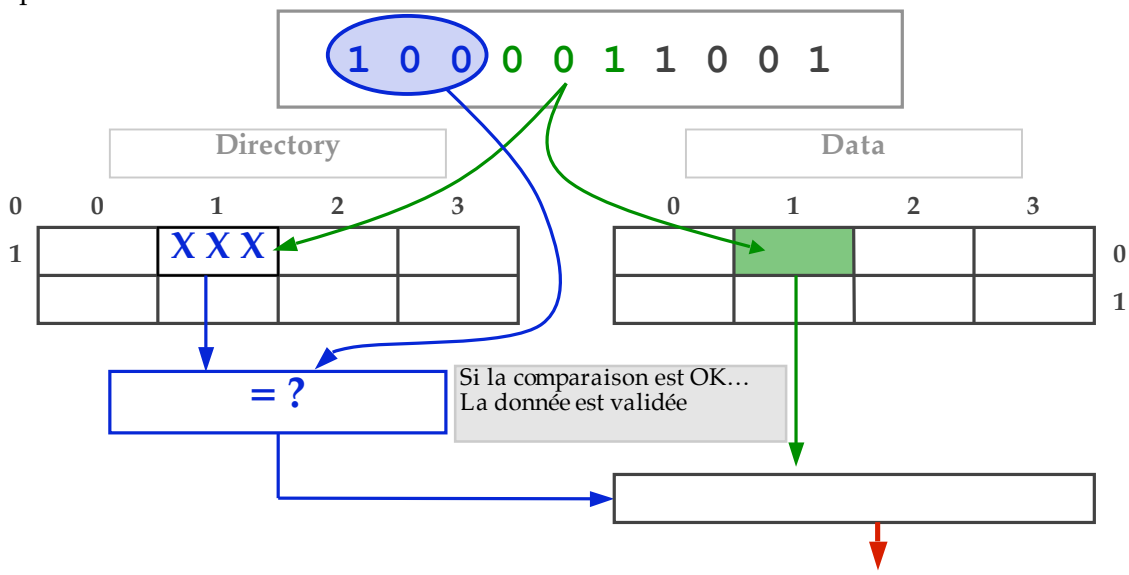
La logique voudrait que l'on envoie au processeur ce bloc puisque désigné par l'analyse de l'adresse. Cependant, les données de ce bloc **ne sont absolument pas celles qui sont attendues !** Souvenez-vous, il s'agit des données de la case 1-1 de la mémoire dynamique (voir ci avant)

Il faut donc mettre en place un mécanisme qui permette de distinguer les divers blocs ramenés dans la mémoire statique (cache). On implémente donc un système de **directory**, qui stocke, associé à chaque bloc du cache, un tag (une signature) permettant de distinguer les blocs.



Ce mécanisme de « **Directory** » permet donc de distinguer les divers blocs de la mémoire dynamique qui sont associés à une même case de la mémoire statique.

La comparaison entre les tags est assurée par un câblage utilisant entre autres des comparateurs :



Cette méthode impose cependant un espace supplémentaire pour stocker les tags.

Dans cet exemple :

- La mémoire dynamique représente : **1024 octets**
- La mémoire statique (cache) :
  - La partie données (DATA) : **128 octets**
  - La partie tag (DIRECTORY) : 8 x 3bits = 24 bits = 24/8 octets = **3 octets**

Dans un exemple plus proche de la réalité :

- La mémoire dynamique : 128 Moctets (en blocs de 16 octets)

L'adresse est donc donnée sur : 2^{27} **= 128 x 10^6 ⇨27 bits**
La partie « **déplacement** » de l'adresse est donnée sur : 2^4 = 16 ⇨ **4 bits**

- La mémoire statique (cache) :
  - La partie données (DATA) : 1 Kbloc = 1024 blocs = 16 Koctets

La partie « **famille** » de l'adresse est donnée sur : 2^{10} **= 1024 ⇨ 10 bits**
Le tag est donc sur : 27-4-10 = 13bits

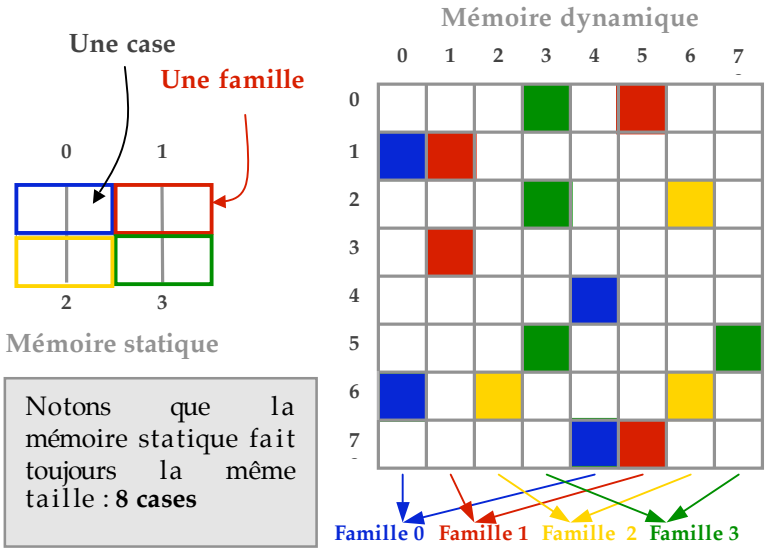
- La partie tag (DIRECTORY) représente donc : 13 x 1024 = 13312 bits
  = 1,6 Ko

C. *Le « Set Associative »*
(Cache associatif à ensemble de blocs)

Cette technique est une évolution de la précédente, dans le sens ou les blocs provenant de la mémoire dynamique disposent de plus de « choix de destination ».

En effet, là où dans la technique précédente, chaque bloc de la mémoire dynamique se voyait associer un unique bloc de la mémoire statique, les blocs de la mémoire dynamique disposent maintenant de plusieurs « destinations » possibles dans la mémoire statique.

La technique de la répartition par colonne est encore une fois utilisée…



Notons que la mémoire statique fait toujours la même taille : **8 cases**

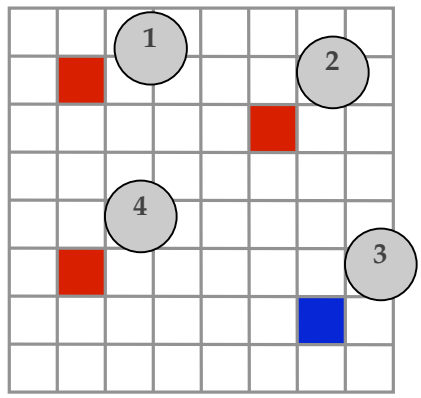
Ce schéma nous indique que chaque bloc de la mémoire dynamique peut être stocké dans 2 emplacements de la mémoire statique. On appelle donc ce cache : **Cache 2 Set Associative**

Si l'on considère l'adresse fournie par le processeur, le découpage :
« **tag - famille - déplacement** » reste le même. Cependant, on dénombre deux fois moins de familles que dans la stratégie Direct Mapping (dans cette stratégie : 1case = 1 famille).

Ainsi, là où on avait 3 bits pour désigner la famille, on n'en compte plus que 2 (puisqu'il ne reste que 4 familles possibles).

Le système de tag est conservé, et deux tags sont enregistrés pour les deux cases d'une même famille (voir exemple suivant).

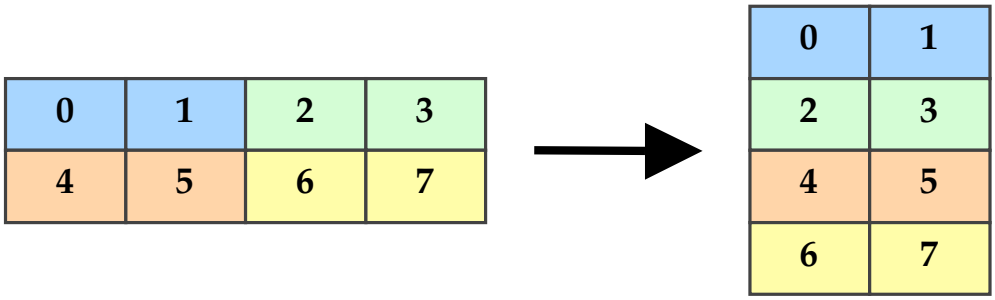
L'exemple suivant s'appuie sur la disposition suivante de la mémoire.



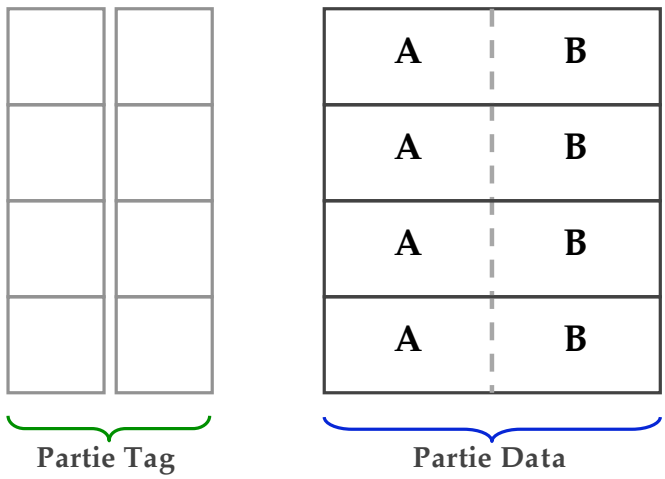
Adresse **Case 1** : `0010010000`
Adresse **Case 2** : `0111010000`
Adresse **Case 3** : `1101100000`
Adresse **Case 4** : `1010010000`

Avant de détailler l'exemple, définissons la structure du cache :

Notons le changement de « présentation » :



Le cache dans le reste de l'exemple sera présenté sous la forme suivante :



Nous rajoutons aussi à cette méthode quelques « bits de contrôle » dans la partie Tag.

- Un bit LRU
- Un bit de présence

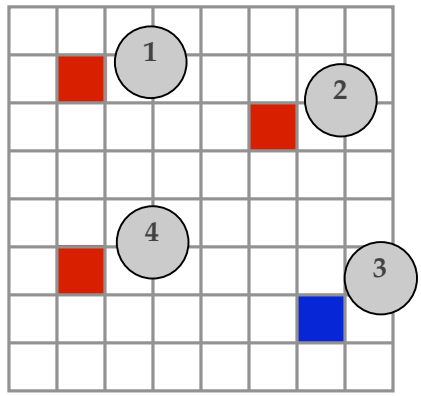
Le bit LRU est là pour indiquer la case de la famille qui doit être allouée au bloc arrivant dans la mémoire statique. En effet, ce type de cache permet de gérer une nouvelle caractéristique de nos programmes : **la localité temporelle**.

Le fait de disposer de deux cases pour une même famille donne 2 fois plus de chances à un bloc d'être utilisé avant d'être remplacé. Le bit LRU est justement là pour indiquer lequel des deux blocs est le plus ancien dans la famille… et donc le plus apte à être remplacé.

Le bit de présence quant à lui permet d'indiquer si le bloc est effectivement chargé en mémoire. Cette stratégie permet notamment de « supprimer » un bloc de la mémoire statique sans avoir besoin de vider le bloc en question. Il s'agit, tout simplement, de mettre le bit de présence à zéro pour indique que la case est non utilisable.

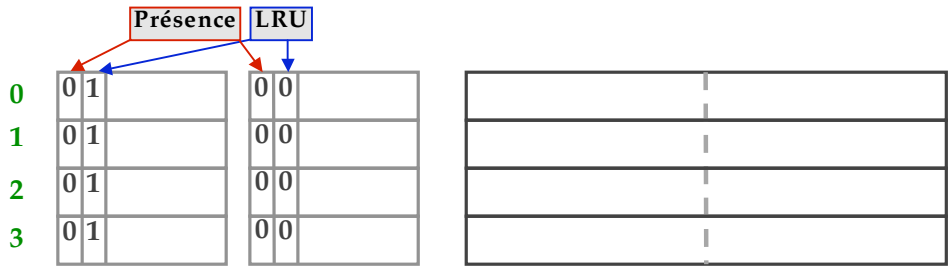
Détails de l'exemple

Rappel : L'exemple suivant s'appuie sur la disposition suivante de la mémoire.



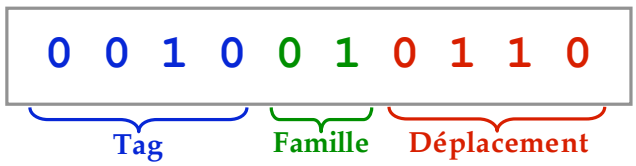
Adresse **Case 1** : **0010010000**
Adresse **Case 2** : **0111010000**
Adresse **Case 3** : **1101100000**
Adresse **Case 4** : **1010010000**

↳ **Etat initial :**

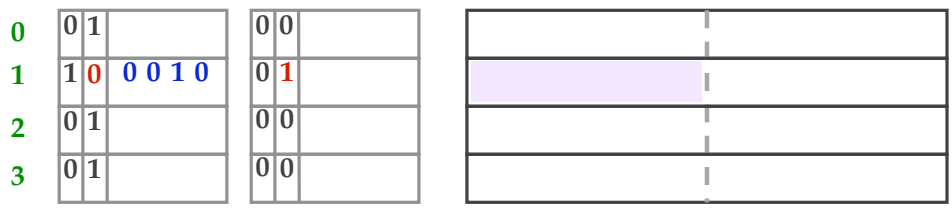


↳ **Le processeur demande l'adresse : 0010010110.**

L'adresse 0010010**110** correspond au bloc 0010010**000**.
Nous devons dans un premier temps vérifier que le bloc n'est pas déjà chargé en mémoire statique. Il faut donc « découper » l'adresse pour trouver la case associée du cache.



Aucun des deux bits de présence de la famille 1 n'est à 1.
Le bloc doit donc être chargé depuis la mémoire dynamique et stocké dans la famille 1, plus particulièrement dans la case du cache possédant le bit LRU à 1

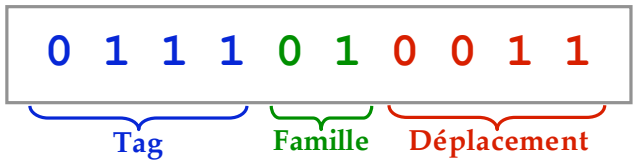


Notons l'inversion du bit LRU dans la famille n°1. La case la plus apte à recevoir une donnée étant la case la plus anciennement modifiée (ici, celle qui n'a pas encore été modifiée)

☑ **Une fois la donnée chargée dans le cache, le processeur peut lire la donnée voulue**

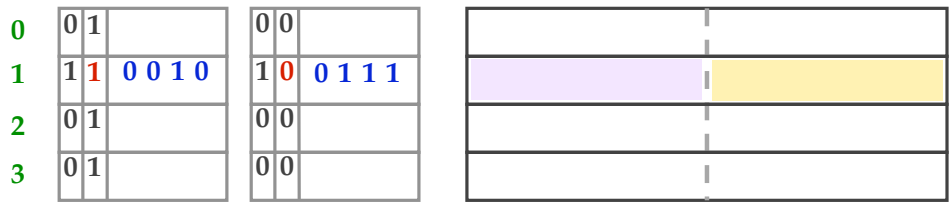
↳ **Le processeur demande l'adresse : 0111010011:**

L'adresse 0111010011 correspond au bloc 0111010000.
Nous devons dans un premier temps vérifier que le bloc n'est pas déjà chargé en mémoire statique. Il faut donc « découper » l'adresse pour trouver la case associée du cache.



Un des bits de présence des cases de la famille est positionné à 1.
Le tag correspondant est : **0010**. Ce dernier ne coïncide pas avec le tag du bloc que nous essayons d'accéder (**0111**).

Le bloc doit donc être chargé depuis la mémoire dynamique et stocké dans la famille 1, plus particulièrement dans la case du cache possédant le bit LRU à 1 (case de droite)

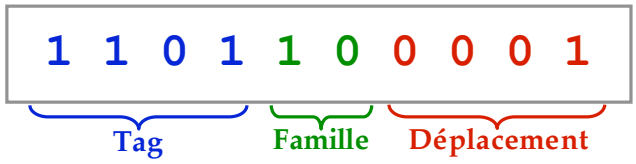


Encore une fois le bit LRU est inversé dans les cases de la famille concernée par le chargement.

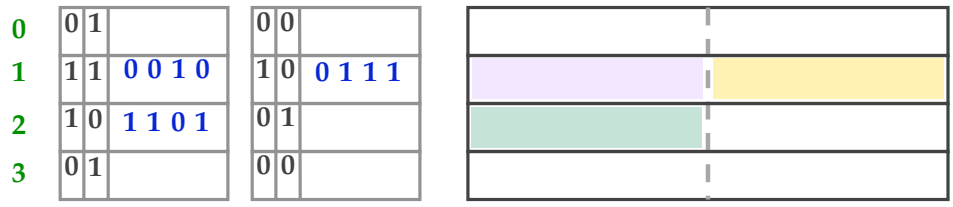
☑ **Une fois la donnée chargée dans le cache, le processeur peut lire la donnée voulue**

↳ **Le processeur demande l'adresse : 1101100001:**

L'adresse 1101100001 correspond au bloc 1101100000.
Nous devons dans un premier temps vérifier que le bloc n'est pas déjà chargé en mémoire statique. Il faut donc « découper » l'adresse pour trouver la case associée du cache.



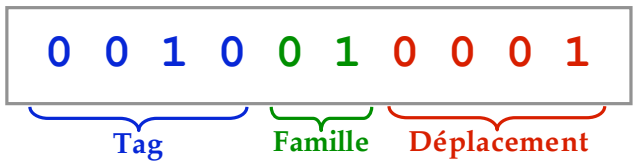
Aucun des bits de présence des cases de la famille n'est positionné à 1.
Le bloc doit donc être chargé depuis la mémoire dynamique et stocké dans la famille 2, plus particulièrement dans la case du cache possédant le bit LRU à 1 (case de gauche)



☑ Une fois la donnée chargée dans le cache, le processeur peut lire la donnée voulue

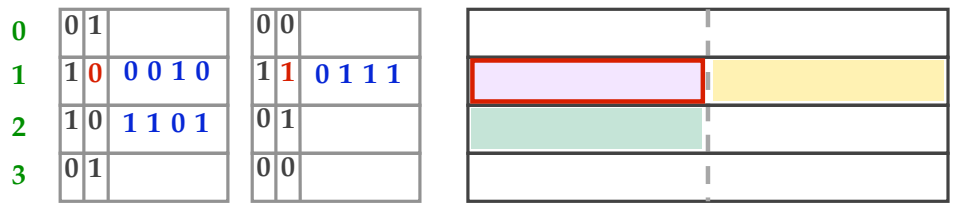
↳ Le processeur demande l'adresse : 0010010001:

L'adresse 0010010001 correspond au bloc 0010010000.
Nous devons dans un premier temps vérifier que le bloc n'est pas déjà chargé en mémoire statique. Il faut donc « découper » l'adresse pour trouver la case associée du cache.



Les bits de présence des cases de la famille sont tous positionnés à 1.
Nous devons donc vérifier les tags des blocs chargés. Notre tag (**0010**) correspond au tag de la case de gauche de la famille 1 (**0010**). Le bloc présent dans cette case est donc celui recherché.

Nous pouvons donc directement récupérer la donnée voulue (indiquée par le déplacement).



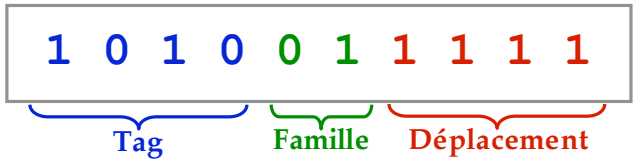
☑ La donnée est directement lue par le processeur dans le cache (pas d'accès au disque).

Notons le changement du bit LRU. En effet, même lors d'un accès, ce dernier est inversé. La localité temporelle indiquant que la donnée la plus récemment accédée (chargée ou lue) a plus de chance d'être réutilisée, que d'autres plus anciennes, dans un futur proche.

↳ Le processeur demande l'adresse : 1010011111:

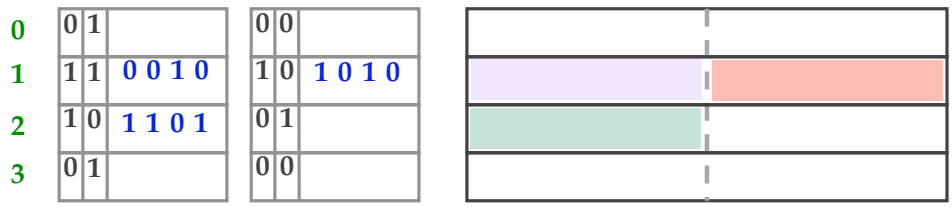
L'adresse 1010011111 correspond au bloc 1010010000.

Nous devons dans un premier temps vérifier que le bloc n'est pas déjà chargé en mémoire statique. Il faut donc « découper » l'adresse pour trouver la case associée du cache.



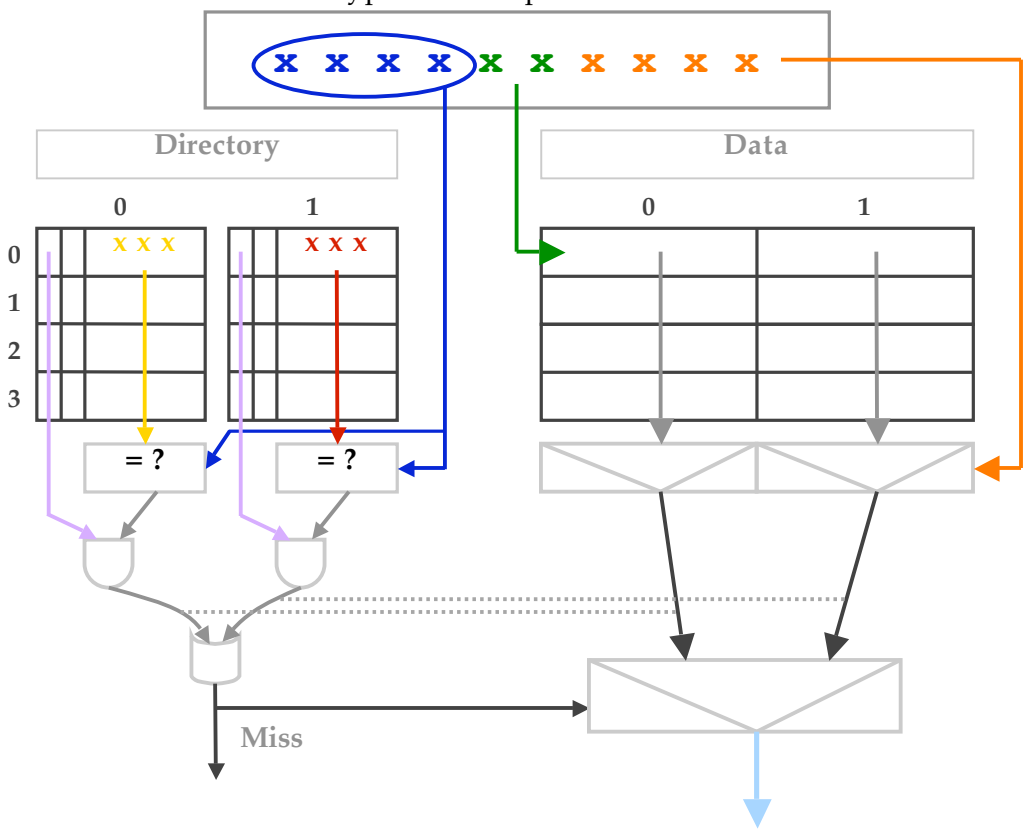
Les bits de présence des cases de la famille sont tous positionnés à 1.
Nous devons donc vérifier les tags des blocs chargés.
Aucun tag présent dans les cases de ce cache ne correspond au nôtre (1010).

Le bloc va donc être chargé depuis la mémoire dynamique dans la case indiquée par le bit LRU (case de droite)



☑ **Une fois la donnée chargée dans le cache, le processeur peut lire la donnée voulue**

L'implémentation matérielle de ce type de cache peut-être résumée dans un schéma comme :



En résumé, cette technique du **Cache Set Associative** se révèle plus complexe que le cache **Direct Mapping**, mais plus performante dans un certain nombre de cas (que nous allons énumérer dans peu de temps).

En effet, alors que la technique du Direct Mapping nécessite la présence d'un comparateur, le cache utilisant la méthode du Set Associative en utilise deux. Plus généralement, tout le matériel d'un cache Direct Mapping est doublé dans l'utilisation d'un cache Set Associative.

Du coté des avantages, nous avons vu plus haut que le cache Set Associative permettait de prendre en compte la **localité temporelle** des programmes. Le taux de miss est donc censé diminuer.

Nous pourrions alors imaginer des caches Set Associative à plus de deux niveaux (4 ou 8)…

Cependant, il faut bien reconnaître que le principe de la localité temporelle dans les programmes n'est pas aussi marqué que celui de la localité spatiale. Il peut donc être intéressant de le prendre en compte sans pour autant en faire une priorité dans la gestion du cache.

Notons de plus que si un cache Associatif de niveau 2 double le matériel par rapport à un cache Direct Mapping, un cache associatif de niveau 8 demandera 8 fois plus de matériel qu'un cache Direct Mapping.

Le temps d'accès est donc dans ces derniers cas, et malgré la diminution potentielle du taux de miss, bien trop important. Le gain devient alors tout relatif voir négatif.

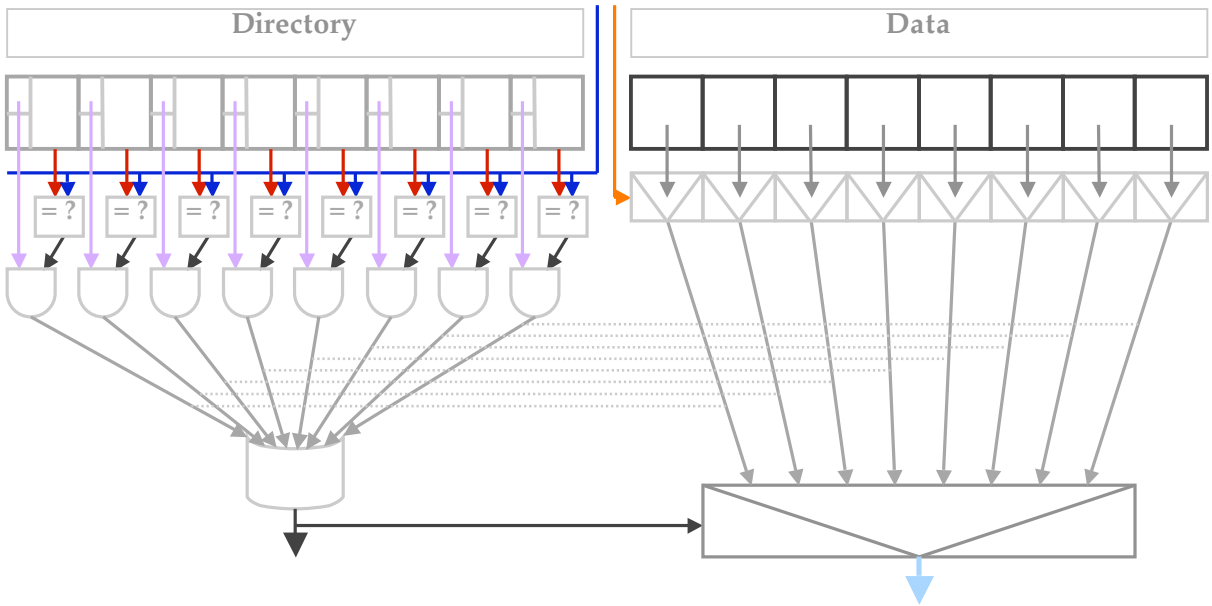
Il convient alors, pour trouver un bon compromis, de considérer et d'étudier les programmes amener à « tourner » sur ce processeur pour optimiser la gestion du cache en conséquence.

D. *Le « Full Associative »* ***(Cache totalement associatif)***

On peut remarquer d'après tous les exemples et démonstrations précédents, que le cache Direct Mapping n'est autre qu'un cache **Set Associative de niveau 1**. On peut alors naturellement se demander qu'elle serait la nature d'un cache Set Associative de niveau « maximum »…

Le fait d'être maximum impliquerait dans un premier temps **la disparition des familles**. Chaque bloc en provenance de la mémoire dynamique ayant par conséquent autant de choix de destination que de cases dans la mémoire statique.

Suivant les représentations utilisées depuis le début de ce cours, on pourrait alors proposer l'implémentation suivante :



La solution illustrée ci-dessus présente clairement la quantité de matériel nécessaire à l'implémentation d'une telle stratégie de gestion de cache. Notons que la solution proposée correspond à un cache de 128 octets, alors que les caches actuels proposent plusieurs Ko de mémoire.

Cette solution est donc rarement, voire jamais utilisée lors de la conception de cache.

E. *Conclusion*

L'étude de ces trois stratégies de gestion de cache nous permet d'en déduire qu'il n'existe en fait qu'une seule catégorie : **le Set Associative** :

- Direct Mapping = Set Associative de **niveau 1**
- Set Associative de **niveau n** (avec n < m) m étant le nombre de cases disponibles
- Full Associative = Set Associative de **niveau m**

Il s'agit donc de trouver le bon compromis sur la profondeur (le nombre de niveau) du cache. Les caches de profondeur 2 ou 4 se révèlent assez performant. Une profondeur supérieure est moins intéressante : la localité temporelle n'étant pas assez marquée dans la plupart des programmes.

Chapitre 10

Caches & Stratégies

La présentation générale des caches et des manières de les gérer nous a permis de prendre conscience qu'il existait de multiples façons d'implémenter ces mécanismes dans un système… Ce cours présente et détaille de nouvelles implémentations de caches, en les insérant dans l'ensemble : processeur-cache-bus-mémoire.

Si l'on considère un programme « normal » en assembleur MIPS, on peut aisément constater que 2 accès-mémoire peuvent être nécessaires à chaque cycle. Cette constatation est de plus renforcée par le détail de opérations faites lors de l'exécution d'une instruction au sein du processeur.

- **Etage I** (IFETCH) : Récupère la nouvelle instruction en mémoire
- **Etage M** (MEM) : Lecture ou Ecriture d'une donnée en mémoire

Notons que les données accédées en mémoire ne sont pas du même type dans les deux cas. Le premier étage accède aux instructions alors que le second vise plus particulièrement les données (« pures »)

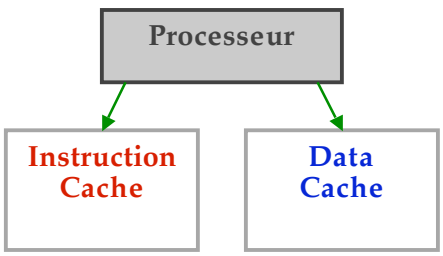
1. Gestion des lectures

Le cours précédent ne traitait que du cas d'un accès unique à la mémoire. Souvenez-vous, le processeur ne demandait qu'une adresse à la fois, et tout un cycle était nécessaire pour faire « remonter » la donnée en mémoire, si celle-ci se trouvait dans le cache.

Il est donc, pour le moment, impensable de demander 2 données simultanément au cache.

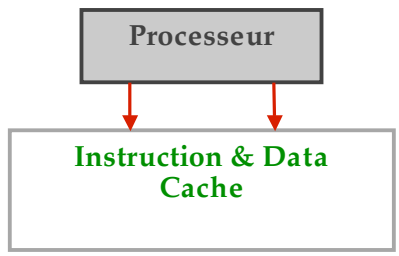
Ce problème devant être résolu, nous proposons deux implémentations différentes. Ces deux propositions vont être, dans les lignes suivantes, étudiées en parallèle afin de simplifier la comparaison entre les deux.

Solution n°1



Le processeur accède ici à deux caches distincts, de tailles a priori différentes et gérés de façons indépendantes.

Solution n°2

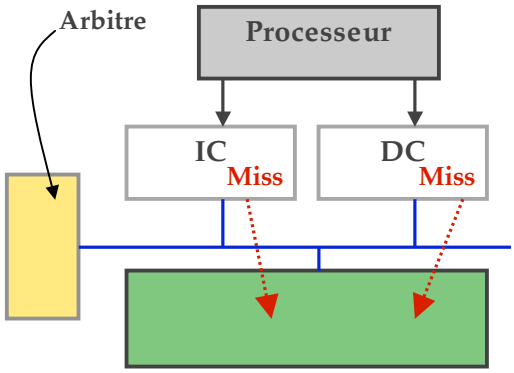


Le processeur exploite un cache unique qui contient aussi bien les instructions que les données. *La double liaison (en rouge) est optionnelle…*
Les deux caches présentés évoluent de façon autonome. Ils disposent chacun de leur partie **data** et **directory**. L'un peut être Direct-Mapping, l'autre Set-Associative ; les deux Set-Associative… D'autres combinaisons sont aussi envisageables
Il faut cependant faire remarquer que **l'espace d'adressage du processeur est**

toujours unique et qu'un problème peut donc apparaître en cas de double accès des caches à la mémoire.

En effet, si les caches sont séparés, la mémoire est, quant à elle, toujours unique et si, au cycle i, le cache d'instruction ET le cache de données provoquent un miss, **l'accès mémoire ne pourra être simultané**.

Cette situation est illustrée sur le schéma suivant :



L'intervention de l'arbitre du bus est donc nécessaire pour réguler les accès et donner la priorité à l'un ou l'autre des caches.

Avantages :
- Simple
- Souple

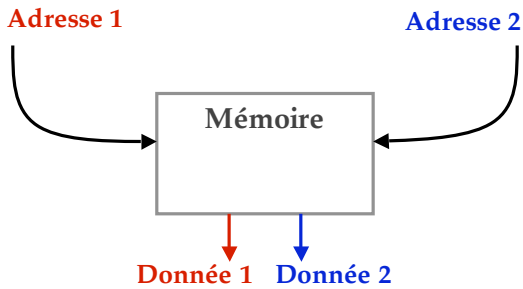
Inconvénients :
- Gaspillage de mémoire facile
- Repousse la difficulté sur le bus

Le cache étant unifié, **la gestion est donc la même** pour les instructions et pour les données. Cependant, la contrainte de taille est moins présente, et l'espace occupé par les cases de donnée est assez élastique ; pouvant, si les instructions sont peu nombreuses, « empiéter » sur l'espace des instructions.

Le gaspillage d'espace est donc moins présent dans cette technique.

Du point de vue de l'accès, soit le processeur est capable de faire deux accès simultanés soit il doit pouvoir donner la priorité à une requête (plutôt qu'à l'autre).

Cette constatation implique une structure beaucoup plus complexe du coté du cache unifié. On utilise par exemple des mémoires « double accès » :



Dans le cas de l'accès unique, les priorités peuvent aboutir à des *dead-locks* provoquant un blocage général du processeur.

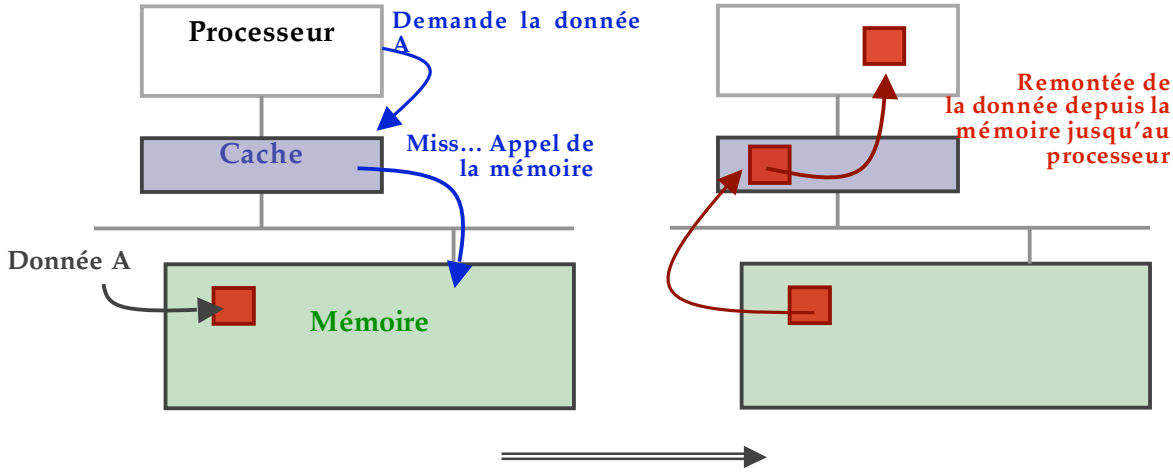
Cette solution apparaît a contrario beaucoup **plus clémente avec le bus**. Le cache étant unique, un seul accès peut avoir lieu simultanément à la mémoire. Le trafic est par conséquent beaucoup moins important.

Avantages :
- Adaptatif

Inconvénients :
- Complexe
- Peu souple

2. Gestion des écritures

Un autre axe stratégique dans la conception et l'implémentation des caches concerne la **politique d'écriture en mémoire**. Cette question concerne principalement les données qui sont susceptibles d'être écrites en mémoire (plus que les instructions).



Les données présentes dans le cache sont donc des copies de ce qu'il y a en mémoire. On a donc, la même donnée trois fois présente dans le système : dans le processeur, dans le cache et dans la mémoire.

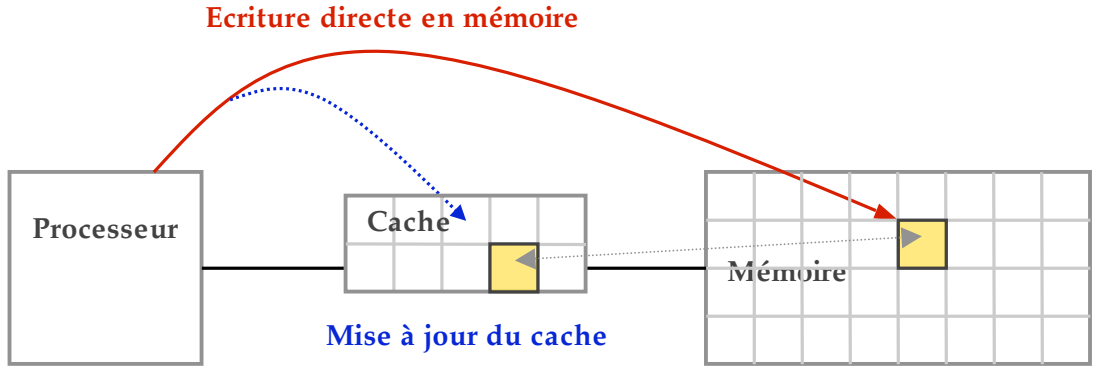
Cette présence multiple entraîne un problème lorsqu'il s'agit de modifier cette donnée :

- **Où doit-on écrire les modifications ?**

Plusieurs propositions ont été faites, et plusieurs solutions sont donc envisageables :

A. *La technique du « Write Trough »*

Dans cette stratégie, **l'écriture se fait directement en mémoire**. La mise à jour du cache se fait dans le même temps. La cohérence entre la mémoire primaire et le cache est donc assurée en permanence !



Cette stratégie est relativement simple à mettre en place et particulièrement souple (aucun cas particulier). Elle est cependant **très coûteuse en ressources processeur**. En effet,

l'écriture se faisant depuis le processeur, des cycles de gel seront introduits afin d'attendre la fin de l'écriture. (Caractéristique propre au mips R3000 mais rarement vérifiée sur les autres processeurs)

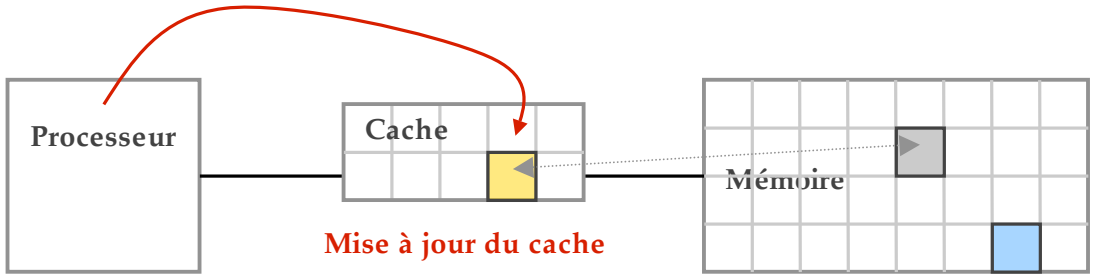
Une autre solution est donc proposée pour pallier cette lenteur :

B. *La technique du « Write Back »*

Le processeur commande ici l'écriture dans le cache. Souvenons nous que ce cache est accessible en un cycle. **Cette propriété garantie qu'aucun cycle de gel ne sera nécessaire**, et que l'exécution du programme pourra se poursuivre normalement.

La mise a jour dans la mémoire est toutefois nécessaire et a lieu au tout dernier moment, juste avant que le bloc ne soit supprimé du cache (lors d'un remplacement par exemple).

1° étape : Mise à jour du cache

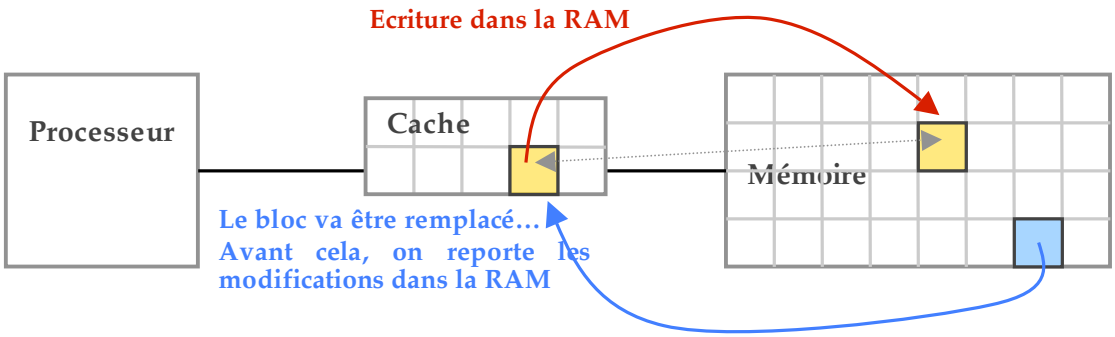


A ce stade, la donnée cohérente (jaune) est dans le cache mais pas dans la mémoire.
Si le processeur redemande la donnée, le cache détectera qu'il l'a, et ne fera aucun appel à la mémoire (fonctionnement logique du cache)

Imaginons maintenant, que le processeur fasse appel à une donnée (bloc bleu) qui ne se trouve pas dans le cache. Ce dernier répondra alors « miss » à la requête du processeur et provoquera donc le chargement du bloc nécessaire en provenance de la mémoire.

Si le bloc que nous rapatrions de la mémoire doit prendre la place du bloc mis à jour un peu plus tôt (le jaune), nous devons absolument reporter les modifications en mémoire. C'est donc à ce moment-là que l'écriture a véritablement lieu sur la RAM.

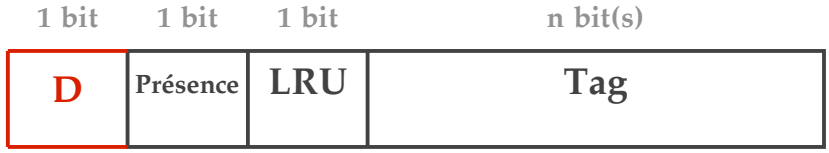
2° Etape : Report dans la mémoire primaire (au dernier moment)



Cette technique permet notamment d'**éviter les écritures à répétition** (dans la mémoire primaire) lorsqu'une donnée est très souvent modifiée. Toutes les modifications ont

maintenant lieu dans le cache, n'étant reportées dans la mémoire primaire uniquement lors de la suppression du bloc du cache.
Cette méthode implique cependant une écriture dans la mémoire primaire à chaque nouveau chargement de bloc dans le cache. Pour éviter cela, il nous faut un moyen de détecter si le bloc a effectivement été modifié durant son « séjour » dans le cache et si il nécessite, par conséquent, une écriture dans la RAM.

On propose pour ce faire d'ajouter un **nouveau bit de contrôle** à la **partie Directory** du cache :



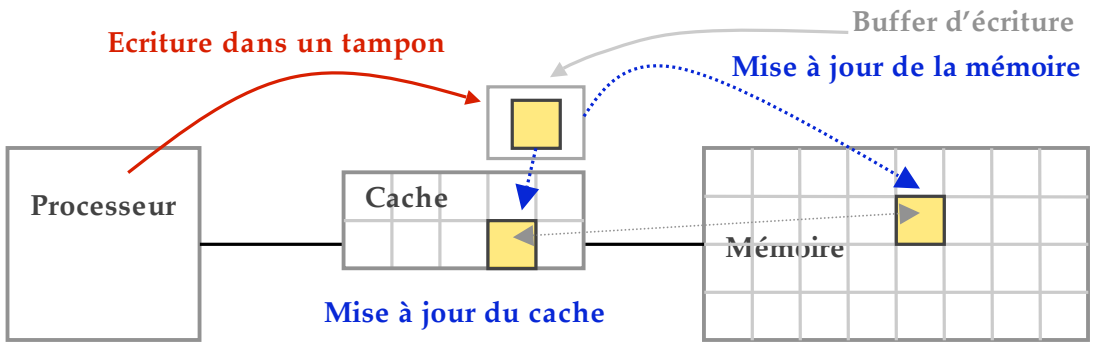
Ce **bit D** (comme Dirty) indique si les données sont « sales », ie. elles ont besoins d'être reportées en mémoire. Les valeurs proposées sont :

0 : les données du bloc sont conformes au contenu de la mémoire (pas de modifications)
1 : La donnée a été modifiée et nécessite un report (lors du prochain remplacement de page)

C. *Retour sur la technique du « Write Trough »*

Nous avons vu précédemment, que la technique du « Write Trough » (première méthode proposée) provoquait le gel du processeur pendant l'écriture en mémoire. Une amélioration a donc été proposée pour supprimer ce temps d'attente.

Dans cette amélioration, la responsabilité de l'écriture n'incombe plus au processeur mais au cache. La gestion de l'écriture a donc été déplacée du processeur vers le cache. On nomme donc cette stratégie : **stratégie de l'écriture postée.**



Le processeur peut donc continuer à exécuter le programme en séquence. Le seul risque de blocage provenant de la taille limite des tampons d'écriture. Si ces derniers sont pleins, le processeur devra alors la fin d'une écriture pour « poster » sa donnée.

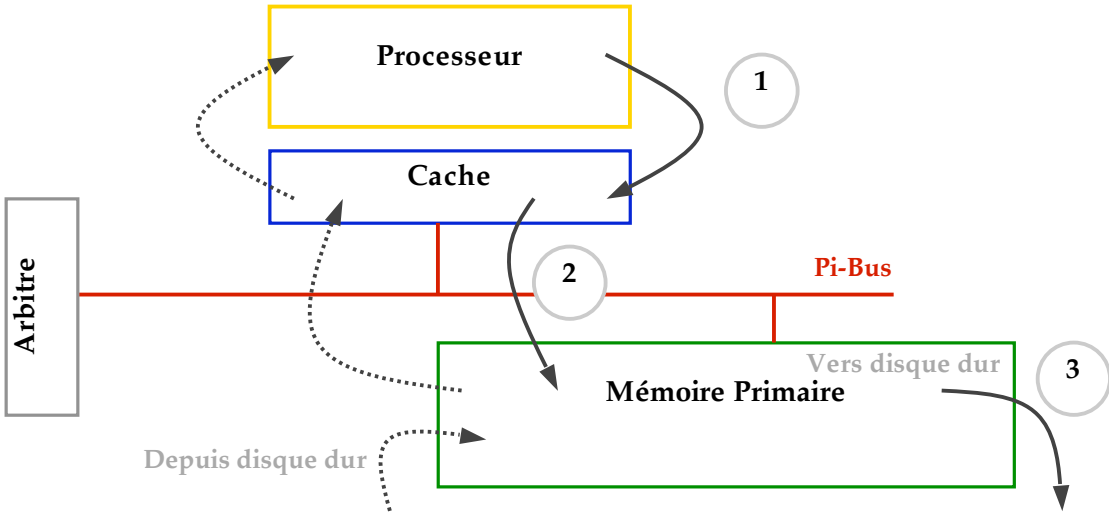
Chapitre 11

Optimisations & Adressage

Pour ce dernier cours… Champagne ! Nous avons pu détailler à loisir, le fonctionnement du processeur, du bus et du cache dans les cours précédents ; il est donc temps de « relier » tous ces éléments pour étudier les mécanismes qui leur permettent de dialoguer entre eux. Nous rajouterons aussi un élément dont nous avons beaucoup parlé jusque-là, mais peu étudié : la mémoire primaire.

1. Le système processeur + mémoires

Il peut être utile, avant de détailler les échanges entre le processeur, le cache et la mémoire primaire (la RAM), de rappeler le fonctionnement global du système.



Etape 1 : Le processeur émet une adresse.

Qu'il s'agisse d'une instruction, ou d'une donnée, le processeur extrait ses données depuis les mémoires sous-jacentes. Il communique avec elles à l'aide d'adresses qui permettent de localiser l'information.

La première mémoire à laquelle le processeur d'adresse est le cache.

Etape 2 : Le cache recherche la donnée demandée

Après diverses comparaisons, le cache est capable de répondre :

- **Hit** : La donnée est présente dans le cache
- **Miss** : La donnée n'est pas dans le cache

Si le cache répond Hit, la donnée est envoyée vers le processeur.
Si le cache répond Miss, la requête est transférée à la mémoire primaire via le bus système.
Etape 3 : La mémoire primaire recherche la donnée demandée

Tout comme le cache, la mémoire primaire est capable après une série d'opérations (que nous détaillerons plus loin) de répondre :

- **Hit** : La donnée est présente dans la mémoire primaire
- **Miss** : La donnée n'est pas présente dans la mémoire primaire

Si la mémoire répond Hit, la donnée est alors transférée via le bus système vers le cache qui se charge de renvoyer cette donnée vers le processeur.

Si la mémoire répond Miss, la requête est transférée vers la mémoire secondaire (disque dur). Nous sommes alors sûr que le disque dispose de la donnée et ce dernier la fournira alors à la mémoire primaire qui la fera remonter jusqu'au cache, où elle sera finalement envoyée vers le processeur.

2. Mémoire primaire & secondaire

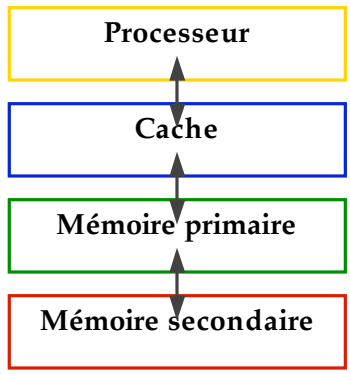
Nous avons pu, dans les cours précédents, analyser les échanges entre processeur et cache, ainsi qu'entre cache et mémoire primaire. Le passage flou du déroulement que nous venons de détailler plus haut concerne les échanges entre la mémoire primaire et la mémoire secondaire.

Cette mémoire secondaire est indispensable car il est très clair, qu'il n'est pas possible de représenter les 4 Goctets de l'espace d'adressage en mémoire primaire. **Que se passe-t-il donc si la donnée demandée n'est pas présente** (ie. La mémoire primaire répond miss) ?

Il faut remarquer, avant tout, dans l'organisation générale des mémoires, que la mémoire primaire agit comme un cache, entre le cache et la mémoire secondaire :

Partant de cette constatation, on utilise les mêmes principes que pour le cache. Ainsi, le principe de la localité spatiale est préservé.

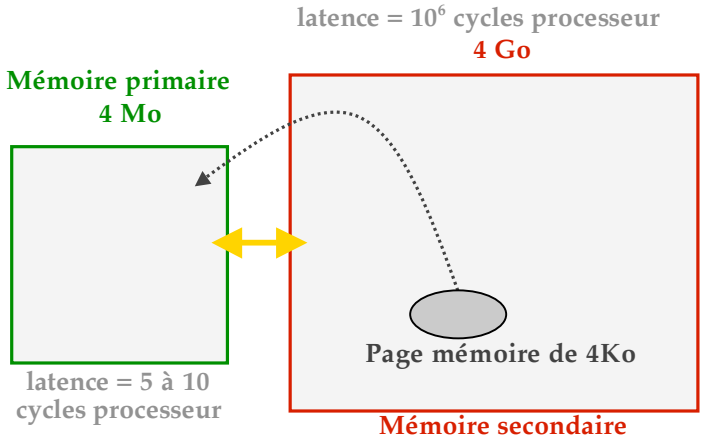
Plus simplement, on ne se contente pas de « remonter » de la mémoire secondaire la donnée seule, mais un bloc entier de données en « pariant » sur le fait qu'une autre donnée proche physiquement de celle qu'on demande sera bientôt requise par le cache.



Alors que les transferts entre le cache et la mémoire primaire se faisait en blocs (16 octets), les mémoires primaires et secondaires s'échange des **pages mémoires (4 Ko)**.

Il ne faut pas oublier, non plus, que les délais d'accès sont très différents entre la mémoire primaire et la mémoire secondaire. Pour la première la latence est comprise entre **5 et 10 cycles processeur**, alors que la seconde se compte en **millions de cycles processeur**.

C'est cette différence de latence qui nous obligé à concevoir une mémoire primaire très performante pour éviter au maximum les accès disque. **Le taux de miss doit donc être très faible** dans la mémoire principale, c'est-à-dire que toutes les requêtes doivent aboutir.
Les performances de tout l'ensemble (processeur, cache, mémoires) en dépendent.



Nous avons découvert dans les cours précédents que diverses stratégies de gestion de caches pouvaient être implémentées. **Quelle stratégie est donc utilisée dans la mémoire primaire ?**

▶ Cache Direct Mapping

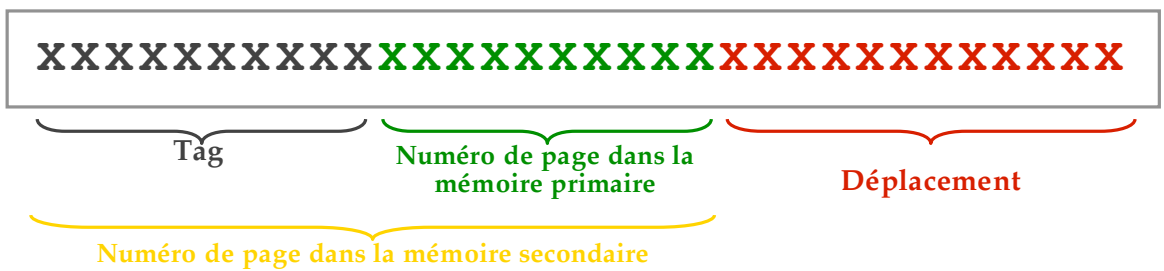
Cette technique empêche toute prise en compte du principe de la localité temporelle.
Une performance optimale étant recherché, nous devons, en plus de la localité spatiale, tenir compte de cette propriété.

▶ Cache Full Associative

Version extrême du cache Set Associative, cette technique prend en compte parfaitement la localité temporelle (en plus de la localité spatiale). Une page provenant de la mémoire secondaire peut donc être stockée à **n'importe quel endroit** dans la mémoire primaire.
La stratégie LRU est utilisée pour gérer les remplacements de pages.

A priori, nous choisirons donc cette dernière stratégie pour gérer notre mémoire primaire.
Quelques adaptations seront toutefois nécessaires pour contourner les inconvénients exposés dans le cours précédent.

En tenant compte de ces informations, nous pouvons proposer un découpage de l'adresse. Notons que le processeur émet désormais des adresses sur 32 bits, puisque disposant d'un espace d'adressage de 4Go et que la mémoire primaire ne dispose de 4 Mo d'espace.



12 bits sont réservés pour le déplacement dans la page et permettent ainsi de distinguer chaque octet de la page (qui en compte 4096…)

Les 10 bits suivants (poids faibles) indiquant le numéro de page dans la mémoire primaire pages disponibles (1024 pages de 4 Ko chacune = 4 Mo de mémoire primaire)

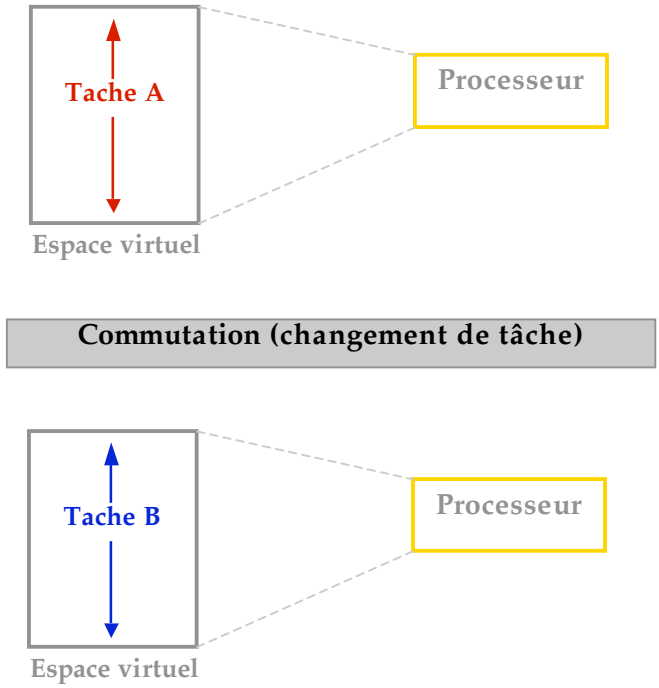
Les 10 derniers (poids forts) constituant le tag.

3. Adressage

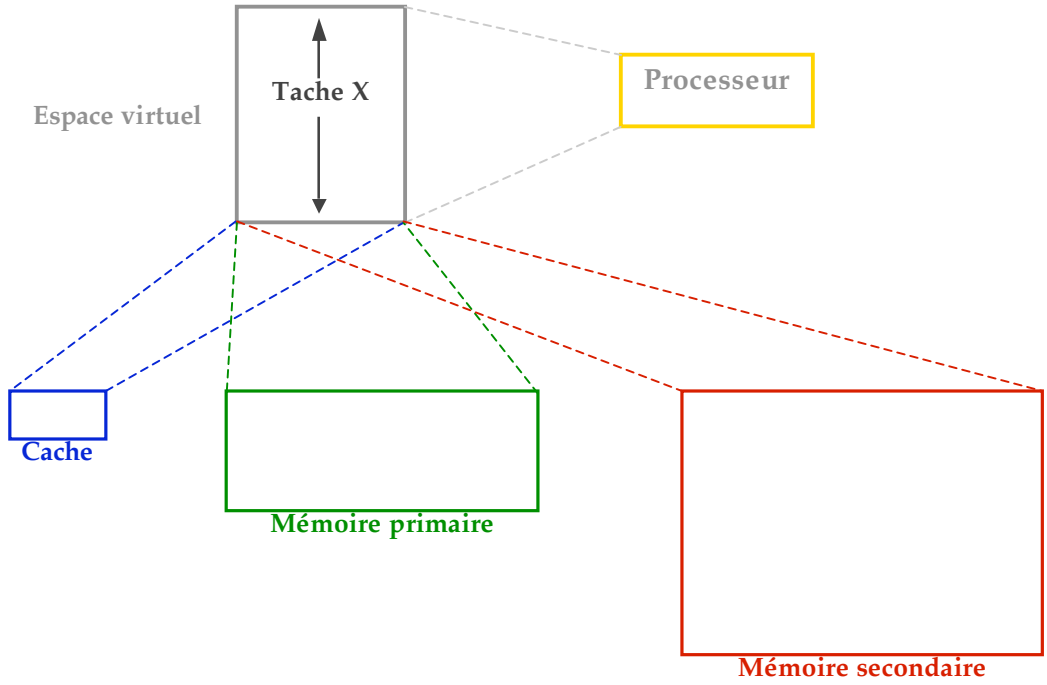
Avant de détailler la gestion de la mémoire primaire, arrêtons nous un instant sur les adresses émises par le processeur.

Nous savons que le processeur **émet des requêtes sous forme d'adresses** concernant la tâche (ou le processus) qui est entrain de s'exécuter. Cette tâche-ci dispose d'un espace virtuel d'exécution de 4Go. Dans un environnement multitâches, une autre tache aurait, elle aussi, un espace d'exécution de 4Go.
Plus généralement, toutes les taches disposent du même espace virtuel, à la simple différence qu'elles ne disposent pas du processeur au même moment.

Une représentation de cette organisation peut être donnée sur le schéma suivant :



On déduit aisément de ce schéma, que **deux adresses équivalentes de l'espace virtuel ne correspondent pas forcément à la même donnée pour les deux taches**. La commutation est donc le mécanisme qui permet le « remplissage de l'espace virtuel » avec les données correspondant à la tâche qui va être exécutée.
Si on considère maintenant l'espace virtuel, nous pouvons constater qu'il est **réparti** entre le cache, la mémoire primaire et la mémoire secondaire.



Le processeur ne connaît pas l'organisation générale des mémoires, et ne peut donc s'adresser directement à elles. **Il émet donc des adresses correspondant à l'espace virtuel**. Mais, comme nous avons pu le remarquer dans la page précédent, deux adresses virtuelles similaires, ne correspondent pas forcément à la même donnée.

En effet, les mémoires et le cache utilisent des adresses réelles issues de l'adressage de la mémoire secondaire.

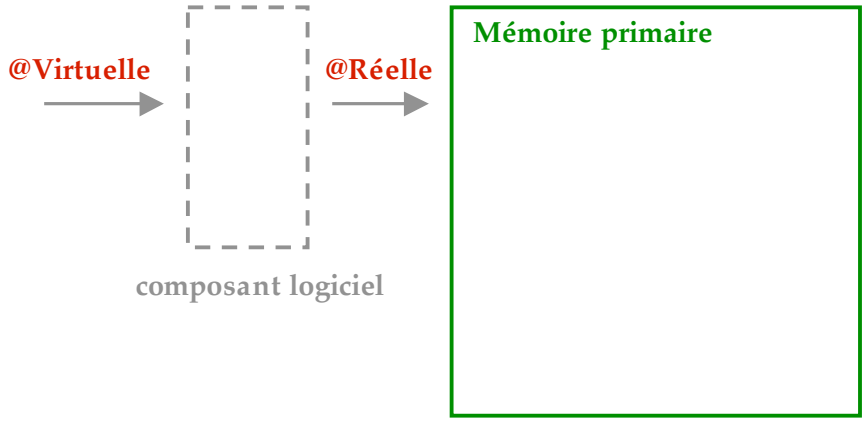
A tout moment, les adresses émises par le processeur (virtuelles) doivent donc être traduites en adresses accessibles par les mémoires et le cache (réelles).

4. Gestion de la mémoire primaire (Retour)

Nous avons choisi la gestion par cache Full Associative pour la mémoire primaire.
Le schéma général proposé pour cette stratégie repose sur l'utilisation d'une partie Directory qui **compare le tag de l'adresse demandée avec celui de l'adresse présente en mémoire**.

Cependant, la mémoire comportant plus de 1000 pages différentes, soit 256 000 blocs de données, le nombre de comparateurs nécessaires à l'implémentation de cette stratégie (256 000) serait trop important pour rendre possible la réalisation d'une telle gestion.

Il faut donc réfléchir à une **solution logicielle** pour permettre la gestion de la mémoire.



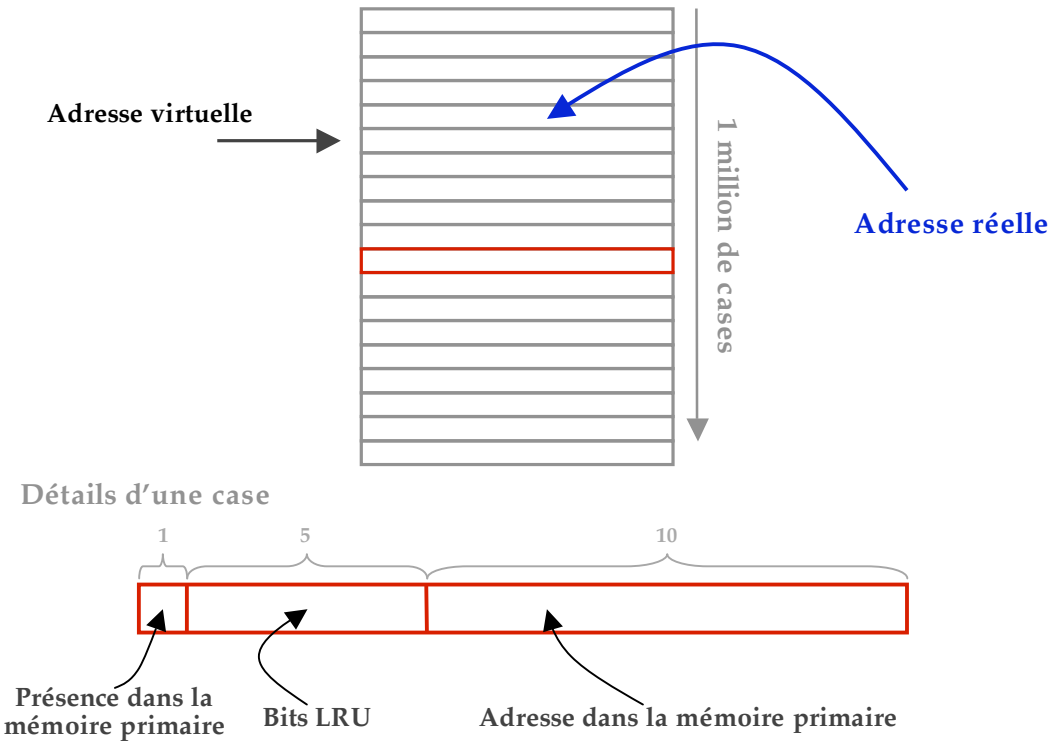
Le composant logiciel dans cet exemple est chargé de :

- **traduire l'adresse virtuelle en adresse réelle**
- **s'assurer que la donnée est présente** dans la mémoire primaire.

A. *Tables des pages*

On pourrait dans un premier temps penser à une table associant un numéro de page virtuelle à un numéro de page réelle : **la table des pages.**

Comme nous avons pu le voir, l'adresse comporte 20 bits déterminant le numéro de page dans la mémoire secondaire. Il y a donc $2^{20} = 1 \times 10^6$ pages virtuelles possibles.



La performance de cette solution est très intéressante. En effet, l'accès à la page se fait dans un temps très court avec une complexité en O(1)

Cependant, une case nécessite 16 bits et la table des pages comportant 10^6 cases, nécessitera donc 16Mbits d'espace soit 2Mo.

Mis en rapport avec la taille générale de la mémoire primaire (4Mo), cet espace semble tout a fait **disproportionné**.

Cette solution est définitivement rejetée en considérant, en plus, que chaque changement de tâche provoquerait un vidage de la table pour qu'elle soit réinitialisée avec les valeurs adéquates pour l'exécution de la nouvelle tache.

B. *Table des pages optimisée*

Une **seconde solution** consiste à optimiser la proposition précédente. Dans notre exemple, la table des pages (TDP) comportait 10^6 cases recensant ainsi toutes les pages adressables de la **mémoire secondaire**. Nous avions donc dans cette table (placée avant la mémoire primaire) une grande majorité de cases avec un bit de présence à 0, puisque la donnée, bien que présente en mémoire secondaire, n'était pas présente dans la mémoire primaire.

En général, seul 1 millier de cases (10^3) possèdent un bit de présence à 1, ce qui signifie que seul 1 millier de pages sont effectivement accessibles en mémoire primaire.

De cette constatation, on peut proposer l'idée d'une table des pages qui « grandit » au fur et à mesure des pages demandées :

Etat initial

TLB vide

Demande de la page @00000001

@00000001	1		

Demande de la page @00000005

@00000001	1		
@00000002	0		
@00000003	0		
@00000004	0		
@00000005	1		

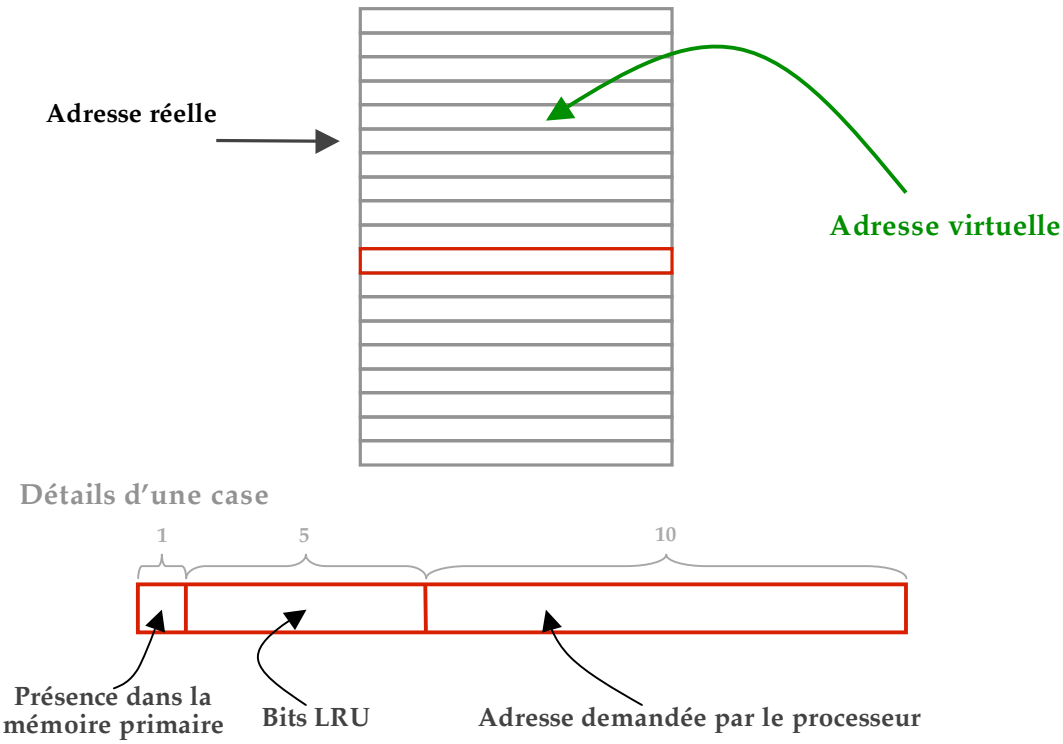
Cet exemple conduit forcément à l'inconvénient majeur de ce type de table des pages. En effet, nous savons que les instructions des appels système sont rangés à l'adresse : 0x80000000.

Un simple appel système provoquerait l'agrandissement brutal de notre table des pages.

C. Table des pages inverse

Pour résoudre ce problème, nous proposons d'indexer notre table des pages, non plus par les adresses de pages virtuelles mais par les adresses de pages réelles.

Mais comment construire une telle table ?



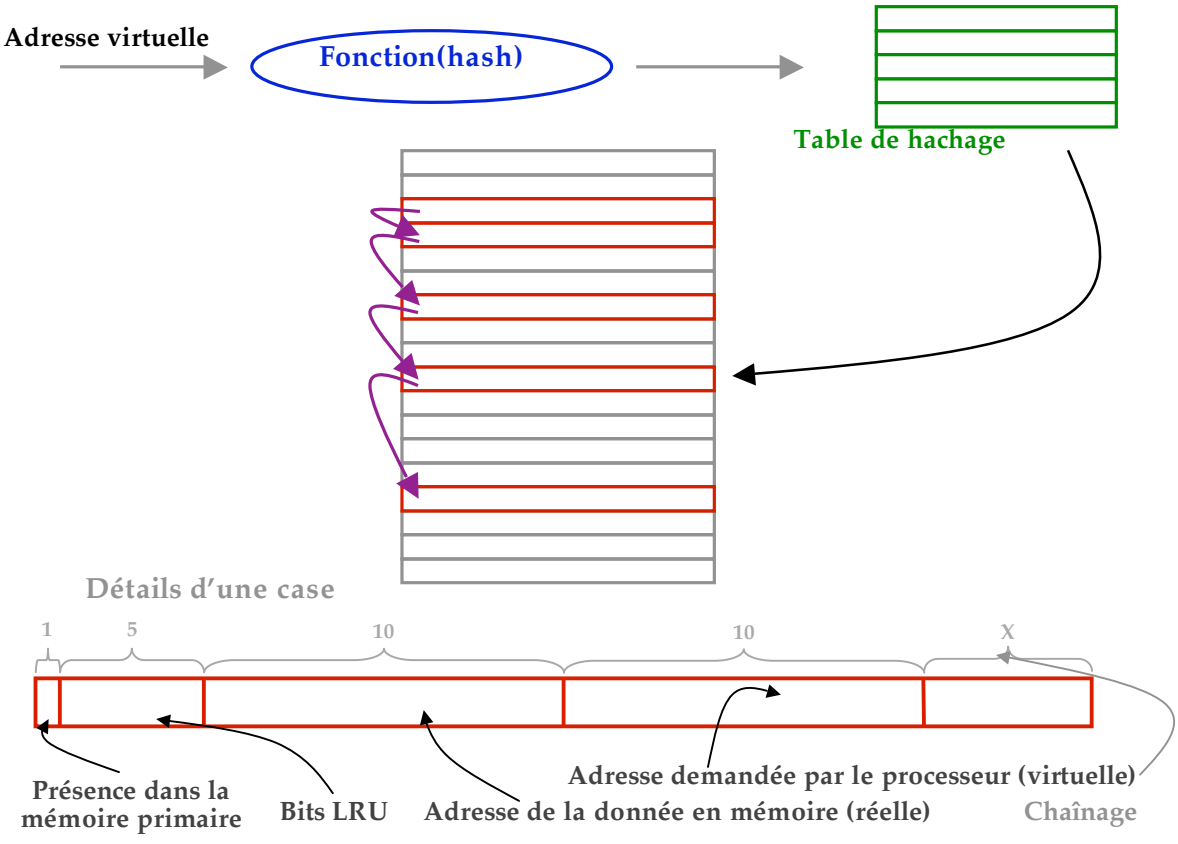
Le processeur nous fournit toujours une adresse virtuelle !

Une première solution consisterait donc à **parcourir toutes les cases de la table** des pages en comparant à chaque fois l'adresse virtuelle de la case testée et l'adresse proposée par le processeur. Cependant, le parcours est trop long et trop coûteux (O(n)), il faut optimiser.

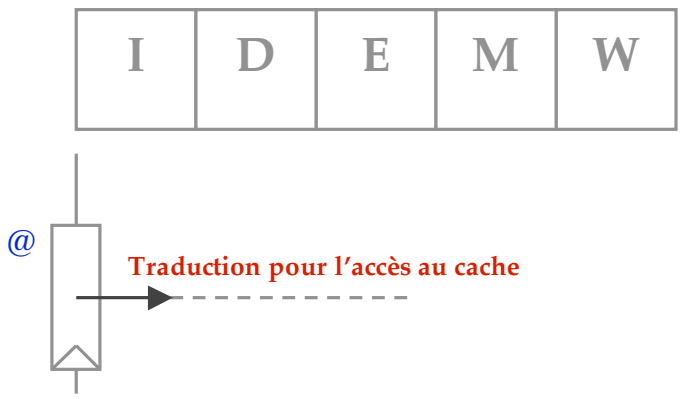
On propose donc l'utilisation d'une table de hachage qui indique, dans une table toujours indexé par les adresses réelles, ou se trouve l'adresse virtuelle ou, au moins, la liste chaînée dans laquelle on peut la trouver. (voir schéma page suivante)

La complexité est ici bien meilleure : $O\left(\frac{n}{\text{taille hash}}\right)$

Cette solution est sans aucun doute la plus performante des trois, même si son implémentation est un peu moins évidente.



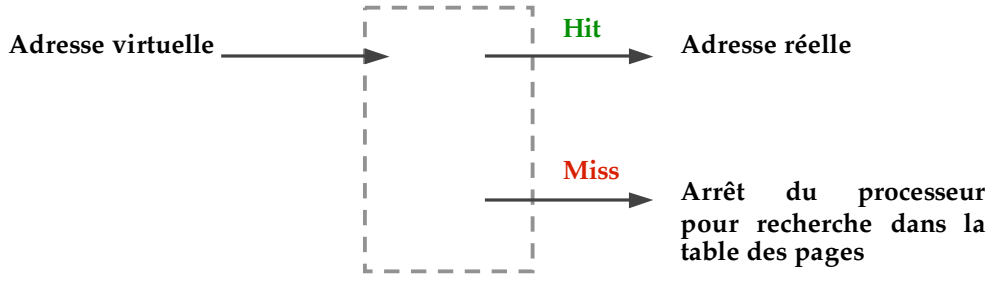
5. Cheminement global d'une requête



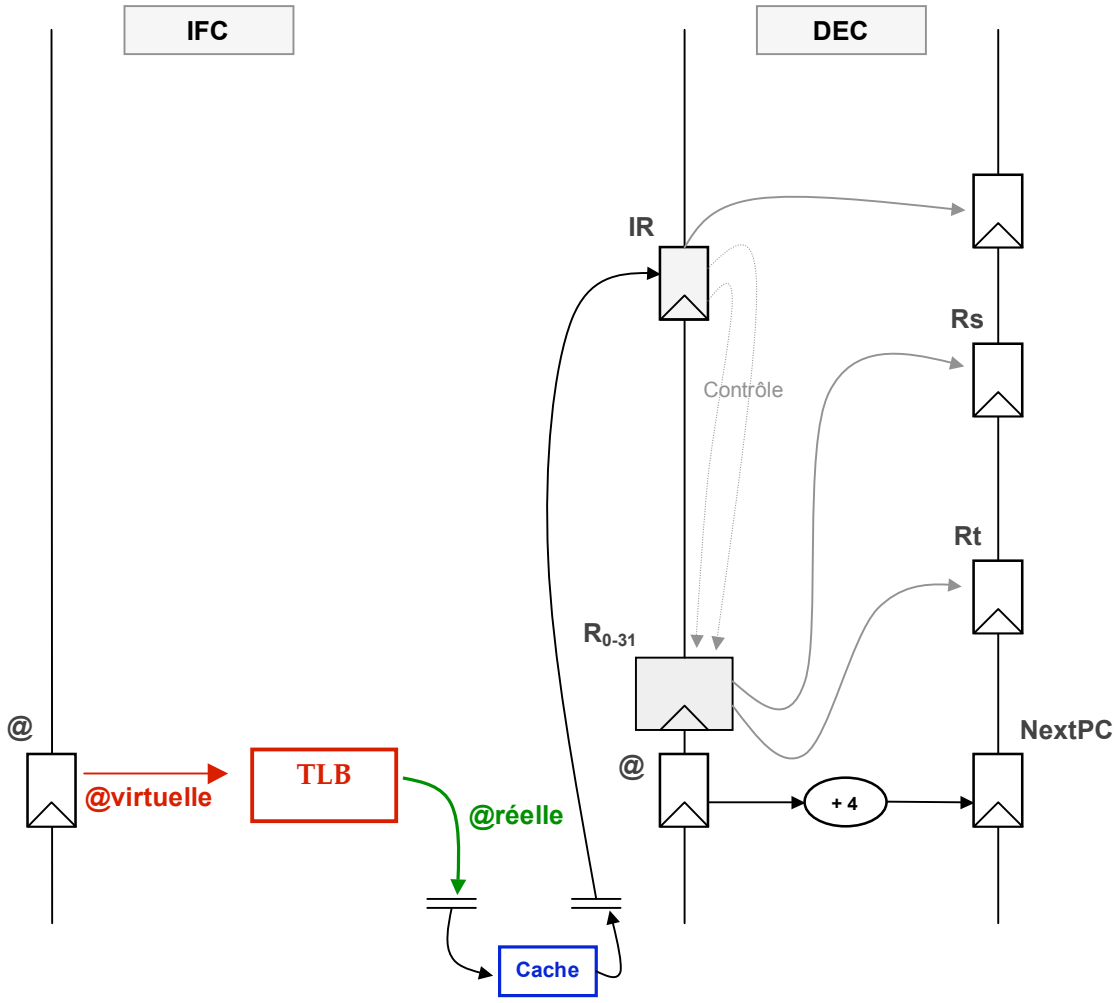
La première étape consiste à vérifier la présence de la donnée dans le cache. Cependant, comme vu précédemment, **le cache utilise un adressage réel** alors que **le processeur manipule des adresses virtuelles**. Une traduction est donc nécessaire pour pouvoir faire une requête au cache.

Le mécanisme précédent de table des pages logiciel ne peut être appliqué en l'état puisque beaucoup trop lent. C'est vers une solution matérielle que nous devons nous orienter pour éviter au processeur de se tourner constamment vers le logiciel pour déterminer les correspondances entre adresses virtuelles et réelles.

On met donc en place une table associant ces deux adresses : **La TLB** *(Traduction Lookside Buffer)*. Ce matériel conserve les associations pour une tache donnée, chaque changement de tache impliquant un **renouvellement de la TLB**.



Une partie d'un schéma d'exécution détaillé pourrait donc se présenter de la façon suivante :
On notera les cas exceptionnels suivant impliquant une démarche particulière :



- **Miss dans le cache** : rechargement depuis la mémoire primaire
- **Miss dans la TLB** : On s'adresse à la table des pages (logiciel)
- **Miss dans la TLB et miss dans la table des pages** : remplissage des tables

Conclusion

1. Bibliographie & Sitographie

- Architecture des Ordinateurs : Une approche quantitative
  David Patterson & John Hennessy

- Simulateur SPIM
  http://www.cs.wisc.edu/~larus/spim.html

- LIP6 : Département Architecture des Systèmes intégrés et Micro électronique (ASIM)
  http://www-asim.lip6.fr/

2. Notes du rédacteur

Ce document est destiné aux étudiants du cours d'architecture des systèmes intégrés de M1. Il constitue un receuil de notes et en aucun cas un cours magistral. Je vous recommande vivement de parcourir les pages du site de l'association des Etudiant en Informatique de Paris VI (A.E.I.P.6.) à l'adresse : http://www.aeip6.com

Toutes les remarques sont les bienvenues : jean-baptiste.voron@aeip6.com