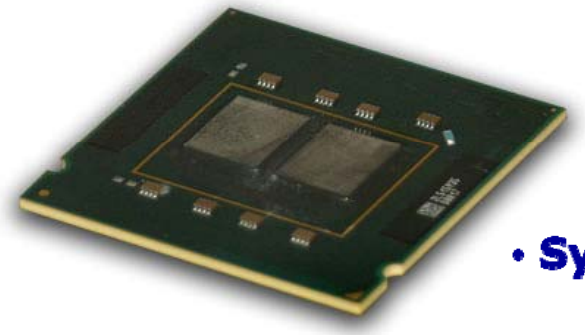
Future computing systems



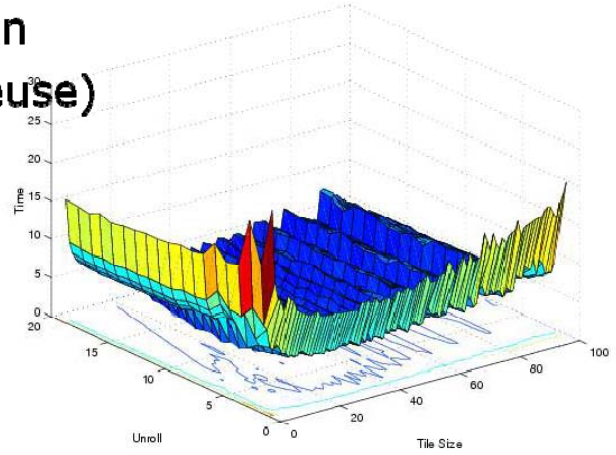
- **Current evolution of technology**
  - More transistors, multi-cores, heterogeneous systems, more parallelism
- **System complexity**
  - Understanding, analysis, evaluation
  - Programming issues

  (6h - Daniel Gracia)
- **Program optimizations**
  - Compilation and architecture specific optimizations
  - Polyhedral optimizations
  - Reversible computing

  (6h - Christine Eisenbeis, Anna Beletska)
- **Toward self-tuning adaptive systems**
  - Iterative feedback-directed compilation
  - Dynamic compilation and run-time adaptation
  - Machine learning (optimization knowledge reuse)

  (6h - Grigori Fursin)
- **Other potential future directions**
  - End of the "Moore law" (CMOS technology)
  - New spatial parallel programming paradigm

  (6h - Daniel Gracia, Frederic Gruau)



Embarqués vs. Généralistes

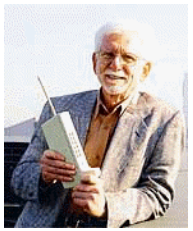
- Lesquels ?
  - généralistes
  - embarqués/enfouis haute-performance
- Convergence
  - Besoins en performance
  - Similarités croissantes des architectures
  - Contraintes de l'embarqué: consommation, taille de code, coût, temps-réel

PC

PDA:
MS/HP iPaq (Intel XScale 400MHz)

Console de jeux:
MS XBOX (PentiumIII+Nvidia)

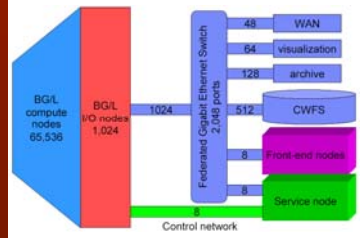
Magnétoscope à disque dur:
Asus DigiMatrix (Pentium4/Celeron)



Téléphone

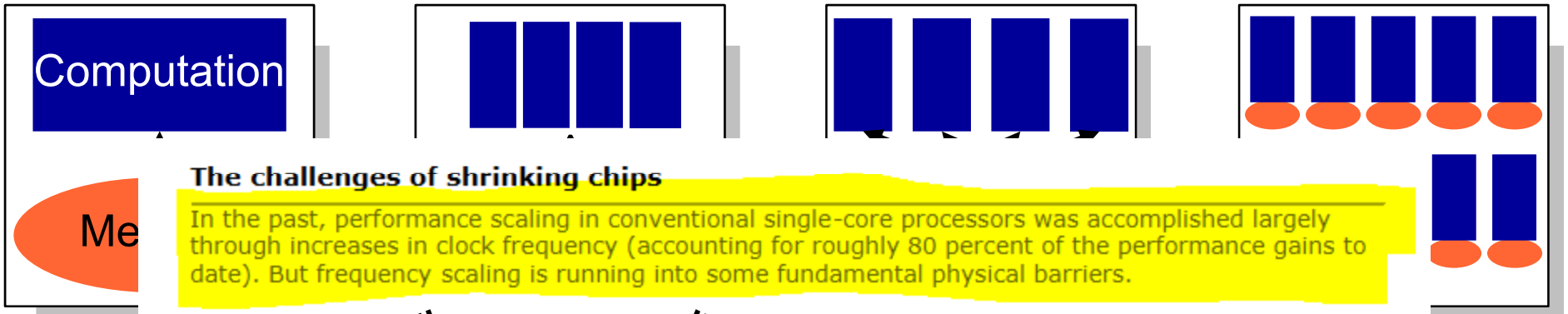
Téléphone+PDA:
PalmOne Treo600 (ARM 144MHz)

Téléphone + console de jeux:
Nokia NGage (ARM+proc. 3D)



Supercalculateur:
IBM BlueGene/L (PowerPC 440)

Contexte



The challenges of shrinking chips

In the past, performance scaling in conventional single-core processors was accomplished largely through increases in clock frequency (accounting for roughly 80 percent of the performance gains to date). But frequency scaling is running into some fundamental physical barriers.

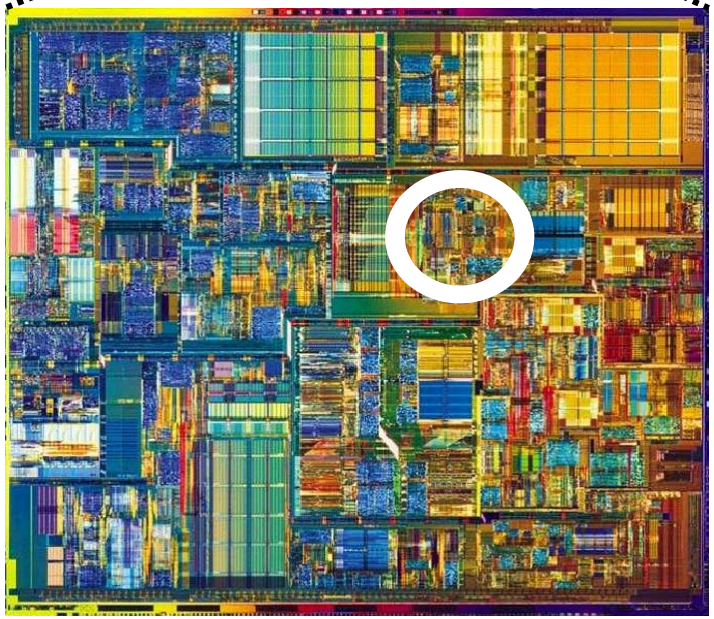
the transistor leakage current
dly, the advantages of higher clock
cess times have not been able to
plications, traditional serial
(due to the so-called Von Neumann
ses might otherwise buy. In
n are growing as feature sizes
s don't address.

oosting the clock speed of large
reaking up functions into many
rocessing units. Rather than
ency, Intel's multi-core processors

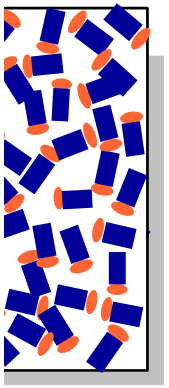
sue 02, May 19, 2005

?

- Cor
- Ter
  d'e
  vite



Pentium IV



Plan

- Processeurs haute-performance
  - Pipeline
  - Prédiction de branchement
  - Cache
  - Exécution superscalaire/parallèle

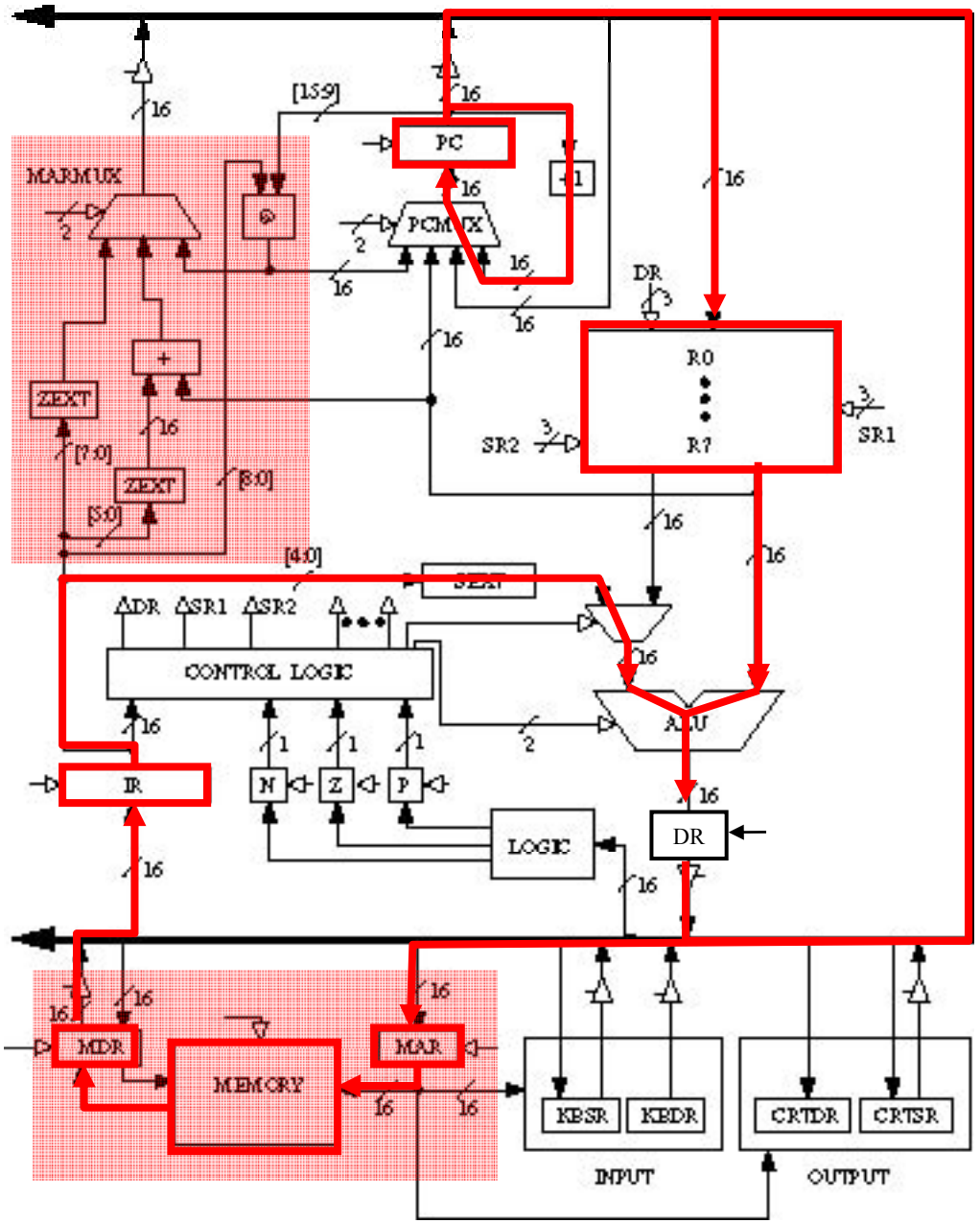
Exploitation Efficace des Composants

- Etapes d'exécution d'une instruction:
  1. Envoi adresse instruction (IA)
  2. Chargement instruction (IF)
  3. Stockage instruction (SI)
  4. Décodage; lecture des opérandes (DI)
  5. Calcul d'adresse (AC)
  6. Accès mémoire (ME)
  7. Exécution (EX)
  8. Ecriture du résultat (WB)

0001	101	100	1	00011

ADD R5, R4, #3

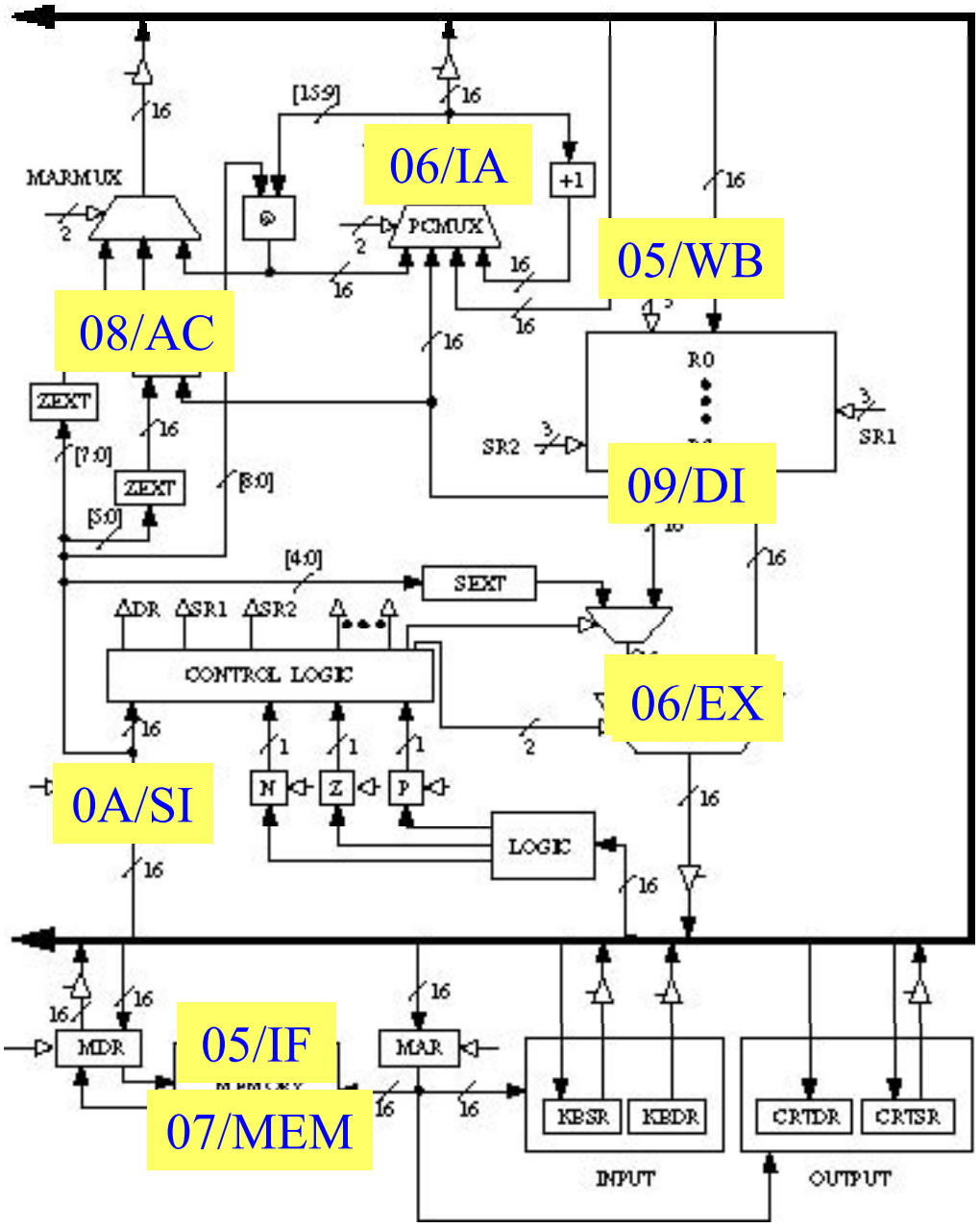
Un seul composant utilisé à chaque cycle



Exploitation Efficace des Composants

```
for (i=0; i < 100; i++) {
   a[i] = a[i] + 5
}
```

```
                ...
05  LOOP        LDR R1, R0, #3
06              ADD R1, R1, #5
07              STR R1, R0, #30
08              ADD R0, R0, #1
09              ADD R3, R0, R2
0A              BRn LOOP
                ...
```



- **Tous les composants sont utilisés**
- **Une instruction termine à chaque cycle**

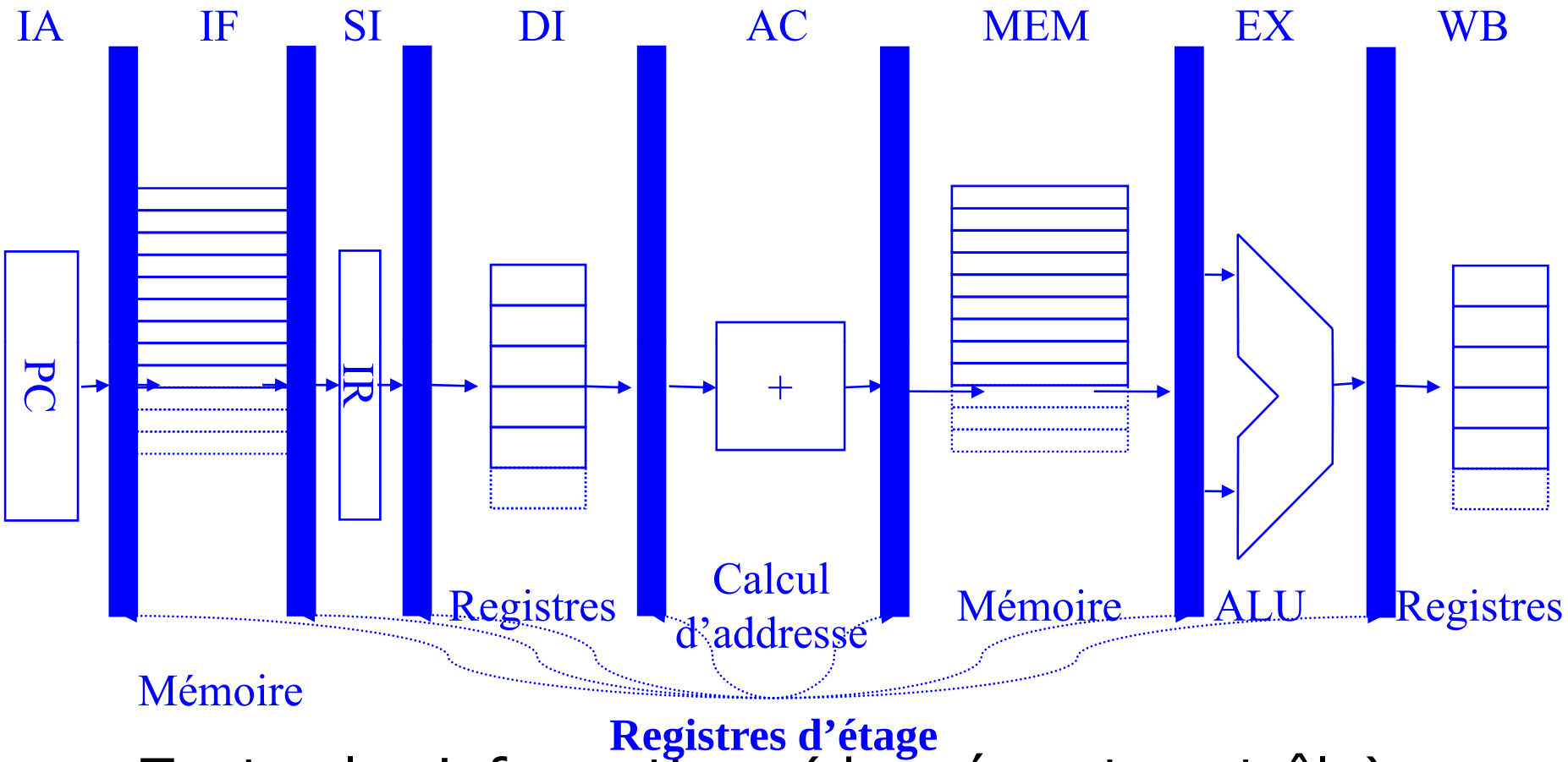
Pipeliner

Tous les composants sont utilisés

Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12	13
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM	EX	WB						
ADD R1,R1,#5		IA	IF	SI	DI	AC	MEM	EX	WB					
STR R1,R0,#30			IA	IF	SI	DI	AC	MEM	EX	WB				
ADD R0,R0,#1				IA	IF	SI	DI	AC	MEM	EX	WB			
ADD R3,R0,R2					IA	IF	SI	DI	AC	MEM	EX	WB		
BRn LOOP						IA	IF	SI	DI	AC	MEM	EX	WB	
LDR R1,R0,#30							IA	IF	SI	DI	AC	MEM	EX	WB
ADD R1,R1,#5								IA	SI	IF	DI	AC	MEM	EX

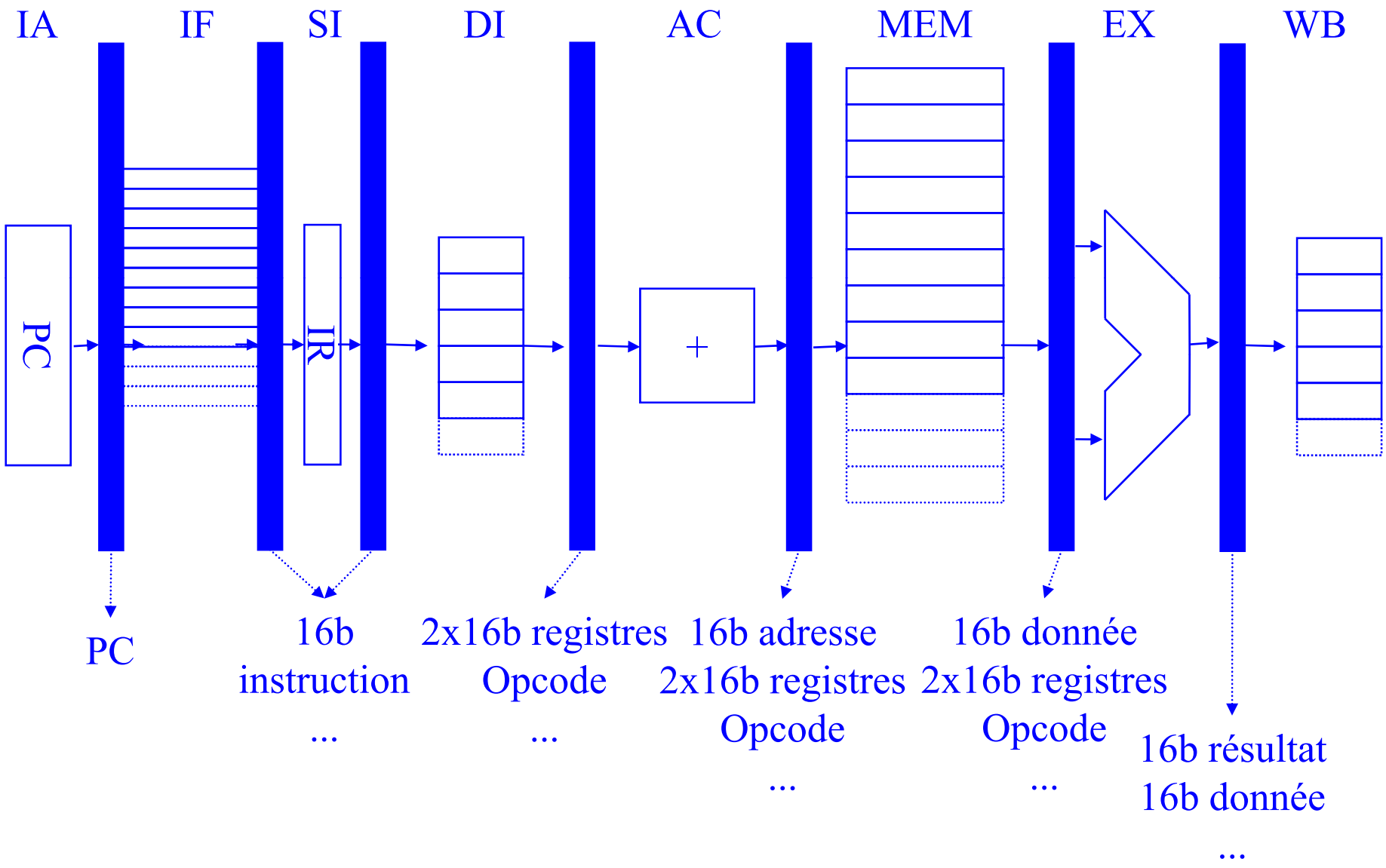
- L'exécution du programme est jusqu'à **8 fois plus rapide**.
- Le temps d'exécution d'une instruction est inchangé.

Implémentation du Pipeline



- Toutes les informations (données et contrôle) nécessaires à l'exécution d'une instruction sont stockées et propagées dans les registres d'étage.

Implémentation du Pipeline



Tendances

- Le pipeline du Pentium IV contient 20 étages + 8 étages de conversion `x86 → µInstructions`.

- La motivation initiale du pipeline était:
  - L'exploitation efficace des composants de l'architecture
  - Et par voie de conséquence: l'augmentation du débit des instructions.
- La motivation actuelle du pipeline est l'augmentation de la fréquence d'horloge:
  - On segmente les étapes/composants en sous-étapes/sous-composants.
  - La durée maximale d'une (sous-) étape est donc réduite.

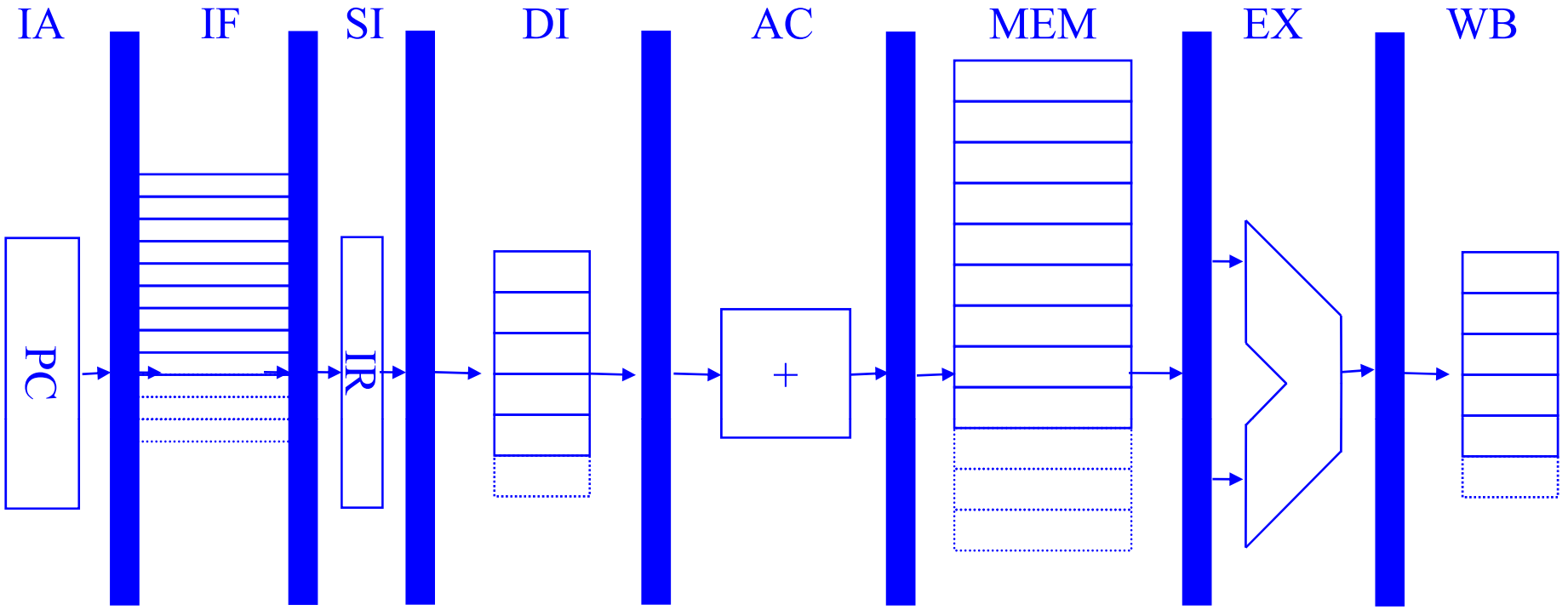
  ⇒ On peut réduire le temps de cycle donc augmenter la fréquence d'horloge.
- Cette stratégie rend difficile l'obtention de performances **soutenues** élevées.

Aléas de Pipeline

- Il n'est pas toujours possible d'émettre/sortir une instruction à chaque cycle dans un pipeline.
- Lorsque l'avancement d'une instruction est bloqué, on parle d'**aléa de pipeline** (*pipeline hazard*).
- On distingue 3 types d'aléas:
  - Aléas structurels (conflits de ressources).
  - Aléas de données (dépendances de données).
  - Aléas de contrôle (branchements).
- En cas d'aléa, le circuit de contrôle introduit une **bulle** dans le pipeline; on parle de **gel de pipeline** (*stall*); l'utilisation du pipeline n'est plus optimale.
- Mesure de performance: IPC (*Instructions Per Cycle*)

  < 1 (optimal) en cas d'aléas

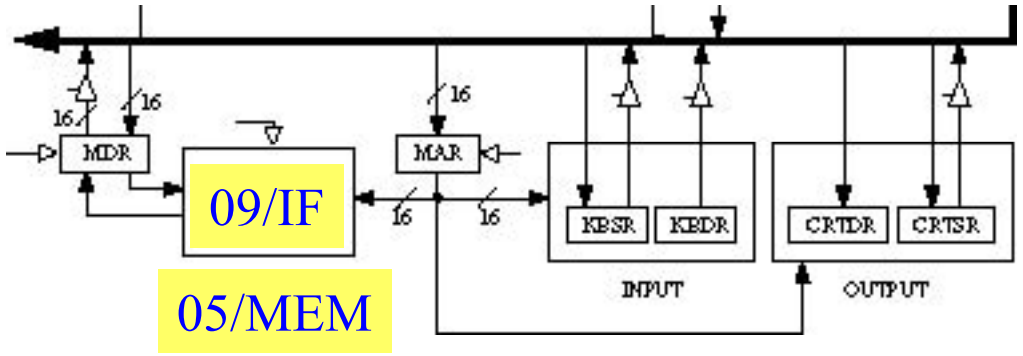
Gestion du Pipeline en Cas d'Aléa



Inst.	0	1	2	3	4	5	6	7	8	9	10	
Inst 1	IA	IF	SI	DI	AC	MEM	EX	WB				8 cycles
Inst 2		IA	SI	IF	DI	AC	MEM	EX	WB			
Inst 1	IA	IF	SI	DI	AC	MEM	EX	WB				10 cycles
Inst 2		IA	IF	SI	DI	●	●	AC	MEM	EX	WB	

Aléas Structurels

- Accès au même composant au même cycle.
- Solutions:
  - Accepter un gel de pipeline à chaque conflit (SPARC).
  - Augmenter les ressources (ici, deux ports d'accès à la mémoire).

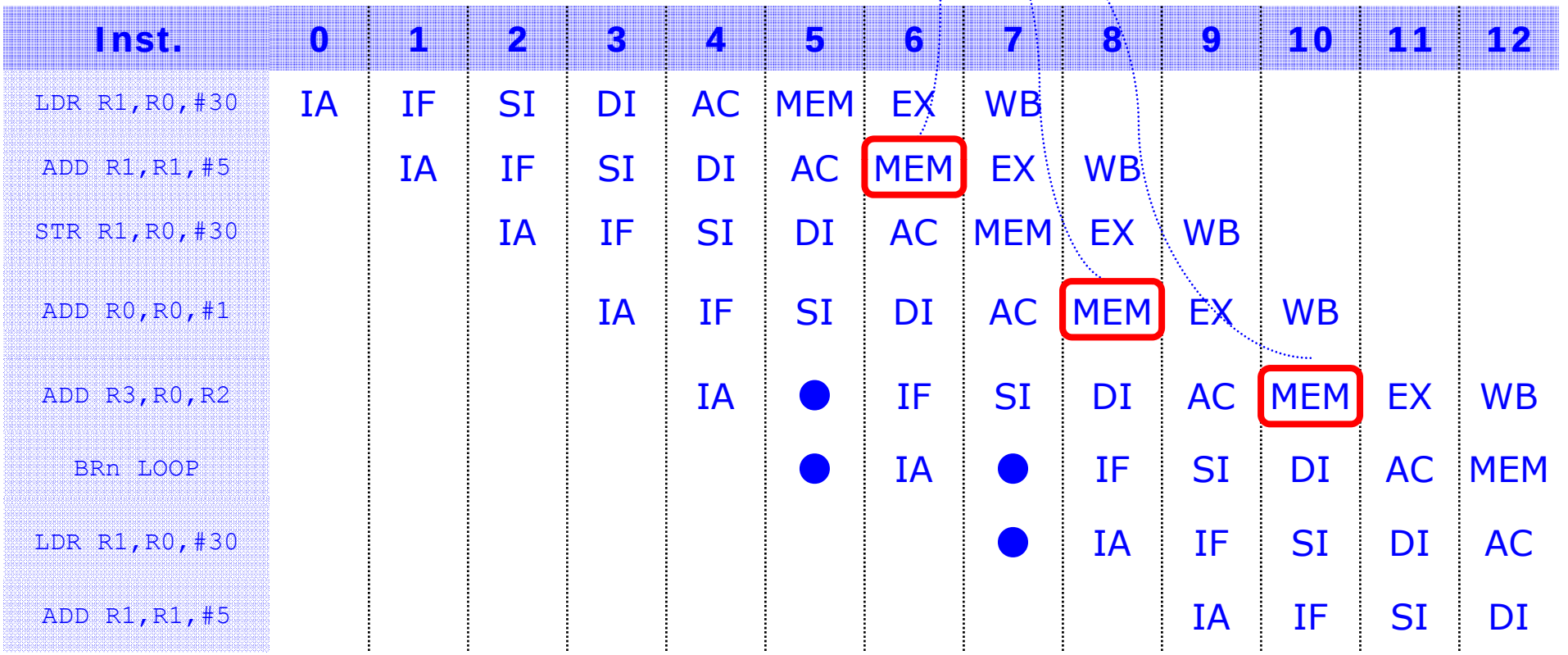


Inst.	0	1	2	3	4	5
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM
ADD R1,R1,#5		IA	IF	SI	DI	AC
STR R1,R0,#30			IA	IF	SI	DI
ADD R0,R0,#1				IA	IF	SI
ADD R3,R0,R2					IA	IF

Conflit de ressource

Aléas Structurels

Pas d'accès à la mémoire ⇒ pas d'aléa structurel



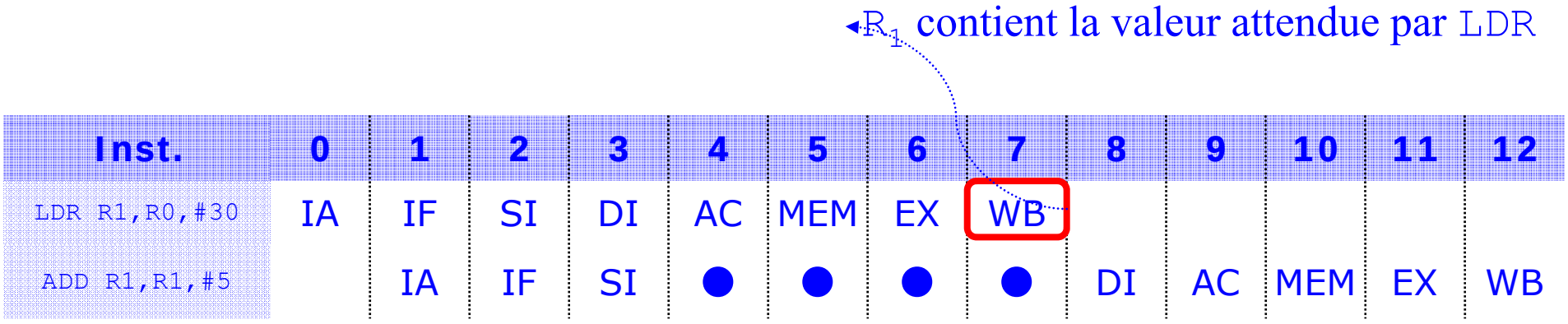
Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM	EX	WB					
ADD R1,R1,#5		IA	IF	SI	DI	AC	MEM	EX	WB				
STR R1,R0,#30			IA	IF	SI	DI	AC	MEM	EX	WB			
ADD R0,R0,#1				IA	IF	SI	DI	AC	MEM	EX	WB		
ADD R3,R0,R2					IA	●	IF	SI	DI	AC	MEM	EX	WB
BRn LOOP						●	IA	●	IF	SI	DI	AC	MEM
LDR R1,R0,#30								●	IA	IF	SI	DI	AC
ADD R1,R1,#5										IA	IF	SI	DI

Aléas de Données

- Dépendance de données entre deux instructions.

```
LDR R1 ← R0,#30
ADD R1 ← R1,#5
```

- Une instruction ne peut effectuer son étape DI (chargement des opérandes) que lorsque ses opérandes sont disponibles.



Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM	EX	WB					
ADD R1,R1,#5		IA	IF	SI	●	●	●	●	DI	AC	MEM	EX	WB

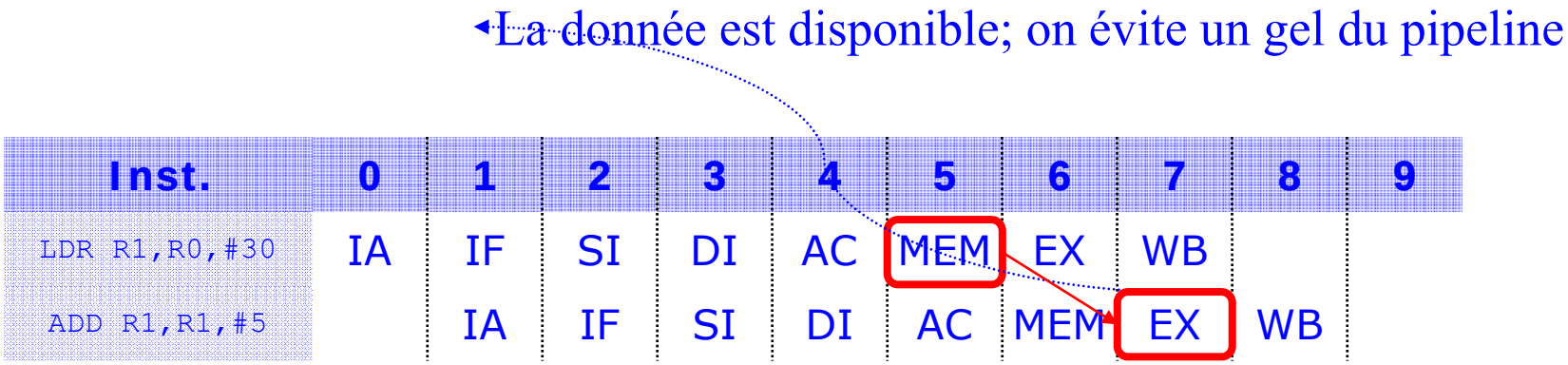
Le *Forwarding*

- En cas de dépendance, la donnée attendue est souvent disponible avant d'être écrite dans le registre

  ⇒ Passer immédiatement la donnée au composant qui l'attend sans attendre son écriture dans le registre.

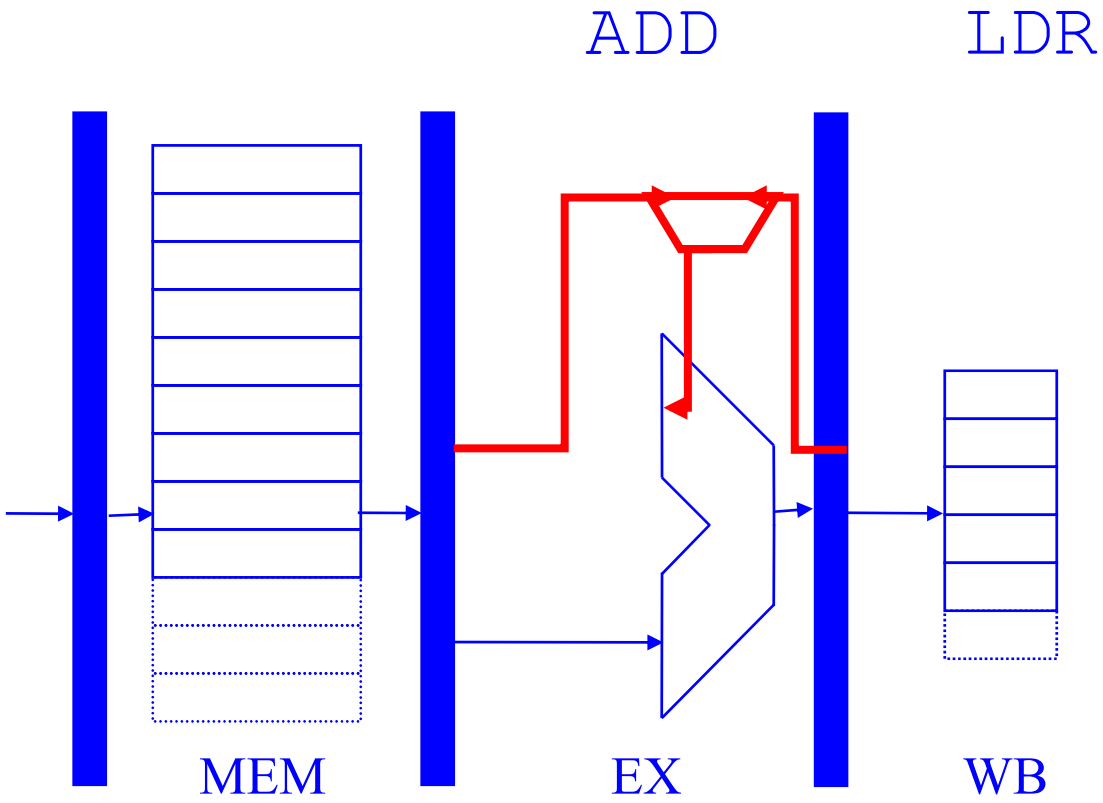
  = ***Forwarding***

- Le pipeline est bloqué seulement lorsque la donnée est indispensable.



Implémentation du *forwarding*

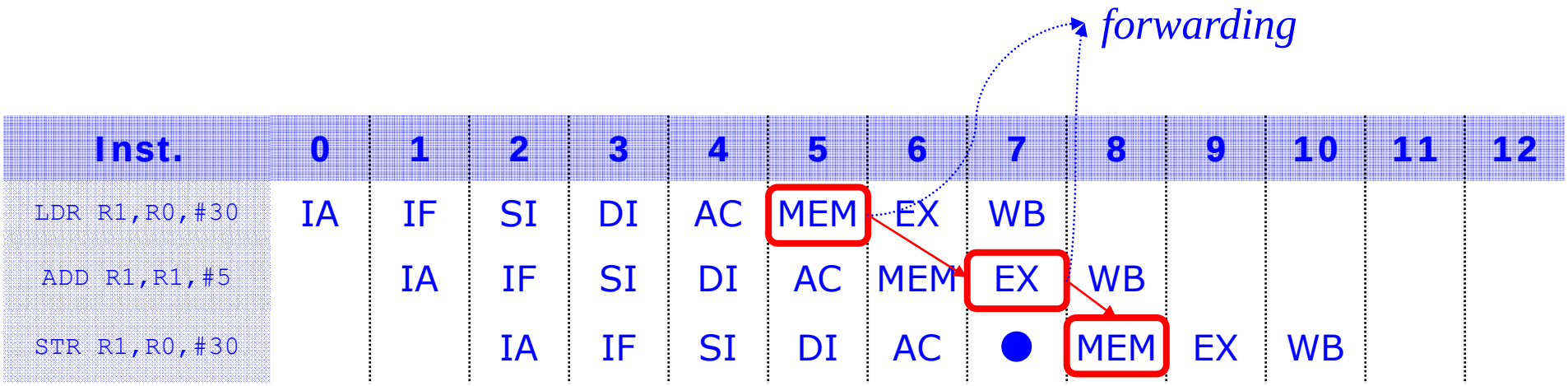
- Le *forwarding* nécessite:
  - de rajouter des chemins de données
  - d'augmenter la taille de multiplexeurs ou d'en ajouter
  - de modifier le circuit de contrôle (détection/activation du *forwarding*)



Le *Forwarding*

- Le *forwarding* ne permet pas d'éviter tous les gels de pipeline:

```
ADD R1 ← R1,#5
STR R1 → R0,#30
```



Instructions Multicycles

- Exemple: instructions flottantes.
  - FPADD: 2 cycles d'exécution
  - FPMUL: 5 cycles d'exécution

Conflit de ressource
si FPADD et FPMUL
dans le même composant

Inst.	0	1	2	3	4	5	6	7						
FPMUL F2←F1,F0	IA	IF	SI	DI	AC	MEM	EX	**EX**	**EX**	**EX**	**EX**	WB		
FPADD F4←F2,F3		IA	IF	SI	DI	AC	MEM	●	●	●	●	EX	EX	
FPADD **F2**←F1,F3			IA	IF	SI	DI	AC	MEM	EX	**EX**	WB			

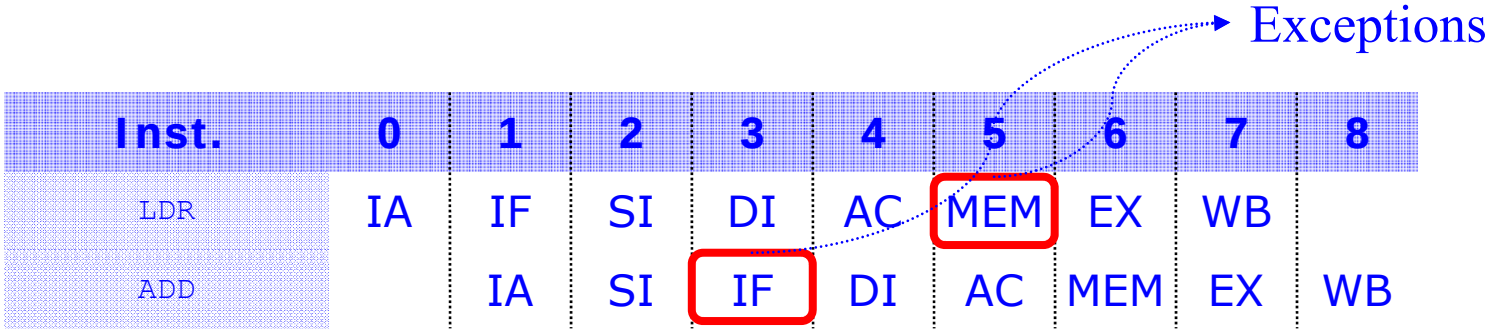
- Nouvelles dépendances de données:
  - Ecriture des registres dans le désordre
- Nouveaux conflits de ressources (ports du banc de registres)

Ecriture
dans le désordre

Inst.	0	1	2	3	4	5	6	7						
FPMUL F2←F1,F0	IA	IF	SI	DI	AC	MEM	EX	EX	EX	EX	EX	WB		
FPADD F4←F2,F3		IA	IF	SI	DI	AC	MEM	●	●	●	●	EX	EX	
FPADD **F5**←F1,F3			IA	IF	IF	DI	AC	MEM	EX	EX	●	●	WB	

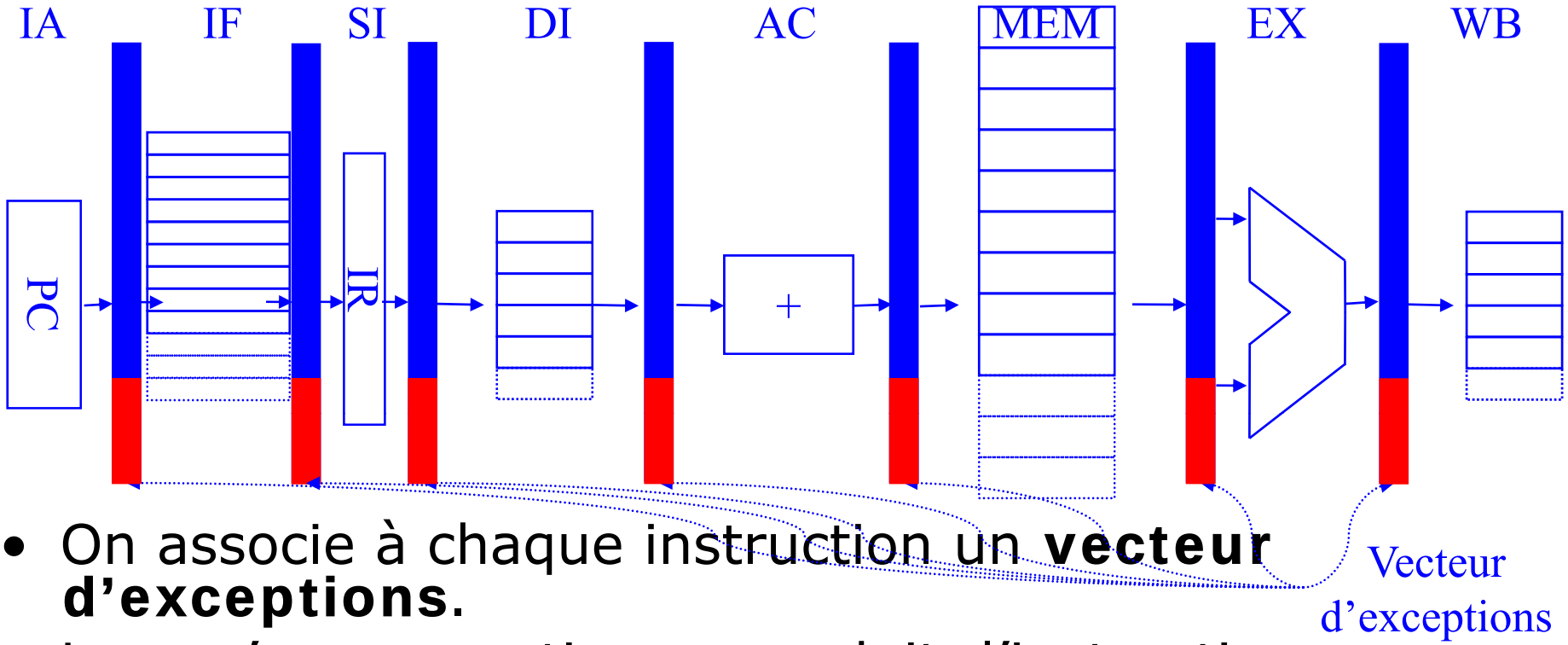
Pipeline et Exceptions

- Le pipeline rend la gestion des exceptions plus difficile. Exemple:
  - `LDR` fait une faute de page en `MEM`
  - `ADD` fait une faute de page en `IF`
- Exception précise sur l'instruction *i*:
  - les instructions < *i* terminent normalement,
  - les instructions > *i* peuvent être interrompues, puis réexécutées (à partir du premier étage du pipeline) après gestion de l'exception.



- Les exceptions doivent être traitées dans l'ordre d'exécution des instructions.
- L'exception de `ADD` intervient avant celle de `LDR`; dans une machine non pipelinée, ce cas ne peut se produire.

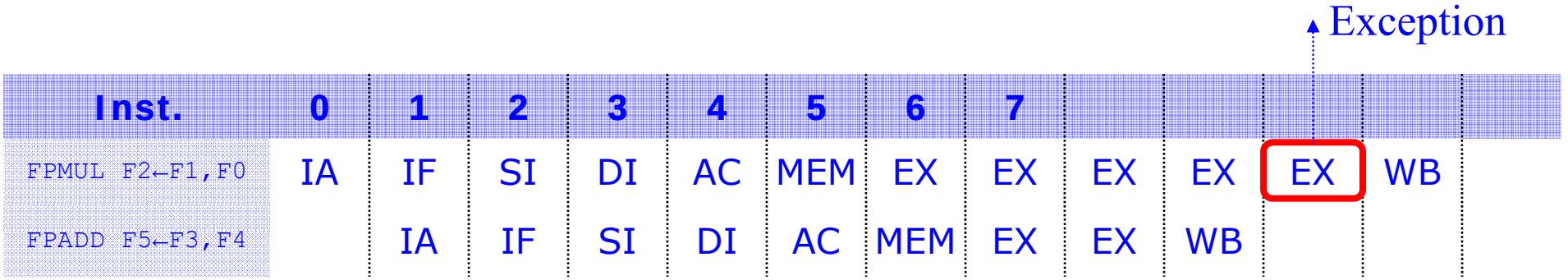
Pipeline et Exceptions



- On associe à chaque instruction un **vecteur d'exceptions**.
- Lorsqu'une exception se produit, l'instruction ne peut plus modifier l'état de la machine (registre/mémoire).
- En WB, le circuit de contrôle examine le vecteur d'exceptions de chaque instruction et détermine si une exception doit être traitée
  - ⇒ Les exceptions sont traitées dans l'ordre d'exécution des instructions.

Exceptions et Instructions Multicycles

- Exemple: l'instruction `FPADD` termine et modifie l'état de la machine avant la détection de l'exception de l'instruction `FPMUL`.
- Le programme ne peut être redémarré normalement à l'instruction `FPMUL`.



- Une solution: retarder les modifications de l'état de la machine (registre/mémoire) jusqu'à ce que toutes les instructions précédentes aient terminé leur exécution.

Aléas de Contrôle

- En cas de branchement, on doit d'abord connaître la destination et le résultat (condition) du branchement, avant de charger l'instruction suivante.

```
LOOP    LDR R1 ← R0,#30
        ...
        BRn LOOP
```

La condition (bits *n*,*p*,*z*) est lue à la fin de cette étape

L'adresse de destination du branchement est disponible à la fin de cette étape

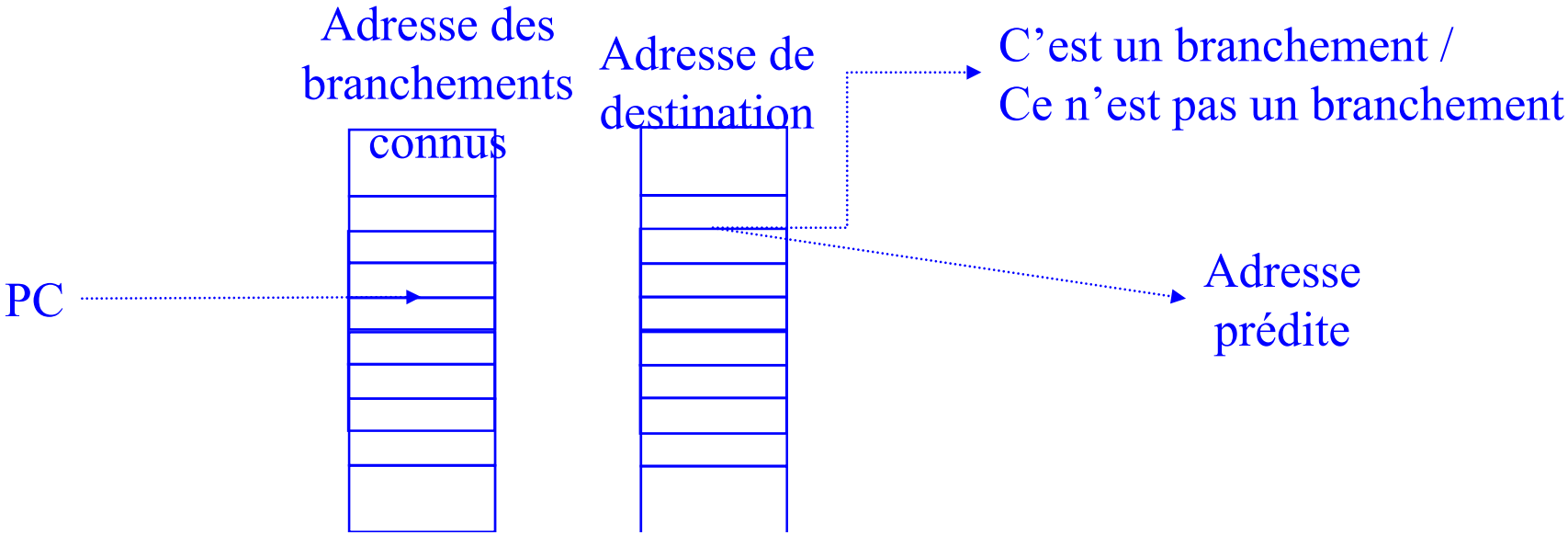
Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12
BRn LOOP	IA	IF	SI	DI	AC	MEM	EX	WB					
LDR R1,R0,#30		●	●	●	●	IA	IF	DI	AC	MEM	EX	WB	

Plan

- Processeurs haute-performance
  - Pipeline
  - Prédiction de branchement
  - Cache
  - Exécution superscalaire/parallèle

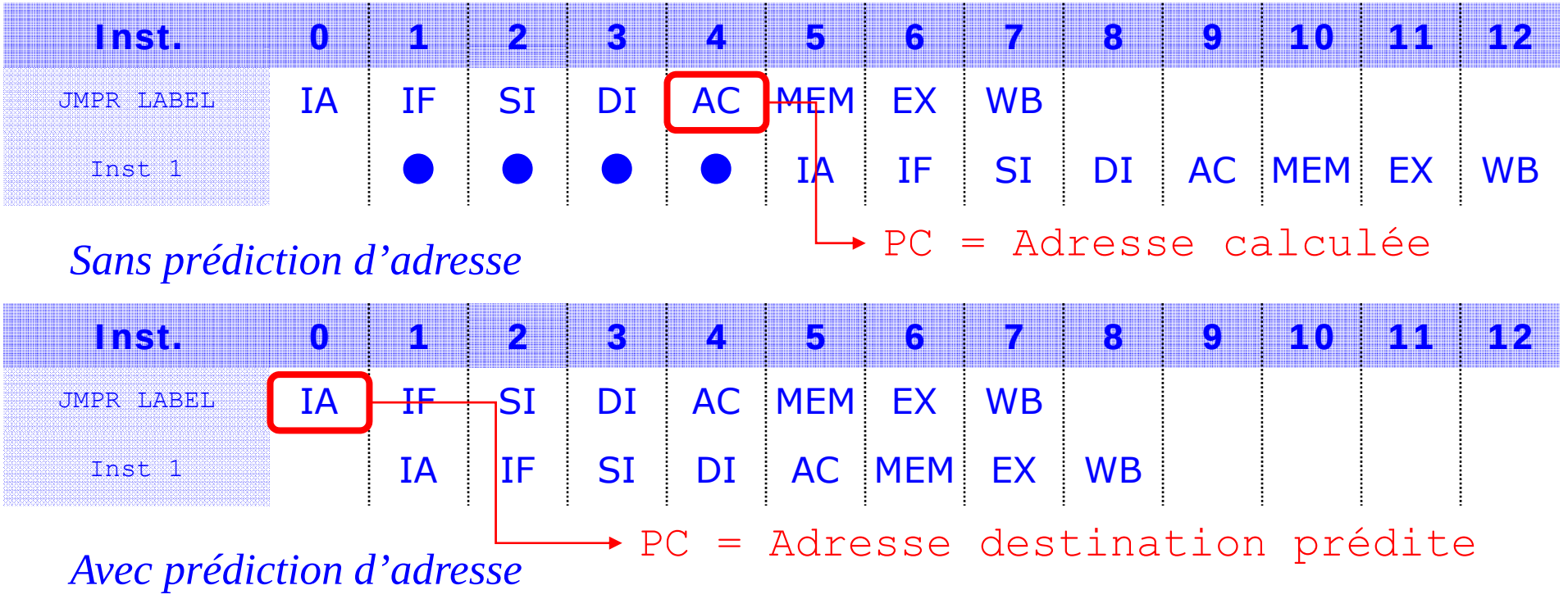
Prédiction de Branchement

- En général, l'adresse de destination d'un branchement est constante (sauf pour `RET` et branchements indirects).
- **Prédire** l'adresse de destination:
  - Les adresses de destination sont stockées dans une table à chaque exécution d'un branchement
  - La table est indexée par le PC de l'instruction (du branchement)
  - Lorsque le PC est envoyé à la mémoire, il est envoyé également à la table qui indique: (1) s'il s'agit d'un branchement, (2) l'adresse de destination si c'est le cas.



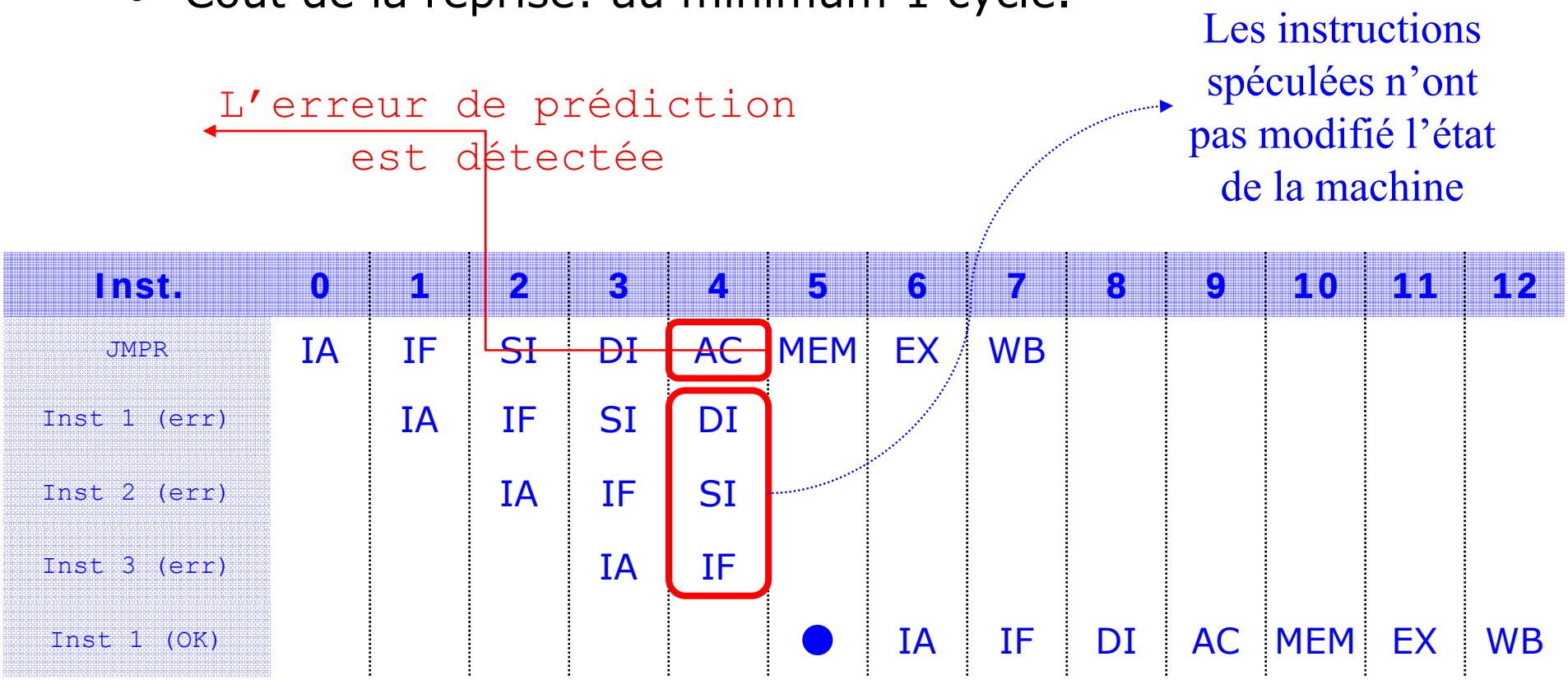
Prédiction d'Adresse

- Si le PC correspond à un branchement, on met à jour le PC: `PC = Adresse de destination`.
- Exemple: branchement inconditionnel

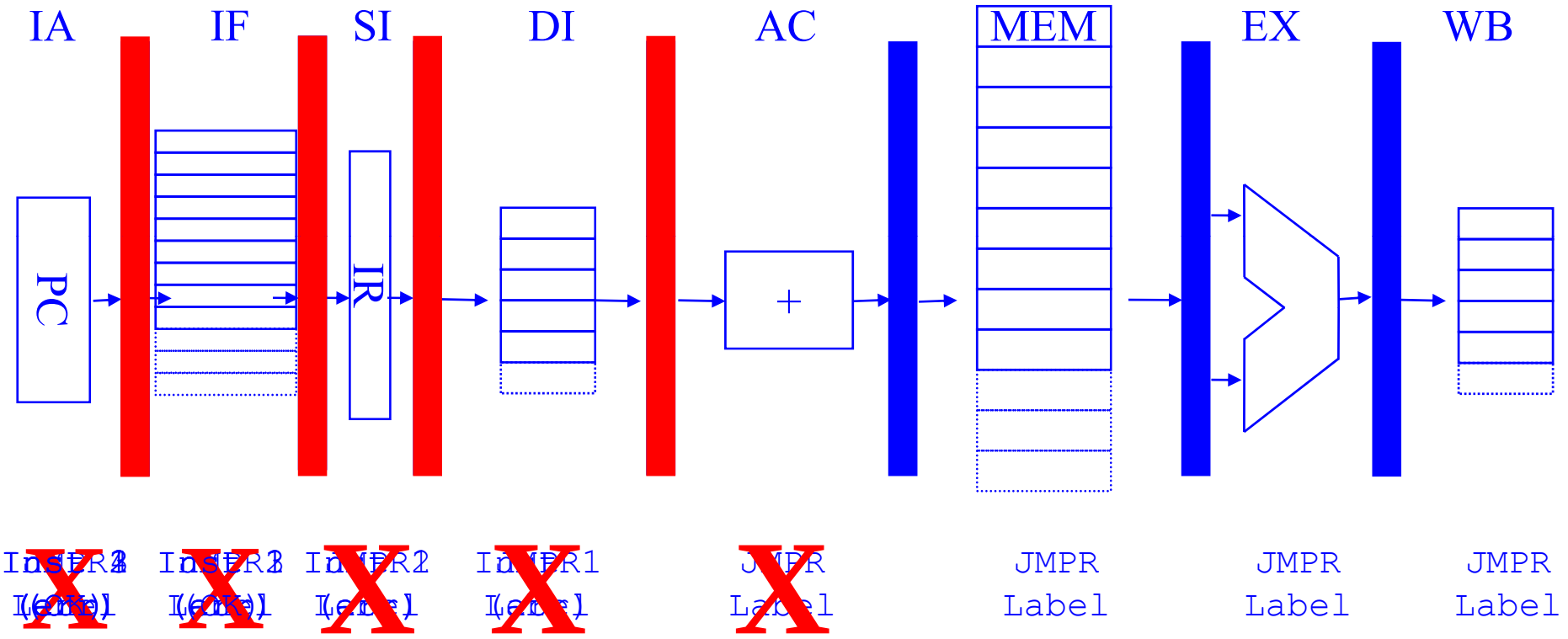


Erreur de Prédiction

- Détecter l'erreur (on effectue toujours le calcul de l'adresse de destination).
- Eliminer du pipeline les instructions chargées spéculativement.
- Les instructions spéculées ne modifient l'état de la machine qu'après vérification de la prédiction.
- Coût de la reprise: au minimum 1 cycle.

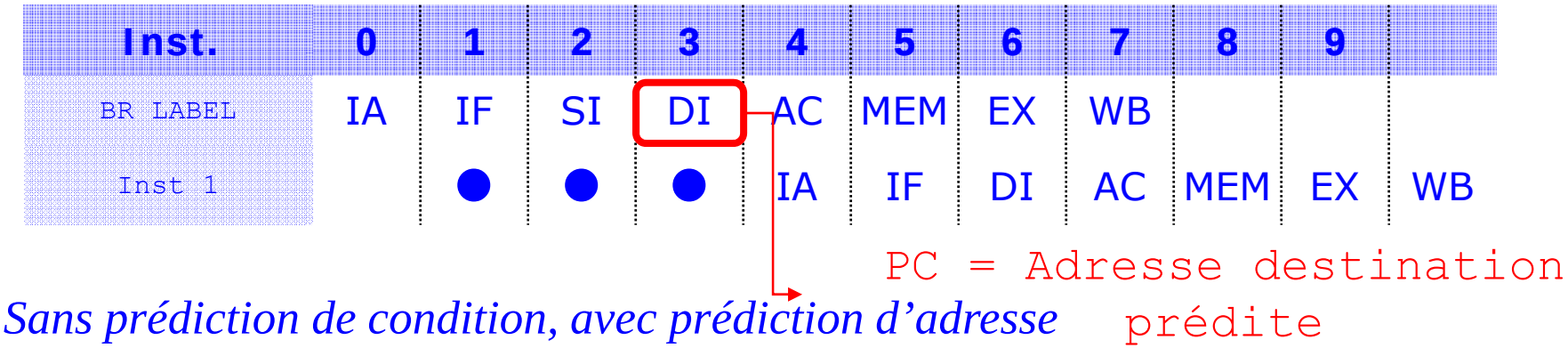


Reprise en Cas d'Erreur

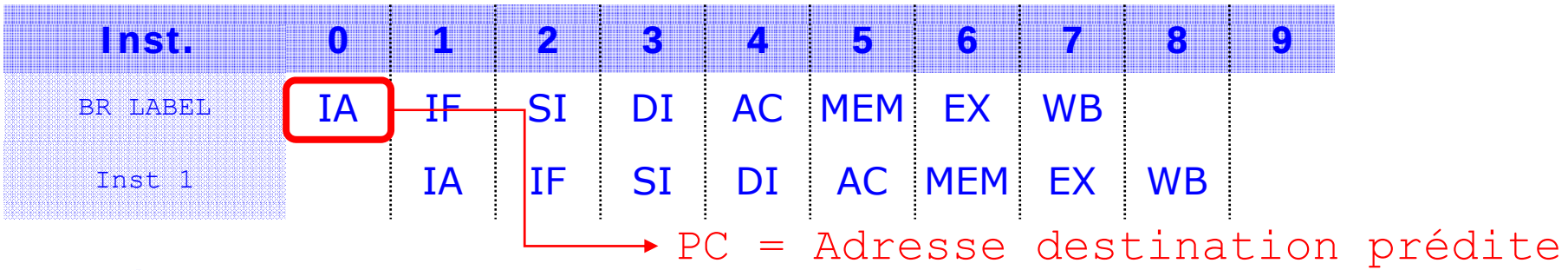


Prédiction de Condition

- Cas des branchements conditionnnels.
- Il faut prédire la valeur de la condition.
- La valeur de la condition varie souvent d'une exécution du branchement à l'autre ⇒ **la prédiction est difficile.**
- Exemple: branchement pris



Sans prédiction de condition, avec prédiction d'adresse



Avec prédiction de condition et d'adresse

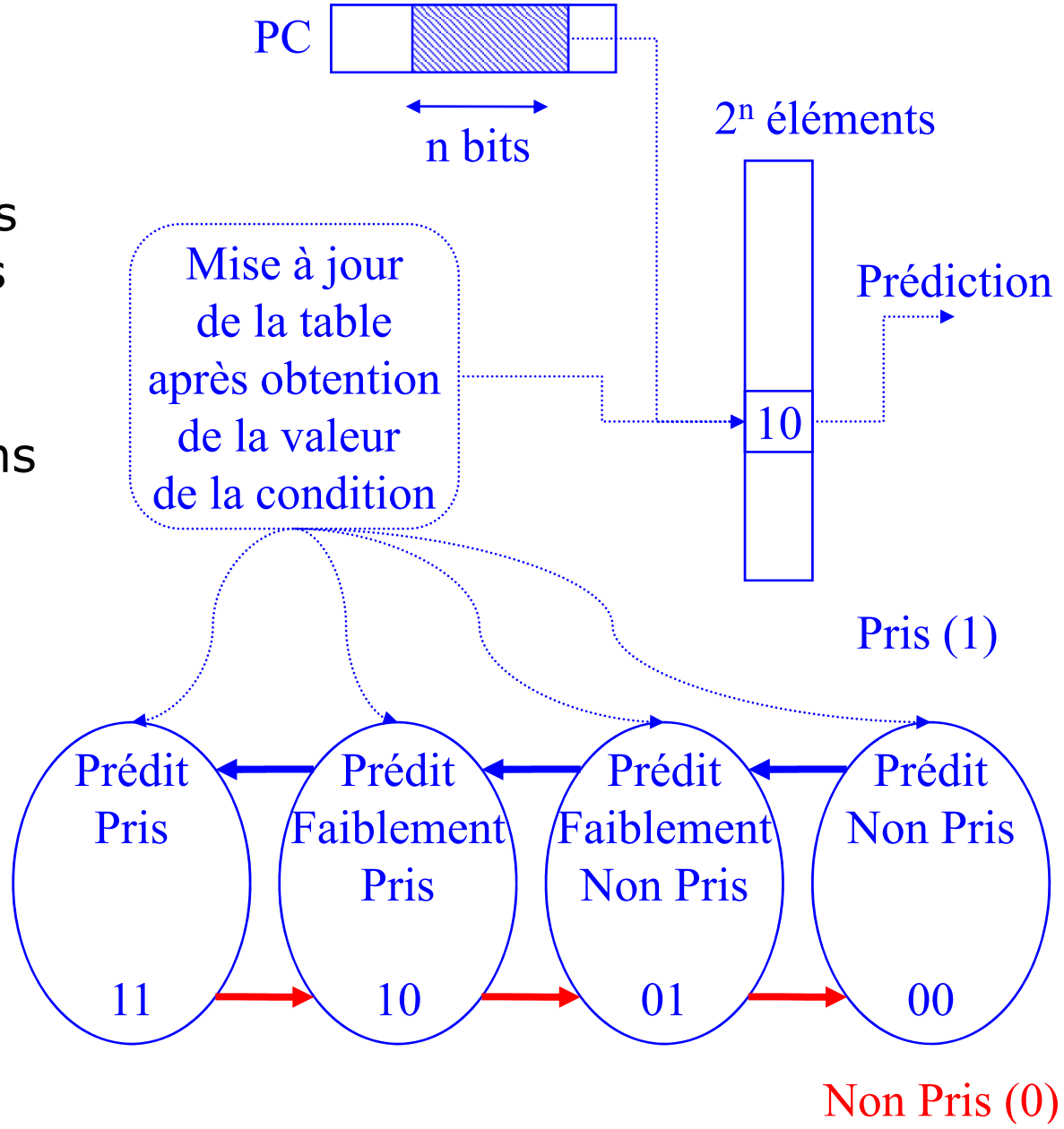
Stratégies de Prédiction

- Prédiction statique:
  - Toujours pris
    - Adéquat pour les boucles
  - Analyse à la compilation
    - EPIC/IA-64; limitations de l'analyse statique
  - Taux de bonnes prédictions: ≈ de 70% jusqu'à 90%

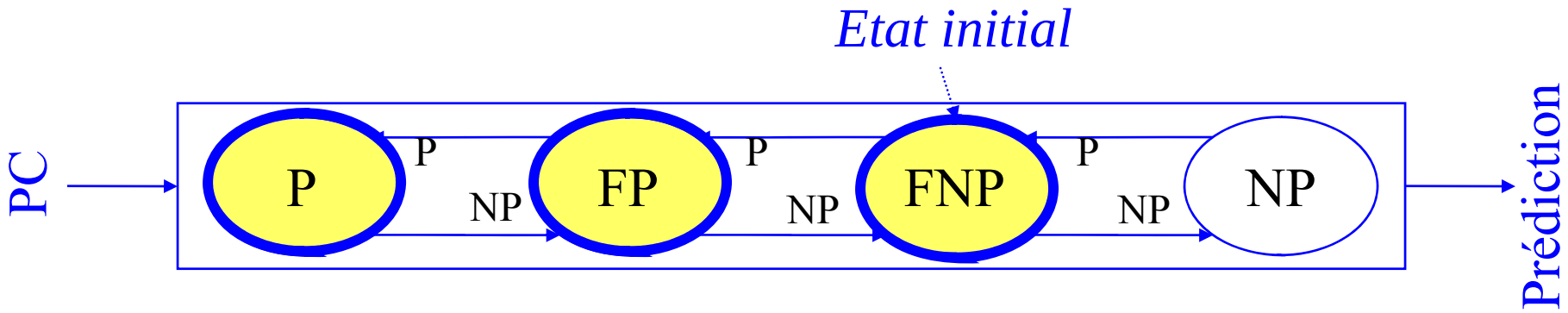
```
                ...
05  LOOP        LDR R1, R0, #3
06              ADD R1, R1, #5
07              STR R1, R0, #30
08              ADD R0, R0, #1
09              ADD R3, R0, R2
0A              BRn LOOP
                ...
```

Stratégies de Prédiction

- Prédiction **dynamique**:
  - Largement utilisée dans les processeurs
  - Mécanismes les plus récents: taux de bonnes prédictions jusqu'à 99% sur certaines applications
  - Principe: apprentissage des comportements individuels des branchements
- Un premier mécanisme: l'historique local
  - Un automate à 4 états par branchement.



Exemple



i=0 (pris)	Non Pris
i=1 (pris)	Pris
i=2 (pris)	Pris
i=3 (pris)	Pris
...	...
i=99 (non pris)	Pris

```
                ...
05  LOOP        LDR R1, R0, #3
06              ADD R1, R1, #5
07              STR R1, R0, #30
08              ADD R0, R0, #1
09              ADD R3, R0, R2
0A              BRn LOOP
                ...
```

Exécution de 100 itérations:

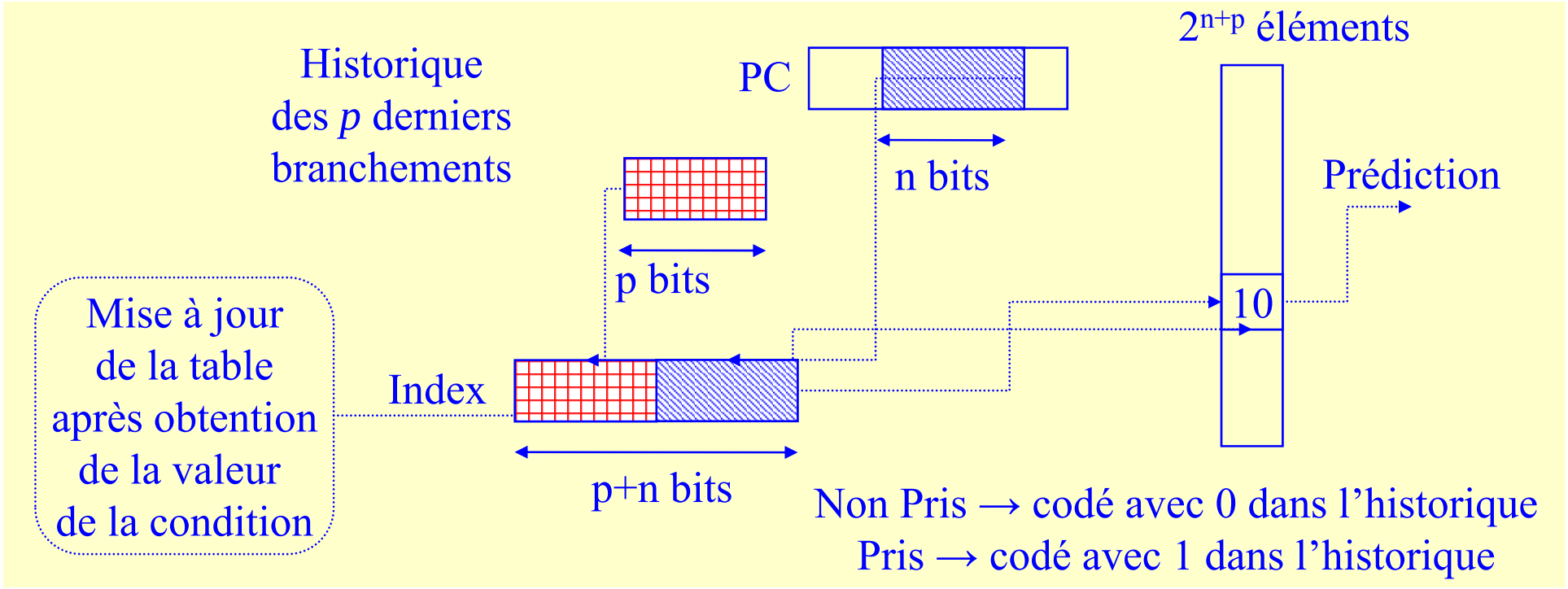
- itération i=0: BRn pris
- itération i=1: BRn pris
- ...
- itération i=99: BRn non pris

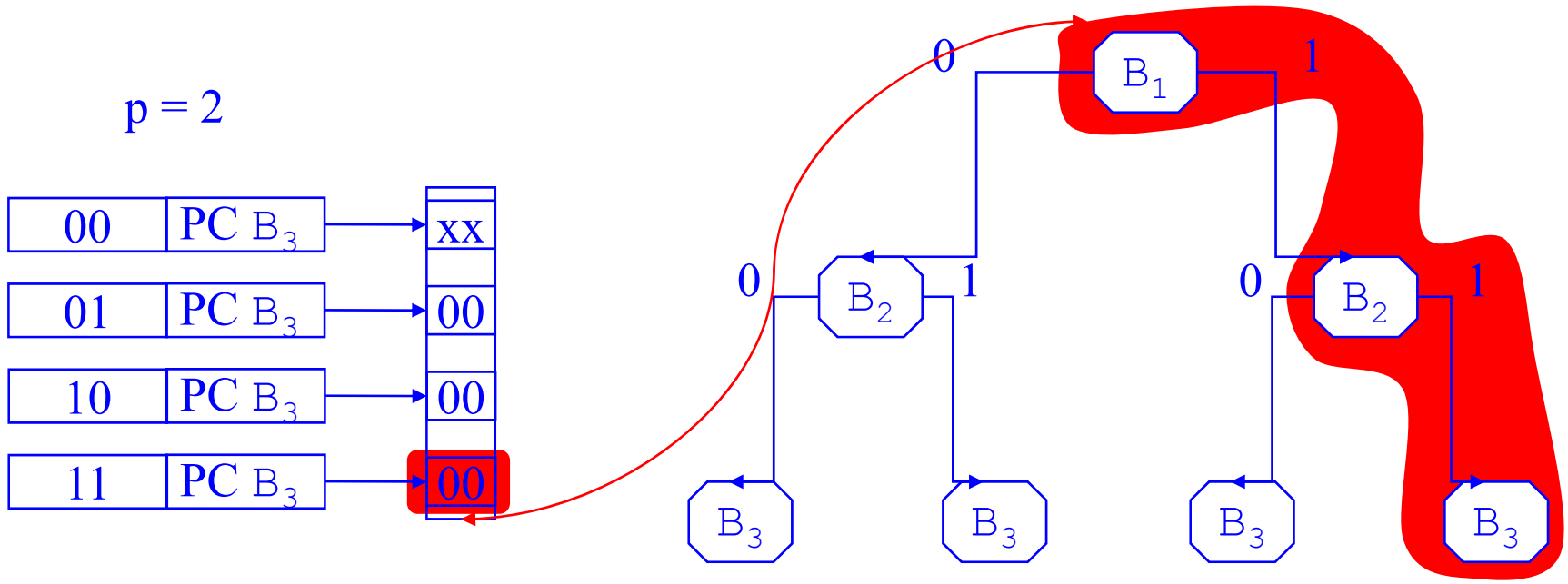
Amélioration de la Prédiction Dynamique

- Une augmentation faible du taux de bonnes prédictions a un impact important sur la performance globale du processeur.

```
if (a == 1) a = 0;        /* Branchement B1 */
...
if (b == 0) b = 1;        /* Branchement B2 */
...
if (a == b) ...;          /* Branchement B3 */
```

- Pour affiner la précision de la prédiction, on utilise le comportement des branchements antérieurs: historique global.


Historique
des p derniers
branchements
PC
2n+p éléments
n bits
Prédiction
p bits
10
Mise à jour
de la table
après obtention
de la valeur
de la condition
Index
p+n bits
Non Pris → codé avec 0 dans l'historique
Pris → codé avec 1 dans l'historique


p = 2
00
PC B3
01
PC B3
10
PC B3
11
PC B3
xx
00
00
00
0
1
B1
0
1
B2
0
1
B2
B3
B3
B3
B3

Impact des Aléas de Contrôle sur la Performance d'un Processeur

- En moyenne, 1 instruction sur 5 est un branchement.
- Sur les pipelines actuels, le coût d'un aléa de contrôle est de 5 à 10 cycles.

- 8 cycles entre le fetch et la résolution du branchement
- 4 cycles pour réinitialiser le pipeline

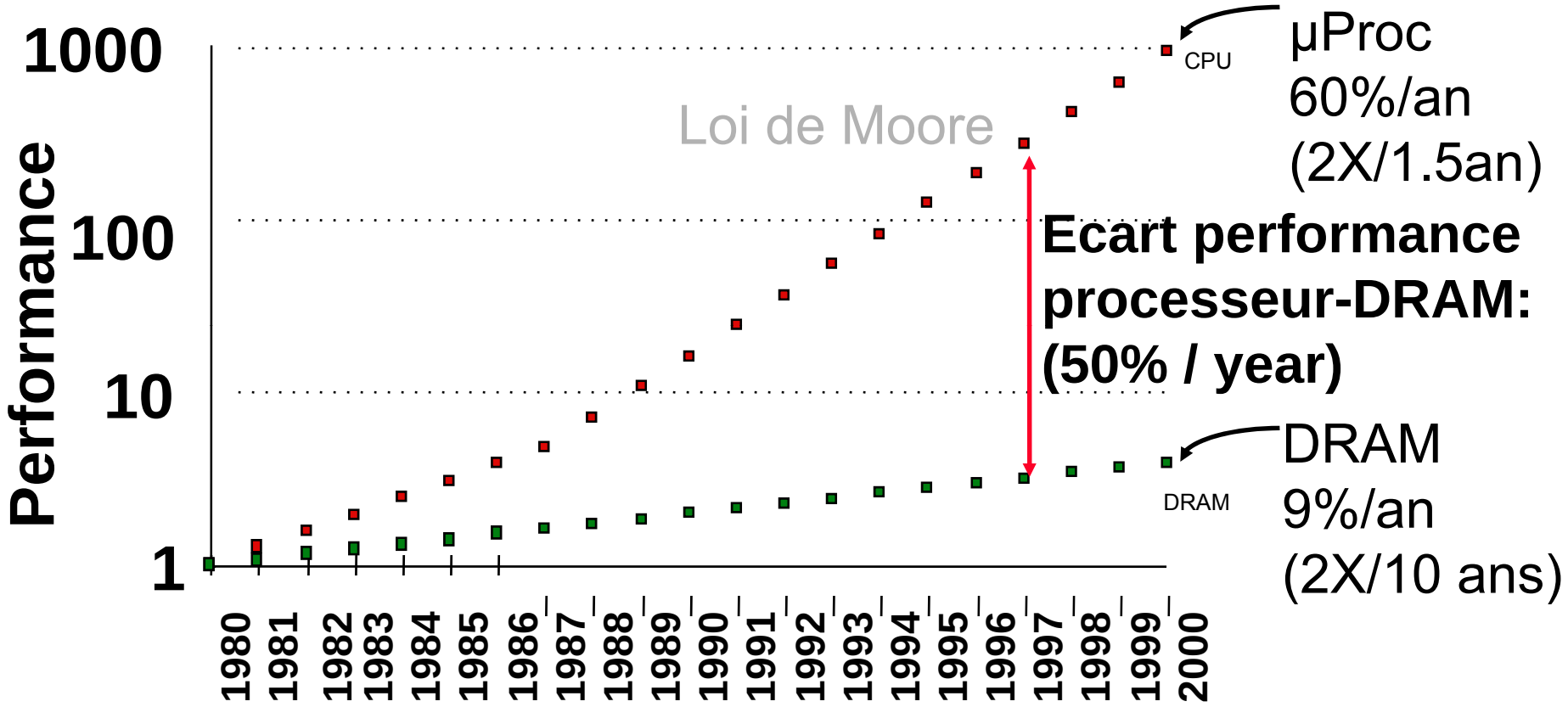
$\Rightarrow$ 12 cycles de pénalité.

- En moyenne 1 branchement toutes les 5 instructions.
- 1 instruction / cycle (1000 instructions) :
  - 50% mauvaises prédictions:
    1000 * (0,8*1 + 0,2*(0,5*1 + 0,5*12)) = 2100 cycles
  - 20% mauvaises prédictions:
    1000 * (0,8*1 + 0,2*(0,8*1 + 0,2*12)) = 1440 cycles
  - 5% mauvaises prédictions:
    1000 * (0,8*1 + 0,2*(0,95*1 + 0,05*12)) = 1110 cycles
- 5 instructions / cycle (1000 instructions) :
  - 50% mauvaises prédictions:
    200 * (0,5*1 + 0,5*12) = 1300 cycles
  - 20% mauvaises prédictions:
    200 * (0,8*1 + 0,2*12) = 640 cycles
  - 5% mauvaises prédictions:
    200 * (0,95*1 + 0,05*12) = 310 cycles

Plan

- Processeurs haute-performance
  - Pipeline
  - Prédiction de branchement
  - Cache
  - Exécution superscalaire/parallèle

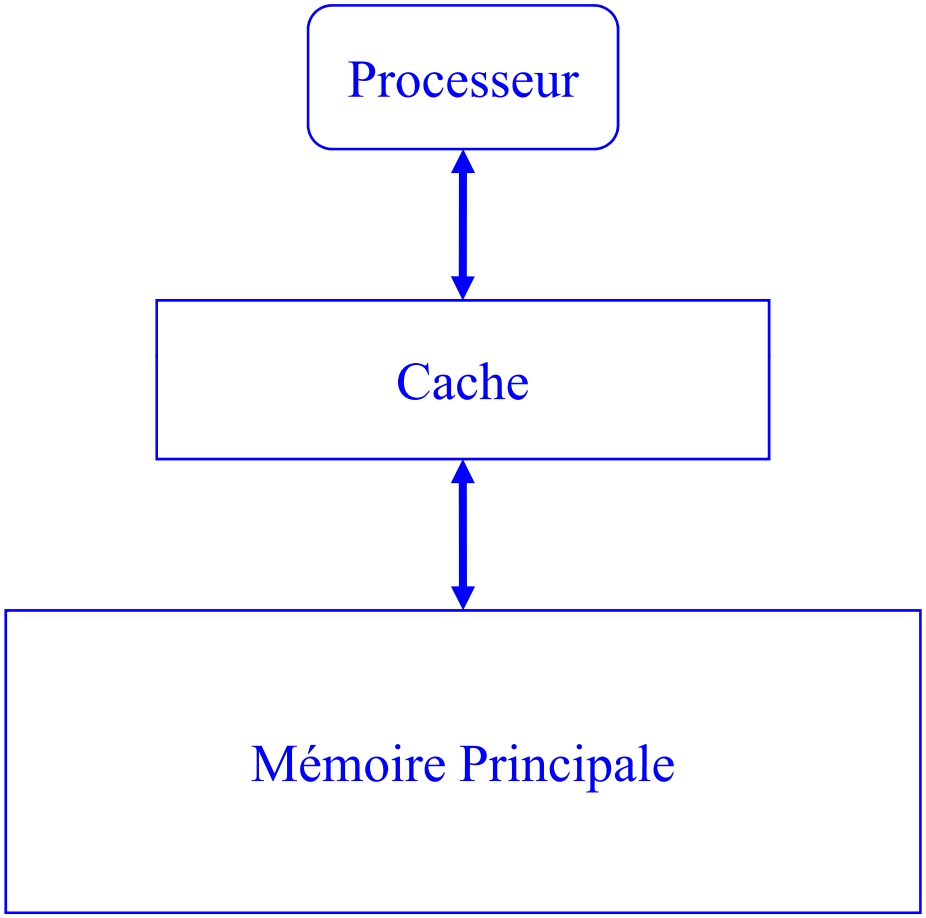
Latence d'Accès à la Mémoire



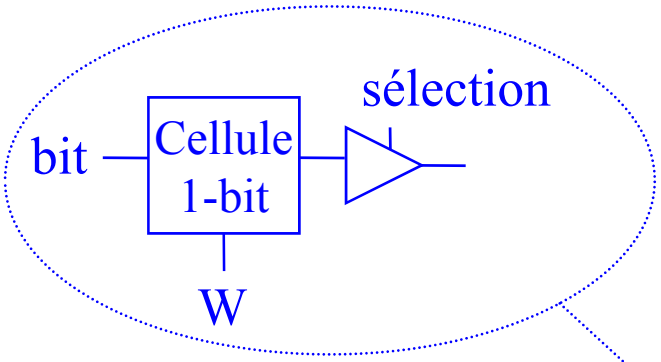
- Temps de cycle du processeur << temps d'accès à la mémoire.

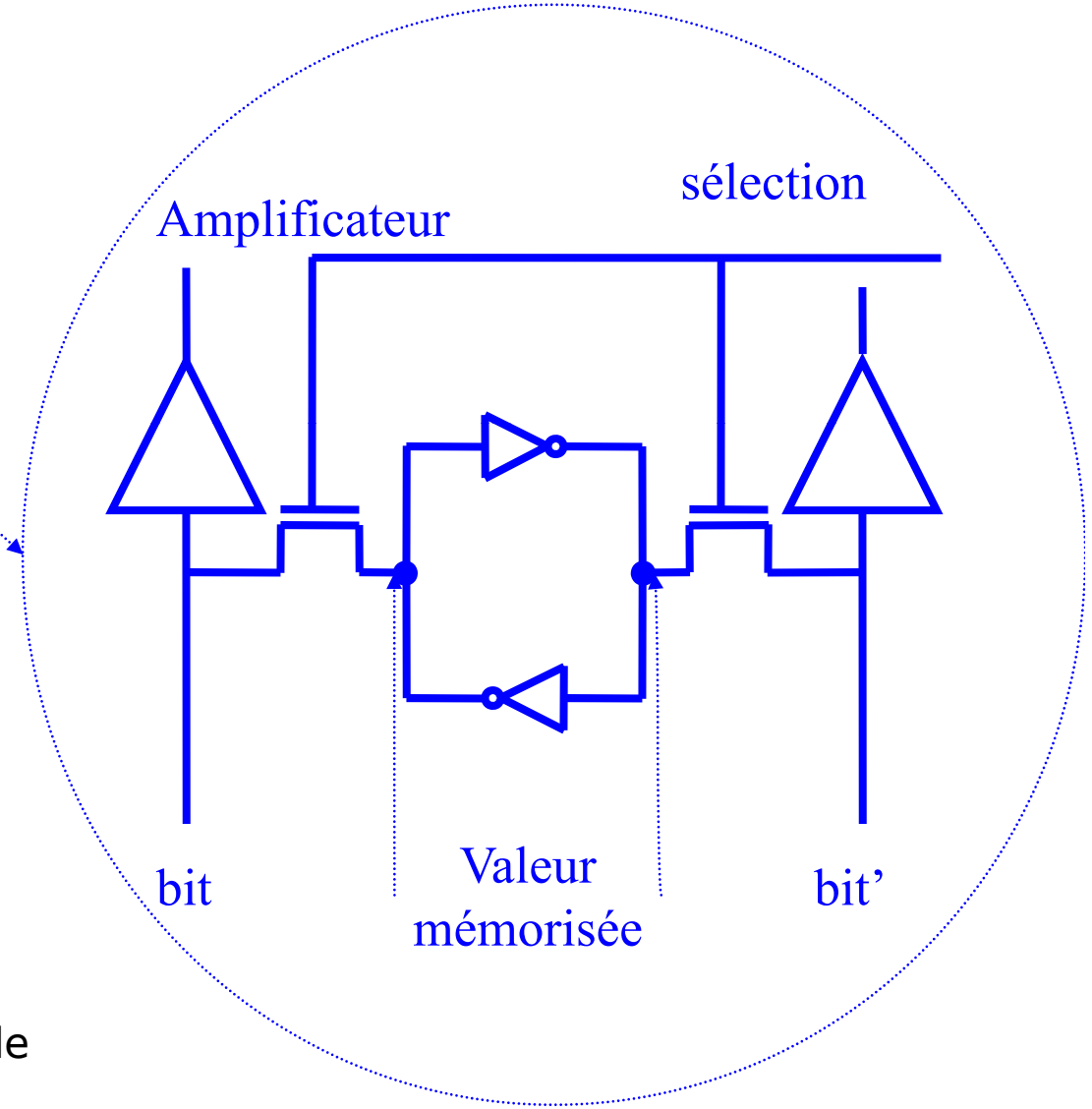
Mémoire Cache

- Mémoire rapide (SRAM) mais petite (coût):
  - ≈ entre 1 et 3 cycles (accès éventuellement pipeliné)
- Le processeur envoie ses requêtes mémoire au cache
  - Donnée dans le cache: succès (*hit*)
  - Donnée hors du cache: défaut (*miss*)
- Performance:
  - Taux de défauts de cache
  - Temps moyen d'accès à la mémoire



Cellule 1-bit SRAM





- SRAM=*Static Random Access Memory.*
- Ecriture:
  - bit=*valeur*; bit'=*valeur'*
  - sélection=1
- Lecture:
  - sélection=0
  - bit=V_{DD}, bit'=V_{DD}
  - sélection=1
  - valeur:
    - $1/V_{DD}$ → décroissance de V sur bit'.
    - $0/V_{SS}$ → décroissance de V sur bit.

Cache & Propriétés de Localité

- La plupart des programmes possèdent de fortes propriété de localité.
- Localité temporelle: une adresse *A* référencée à l'instant *t* a une forte probabilité d'être référencée à nouveau dans un court intervalle de temps.
- Localité spatiale: si une adresse *A* est référencée à l'instant *t*, il y a une forte probabilité qu'une adresse voisine de *A* soit référencée dans un court intervalle de temps.
- Le cache exploite ces deux propriétés de localité.

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    y[i] = y[i] + a[i][j] * x[j]
  }
}
```

- `y[i]`: propriétés de localités temporelle et spatiale.
- `a[i][j]`: propriétés de localité spatiale.
- `x[j]`: propriétés de localité temporelle et spatiale.

Localité des Données et des Instructions

- Les instructions, comme les données, possèdent de fortes propriétés de localité.
- La localité temporelle est simplement exploitée en conservant une donnée dans le cache.
- La localité spatiale est exploitée en chargeant les données par blocs et non individuellement.

```
                ...
05  LOOP        LDR R1, R0, #3
06              ADD R1, R1, #5
07              STR R1, R0, #30
08              ADD R0, R0, #1
09              ADD R3, R0, R2
0A              BRn LOOP
                ...
```

Boucle

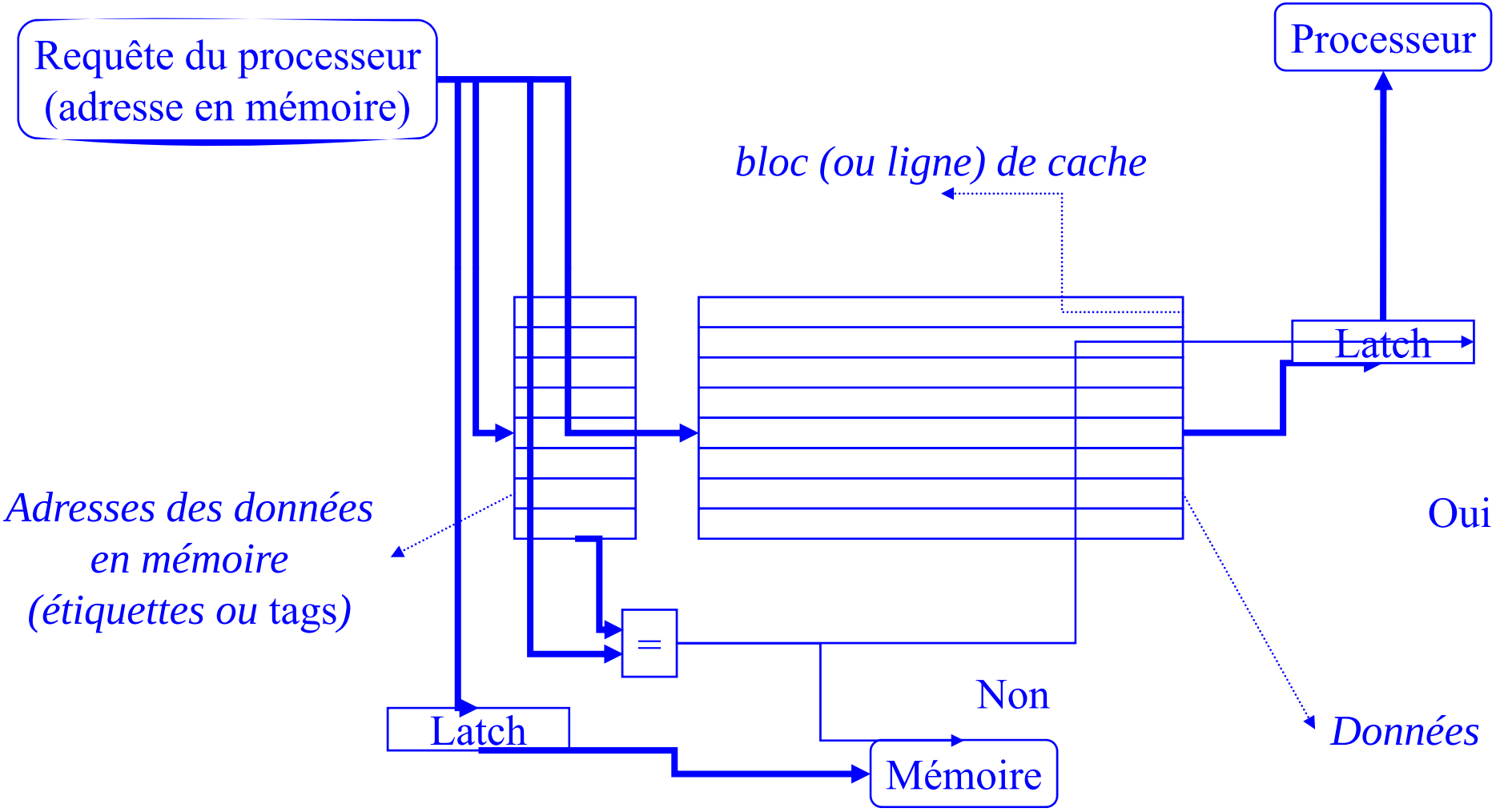
⇒ réutilisation des instructions

⇒ localité temporelle

Instructions consécutives en mémoire

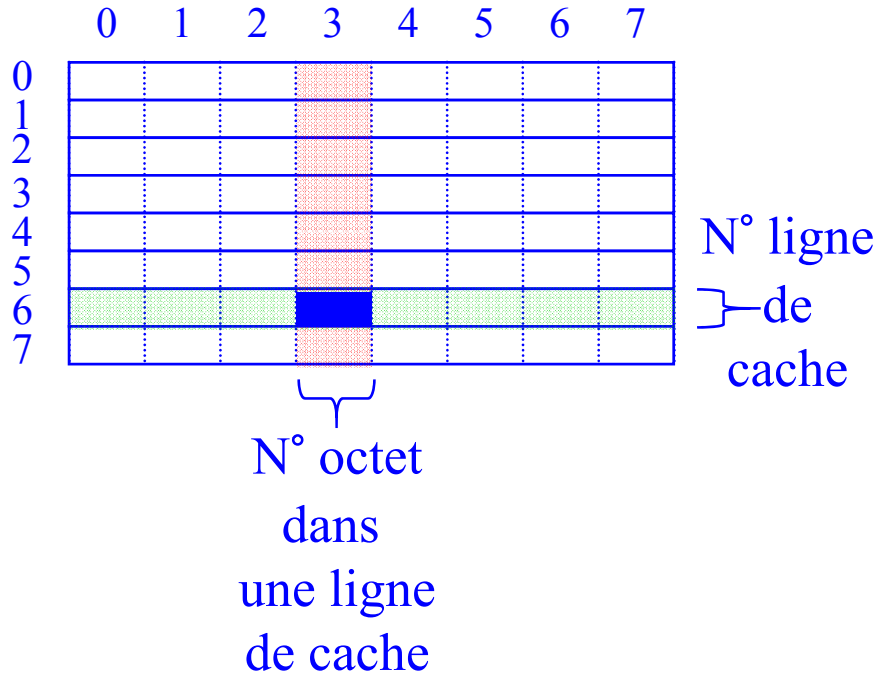
⇒ localité spatiale

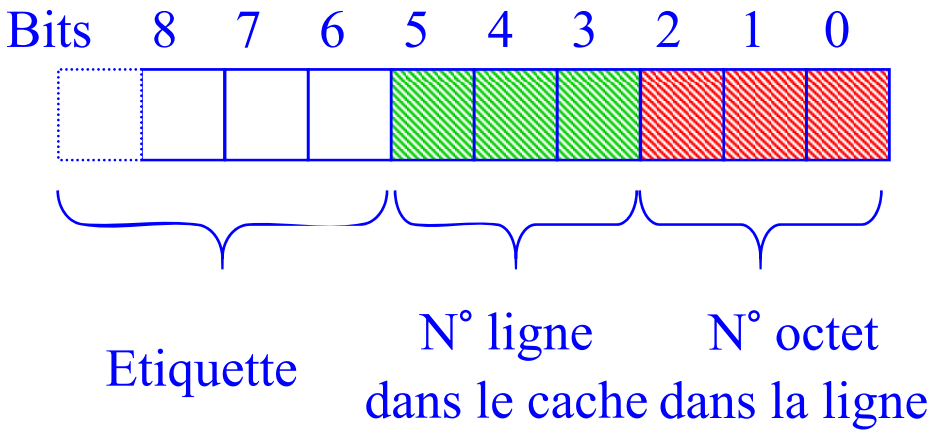
Structure d'un Cache



Placement des Données

- Le placement des données est géré par le matériel
  - ⇒ Le programmeur n'a pas à se soucier du placement des données
  - ⇒ Le fonctionnement du cache est transparent pour le programmeur.
- L'emplacement d'une donnée est déterminé selon une fonction simple de l'adresse.
  - N° ligne dans le cache
  - N° d'octet dans la ligne

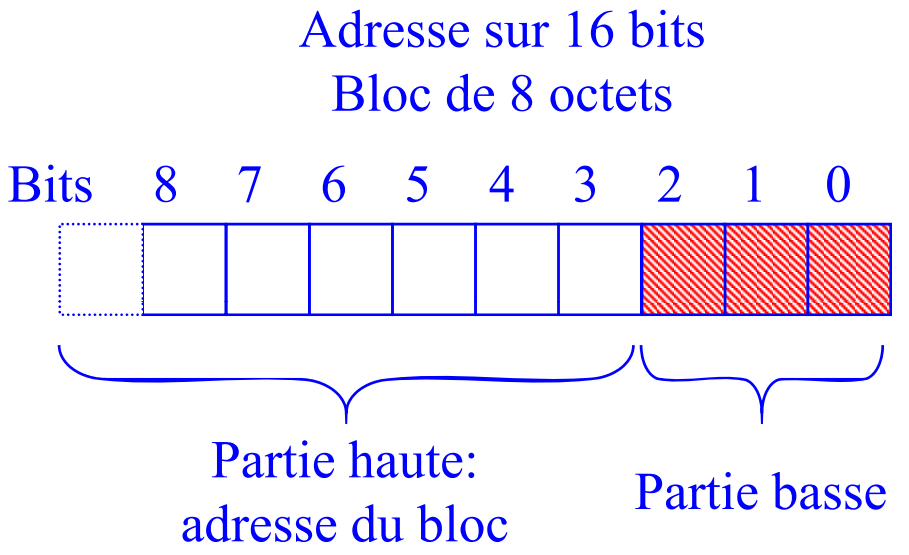




- Cache de C_S octets
- Ligne de L_S octets
- N° octet: $\log_2(L_S)$ bits de poids faible de l'adresse.
- N° ligne: $\log_2(C_S/L_S)$ bits de poids faible suivants de l'adresse.

Ligne de Cache

- Charger les données par blocs (localité spatiale).
- Décrire un bloc de données avec une seule adresse (contraintes du bus mémoire, calcul d'adresse).
  - La partie haute de l'adresse des données d'un même bloc est identique
  - Seule la partie basse de l'adresse varie.
- Rappel: une adresse = 1 octet.



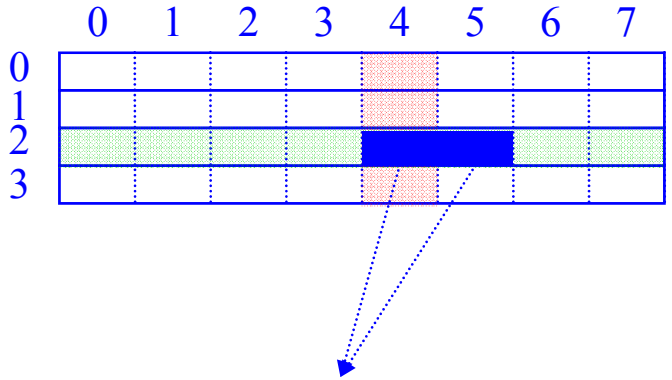
Exemple:

0...010100010
0...010100111
0...010101111

Même ligne de cache
Lignes distinctes,
adresses consécutives

Lecture d'une Donnée

- Exemple:
  - C_S = 32 octets
  - L_S = 8 octets
- Adresse demandée (16 bits):
  - 0110010001010100
  - N° ligne: 10
  - N° octet dans la ligne: 100
- Une requête peut avoir une taille variable:
  - octet, demi-mot, mot...
  - requête = adresse + nombre d'octets
  - adresse = adresse du premier octet
  - exemple: 2 octets (mot de 16 bits)

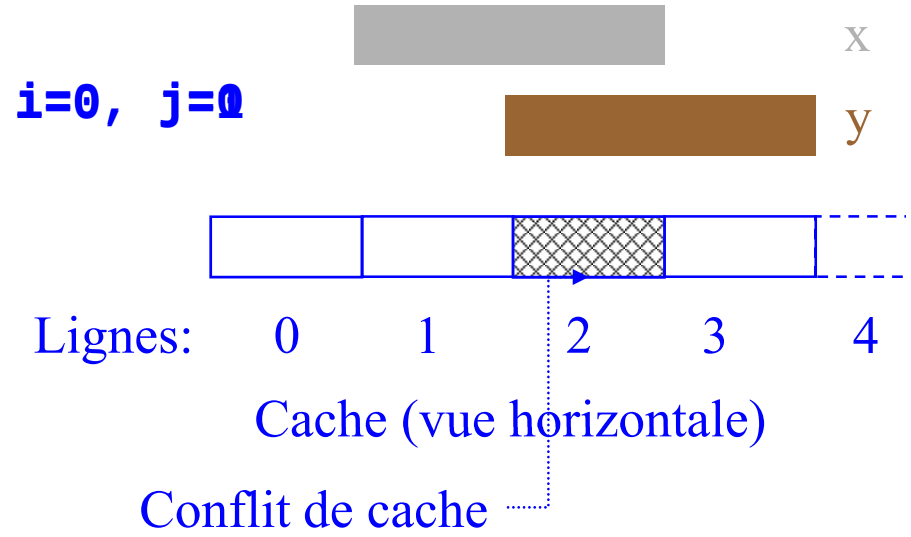


Associativité

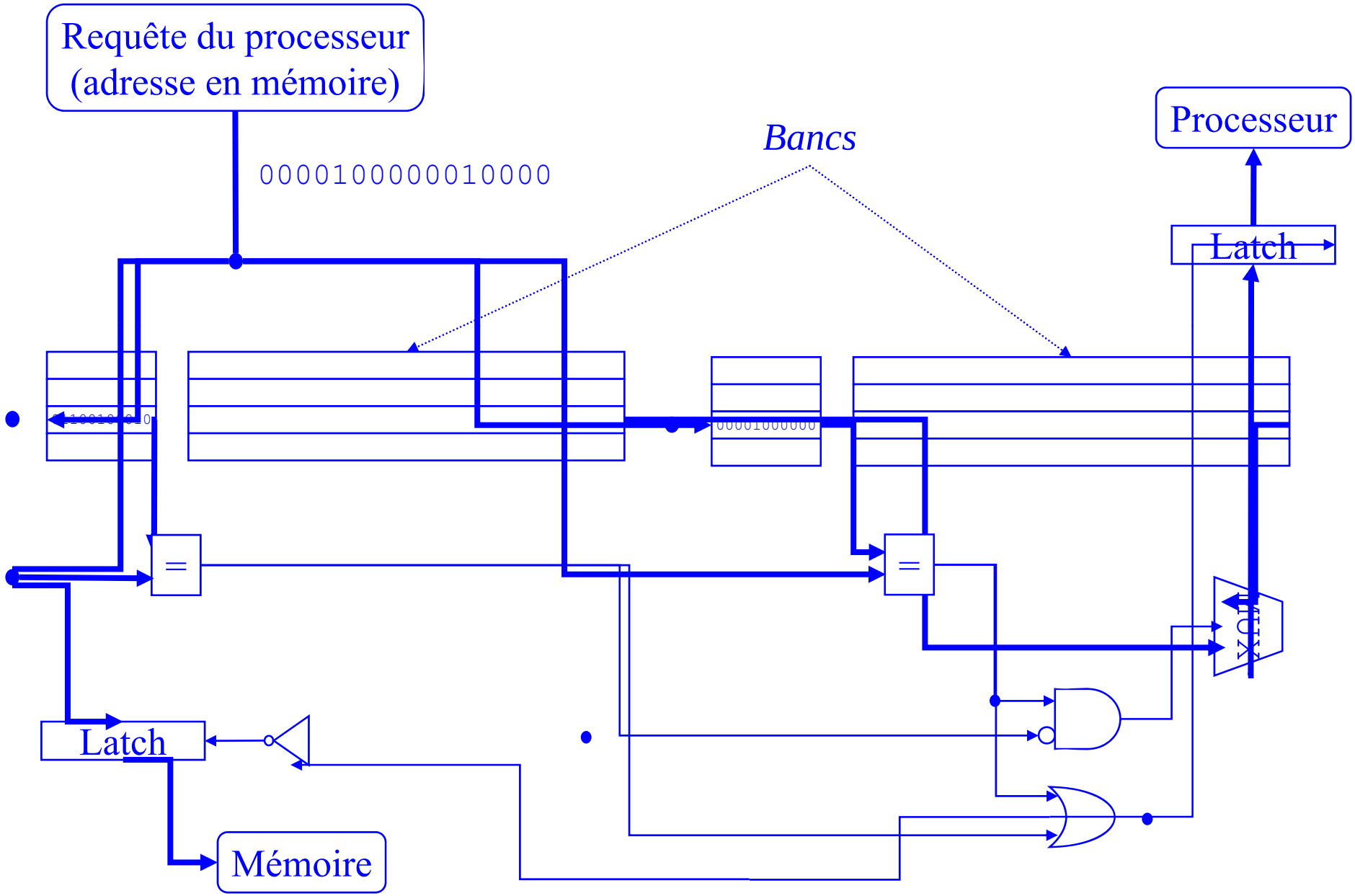
- Taille mémoire physique >> taille cache
- La fonction de placement peut engendrer des conflits entre les données.
- On peut réduire les conflits en augmentant l'**associativité** des caches.

```
for (i=0; i<N; i++) {
  for (j=0; j<2; j++) {
    x[j] = y[j]
  }
}
```

```
@x = 0110010001001000
@y = 0000100000010000
```

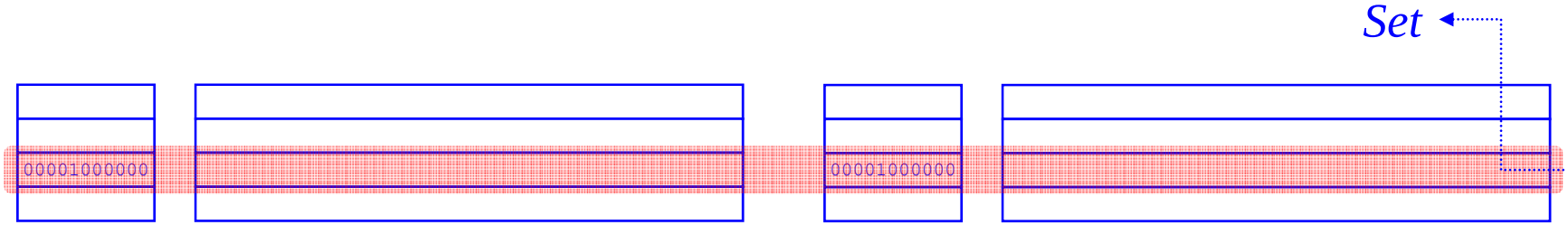


Structure d'un Cache Associatif

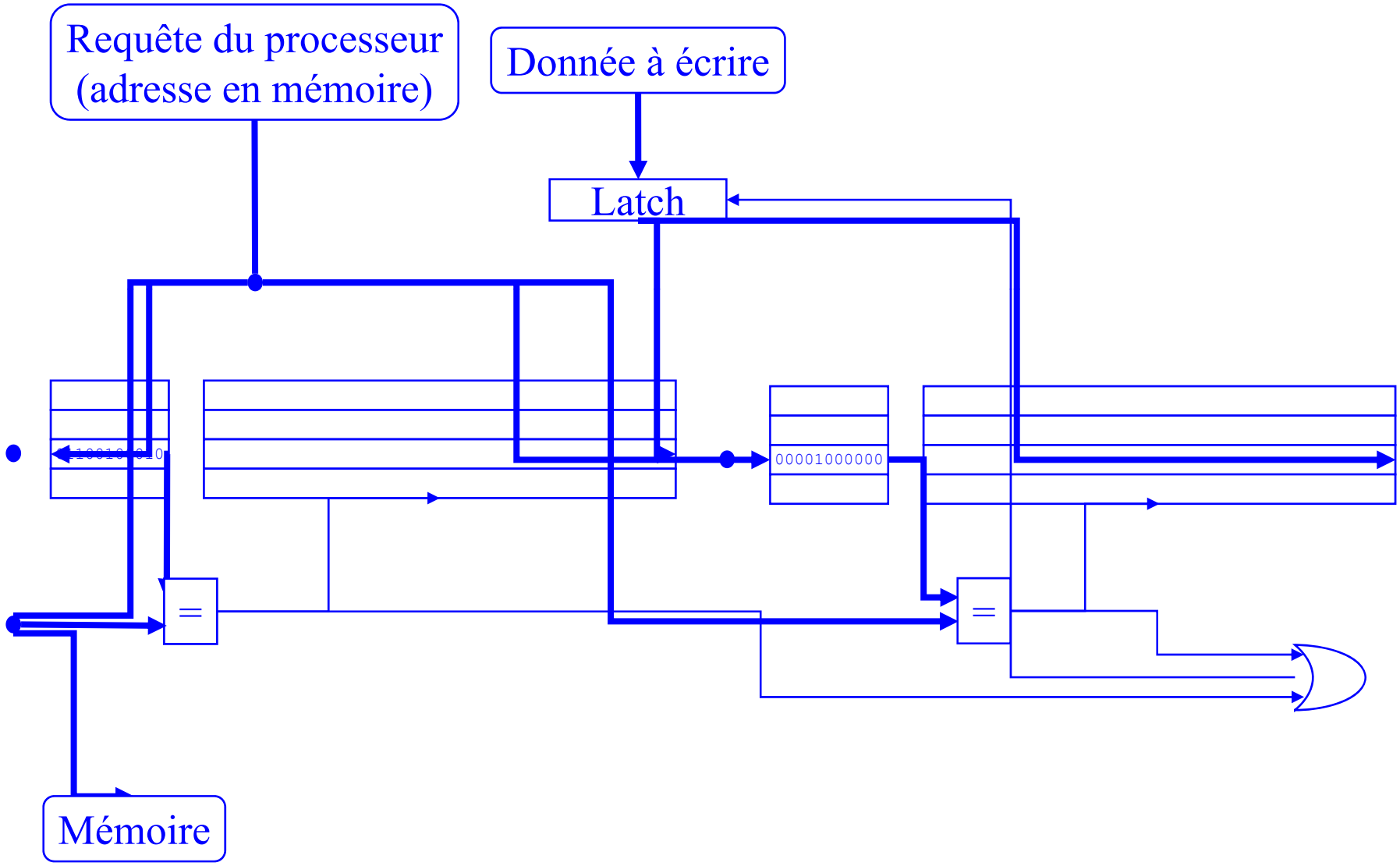


Fonctionnement Cache Associatif

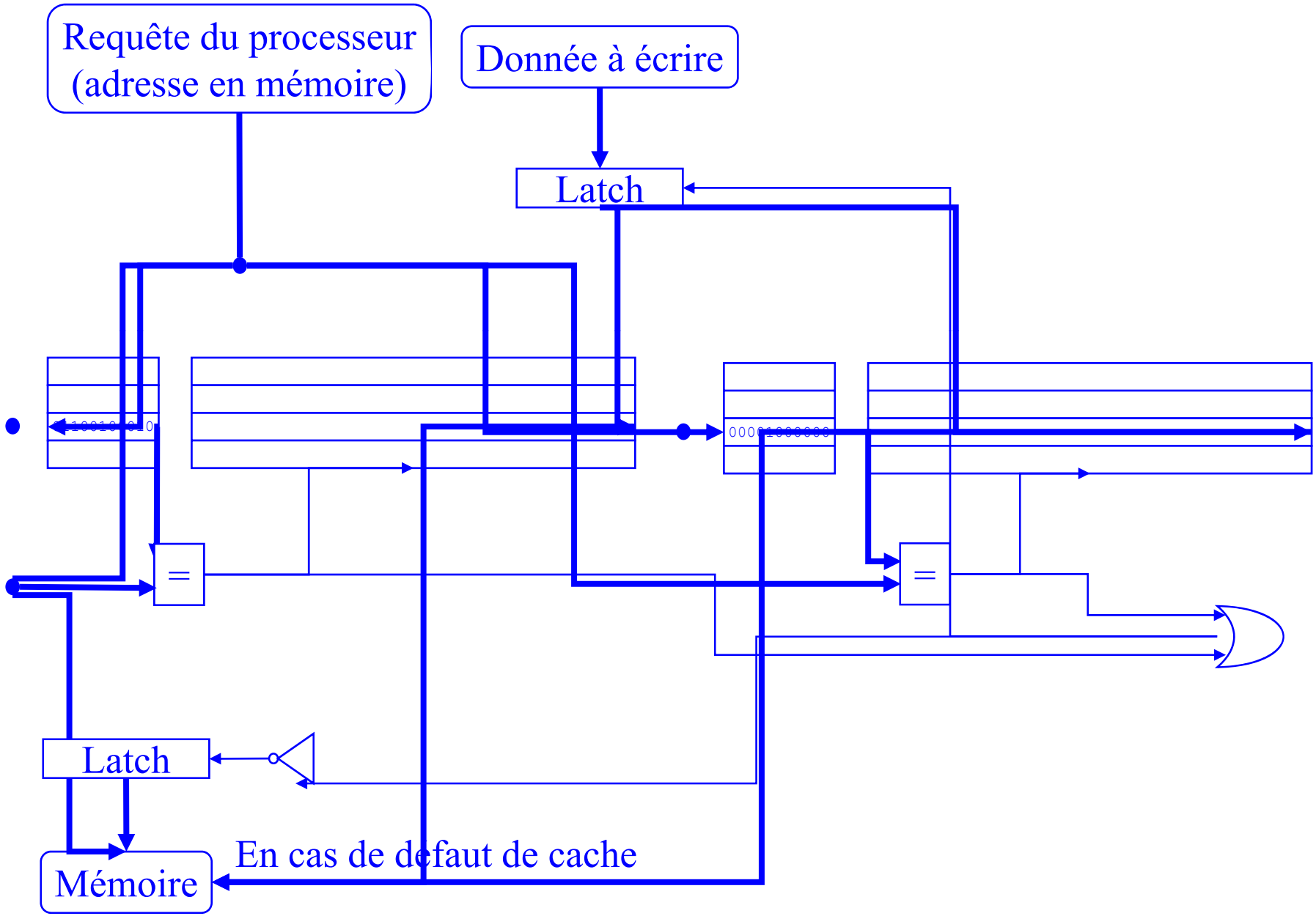
- Degré d'associativité *n*.
- Une donnée peut être stockée dans *n* emplacements différents.
- En cas de défaut de cache, il faut choisir le bloc dans lequel on va charger la nouvelle donnée.
- Le choix est effectué entre les blocs du cache qui peuvent accueillir la donnée (un *set*):
  - LRU (*Least Recently Used*) : on choisit le bloc le plus anciennement accédé.
  - *Random*
  - Pseudo-LRU: la ligne la plus récemment accédée n'est pas remplacée; choix aléatoire entre les autres lignes.
  - FIFO (*First In First Out*)



Ecriture d'une Donnée (*Write-Through*)

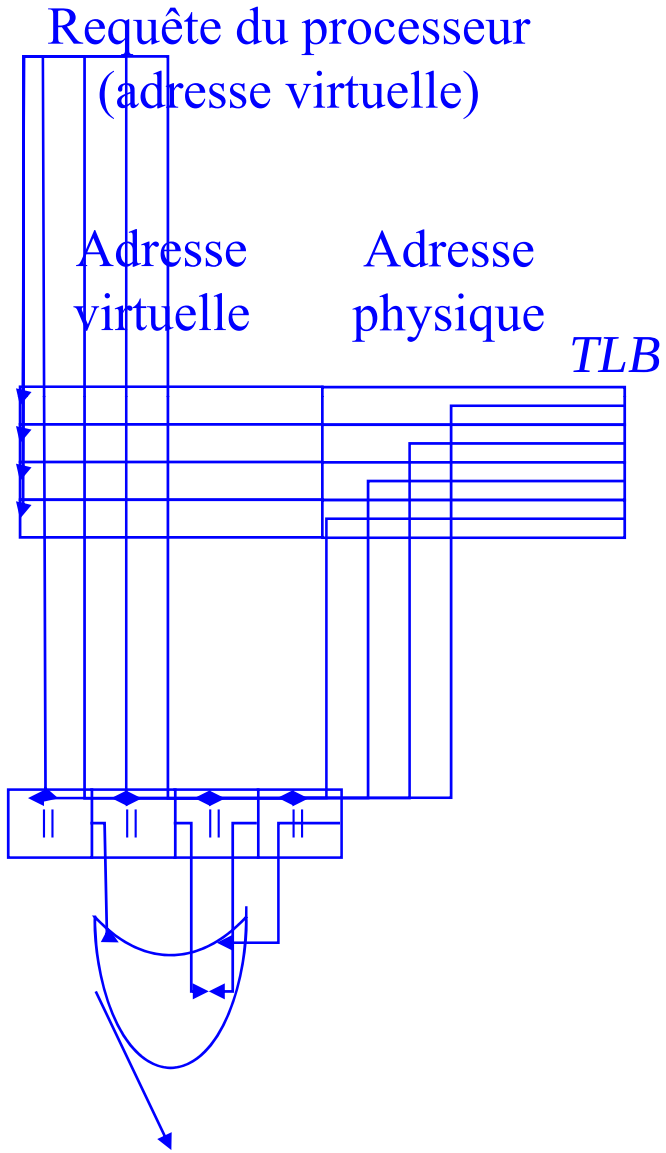


Ecriture d'une Donnée (*Write-Back*)

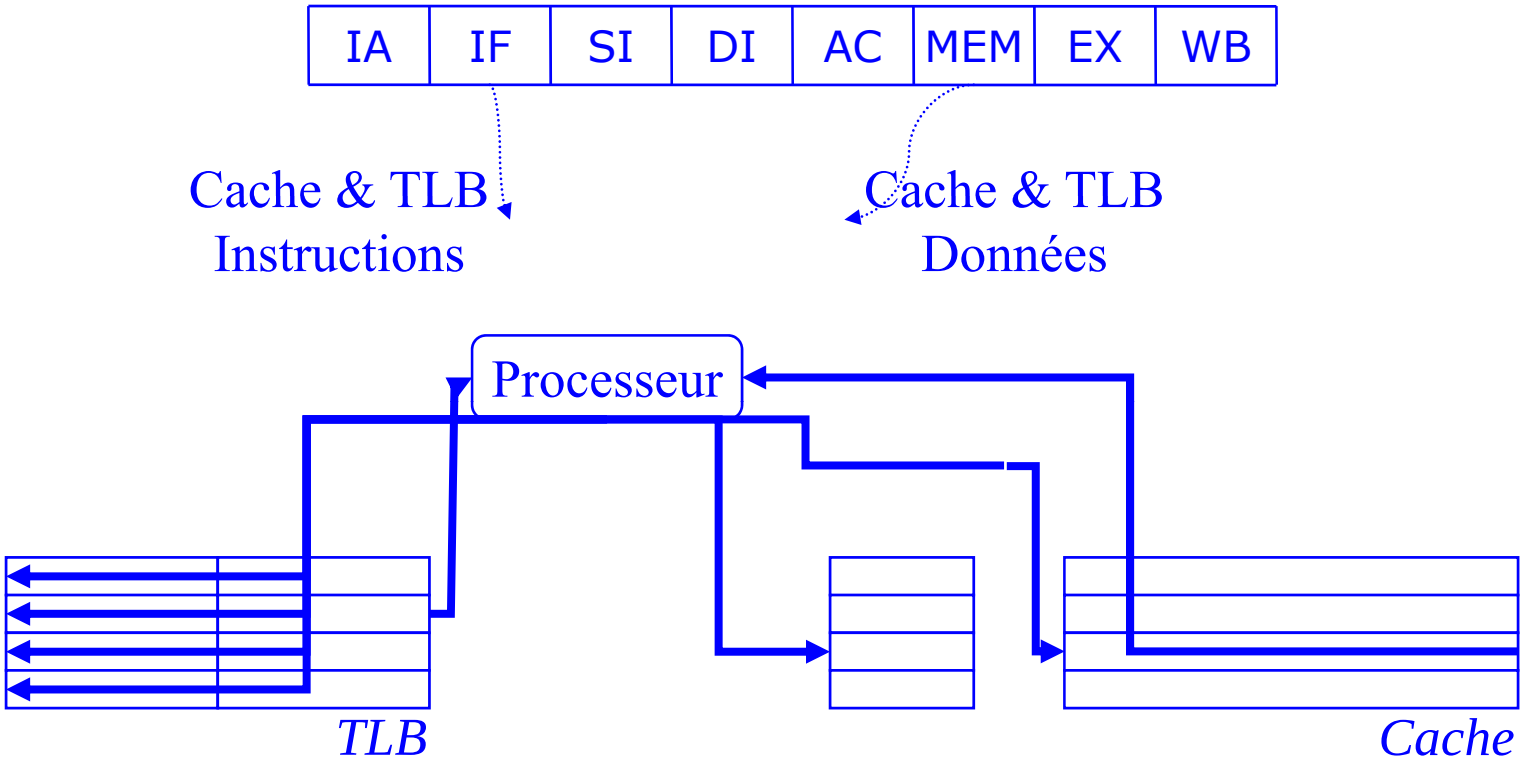


Mémoire Virtuelle / Mémoire Physique: le TLB

- Le processeur utilise des adresses *virtuelles*.
- Les données ont une adresse en mémoire *physique*.
  - ⇒ Il faut faire la traduction adresses virtuelles / adresses physiques
  - ⇒ **TLB** (*Translation Lookaside Buffer*)
- Le TLB est un *cache* de traductions d'adresses.
- Une entrée du TLB correspond à une *page*.
- Souvent le TLB est complètement associatif (*n* = nombre de lignes).



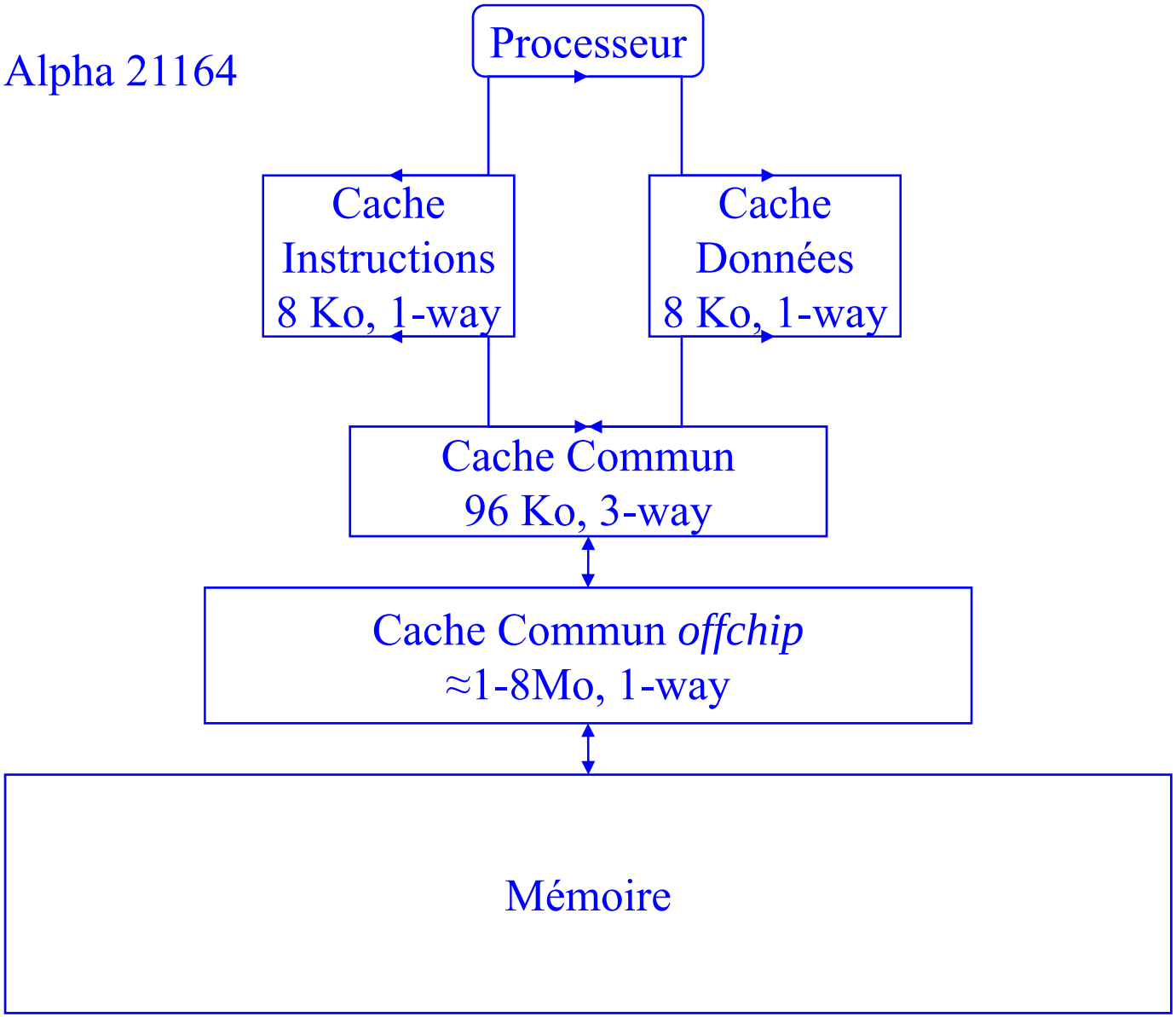
Synthèse



- L'adresse stockée dans le cache est en général l'adresse *physique*.
- Mais le processeur accède au cache avec une adresse *virtuelle*.
- Le processeur accède donc au TLB pour obtenir la traduction et effectue la comparaison ensuite.
- Souvent, on conçoit les caches pour que l'on puisse effectuer *en parallèle* l'accès au cache et au TLB.

Plusieurs Niveaux de Cache

Exemple: Alpha 21164



Impact des Défauts de Cache sur la Performance d'un Processeur

- En moyenne, 1 instruction sur 3 est une instruction mémoire.
- Il y a aussi des défauts de cache instruction.
- Les hiérarchies de cache réduisent la latence moyenne d'accès à la mémoire.

- Processeur à 1GHz, 100ns pour accéder à la mémoire
- 1000 instructions :
  - 50% défauts de cache:
    1000 * (0,67*1 + 0,33*(0,5*1 + 0,5*100)) = 17335 cycles
  - 5% défauts de cache:
    1000 * (0,67*1 + 0,33*(0,95*1 + 0,05*100)) = 2633 cycles
  - 0% défauts de cache:
    1000 cycles

Plan

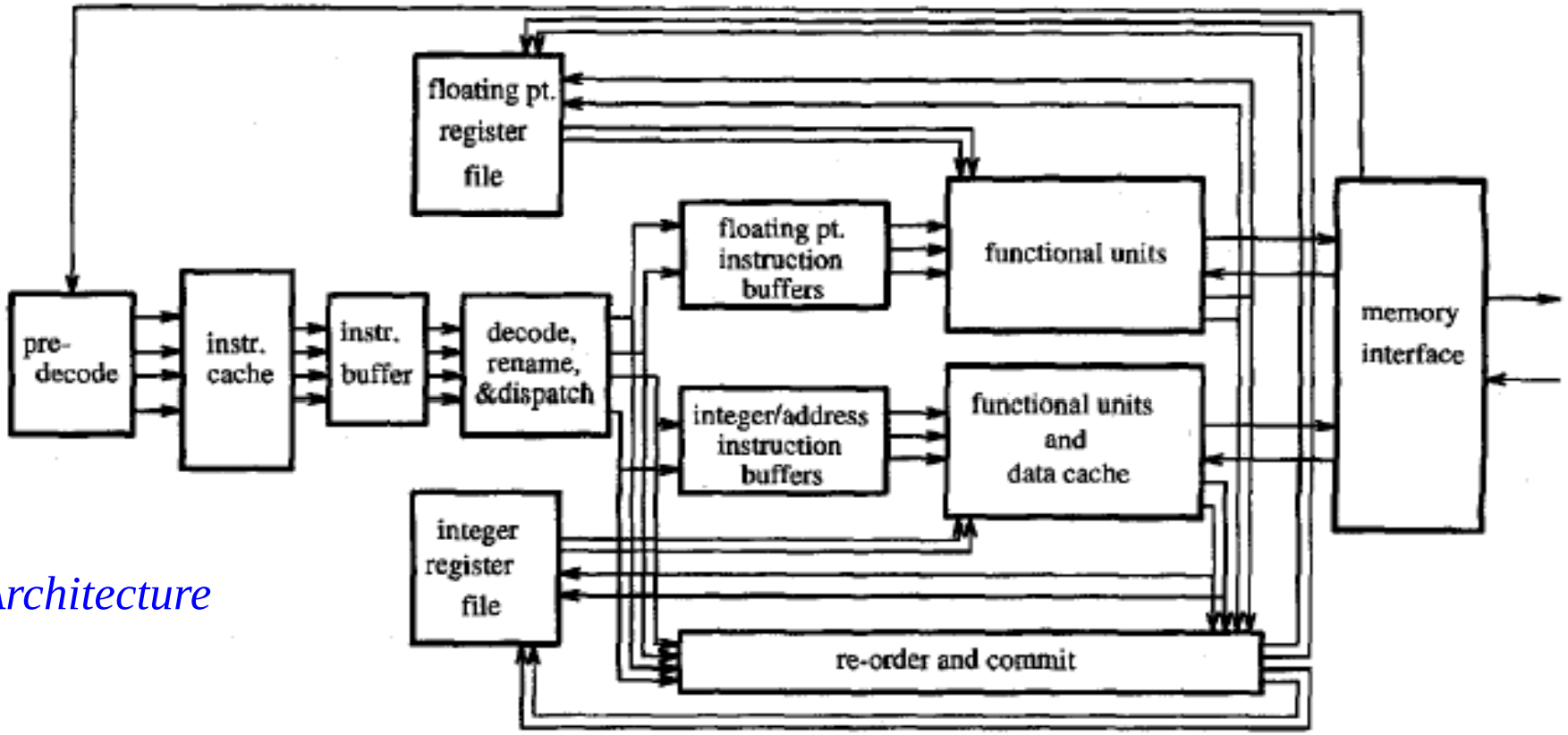
- Processeurs haute-performance
  - Pipeline
  - Prédiction de branchement
  - Cache
  - Exécution superscalaire/parallèle

Processeur Superscalaire

- Pipeline: au plus une instruction terminée par cycle.
- Superscalaire de degré *n*: jusqu'à *n* instructions terminées par cycle (actuellement $n \approx 4$).
- Pour réaliser un processeur superscalaire, il faut:
  - Assurer un flux d'instructions suffisant.
  - Déterminer quelles instructions peuvent s'exécuter en parallèle.
  - Passer les données entre les instructions (le résultat d'une instruction *i* est l'opérande d'une instruction *j*).
  - Disposer de plusieurs unités de calcul en parallèle.
- Contrainte: maintenir des interruptions précises.
- L'architecture des différents processeurs haute-performance est très similaire.

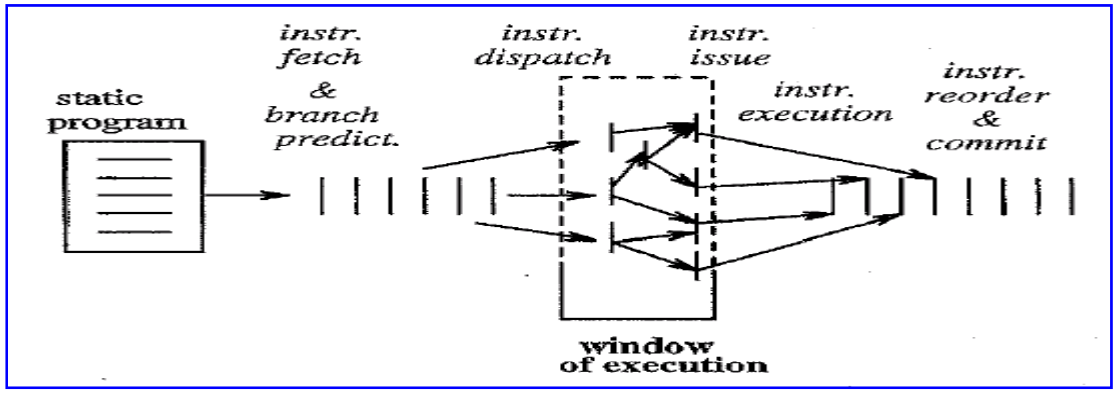
Structure d'un Processeur Superscalaire

Architecture



Source: Proceedings of the IEEE

Pipeline



Parallélisme entre Instructions (*ILP*)
(parallélisme à grain fin)

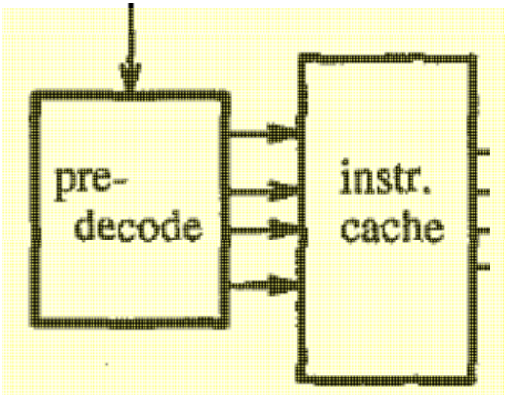
```
for (i=0; i<last; i++) {
  if (a[i] > a[i+1]) {
    temp = a[i];
    a[i] = a[i+1];
    a[i+1] = temp;
    change++;
  }
}
```

```
L2:
    move    r3,r7       #r3->a[i]
    lw      r8,(r3)     #load a[i]
    add     r3,r3,4     #r3->a[i+1]
    lw      r9,(r3)     #load a[i+1]
    ble     r8,r9,L3    #branch a[i]>a[i+1]

    move    r3,r7       #r3->a[i]
    sw      r9,(r3)     #store a[i]
    add     r3,r3,4     #r3->a[i+1]
    sw      r8,(r3)     #store a[i+1]
    add     r5,r5,1     #change++
L3:
    add     r6,r6,1     #i++
    add     r7,r7,4     #r4->a[i]
    blt     r6,r4,L2    #branch i<last
```

Chargement d'Instructions

- Il faut pouvoir déterminer rapidement si une instruction est un branchement pour éviter d'ajouter des bulles dans le pipeline:
  - Lors du chargement d'une instruction dans le cache instruction, passage par un circuit de **prédécodage**.
  - Insertion de bits de prédécodage (branchement oui/non, conditionnel oui/non...) dans le cache instructions.



IA LC-2:

BR $I_{15}I_{14}I_{13}I_{12}$=0000

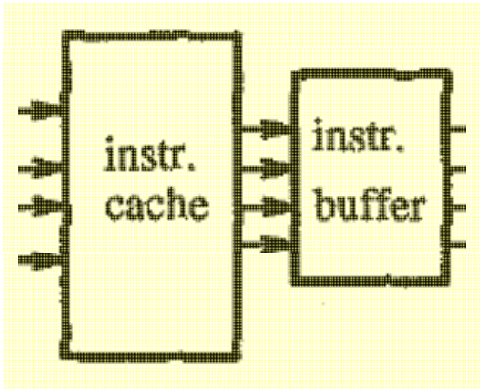
JSR $I_{15}I_{14}I_{13}I_{12}$=0100

JSRR $I_{15}I_{14}I_{13}I_{12}$=1100

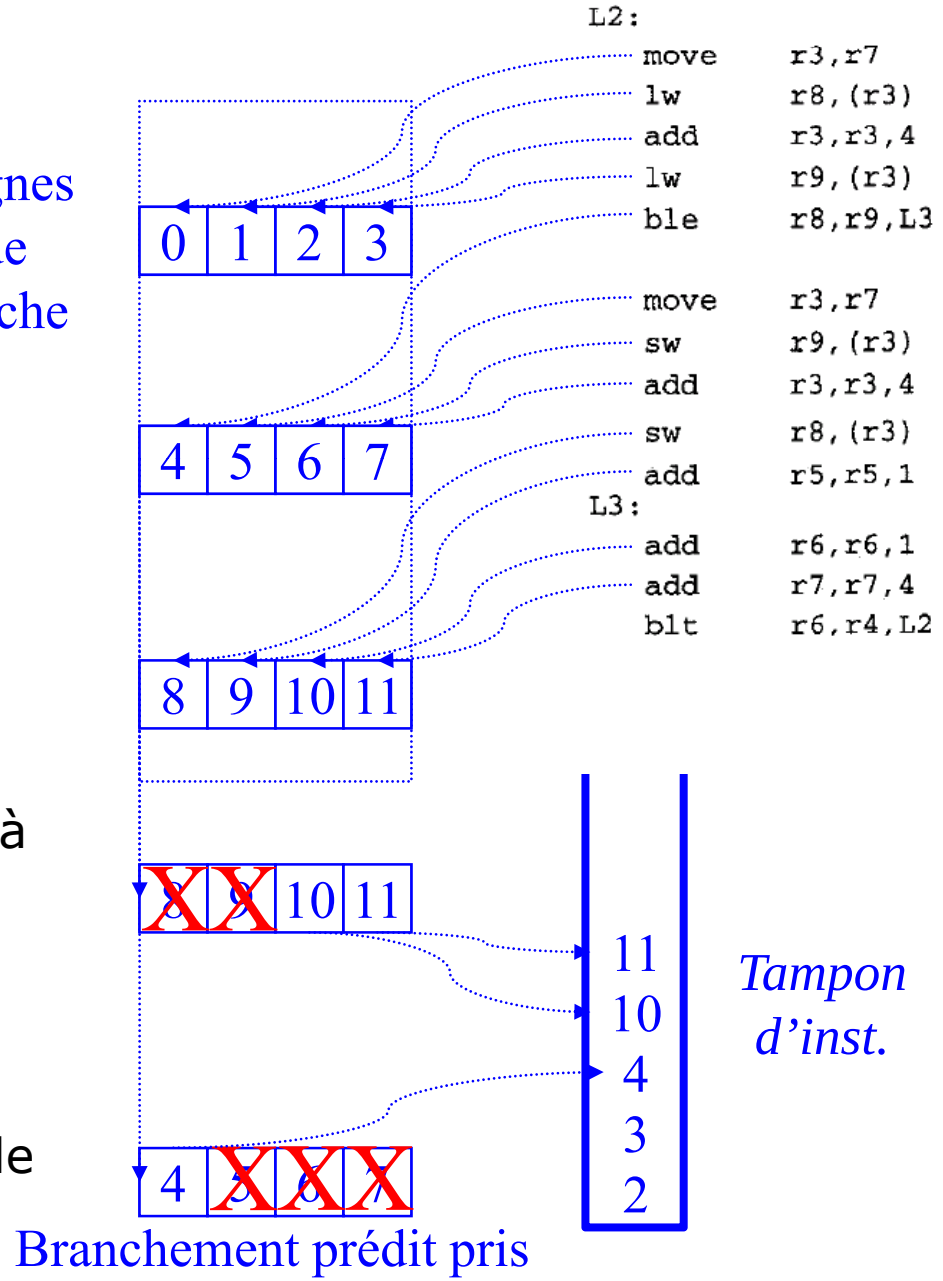
Bit de prédécodage ***isBranch***:

$I_{14} + I_{15}'$

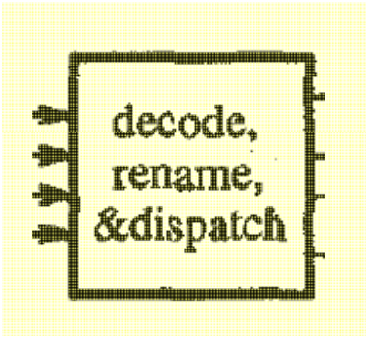
Chargement d'Instructions



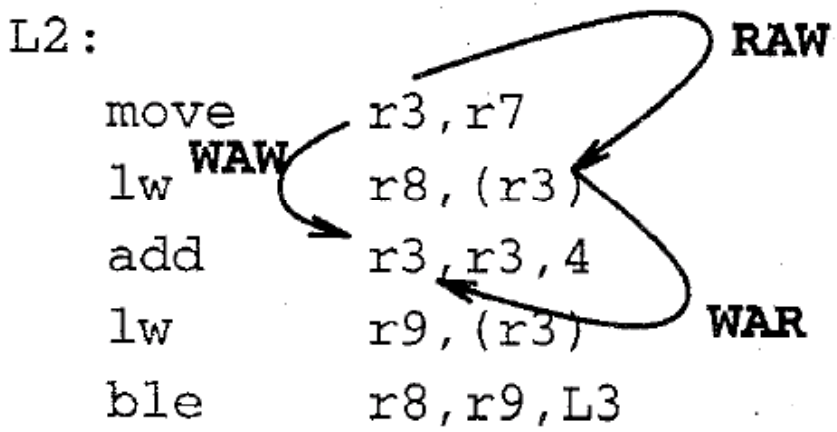
- Interruptions du flux d'instructions:
  - branchements
  - défauts de cache instruction
- Eviter l'interruption du flux d'instructions:
  - Nombre d'instructions chargées à chaque cycle ≥ 4: une ou plusieurs lignes du cache instruction (cache multi-port)
  - Tampon (*buffer*) pour conserver les instructions chargées par anticipation et éviter les bulles de pipeline



Décodage, Dépendances et Mise en Attente



- Les instructions sont sorties du tampon d'instructions.
- On détermine les dépendances de données entre instructions.
- On élimine les «fausses» dépendances liées aux alias de registres.



- RAW (*Read After Write*): vraie dépendance.
- WAW (*Write After Write*): risque d'écriture dans le désordre (fausse dépendance).
- WAR (*Write After Read*): risque d'écriture avant lecture, i.e., avant qu'une donnée ait été utilisée (fausse dépendance).

Alias de Registres: Renommage

- Le maintien de la compatibilité binaire empêche l'augmentation du nombre de registres.
- Nombre de zones de stockage *physiques* > nombre de registres *logiques* (ceux définis par le jeu d'instructions).
- Zones de stockage physiques = banc de registres et **tampon de réordonnancement** (*ReOrder Buffer* ou *ROB*).
- A chaque instruction est associée une entrée dans le ROB; la *valeur* produite par une instruction est stockée dans le ROB; le registre logique de destination est remplacé par le nom d'une entrée dans le ROB: on *renomme* le registre.
- Une table indique où se trouve la valeur courante d'un registre logique (dans le banc de registres ou dans le ROB); lors du chargement d'une instruction, ses opérandes peuvent être dans le ROB ou le banc de registres.

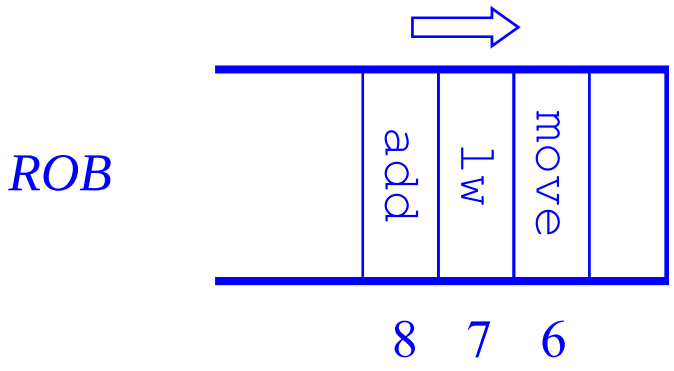
⇒ Elimination des alias de registres.

⇒ Détermination des vraies dépendances.

```
L2:
    move   r3, r7
    lw     r8, (r3)
    add    r3, r3, 4
```

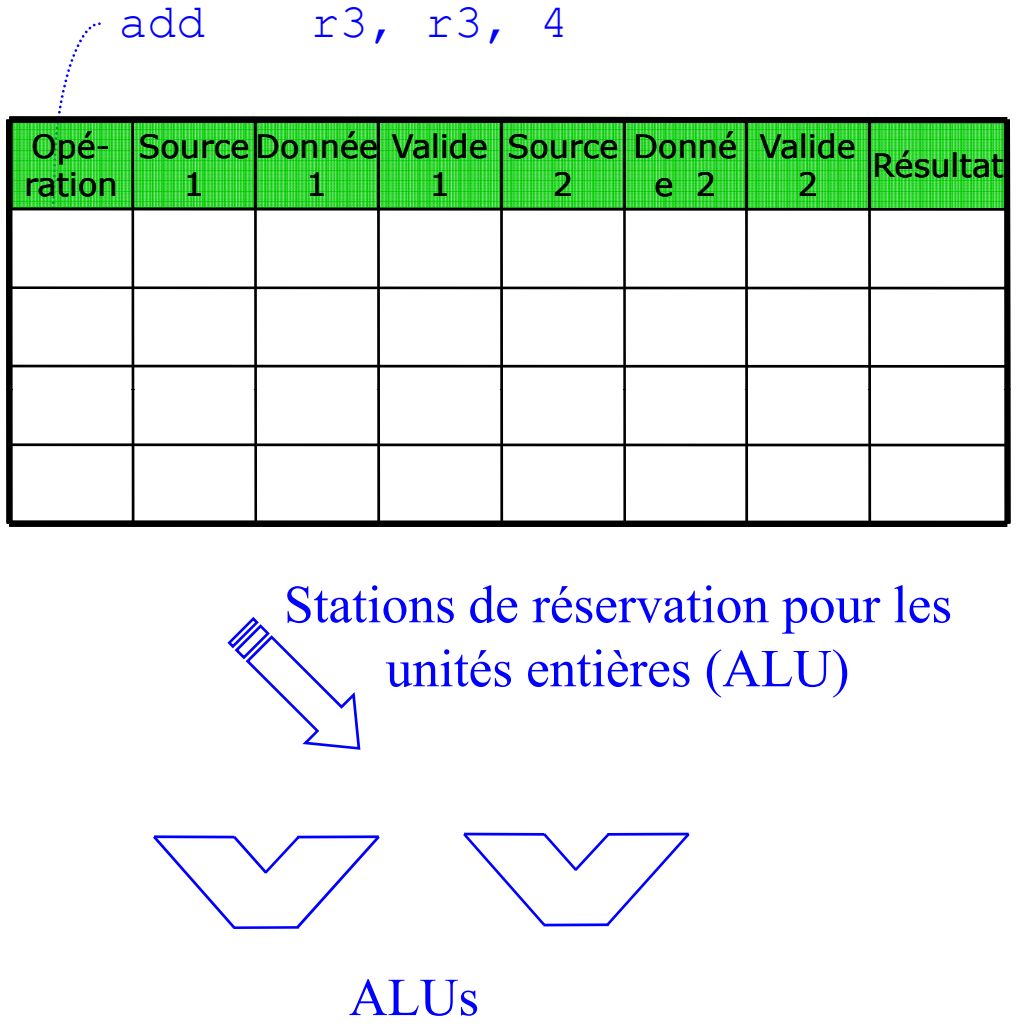
Registre logique	Zone de stockage physique
r_3	ROB_6
...	...
r_8	ROB_7

Registre physique	Valeur
r_3	produite par add
...	...
r_8	produite par lw



Mise en Attente (*Dispatch*)

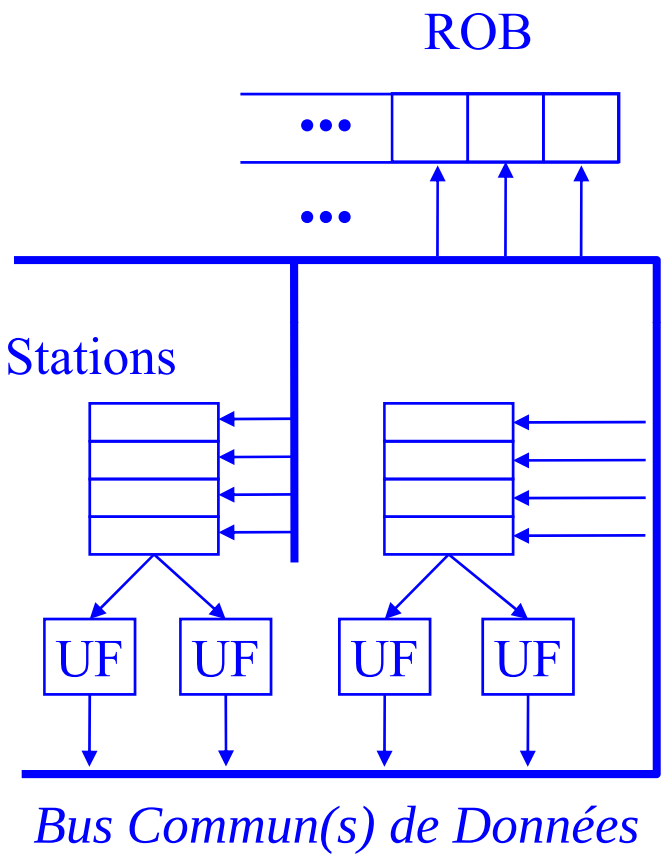
- Après résolution des dépendances, envoi des instructions pour exécution (*Dispatch*).
- Stations de réservation: pour chaque unité fonctionnelle (ou groupe d'unités), une file d'attente des instructions.
- Une instruction est exécutée si:
  - toutes ses opérandes sont disponibles
  - une unité fonctionnelle est disponible
- Algorithme de Tomasulo



Envoi aux Unités Fonctionnelles (*Issue*)

- Lorsqu'une instruction peut s'exécuter, elle est envoyée à l'unité fonctionnelle.
- Après exécution, le résultat est propagée par un (des) bus:
  - à la zone de stockage de destination (entrée du ROB)
  - à toutes les stations de réservation
- Les instructions qui attendaient ce résultat peuvent être exécutées immédiatement.

$\forall$ $\Rightarrow$ Modèle interne plus proche du *dataflow* que de *von Neumann*: les instructions s'exécutent au fur et à mesure que leurs données sont disponibles, et non selon l'ordre du programme.



Accès Mémoire

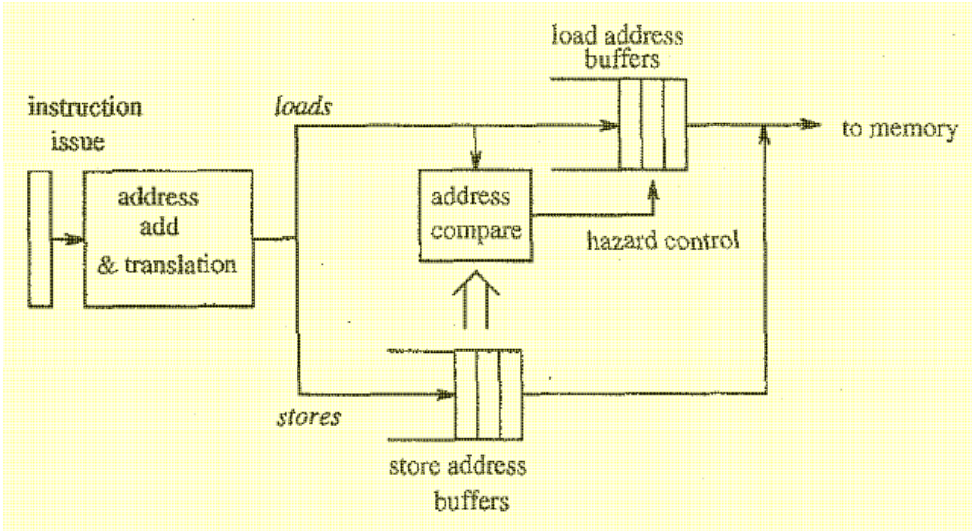
- Calcul d'adresse dans ALU ou unités fonctionnelles dédiées.
- Requête cache+TLB.
- Contraintes:
  - Plusieurs requêtes mémoire par cycle: caches multiports.
  - Garantir une exécution correcte: dépassements `load/store`.
  - `stores` dans l'ordre; exécution spéculative des `loads`

```
sw      r1, 12(r2)    ; 12+r2=100
...
lw      r4, 20(r5)    ; 20+r5=100
```

Cycle	Action
10	sw est dispatché; attend r2
11	lw est dispatché; opérandes prêts
12	Calcul d'adresse pour lw
13	Accès mémoire pour lw **r4 ne contient pas la bonne valeur**
14	r2 est prêt; calcul d'adresse pour sw
15	Accès mémoire pour sw

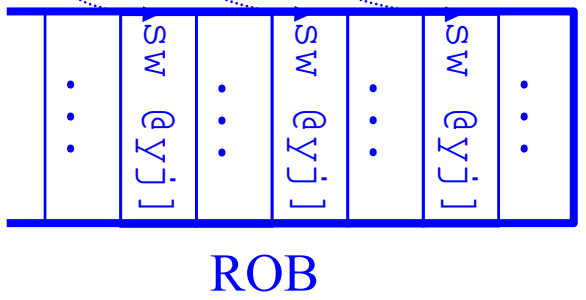
Accès Mémoire

- Lors d'un accès mémoire en lecture, on vérifie qu'il n'existe pas une écriture à la même adresse en cours.
- Si l'adresse de certaines écritures n'est pas connue: attente ou spéculation.
- On peut effectuer la même vérification en écriture (plus efficace mais pas indispensable).




```
for (i=0; i < n; i++) {
  for (j=0; j < n; j++) {
    y[j] += a[i][j]*x[j];
  }
}
```

```
...
sw @y[j]
...
sw @y[j]
...
sw @y[j]
...
```



Accès Mémoire et Exécution dans le Désordre

- L'exécution des instructions dans le désordre et en parallèle permet de masquer partiellement la latence en cas de défaut de cache.

```
lw      r1, 12(r2)
add     r2, r1, 6
inst3
lw      r3, 20(r5)
inst5
add     r4, r1, r3
inst6
```

Cycle	Action
10	lw r1 : *miss*
11	add r2, r1, 6 : bloqué
12	inst3
13	lw r3 : *hit*
14	inst5
15	lw r1 : terminé add r4, r1, r3 add r2, r1, 6 : terminé
16	inst6

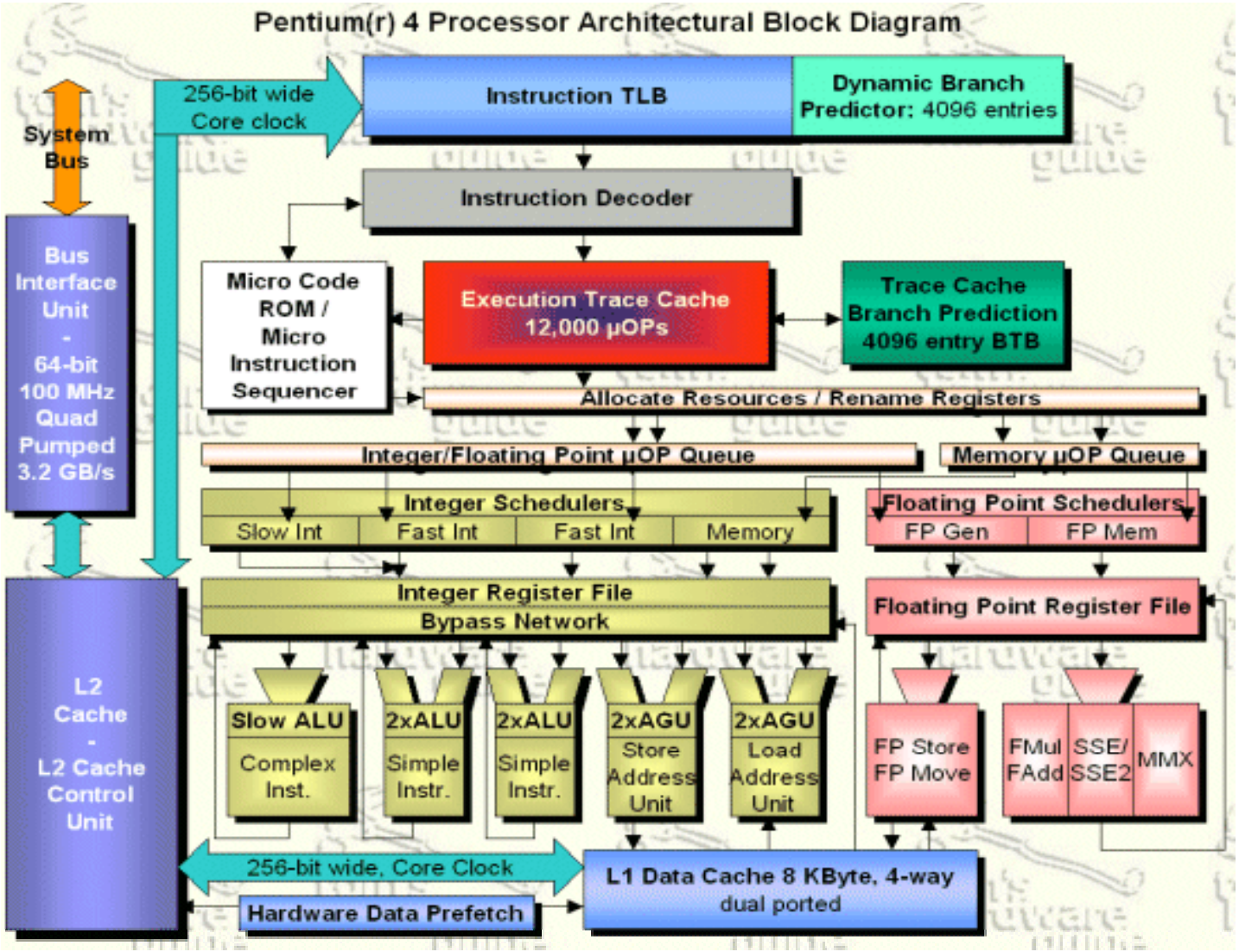
Latence mémoire: 5 cycles

Aucun délai dû au miss

Finalisation (*Commit*)

- Une instruction effectue l'étape de finalisation lorsqu'elle est en tête du ROB
  - ⇒ Les instructions effectuent l'étape de finalisation dans l'ordre du programme.
- L'état *logique* de l'architecture n'est modifié que lors de l'étape de finalisation (*commit*)
  - ⇒ il est possible d'avoir des interruptions précises malgré l'exécution dans le désordre (le ROB contient le vecteur d'exception de chaque instruction).
- Etat logique: registres et mémoire.
- Lorsqu'une instruction sort du ROB:
  - le résultat est écrit dans le registre correspondant (toutes instructions sauf écriture en mémoire)
  - OU la donnée est envoyée à la mémoire (écriture en mémoire)
- Processeur superscalaire de degré *n*: *n* instructions peuvent effectuer simultanément l'étape de finalisation.
- Si l'instruction en tête du ROB n'a pas terminé son exécution: le processeur est bloqué.

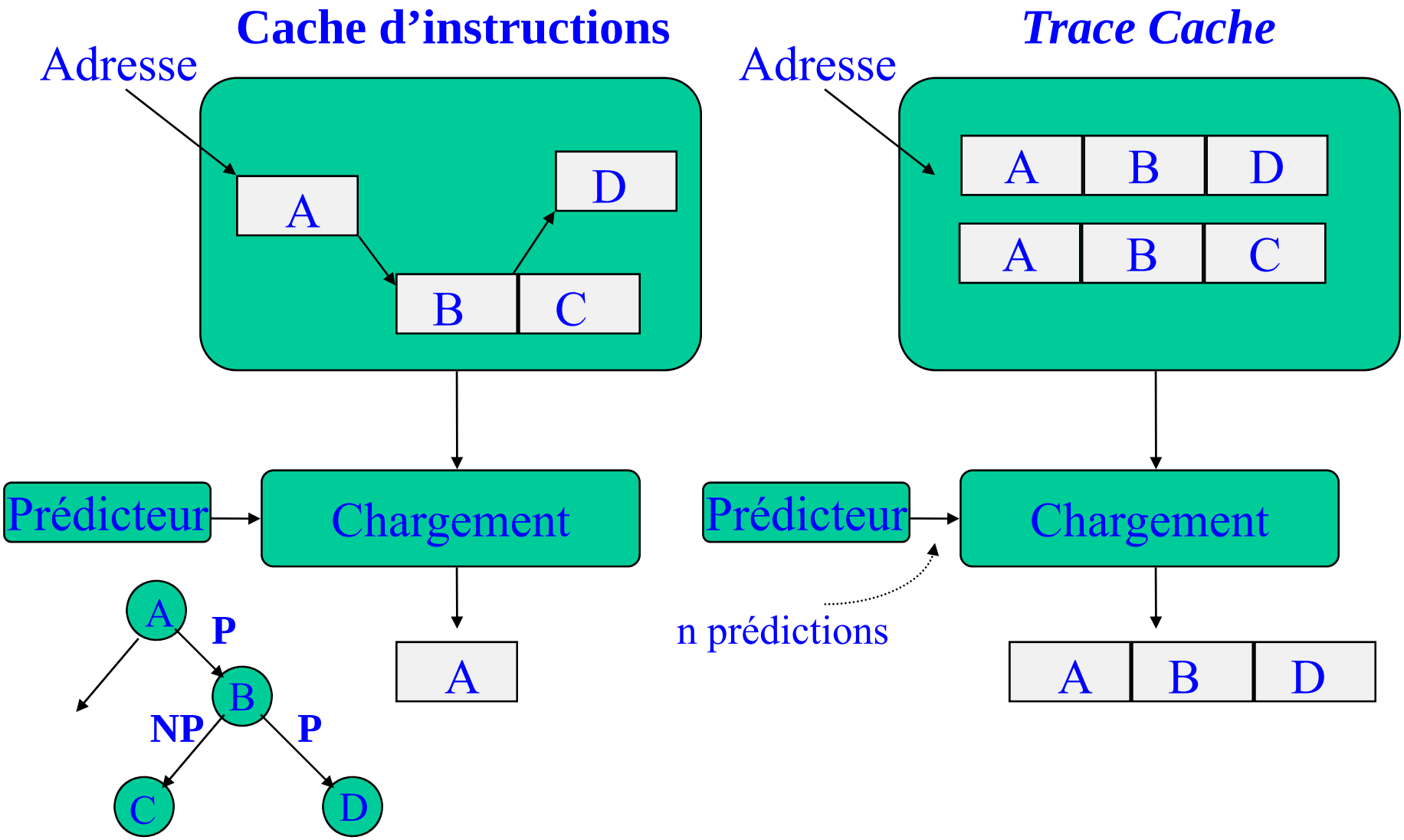
Pentium IV



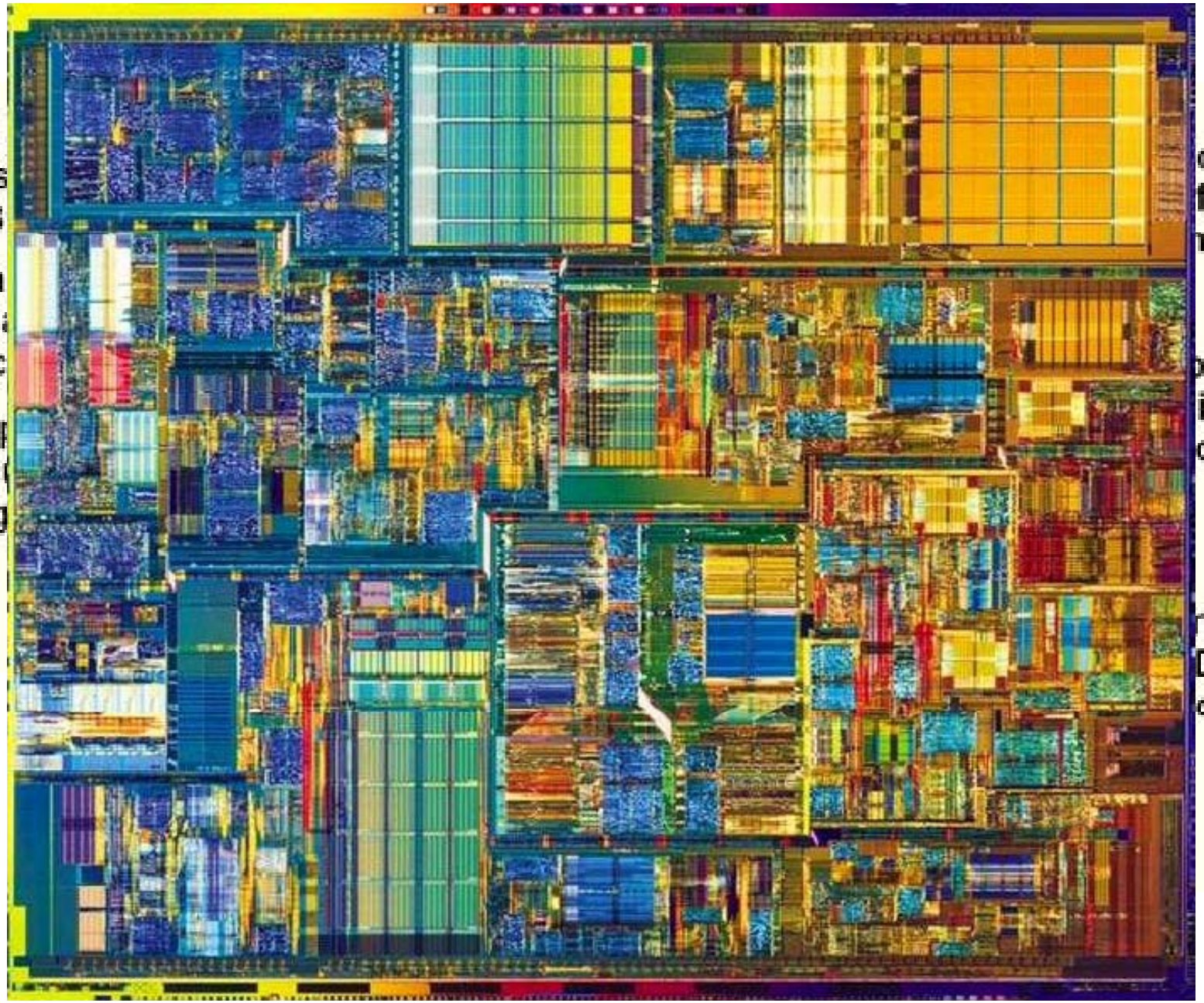
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC	Nxt IP	TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

Source: Tom's Hardware

Principe du *Trace Cache*

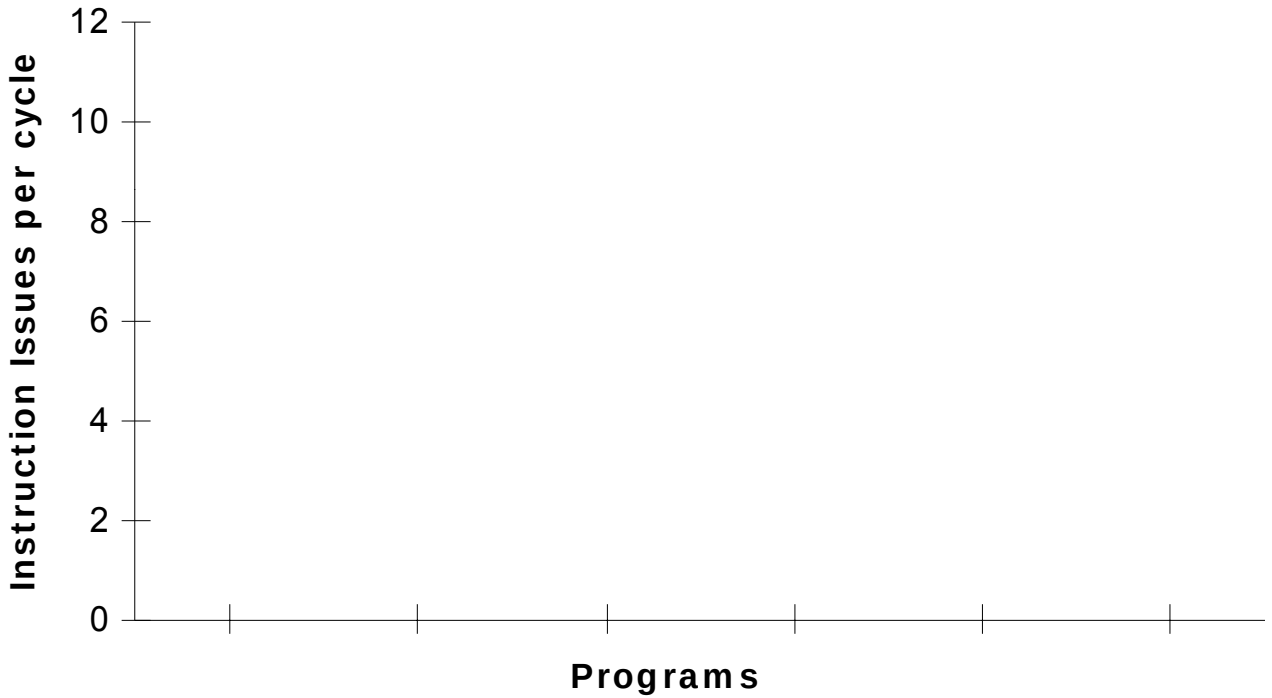


Pentium IV

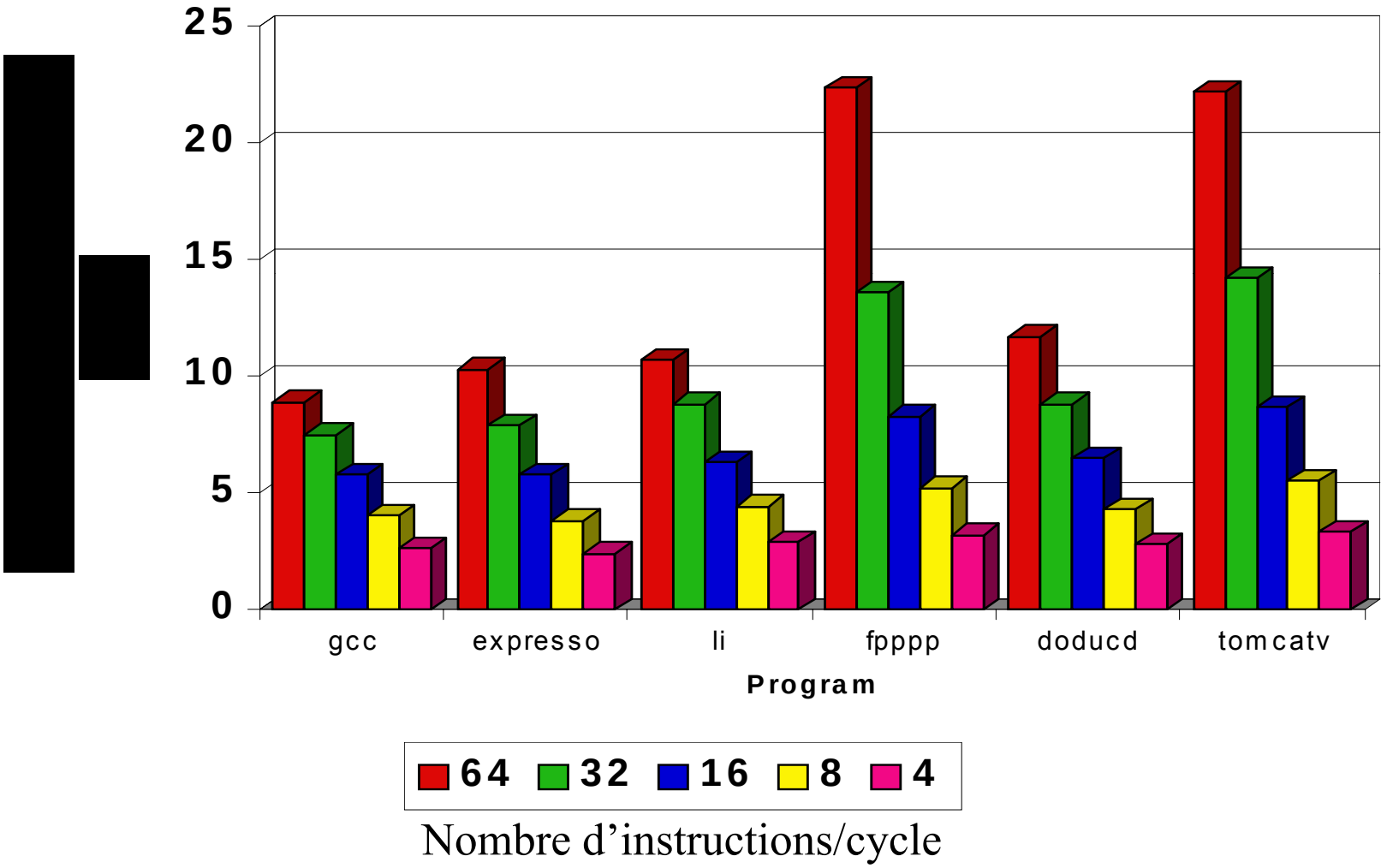


Source: Tom's Hardware

Idéalement...

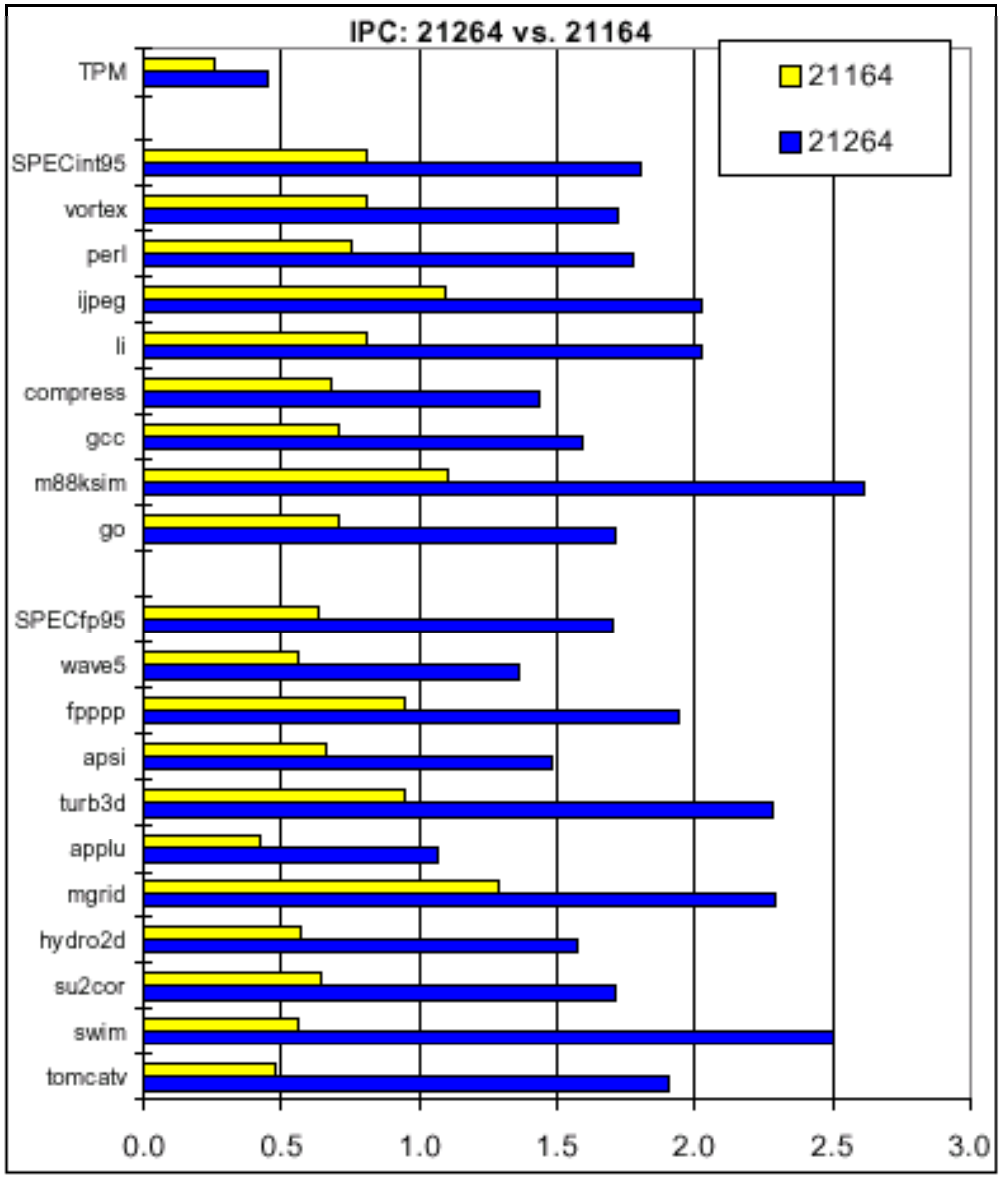


En Prenant en Compte les Principales Contraintes...

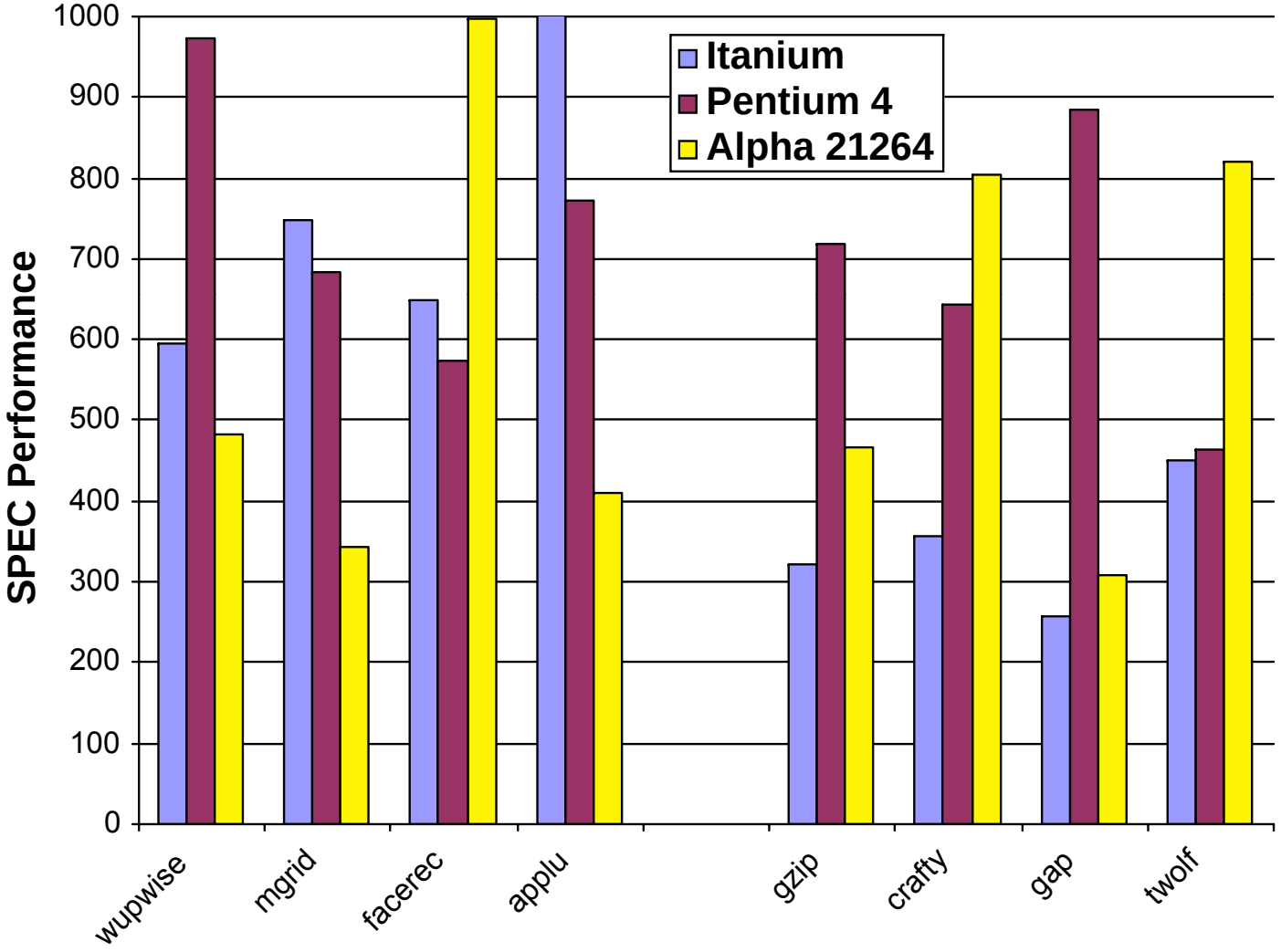


En Réalité...

- 4 instructions/cycle
- Au maximum, 2,6 instructions par cycle; 2 en moyenne (IPC = *Instruction Per Cycle*).



Performance Relative des Principales Architectures



Processor	Alpha 21364	AMD Opteron	HP PA-8700	IBM Power 4+	Intel Itanium 2	Intel XeonMP	Intel Xeon	MIPS R14000	Sun UltraSPARC III
System or Motherboard	Alpha GS1280/7	Rioworks HDAMA	HP9000 C3750	pSeries 650 6M2	HP RX2600	Dell PwrEdg 6650	Dell Prec. 350	SGI 3200	Sun Fire 280R
Clock Rate	1.15GHz	2.0GHz	870MHz	1.7GHz	1.5GHz	2.8GHz	3.06GHz	600MHz	1.20GHz
External Cache	None	None	None	128MB	None	None	None	8MB	8MB
164.gzip	583	1,093	588	799	973	1,089	1,138	322	474
175.vpr	822	1,060	688	1,053	1,119	959	606	572	544
176.gcc	859	1,270	906	1,141	1,554	1,452	1,236	445	674
181.mcf	712	878	494	1,804	2,305	882	773	783	684
186.crafty	982	1,372	751	944	1,221	1,096	1,179	502	660
197.parser	514	1,223	495	453	983	1,125	1,025	409	565
252.eon	958	1,447	592	1,344	1,547	1,236	1,387	507	762
253.perlbmk	768	1,314	619	828	1,210	1,262	1,381	367	636
254.gap	636	1,397	339	1,115	1,074	1,416	1,417	308	462
255.vortex	1,094	1,934	1,196	1,663	1,944	1,794	1,658	679	992
256.bzip2	824	1,040	534	1,130	1,217	1,024	856	493	691
300.twolf	1,018	1,230	911	1,408	1,280	1,383	900	645	652
SPECint_base2000	795	1,248	642	1,077	1,322	1,201	1,085	483	637
168.wupside	883	1,308	446	2,220	1,466	1,287	1,406	434	888
171.swim	3,590	1,733	931	2,140	4,136	1,308	1,837	529	1,023
172.mgrid	708	1,105	621	1,041	2,503	1,025	1,047	379	700
173.applu	1,518	1,033	702	1,393	3,469	845	1,168	381	741
177.mesa	928	1,407	694	856	1,126	1,137	1,165	425	693
178.galgel	2,105	1,782	1,603	3,794	4,225	2,291	1,536	1,398	1,977
179.art	2,014	1,434	670	2,314	6,514	1,489	716	1,436	9,951
183.equake	519	1,037	413	2,950	2,982	1,024	1,291	347	1,537
187.facerec	1,105	1,425	430	1,928	1,925	1,253	1,315	647	1,172
188.ammp	735	1,200	553	1,046	1,413	977	644	573	592
189.lucas	1,522	1,358	448	1,834	1,774	1,153	1,522	442	520
191.fma3d	1,019	1,192	404	1,290	1,091	939	1,089	306	489
200.sixtrack	469	489	471	724	1,402	525	564	298	406
301.apsi	1,242	1,085	696	1,345	1,024	979	833	406	644
SPECfp_base2000	1,124	1,209	600	1,598	2,119	1,103	1,092	499	945

Processor	Alpha 21364	AMD Opteron	HP PA-8700	IBM Power4+	Intel Itanium 2	Intel XeonMP	Intel Xeon	MIPS R14000	Sun Ultra-III
Clock Rate	1.15GHz	2.0GHz	870MHz	1.7GHz	1.5GHz	2.8GHz	3.06GHz	600MHz	1.2GHz
Cache (I/D/L2/L3)	64K/64K/ 1.75M	64K/64K/ 1M	750K/ 1.5M	64K/32K/ 1.5MB	16K/16K/ 256K/6M	12K/8K/ 512K/2M	12K/8K/ 512K	32K/32K	32K/64K
Issue Rate	4 issue	3 x86 instr	4 issue	8 issue	8 Issue	3 ROPs	3 ROPs	4 issue	4 issue
Pipeline Stages	7/9 stages	9/11 stages	7/9 stages	12/17 stages	8 stages	22/24 stages	22/24 stages	6 stages	14/15 stages
Out of Order	80 instr	72ROPs	56 instr	200 instr	None	126 ROPs	126 ROPs	48 instr	None
Rename Regs	48/41	36/36	56 total	48/40	328 total	128 total	128 total	32/32	None
BHT Entries	4K x 9-bit	4K x 2-bit	2K x 2-bit	3x16K x1-bit	512 x 2-bit	4K x 2-bit	4K x 2-bit	2K x 2-bit	16K x 2-bit
TLB Entries	128/128	280/288	240 unified	1,024 unified	32L1I/32L1D/ 256L2D	128I/64D	128I/64D	64 unified	128I/512D
Memory B/W	12GB/s	5.4GB/s	1.54GB/s	12.8GB/s	6.4GB/s	3.2GB/s	4.3GB/s	1.6GB/s	4.8GB/s
Package	FC-LGA1443	PGA-940	LGA-544	MCM	mPGA-700	mPGA-603	PGA-423	FCBGA-1153	FC-LGA 1368
IC Process	0.18µm 7M	0.13µm 6M	0.18µm 7M	0.13µm 7m	0.13µm 6M	0.13µm 6M	0.13µm 6M	0.15µm 7M	0.15µm 7M
Die Size	397mm²	193mm²	304mm²	267mm²**	380mm²*	211mm²	131mm²	142mm²	210mm²
Transistors	135 million	106 million	130 million	184 million**	410 million	160 million*	55 million	7.2 million	29 million
Est Die Cost	$180*	$79*	$96*	$144**	$140*	$64*	$55*	$68*	$72*
Power (Max)	110W*	86*W(MTP)	75W*	100W**	130W	83W(Max)	82W(TDP)	16W*	85W*
Availability	1Q03	3Q03	3Q02	2Q03	3Q03	3Q03	4Q02	2Q02	2Q03

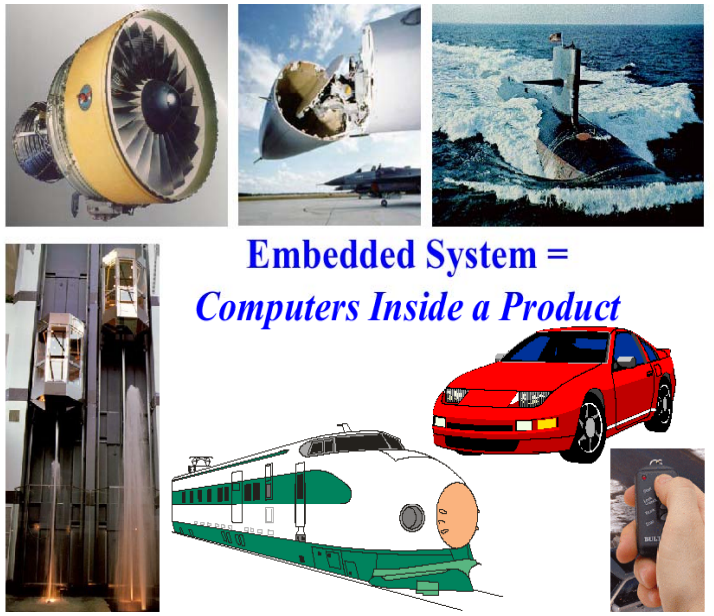
Source: vendors, except *MDR estimates. Estimated manufacturing costs does not include external cache chips. ** Contains two processors on one die.

Définitions

Embedded Processor (EP) : processeur embarqué, processeur enfoui, processeur spécifique.

General-purpose Processor (GP) : processeur généraliste, des PC aux stations haute-performance, dimmensionnement. Voir cours précédents.

Processeur embarqué = Système embarqué = processeurs + applications + OS = indissociables. Système embarqué : système inclus (embarqué) dans un système plus large et son existence comme « ordinateur » n'est pas apparente.



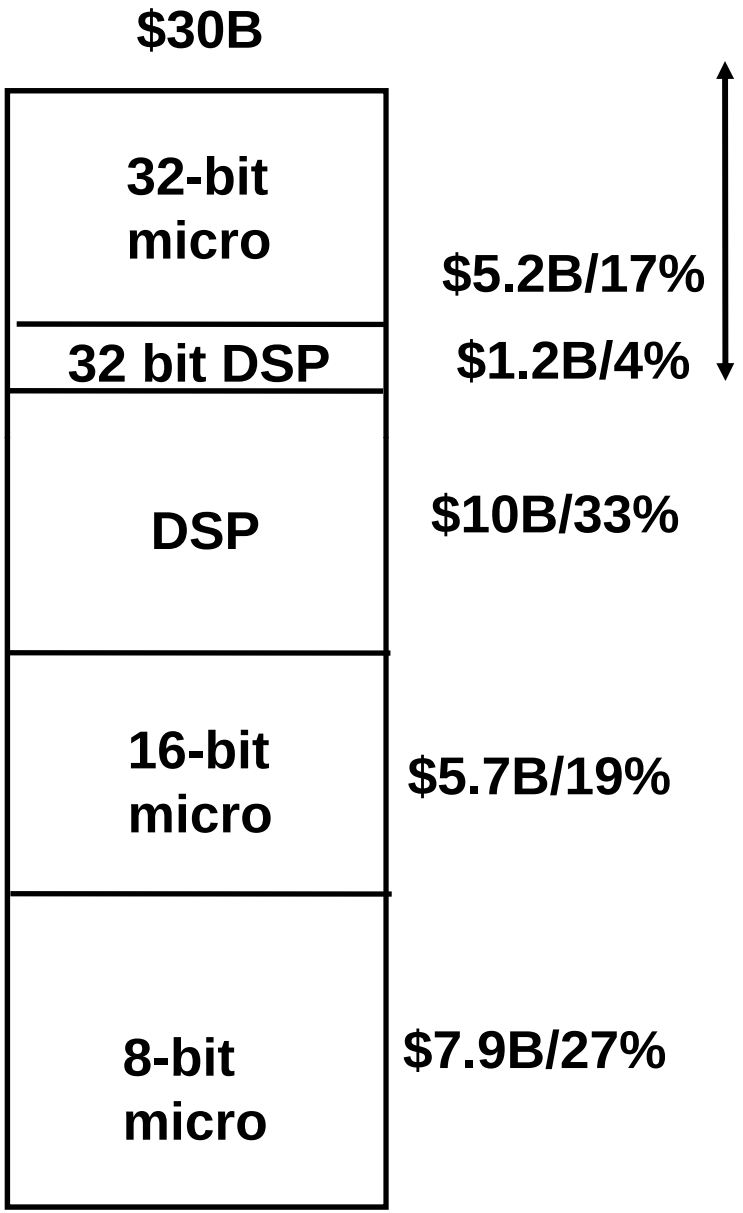
Différences : applications, volume de production (marché) et contraintes d'utilisation.

Similitudes : architectures et applications émergeantes.

Marché

GP représente - **1%** de tous les processeurs vendus chaque année (~ 100 millions de GP).

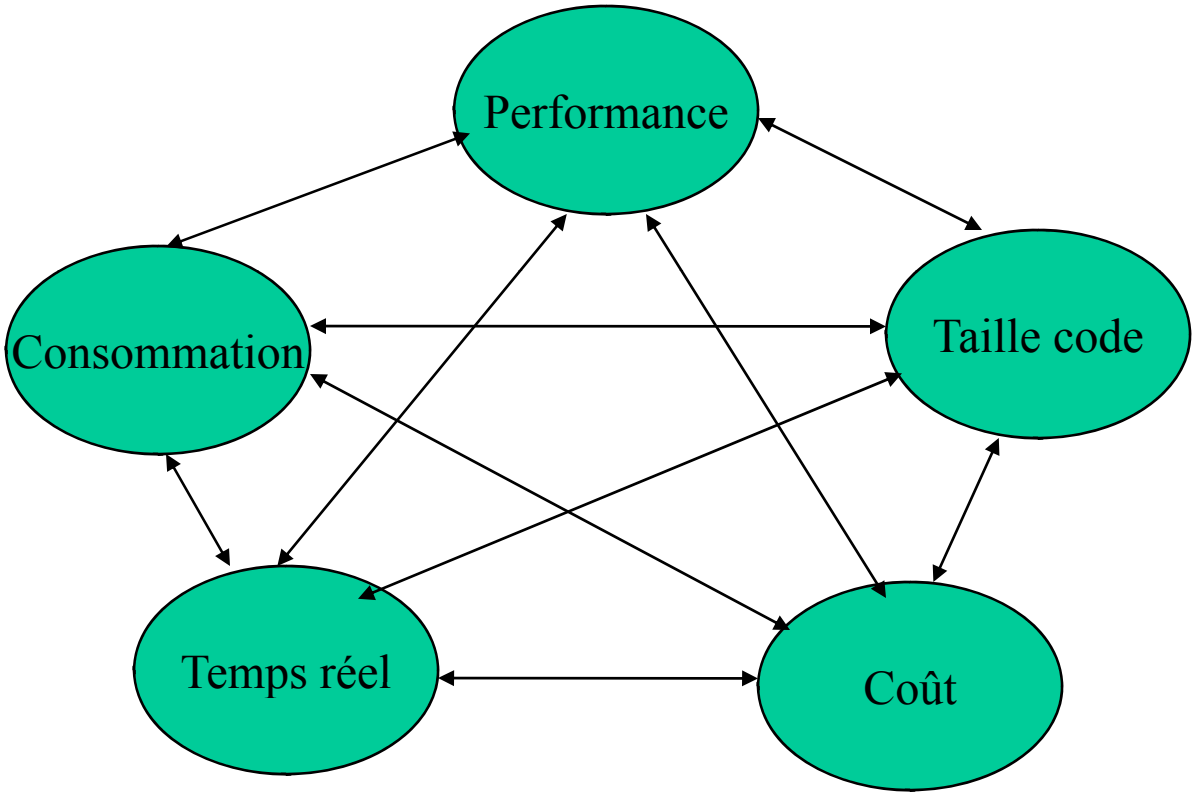
EP représente **99%** des ventes!



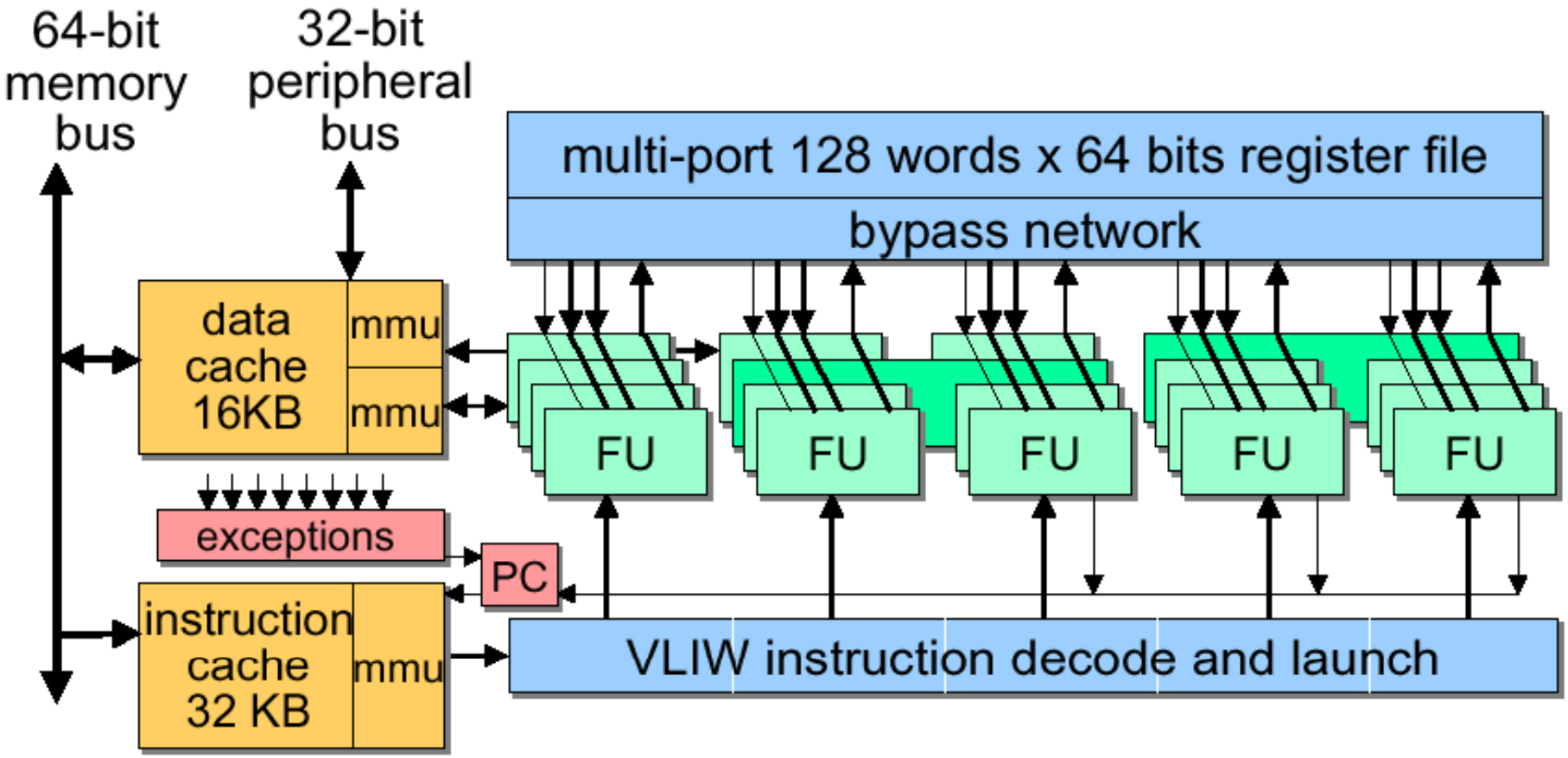
Ominiprésence

- "... *the New York Times has estimated that the average American comes into contact with about 60 microprocessors every day*...." [Camposano, 1996]
- "*Latest top-level BMWs contain over 100 micro-processors*" [Personal communication]

Embarqué = Multiples Contraintes

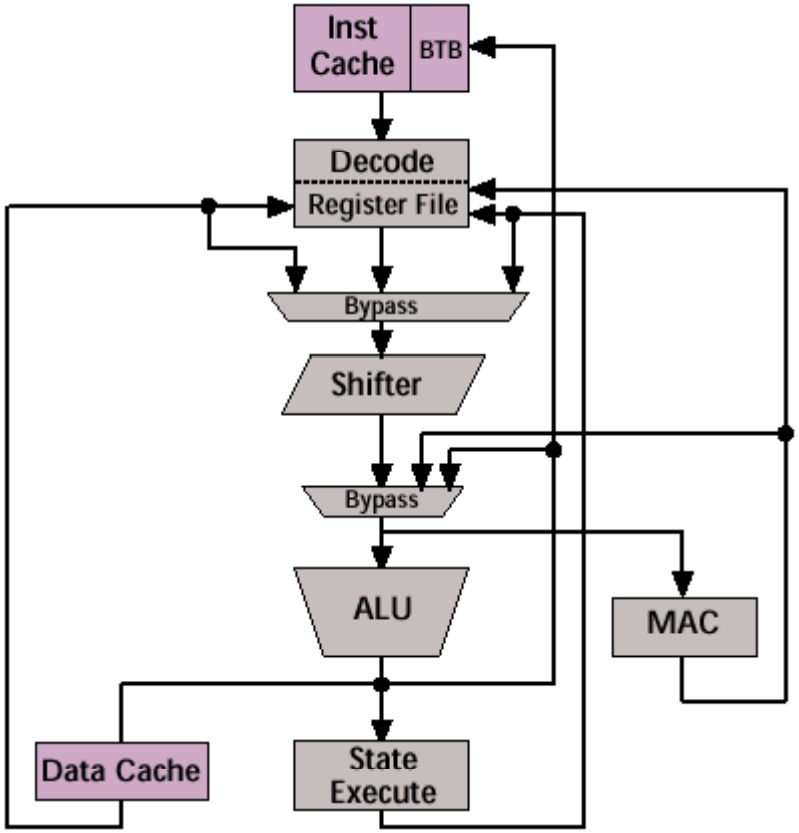


VLIW


64-bit memory bus
32-bit peripheral bus
multi-port 128 words x 64 bits register file
bypass network
data cache 16KB
mmu
mmu
FU
FU
FU
FU
FU
exceptions
PC
instruction cache 32 KB
mmu
VLIW instruction decode and launch

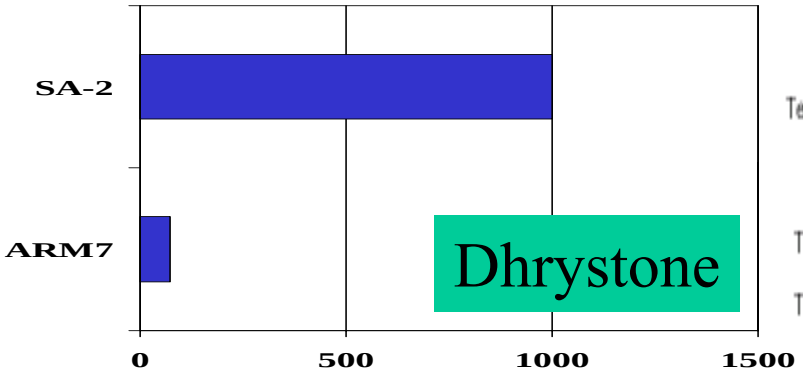
StrongARM2 / XScale

- Cœur classique.
- Très flexible :
  50MHz/10mW/62Mips → 800MHz/900mW/1000Mips.
- Réduction acharnée de la consommation.
- Pipeline long: BTB.
- Caches data/inst.
- *iPaq*, *Palm*.



Principales Différences

Performance



- SA-2 ≈ 14 ARM7
- P266MMX ≈ 8 ARM7
- PIV 1,5 ≈ 8 P266

⇒ PIV 1,5 ≈ 64 ARM7

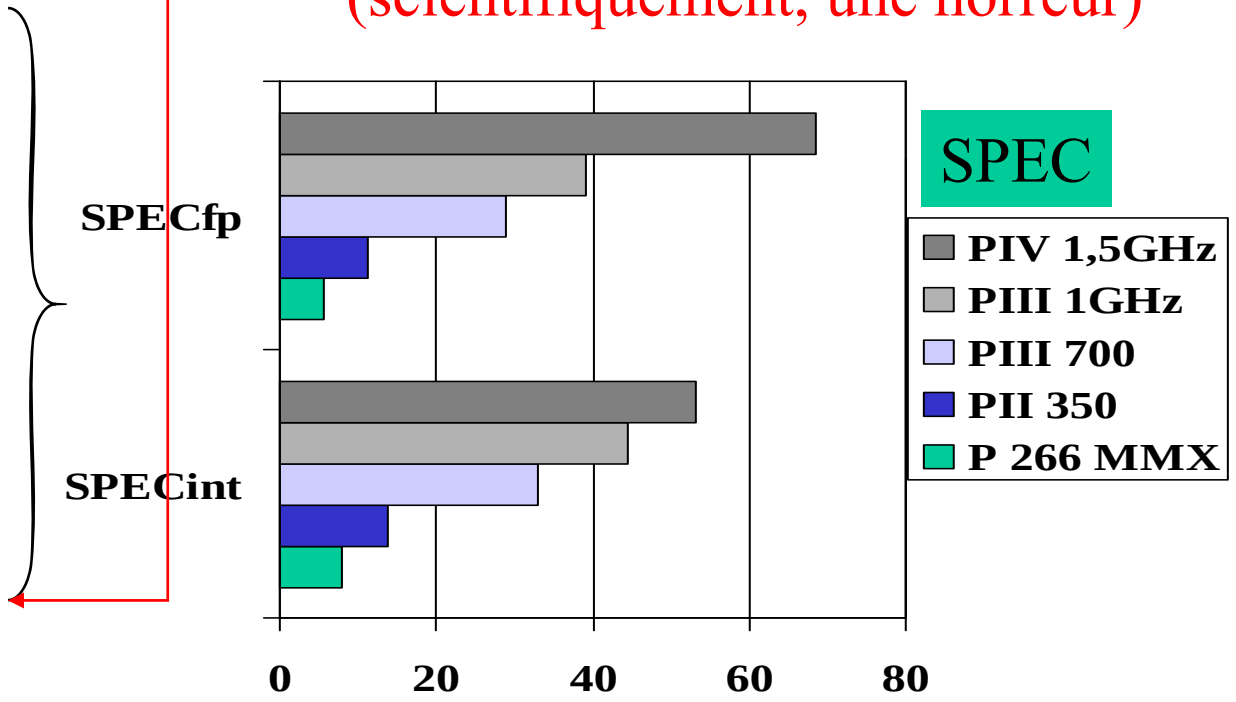
⇒ **PIV 1,5 ≈ 4,5 SA-2**

C62x ≈ 1,5 SA-2

⇒ C64x 1,1 ≈ 4 C62x

⇒ **C64x 1,1 ≥ PIV 1,5**

(scientifiquement, une horreur)



Contraintes

- Performance : essentielle pour les GP (*faster is always better*), mais également importante pour les EP sous certaines contraintes…

- Consommation/dissipation,
- Côut,
- Taille du code,
- Temps réel,
- Fiabilité, sécurité, flexibilité, ...

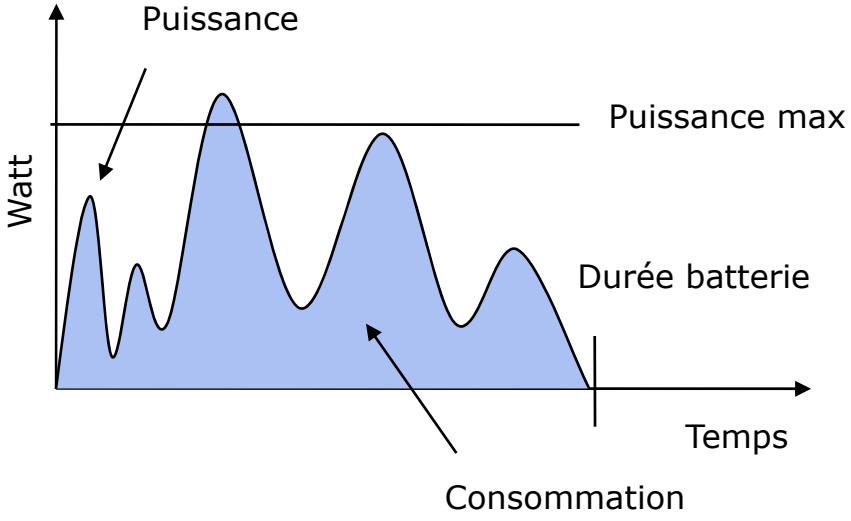
Dissipation et Consommation

- Processeurs généralistes et haute-performance (dissipation) :
  - Augmentation de la fréquence, de la densité d'intégration.
  - Coût du matériel de refroidissement.

- Processeurs embarqués (consommation) :
  - Traitements de plus en plus complexes à des cadences de plus en plus rapides.
  - Durée des batteries.
- + dissipation (idem GP).

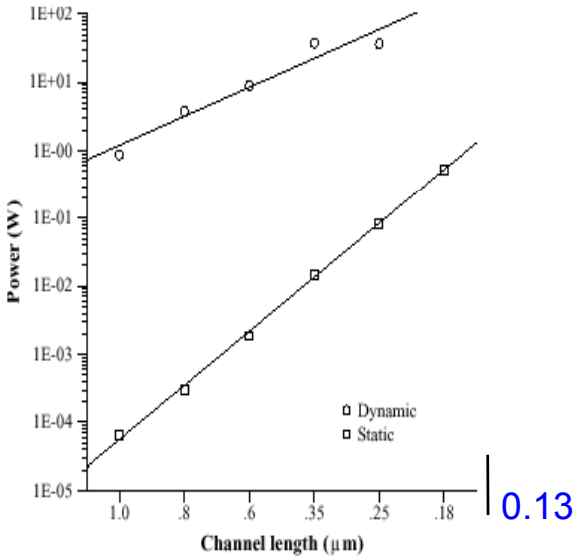
Year	1999	2002	2005	2008	2011	2014
Feature size (nm)	180	130	100	70	50	35
Logic trans/cm^2	6.2M	18M	39M	84M	180M	390M
Cost/trans (mc)	1.735	.580	.255	.110	.049	.022
#pads/chip	1867	2553	3492	4776	6532	8935
Clock (MHz)	1250	2100	3500	6000	10000	16900
Chip size (mm^2)	340	430	520	620	750	900
Wiring levels	6-7	7	7-8	8-9	9	10
Power supply (V)	1.8	1.5	1.2	0.9	0.6	0.5
High-perf pow (W)	90	130	160	170	175	183

SIA RoadMap

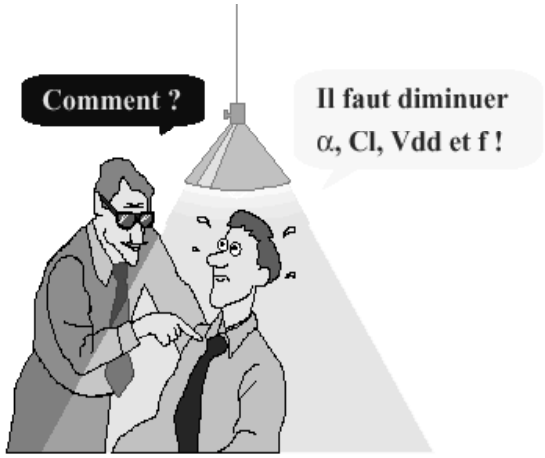


Consommation d'Energie

- Consommation = Puissance x Temps.
- Puissance = P statique + P dynamique (Watt).
  - P statique (courant de fuite) actuellement négligée (-10% consommation totale).
  - P dynamique = ½ x C x a x Fréquence x (Tension)^2 (dépend de l'activité de la porte, c'est à dire, elle dépend des transitions des entrée/sortie).
- *"Plus le transistor est petit et moins il chauffe, mais plus leur nombre par unité de surface est grand".*
- Activité : détermine le nombre de transitions par cycle.
- Puissance dépend de l'activité et des données.

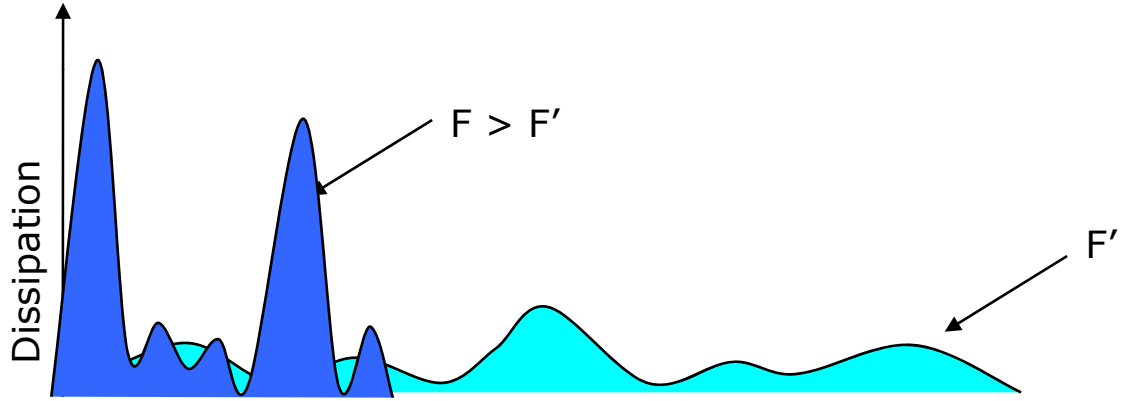


- Consommation importante :
  - Gestion de l'horloge.
  - Gestion des instructions.
  - Accès aux caches.



Réduire la fréquence / le voltage

- Le plus direct (P dynamique = ½ x C x a x Fréquence x $(\text{Tension})^2$):
  - Réduire la fréquence,
  - Réduire le voltage.

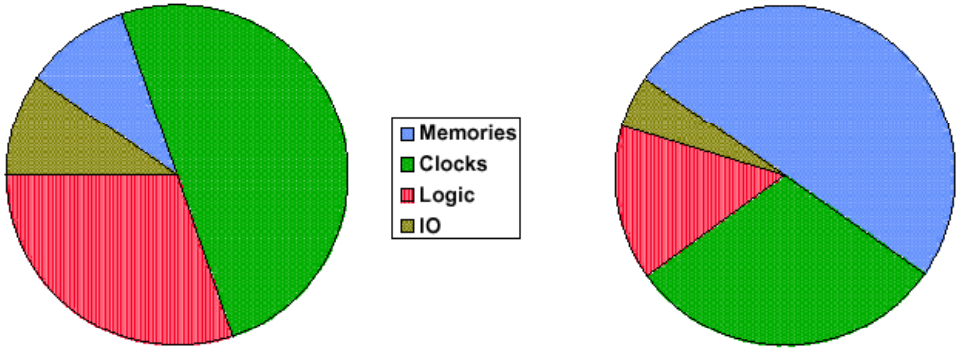


Comparaison généraliste / embarqué

Comparaison performance / consommation (MIPS / Watt)

Processeur	MIPS	Vdd	Pmoy	MIPS/Watt
CoolRisc	14	3V	2.8 mW	5000
TMS C54x	30-200	1.8V	460 mW	1500-3000
TMS C6x	1600	2.5V	2W	800
Dec α	500	1.8-3V	50 W	7-10

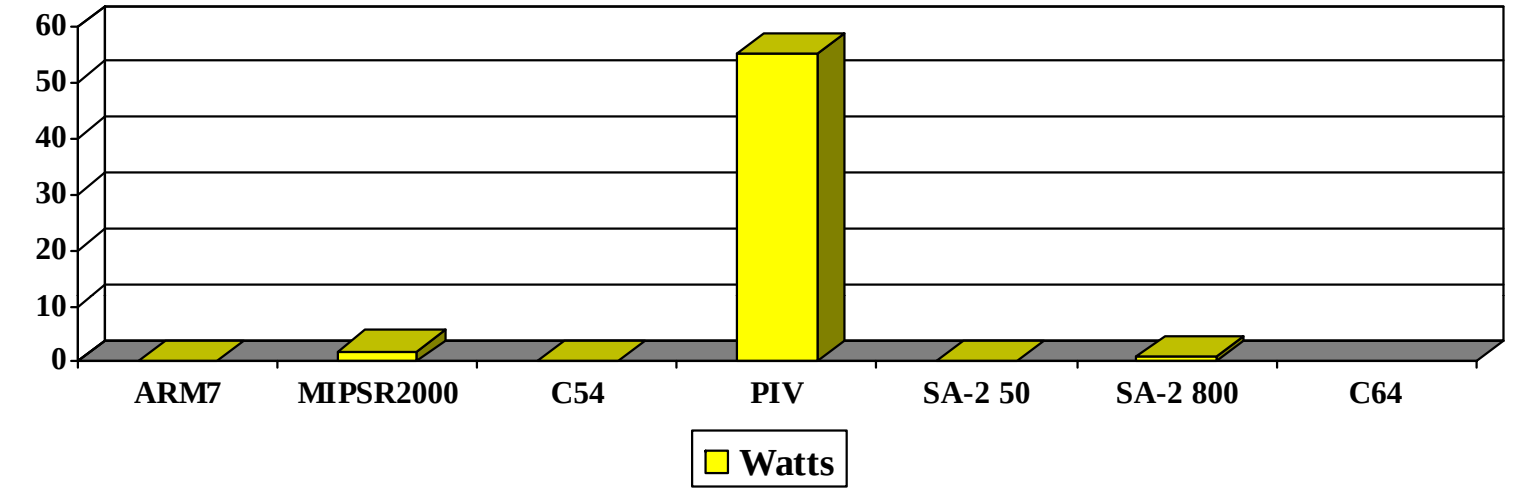
Comparaison répartition

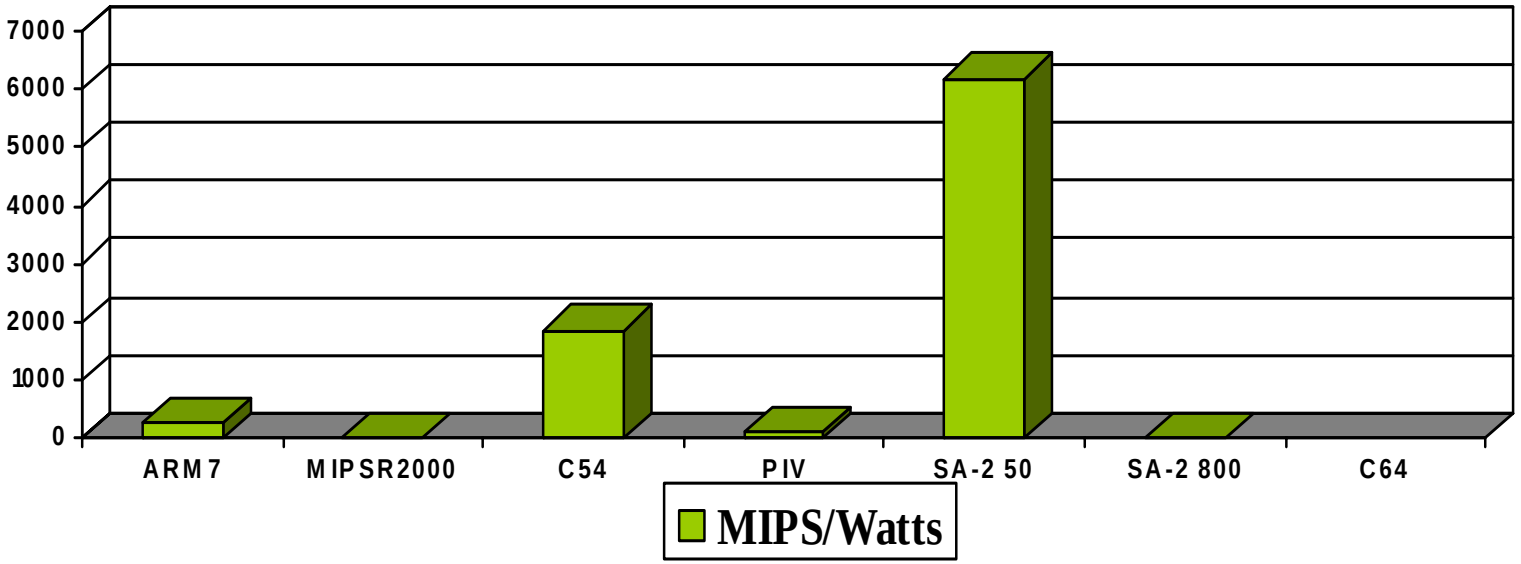


DEC Alpha uP

Alcatel telecom chip (ASIC)

Comparaison Performance/Consommation

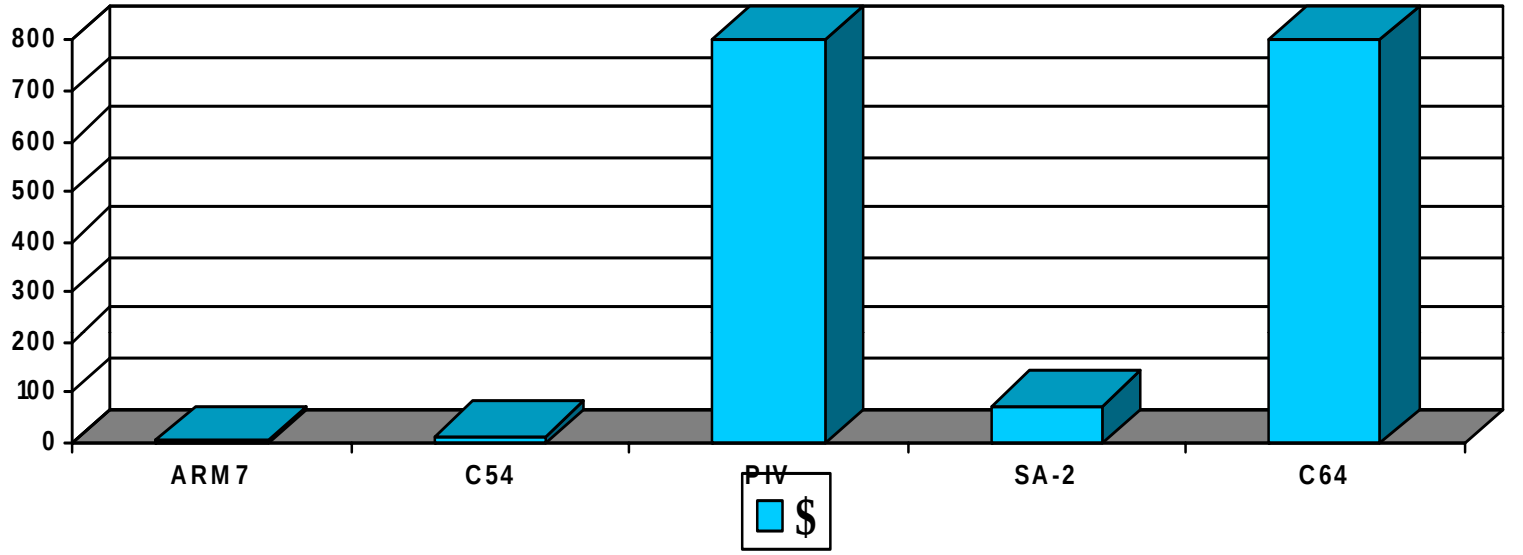


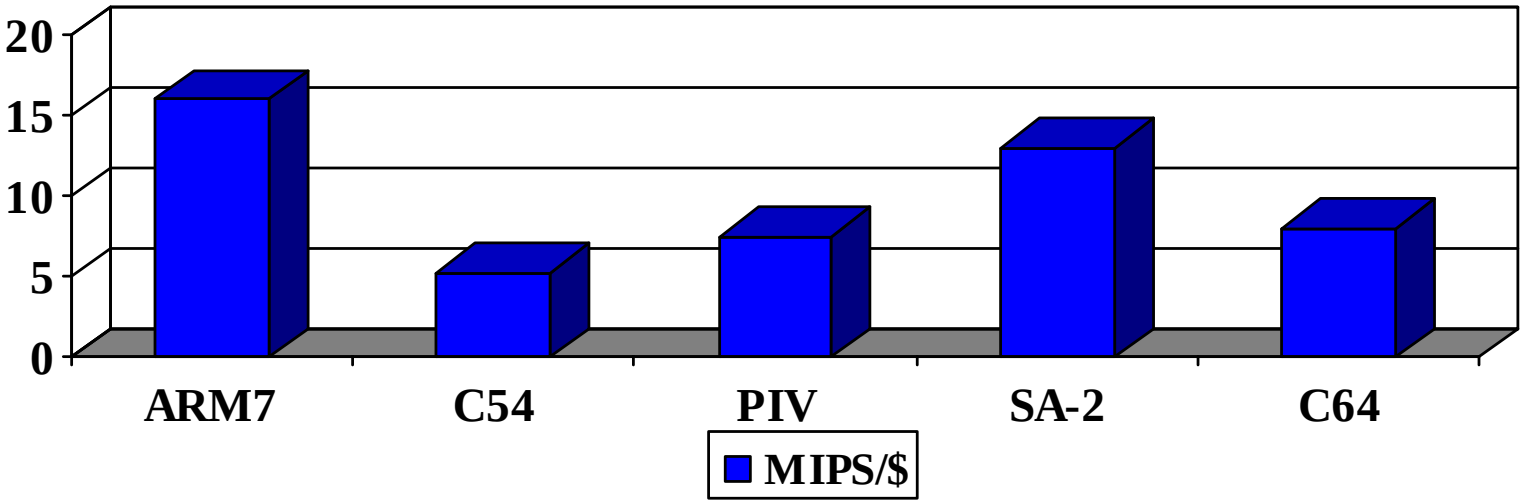


Coût

- Coût dépendant de plusieurs paramètres :
  - coût d'un transistor.
  - coût de la chaîne de conception et des différentes solutions matérielles adoptées.
  - coût des développements logiciels.

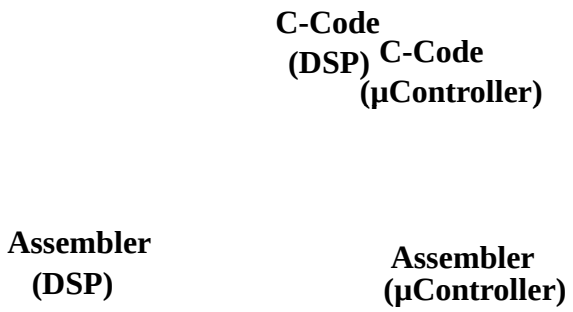
Comparaison Coût





Coût des développements logiciels

- « Historiquement » : développement en assembleur notamment pour les parties critiques.
- Développement HL langage (C et C++) :
  - Coût des développements logiciels et aussi :
  - Augmentation de la complexité des programmes,
  - Portabilité,
  - Progrès des compilateurs,
  - Evolution des architectures.

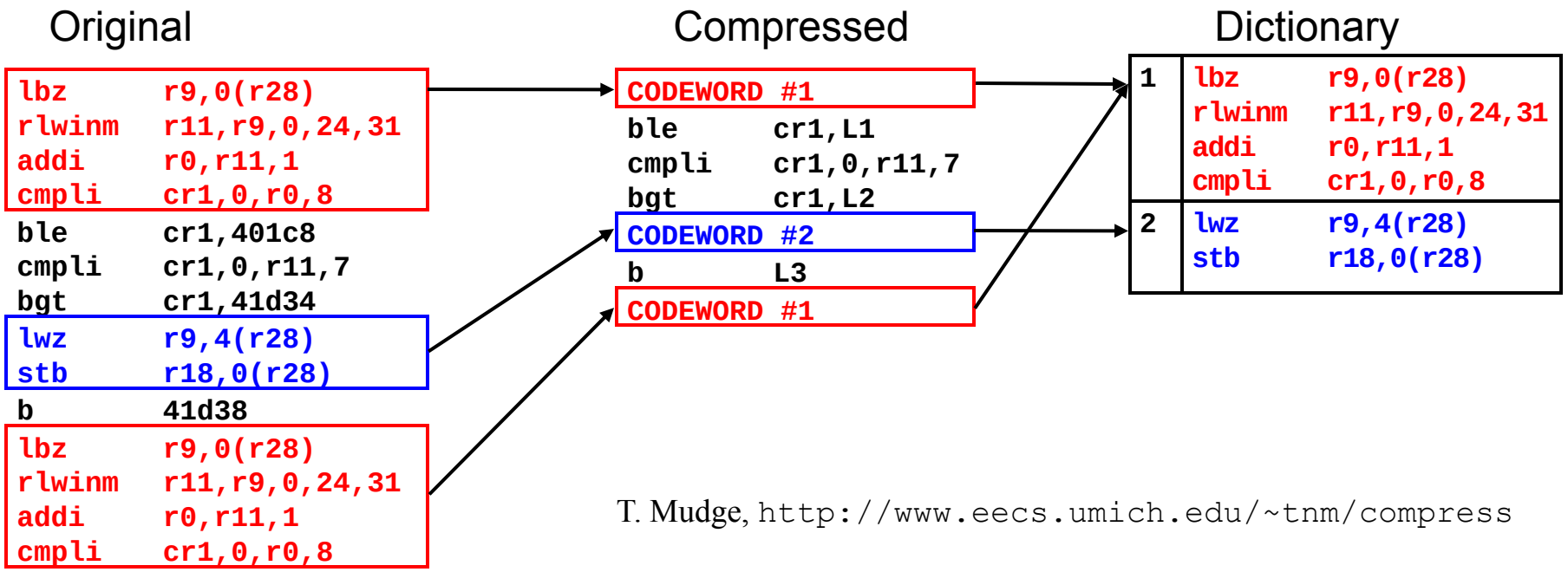


Taille du code

- Problèmes:
  - Coût, taille et puissance.
    - Coût de certaines machines embarquées = coût de la mémoire (et non du processeur).
    - La mémoire instructions domine en taille et en consommation de puissance.
  - Faire tenir le programme sur la mémoire on-chip.
- Solutions:
  - Compilateurs vs assembleur :
    - Taille/vitesse d'optimisation, portabilité, coûts de développements.
  - Compression des programmes:
    - Tenir compte des répétitions.
    - Compromis performance/densité de code.
    - Utiliser des processeurs meilleur marché avec des mémoire on-chip plus petites.

Méthode du dictionnaire : par bloc de base

- Mettre dans un dictionnaire des séquences d'instructions communes (une séquence ne peut pas dépasser un bloc de base).
- Puis, remplacer les séquences du programme avec les codes du dictionnaire.
- Le programme final contient le code compressé et le dictionnaire.



T. Mudge, http://www.eecs.umich.edu/~tnm/compress

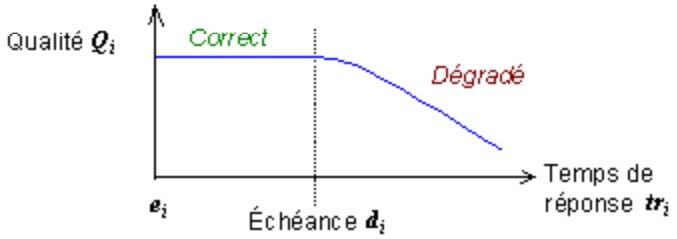
Temps réel

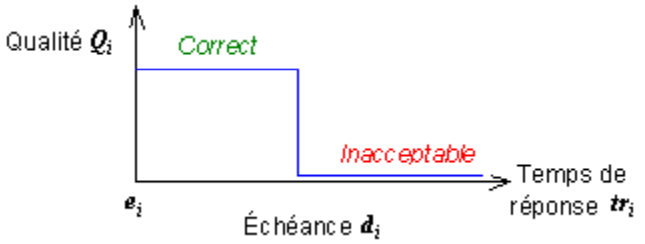
- Système temps réel : système dépendant non seulement de la fiabilité des résultats, mais aussi de **l'instant où les résultats sont produits**.
- « Soft deadline » (*soft real-time systems*), temps réel mou :
  - Multimédia,
  - Jeux vidéo intéractifs.

Ex: *"Missing deadlines in real-time video leads to unacceptable picture quality resulting in a failed product".*

- « Hard deadline » (*hard real-time systems*), temps réel dur :
  - Aviation,
  - Nucléaire,
  - Médical.

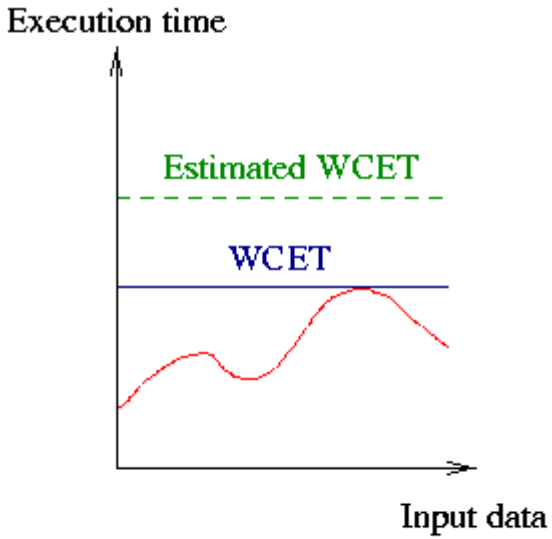
Ex: "*Advanced variable-cycle jet engines can explode if correct control inputs are not applied every 20-50 ms*".

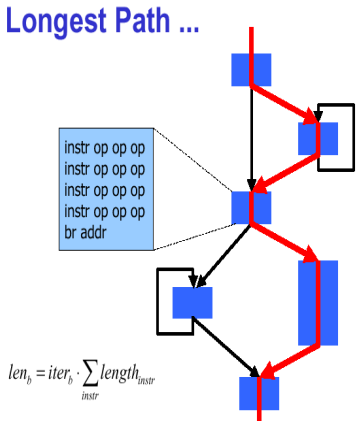




Estimation du temps

- Estimer le temps d'exécution d'une tâche/application donnée. Estimation à architecture et compilateur fixés.
- *Soft real-time systems :* estimation du temps moyen d'exécution.
- *Hard real-time systems* : estimation du WCET (*Worst-Case Execution Time)*, temps d'exécution maximum.
- Le WCET estimé doit être :
  - Sûr (*safe*) i.e. WCET estimé > WCET.
  - Juste (*accurate*) i.e. (WCET estimé – WCET) petit. Ressources sous-utilisées, dimensionner le système.





Embarqué = Multiples Contraintes