

Unité INF-4101C

## Optimisation des temps de calcul et processeur RISC

Mohamed AKIL

Département Informatique

ESIEE Paris

[akilm@esiee.fr](mailto:akilm@esiee.fr)

**Laboratoire d'Informatique**

**(Unité Mixte de Recherche CNRS-UMLV-ESIEE - UMR 8049).**

*<http://www.esiee.fr/dept-info/>*

# Contenu

## ❑ Performances des architectures

- Structures de programme & performances
- Méthodes de conception d'implantation temps réel :  
Adéquation Algorithme Architecture

## ❑ Parallélisme et Processeur RISC

- Parallélisme d'Instruction :  
Processeurs RISC, processeurs VLIW, RISC Multi-core/multi-thread

## ❑ Hiérarchie mémoire et optimisation des temps de calcul

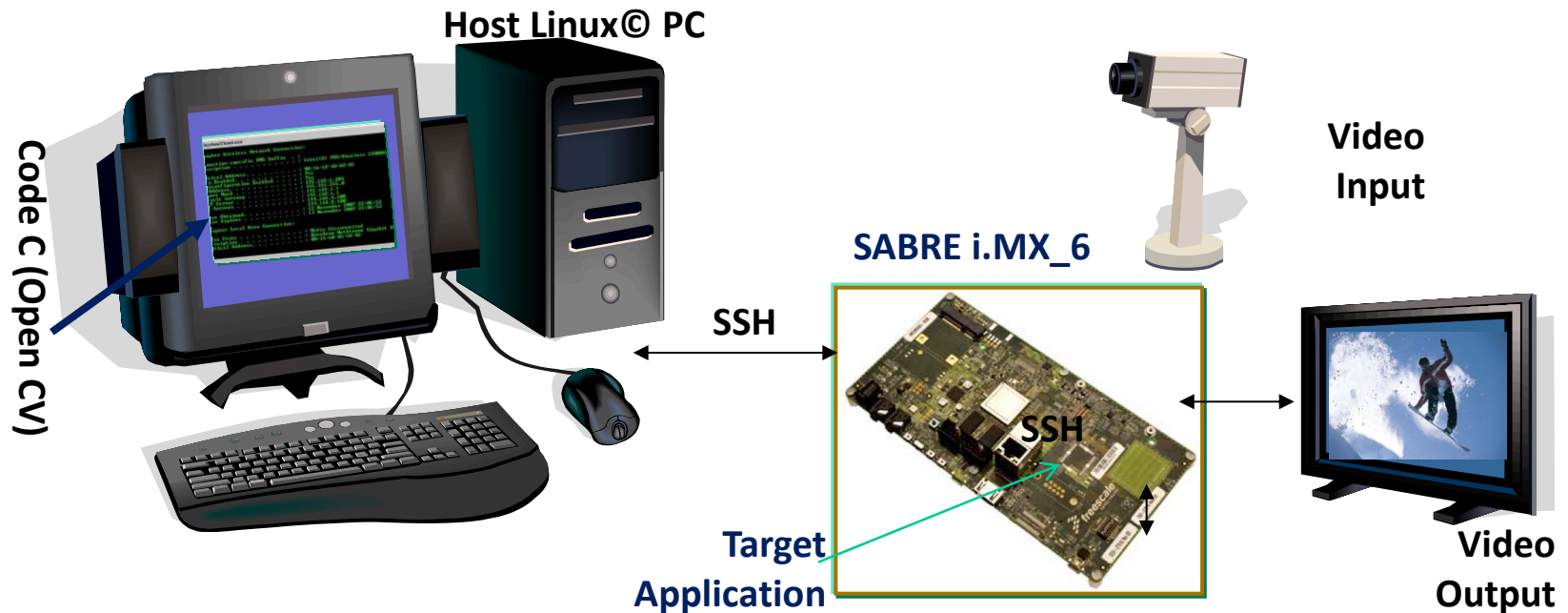
- Mémoire Virtuelle
- Mémoire Cache

# Les compétences à acquérir

- ☐ Analyser la complexité d'un algorithme en termes d'opérations
- ☐ Optimiser (réduire) cette complexité pour respecter la contrainte temporelle de l'application
- ☐ Exploiter le parallélisme d'instructions et appliquer les techniques d'optimisation du code pour réduire le temps d'exécution

# Projet Vision sur carte SABRE i.MX\_6

## Processeur RISC – ARM – Cortex A9 Multi-core/multi-Thread

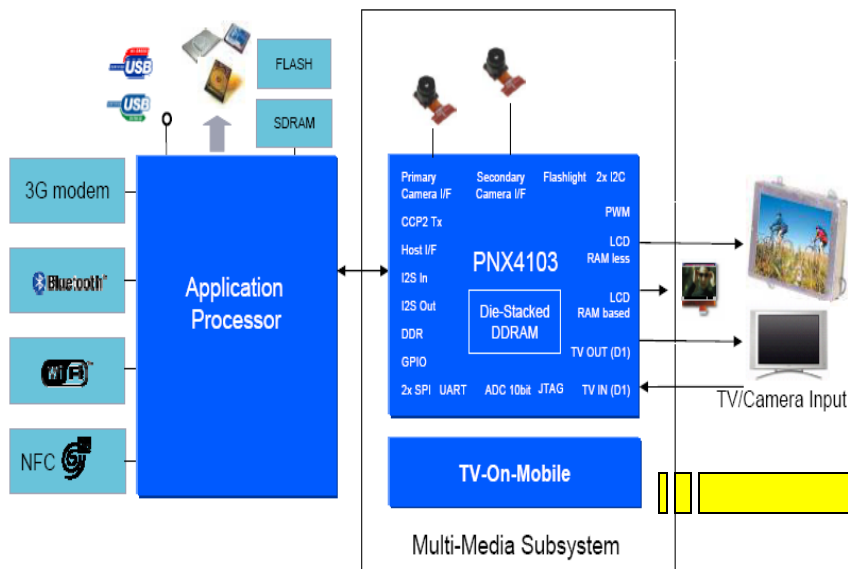
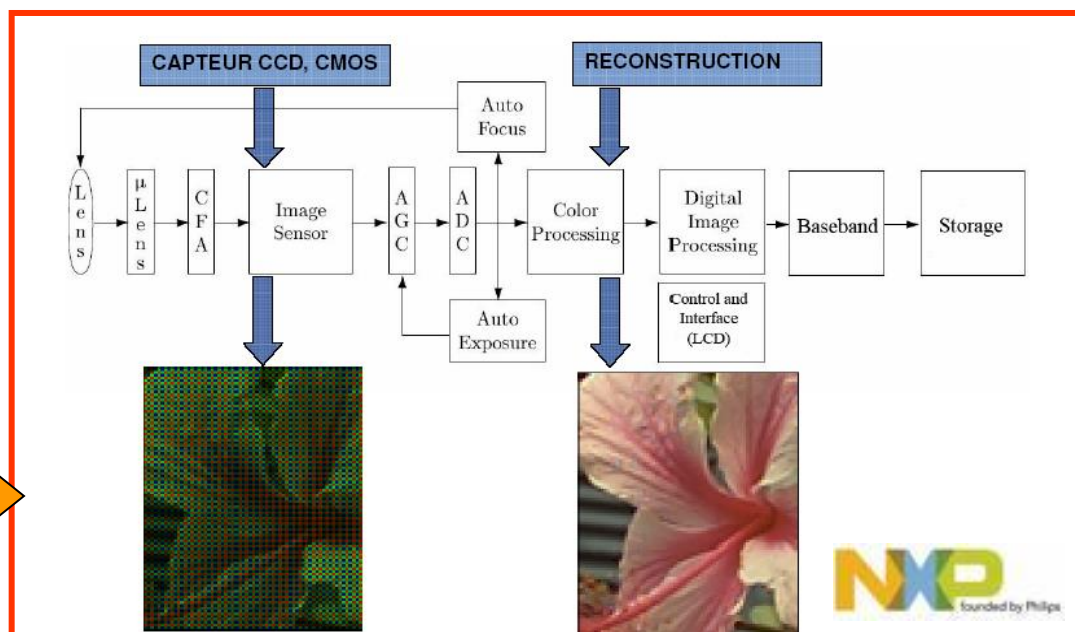


- Optimiser les algorithmes de l'application
- Optimiser le code de l'application
- Exploiter le parallélisme des instructions
- Gérer les threads
- Gérer les accès aux données

# Contextes : quelques exemples d'applications - projets

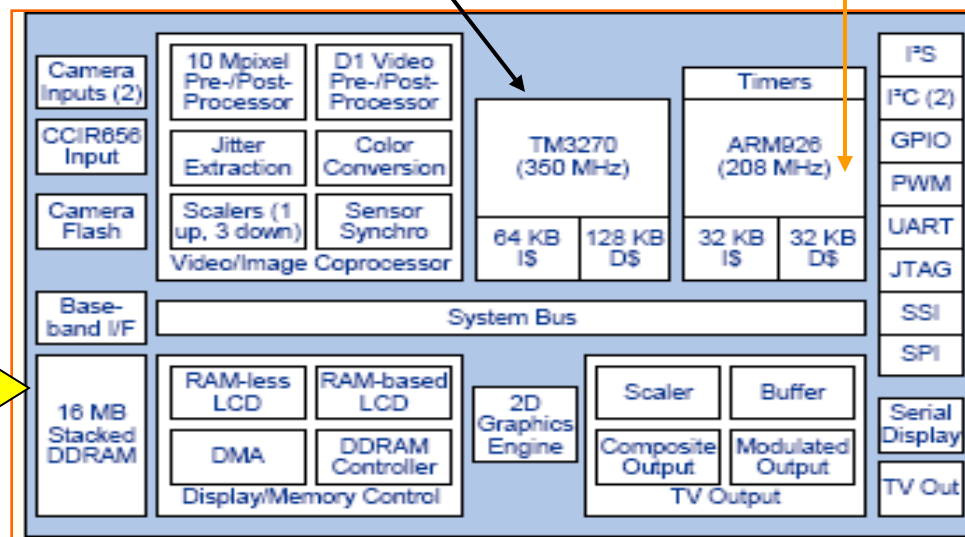
- ❑ Applications → satisfaire la contrainte temporelle
- ❑ Exemples d'Applications :
  - ❖ Téléphone mobile – NxP/ESIEE
  - ❖ Sécurité :
    - Détection de visages – CAOR (ENSMP)/ESIEE
    - Vérification d'identité – THALES/ESIEE

# Traitement des défauts pour l'Imagerie numérique : Étude Algorithmique et Implémentation sur Architecture Mobile



Processeur DSP - VLIW

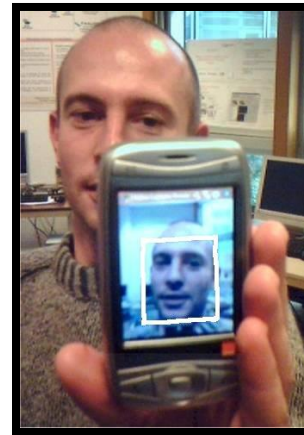
Processeur RISC



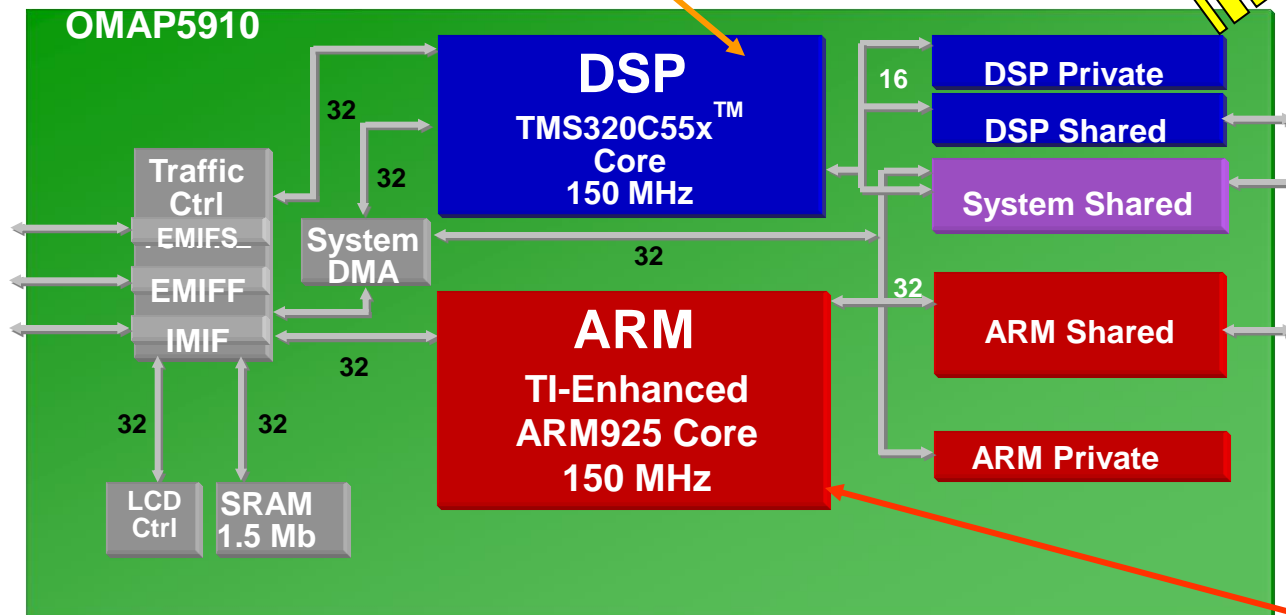
Reduce Instruction Set Computer  
Digital Signal Processor

Digital Signal Processor

# Détection de visage sur système Sur Puce (SoC)



Processeur DSP



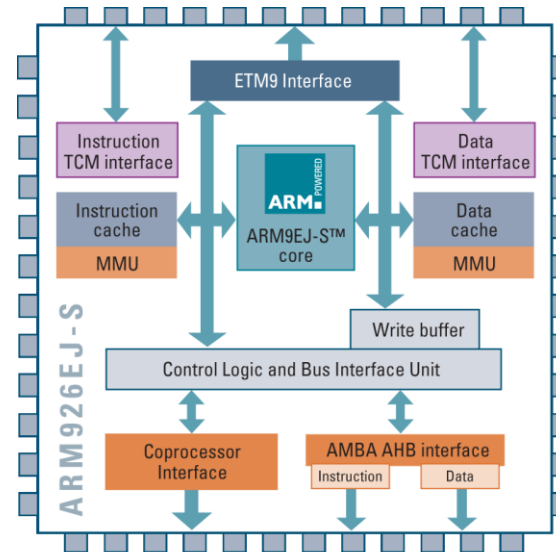
OMAP5910

Processeur RISC - ARM

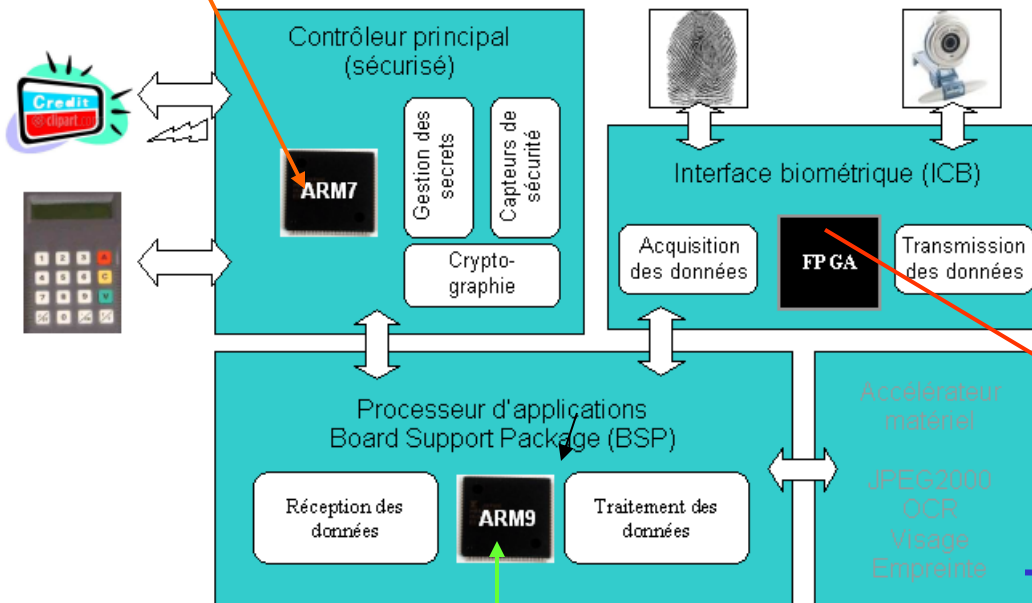
# Projet VINSI (THALES) :

## Vérification d'Identité Numérique Sécurisée Itinérante

### Processeur RISC - ARM



### Processeur RISC



### Processeur RISC

### Circuit Reconfigurable (FPGA)

Field Programmable Gate Array

### Circuit dédié (VLSI)

Very Large Scale Integration



# Optimisation des temps de calcul et processeur RISC

## Objectifs du cours (1):

**Implantation « temps réel » d'algorithmes (code généré) sur des architectures informatiques à base de processeur RISC**

**les techniques étudiées dans le cadre des processeurs s'appliquent et s'étendent aux processeurs RISC  
Multi-core/multi-Thread**

**Implantation optimisée d'algorithmes :  
traitement des données en temps réel →  
respecter les contraintes de latence ou de cadence**

# Optimisation des temps de calcul et processeur RISC

## Objectifs du cours (2) :

**1. Comprendre le fonctionnement et l'organisation des processeurs :**

- **généralistes de type RISC**

**2. Mettre en œuvre des techniques logicielles d'optimisation de code pour obtenir une implantation optimisée (respectant la contrainte temporelle) d'un algorithme**

# Optimisation des temps de calcul et processeur RISC

## Évaluation du coût d'une implantation – à partir de l'algorithme à implanter :

- coût en termes de nombres d'opérations (+,x,..) à implanter
- coût en termes de cycles d'accès aux données à traiter
- coût en termes de taille (encombrement) : traitement, mémoire

### Traitement d'images : Exemples

#### 1. Filtrage spatial d'une image :

- filtre linéaire (Sobel)
- filtre non linéaire (Médian)

## Contexte (4) : Exemple du filtre de Sobel (c=2)

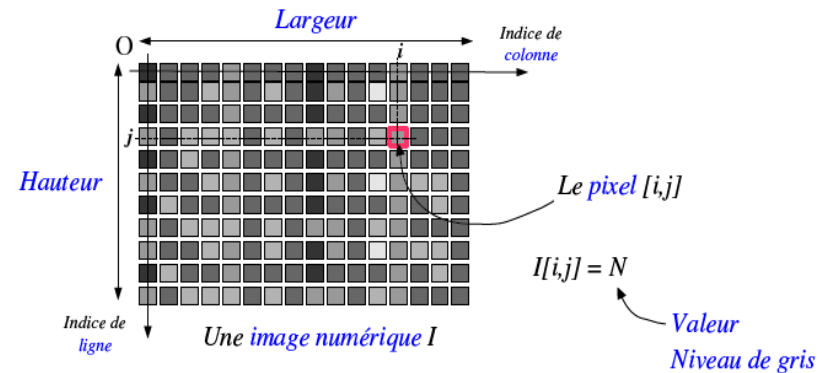
Calcul des dérivées directionnelles en x et y :

→ Convolution avec 2 noyaux  $H_x$  et  $H_y$



$$\begin{aligned} f_x[i, j] &= (f * h_x)[i, j] \\ f_y[i, j] &= (f * h_y)[i, j] \end{aligned}$$

$$H_x = \begin{bmatrix} +1 & +c & +1 \\ +0 & +0 & +0 \\ -1 & -c & -1 \end{bmatrix} \quad H_y = \begin{bmatrix} +1 & 0 & -1 \\ +c & 0 & -c \\ +1 & 0 & -1 \end{bmatrix}$$



Gradient horizontal (Sobel)



Gradient vertical (Sobel)

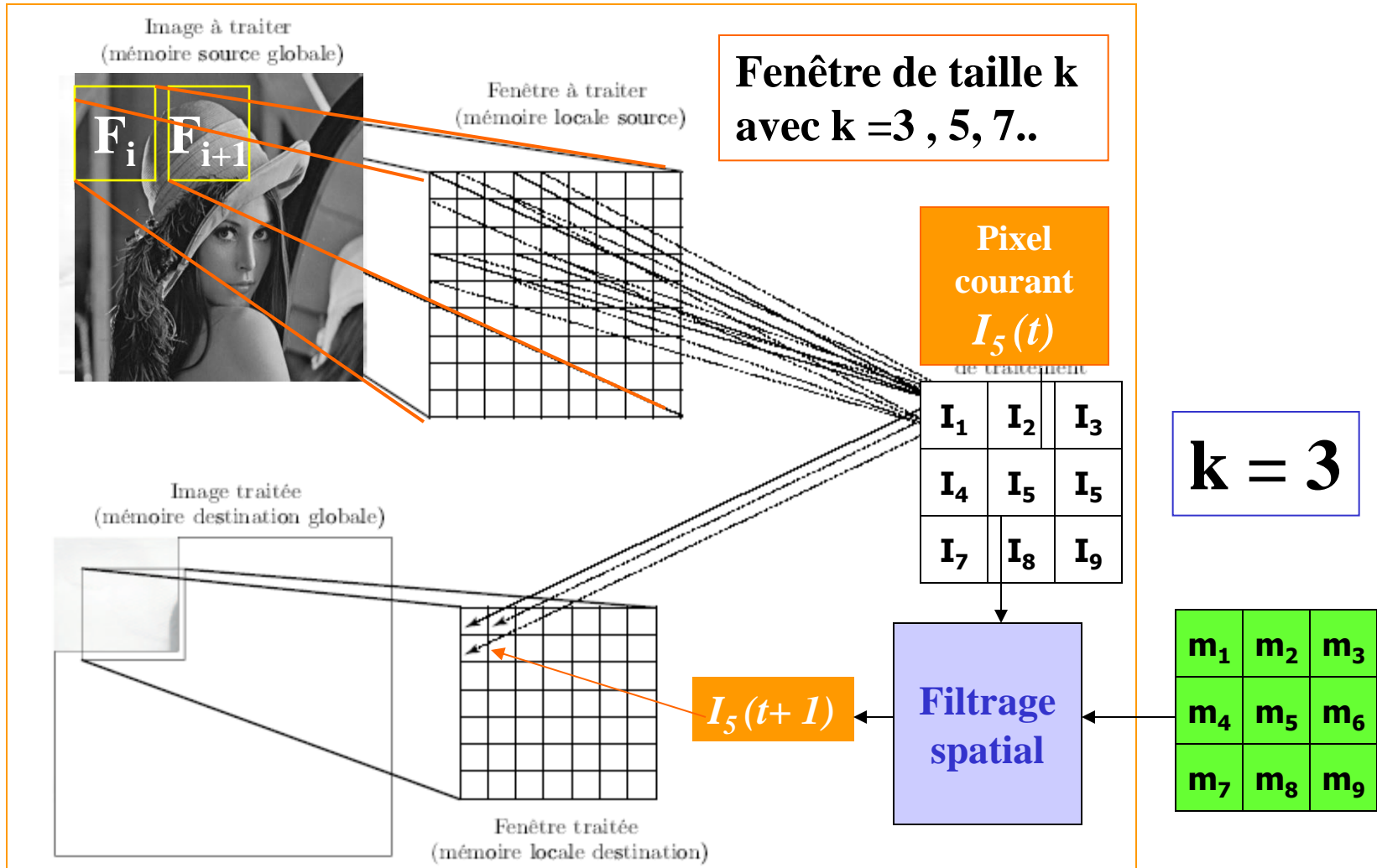


Module du gradient de Sobel

Norme du Gradient

$$\begin{aligned} \|\nabla f[i, j]\|_2 &= \sqrt{f_x[i, j]^2 + f_y[i, j]^2} \\ \|\nabla f[i, j]\|_1 &= |f_x[i, j]| + |f_y[i, j]| \\ \|\nabla f[i, j]\|_\infty &= \max\{|f_x[i, j]|, |f_y[i, j]|\} \end{aligned}$$

## Contexte (5) : Exemple du filtre spatiale – cas général



$$I_5(t+1) = I_1 * m_1 + I_2 * m_2 + \dots + I_9 * m_9$$

## Contexte (6) : Exemple du filtre – cas général

- ❑ Cet algorithme effectue pour un pixel :  $k^2$  multiplications et  $(k^2 - 1)$  additions
- ❑ cet algorithme effectue pour une image de taille  $N*N$  pixels ( $N$  = taille d'une ligne) :
  - ❖  $N^2 [k^2 \text{ multiplications} + (k^2 - 1) \text{ additions}]$
- On dit que la complexité de l'algorithme est de l'ordre de  $N^2$  (noté  $O(N^2)$ )
- ❑ pour un noyau (masque) de taille  $3*3$ , on a  $\approx 18$  opérations
- ❑ pour un noyau (masque) de taille  $31*31$ , on a  $\approx 1900$  opérations
- ❑ pour une image avec  $N=512$  et  $k=3$ , on a :
  - ❖ 9 multiplications par pixel
  - ❖ 2.4 M de multiplications par Image
  - ❖ 59 M de multiplications pour une cadence de 25 images par seconde
  - soit une multiplication (8\*8 bits) tout les 16 ns

## Contexte (7) : Exemple du filtre – cas général

❑ Implantation séquentiel, pour un masque de taille  $k$ , une ligne image de taille  $N$  :

❑ calcul :

❖  $T_k$  = temps de calcul d'un masque =  $[k^2 \text{ multiplications} + (k^2 - 1) \text{ additions}]$

❖  $T_{im}$  = temps de calcul d'image =  $T_k * N^2$

❑ mémoire :

❖  $k^2$  accès en lecture par pixel + un accès en écriture/pixel

❖ 2 bancs mémoire : mémoire (image originale) et mémoire –résultat soit une taille de  $2 * N^2 \text{ Pixels}$  ( $2 * N^2 \text{ octets}$ )

## Contexte (8) : Exemple du filtre – Sobel


$$H_Y =$$

-1	0	1
-2	0	2
-1	0	1

$$H_X =$$

1	2	1
0	0	0
-1	-2	-1

$I_1$	$I_2$	$I_3$
$I_4$	$I_5$	$I_6$
$I_7$	$I_8$	$I_9$

↓ Pixel central

$$\text{Filtrage en } x = (I_1 + 2 \cdot I_2 + I_3) - (I_7 + 2 \cdot I_8 + I_9)$$

$$\text{Filtrage en } y = (I_3 + 2 \cdot I_5 + I_9) - (I_1 + 2 \cdot I_4 + I_7)$$

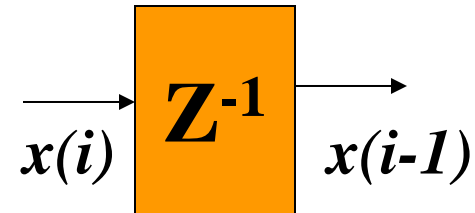


## Contexte (9) : Exemple du filtre – Sobel

□ Filtre séparable :  $N$  = taille d'une ligne

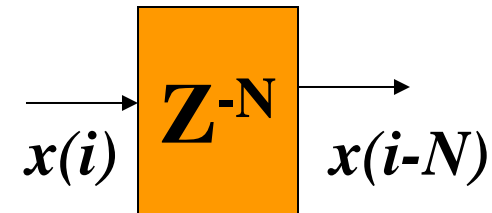
$$H_X = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ -1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

$$H_X = (1 + 2z^{-1} + z^{-2})(1 - z^{-2N})X$$



□ Filtre séparable  $\rightarrow$  factorisable

$$H_X = (1 + z^{-1})^2(1 - z^{-2N})X$$



## Contexte (10) : Exemple du filtre Médian



Image originale

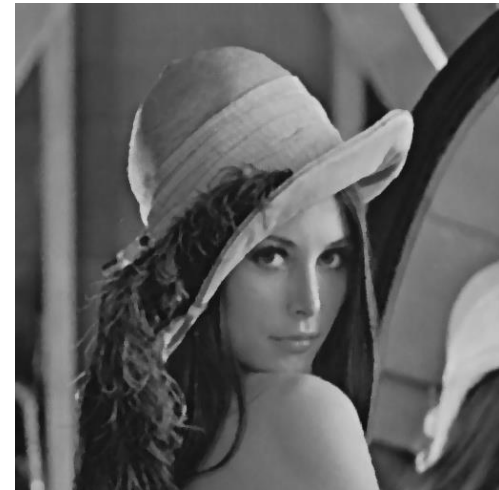


Image bruitée : bruit impulsif

i la liste des niveaux de gris d'un pixel est : [64,64,64,64,255,255,64,64,255].

64 <sub>1</sub>	64	64
64	255	255
64	64	255

Image filtrée par le Médian



TRI : ordre croissant  
→ [64,64,64,64,64,64,255,255,255]

la valeur du pixel courant devient donc la 5<sup>ème</sup> valeur de la liste soit 64

64 <sub>1</sub>	64	64
64	64	255
64	64	255

## Contexte (11) : Exemple du filtre Médian

Le filtre médian repose sur l'utilisation d'un algorithme de tri :

- ❖ tri par sélection
- ❖ tri par insertion
- ❖ tri à bulles
  - **Complexité en  $O(n^2)$**  :  $n^2$  comparaisons pour trier  $n$  valeurs
- ❖ tri rapide
  - **Complexité en  $O(n \log(n))$**  :  $n \log(n)$  comparaisons pour trier  $n$  valeurs

La valeur médiane = valeur se trouvant en  $n/2$  des  $n$  valeurs triées

## Contexte (12) : Exemple du filtre Médian

Médian rapide : soit  $k$  la taille de matrice,  $k$  est un carré

Étape 1 : trier chaque ligne

Étape 2 : trier chaque colonne

Étape 3 : trier la diagonale secondaire

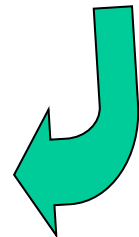
1	4	2
8	7	5
3	6	1



1	2	4
5	7	8
1	3	6



1	2	4
1	3	6
5	7	8



3	4	5
---	---	---



			1	
	1		2	
5		3		4
	7		9	
	8			

## Contexte (13) : Exemple du filtre Médian

### Implantation du Tri :

Soit  $mM$  opérateur *minMax* tel que  $mM(a,b) = (\min(a,b), \max(a,b))$

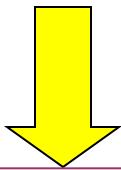
$mM(a,b) = (b,a)$  si  $b < a$

$= (a,b)$  sinon

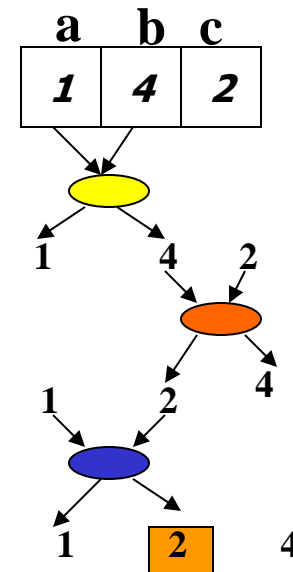
L'opérateur  $mM \rightarrow$  simple permutation

Trier a, b, c consiste à :

1. Trier a,b :  $mM(a,b)$
2. Trier b,c :  $mM(b,c)$
3. Trier a,b :  $mM(a,b)$



min en a, max en c  
la valeur médiane au milieu



# Processeur RISC

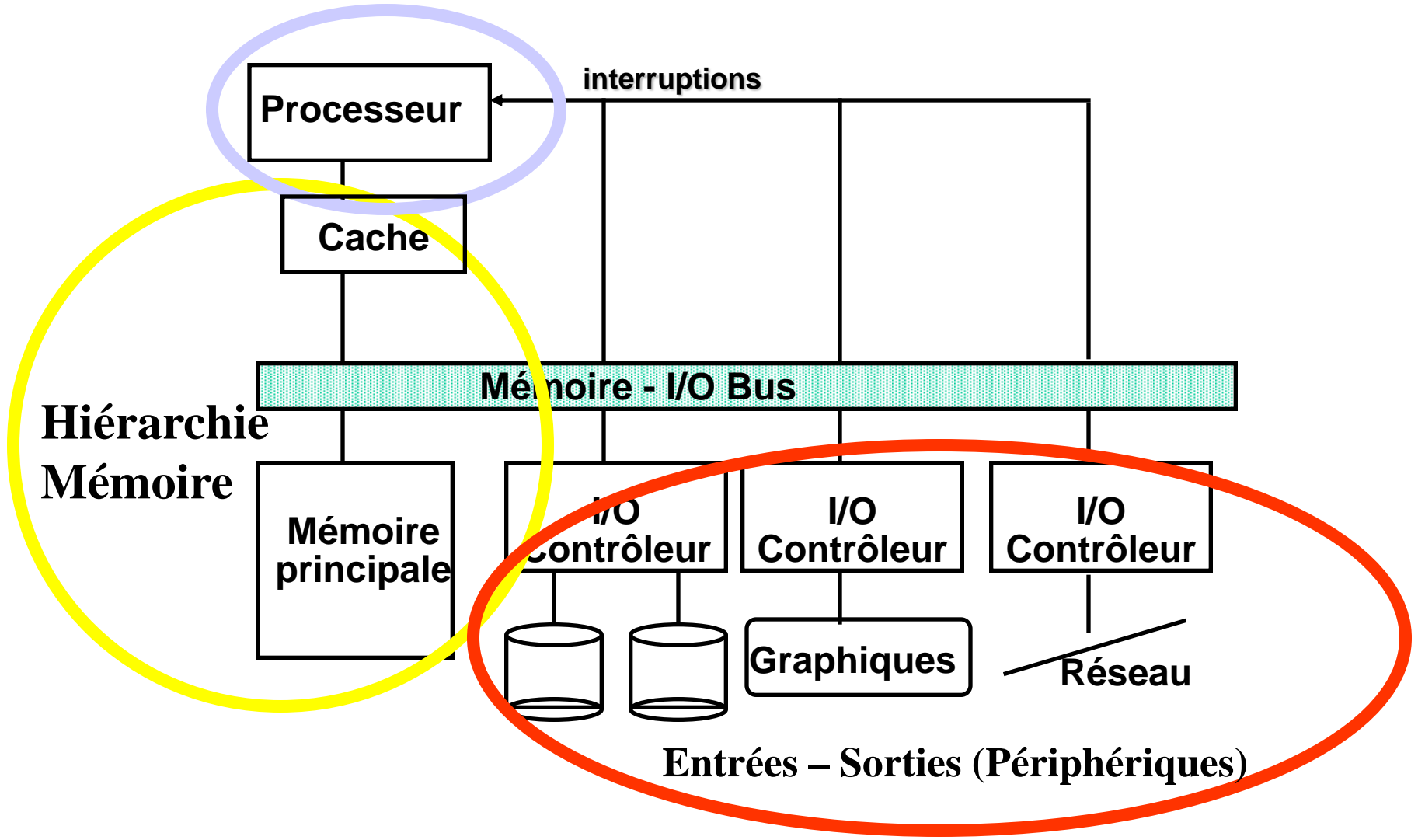
## **Plan :**

- 1. Architecture des ordinateurs et performance**
- 2. Architecture des processeurs RISC**

# **Cours 1 : Architecture et performances**

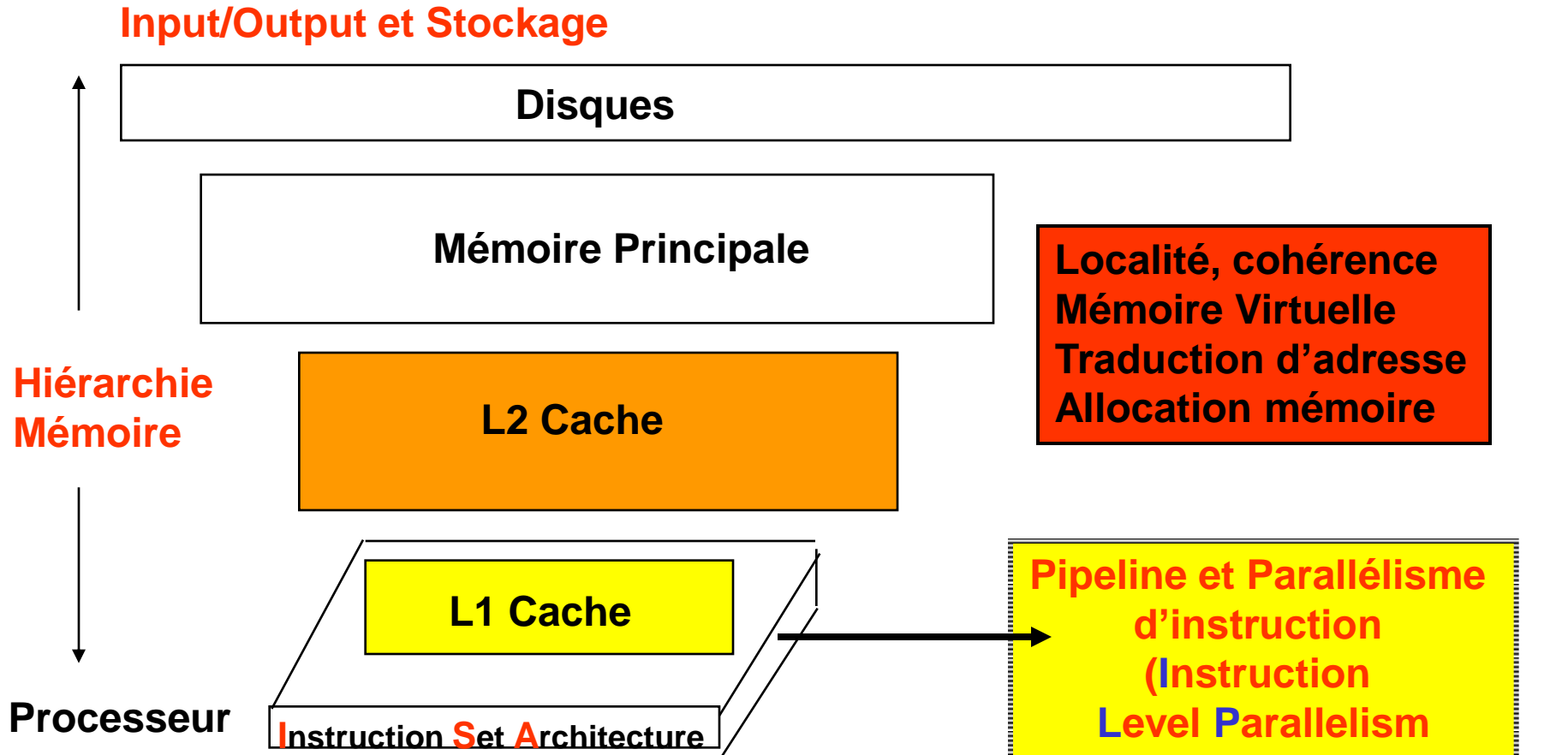
- 1. Evolution technologique : taux de croissance rapide**
- 2. Performance relative**
- 3. La loi d'Amdahl**
- 4. Propriété de la localité des Références**
- 5. Temps UC : Performance du processeur (UC)**

# Organisation d'un ordinateur – rappel (1)



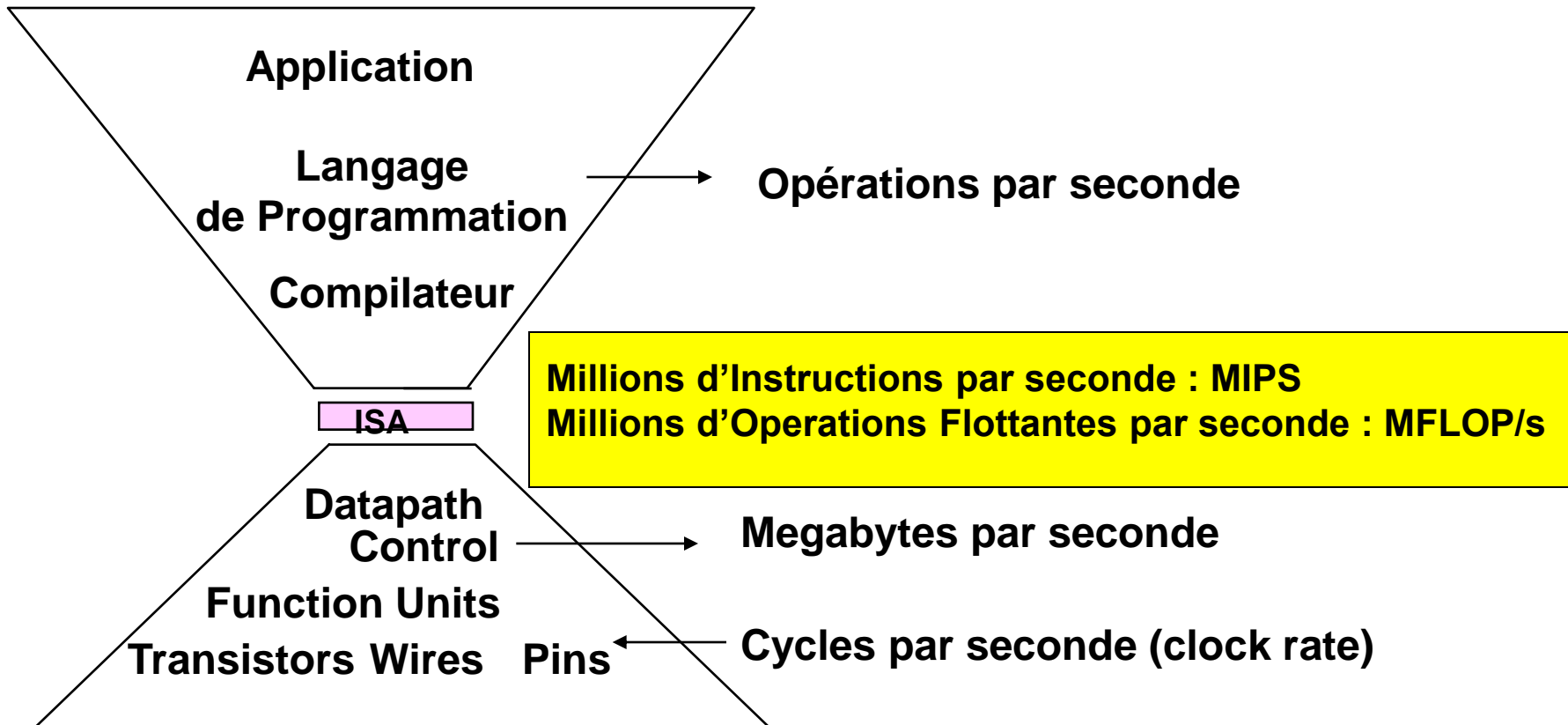


# Organisation d'un ordinateur (2)



1. **Pipeline** : résolution des aléas, Réordonnement des Instructions, Déroulage de Boucles, optimisation de code
2. **Processeurs** : Superpipeline, superscalaire, VLIW, DSP

# Evaluation des Performances (1)



# Evaluation des Performances (2)

- ♦ Deux paramètres peuvent être utilisés pour mesurer la performance d'un processeur:
  - le temps de réponse ou temps d'exécution d'une certaine tâche: temps écoulé entre le début et la fin d'exécution de la tâche
  - throughput: quantité total de travail réalisé dans un certain temps
- ♦ L'amélioration du temps de réponse implique toujours une amélioration du *throughput*. Toutefois, le contraire n'est pas toujours vrai: une augmentation du nombre de processeurs d'un ordinateur augmente le *throughput*, sans améliorer nécessairement le temps de réponse
- ♦ Nous allons considérer le temps d'exécution comme paramètre principal pour le calcul de la performance d'un processeur

# Evaluation des Performances (3) : Débit

- Débit : nombre d'opérations (tâches, instructions,) exécutées par unité de temps
- Exemples:
  - millions d' instructions / sec (MIPS)
  - millions d' instructions-flottantes/ sec (MFLOPS)
  - millions d'octets / sec (MBytes/sec)
  - millions de bits / sec (Mbits/sec)
  - images / sec
  - Échantillons (samples) / sec
  - transactions / sec (TPS)

# Evaluation des performances (4) : temps d'exécution - performance

- **X est n fois plus rapide qu'Y  $\rightarrow$  :**

$$\text{ExTime}(Y) / \text{ExTime}(X) = \text{Performance}(X) / \text{Performance}(Y)$$

- **Soit un programme s'exécutant sur la machine X :**

$$\text{Performance}_X = 1 / \text{Execution time}_X$$

- **X est n fois plus rapide qu'Y  $\rightarrow$  :**

$$\text{Performance}_X / \text{Performance}_Y = n$$

# Evaluation des performances (5) :

## Loi d'Amdahl

$$\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times \left[ (1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right]$$

$$\text{Speedup}_{\text{overall}} = \frac{\text{ExTime}_{\text{old}}}{\text{ExTime}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$\text{Speedup}_{\text{maximum}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})}$$

# Evaluation des performances (6) :

## Loi d'Amdahl

- Instructions Flottantes : amélioration = 2 et seulement 10% des instructions sont des instructions Flottantes

$(1 - \text{Fraction}_{\text{enhanced}})$

$\frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}$

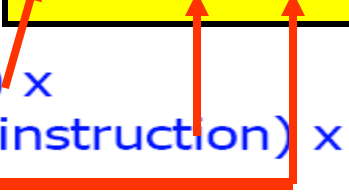
$\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times (0.9 + 0.1/2) = 0.95 \times \text{ExTime}_{\text{old}}$

$\text{Speedup}_{\text{overall}} = \frac{1}{0.95} = 1.053$

# Evaluation des performances (8) :

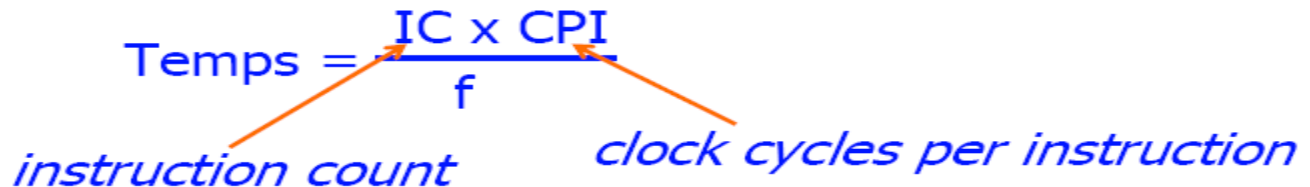
## Temps d'exécution

- ♦ Le temps d'exécution dépend de trois facteurs:
  - le nombre d'instructions machine exécutées,
  - le nombre moyen de cycles d'horloge par instruction machine et
  - la période d'horloge

$$\text{Temps} = 1 / \text{performance} = \frac{\text{(nombre d'instructions)} \times \text{(nombre de cycles par instruction)} \times \text{(période d'horloge)}}{1}$$


$$\text{Temps} = \frac{\text{IC} \times \text{CPI}}{f}$$

*instruction count*      *clock cycles per instruction*



- augmenter la fréquence d'horloge
- améliorer l'organisation interne pour diminuer le CPI
- améliorer le compilateur pour diminuer le IC ou pour augmenter le taux d'utilisation des instructions avec un CPI moindre



# Evaluation des performances (9) : nombre de Cycles Par Instruction

- Nombre de Cycles moyen par Instruction :

$$\begin{aligned}\text{CPI} &= \text{Cycles} / \text{Instruction Count} \\ &= (\text{CPU Time} * \text{Clock Rate}) / \text{Instruction Count}\end{aligned}$$

$$\text{CPU time} = \text{CycleTime} * \sum_{i=1}^n \text{CPI}_i * I_i$$

- Fréquence d'Instruction :

$$\text{CPI} = \sum_{i=1}^n \text{CPI}_i * F_i \quad \text{où} \quad F_i = \frac{I_i}{\text{Instruction Count}}$$

- Temps d'exécution :  $T_{\text{ex}} = \text{NI} * \text{CPI} * T_{\text{C}}$
- $\text{CPI} = \text{CPI}_{\text{processeur}} + \text{CPI}_{\text{mémoire}} + \text{CPI}_{\text{e/s}}$

# Evaluation des performances (10) :

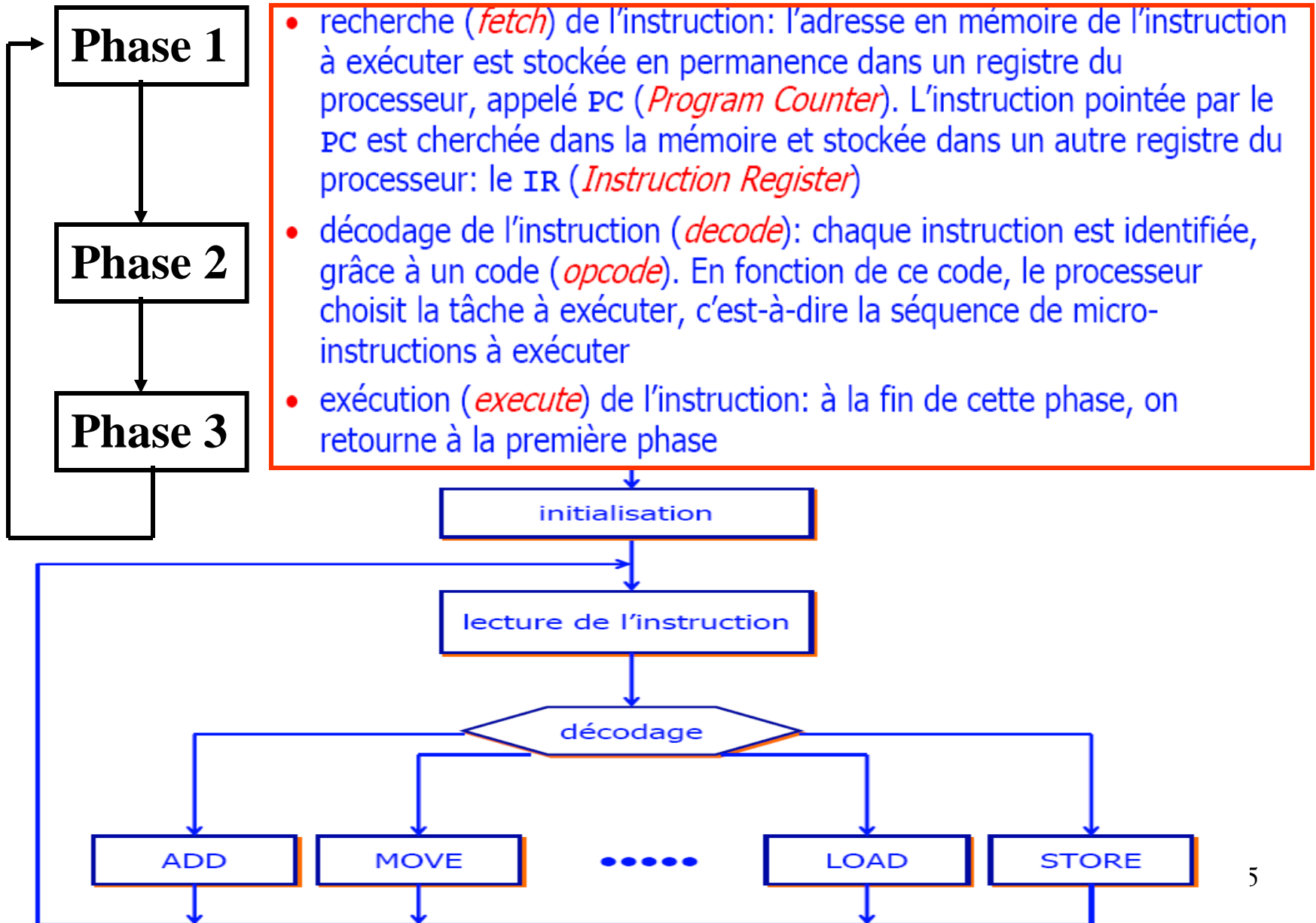
## Mesure du temps d'exécution

$$\text{Exec. time} = \frac{\text{Secondes}}{\text{Programme}} = \frac{\text{Instructions}}{\text{Programme}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Secondes}}{\text{Cycle}}$$

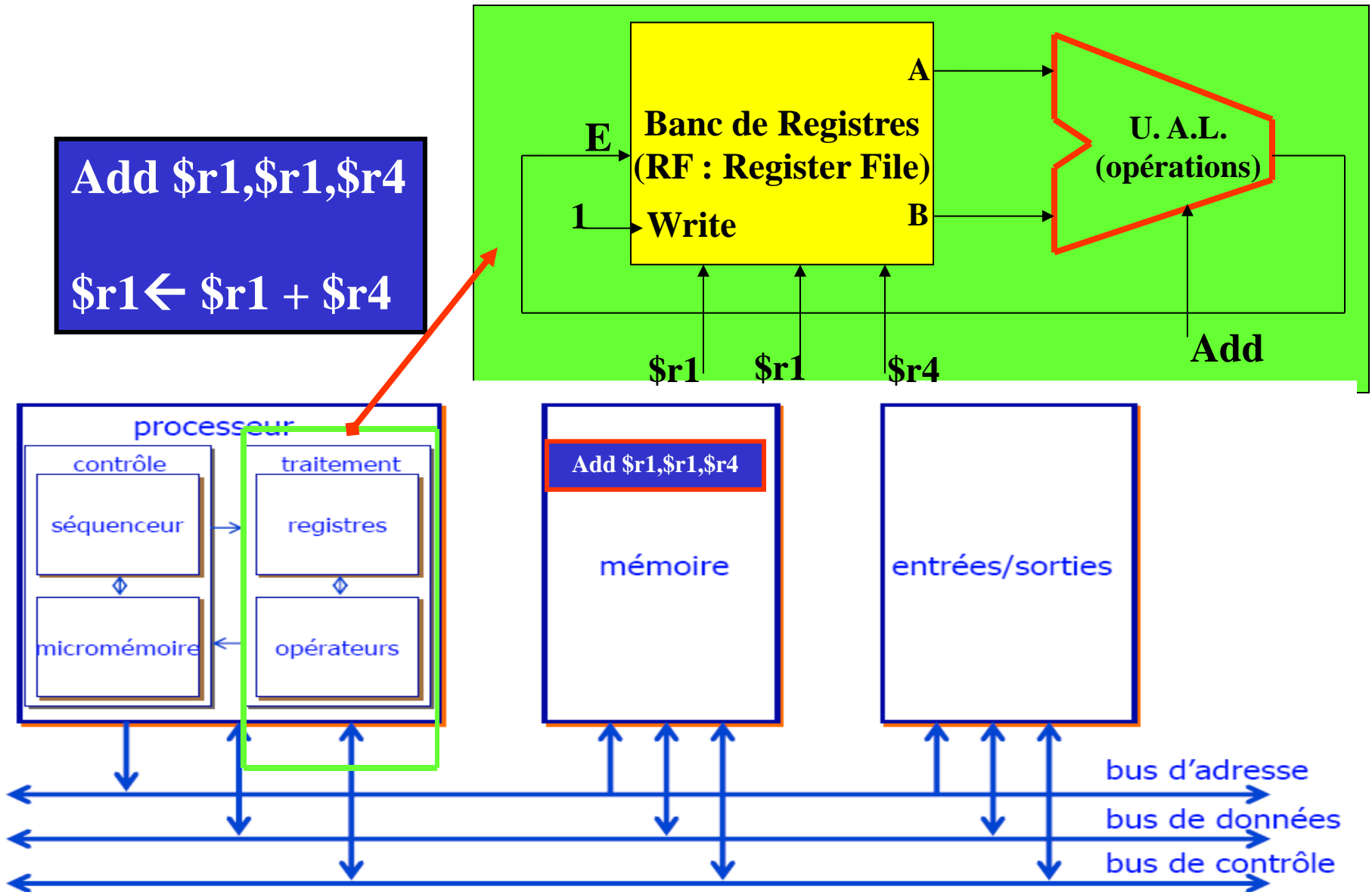
	Inst Count	CPI	Clock Rate
Programme	X		
Compilateur	X	(X)	
Jeu d'Inst.	X	X	
Organisation	X		X
Technologie			X

Domaine de l'Architecture

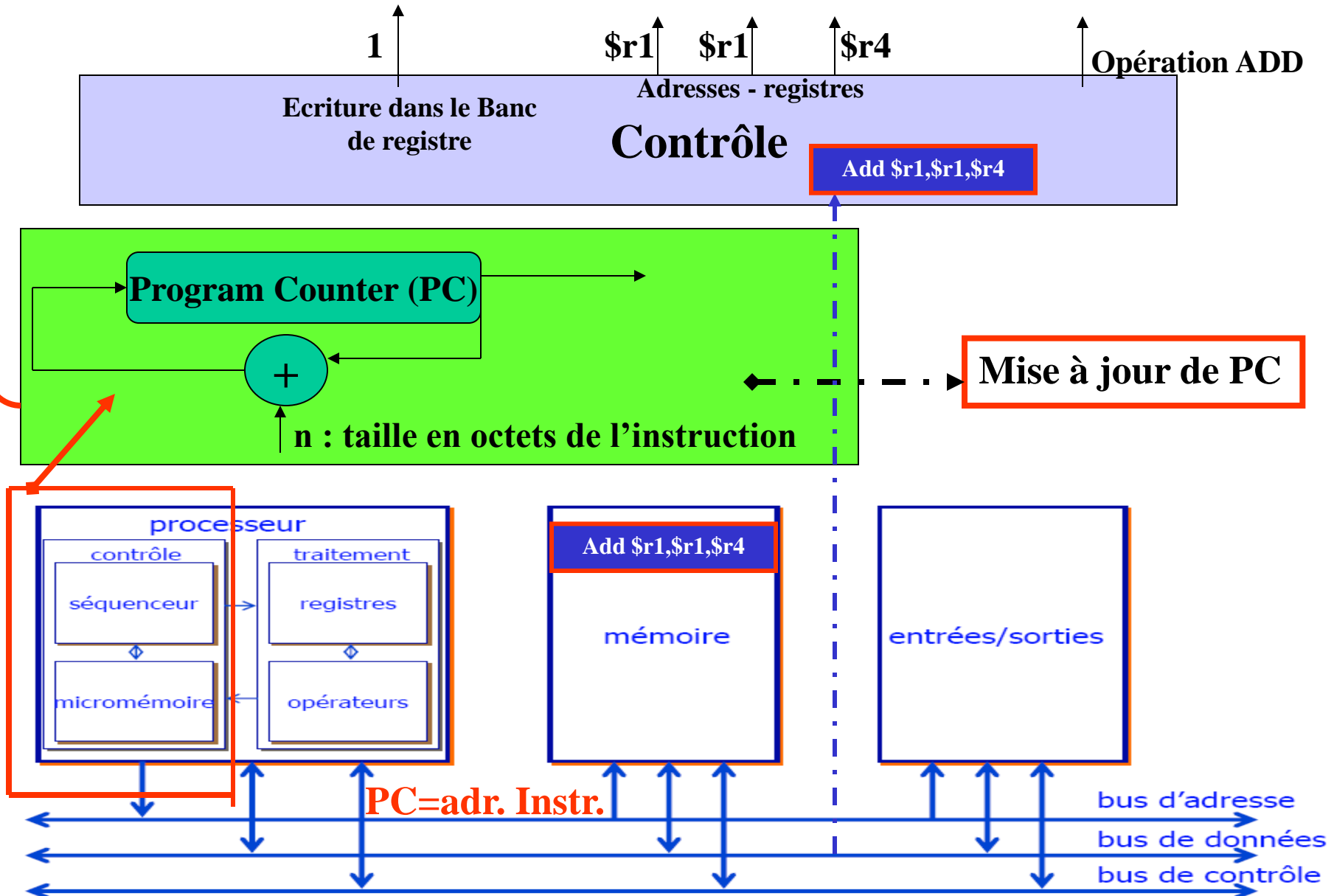
# Instruction : différentes phases d'exécution



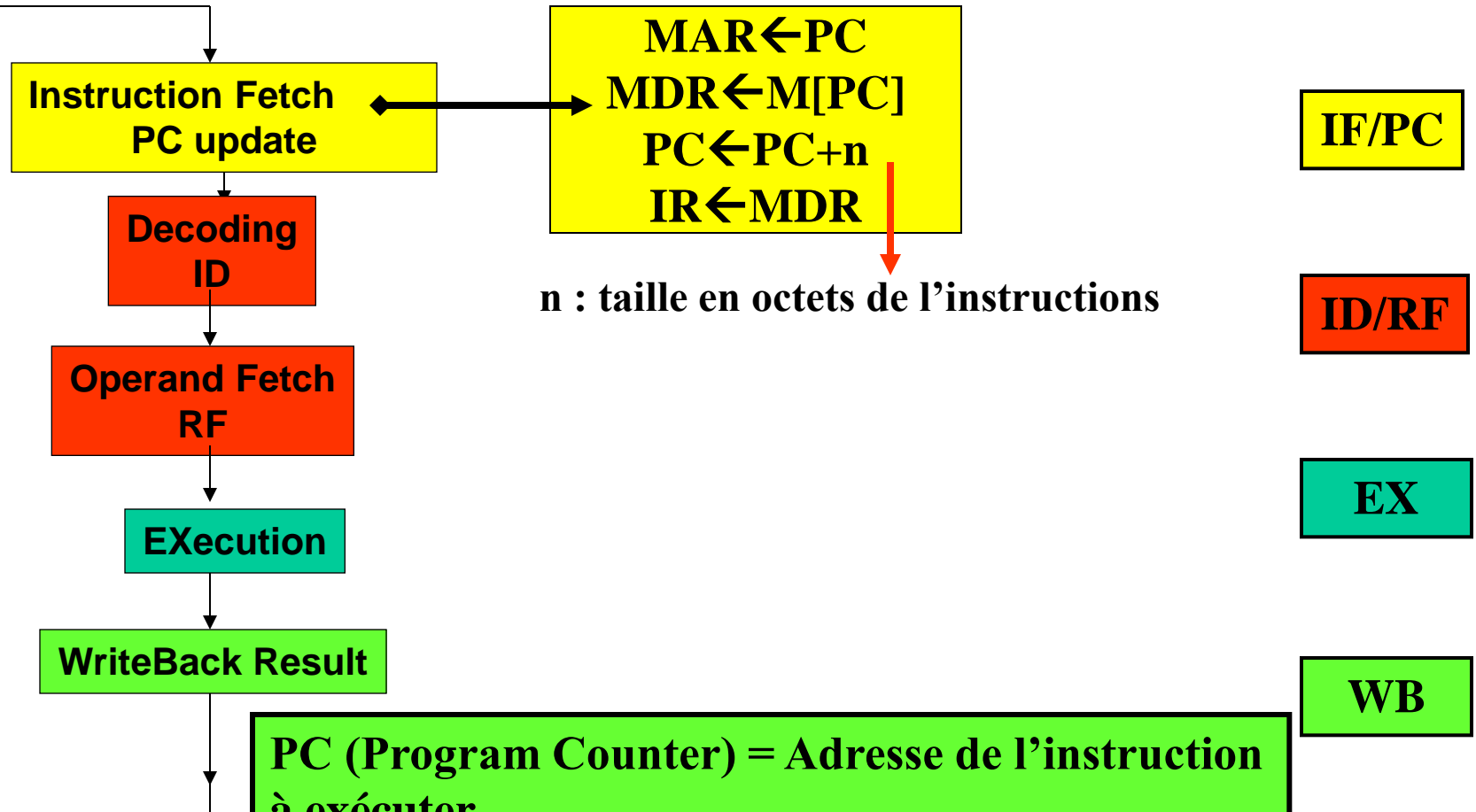
# Processeur : chemin de données (Datapath)



# Processeur : chemin de contrôle (Controlpath)



# Phases d'exécution d'une instruction de type UAL



**PC (Program Counter) = Adresse de l'instruction à exécuter**

**Registres Tampons :**

- **MAR (Memory Adress Register)**
- **MDR (Memory Data Register)**
- IR (Instruction Register)**

# Phases d'exécution d'une instruction :

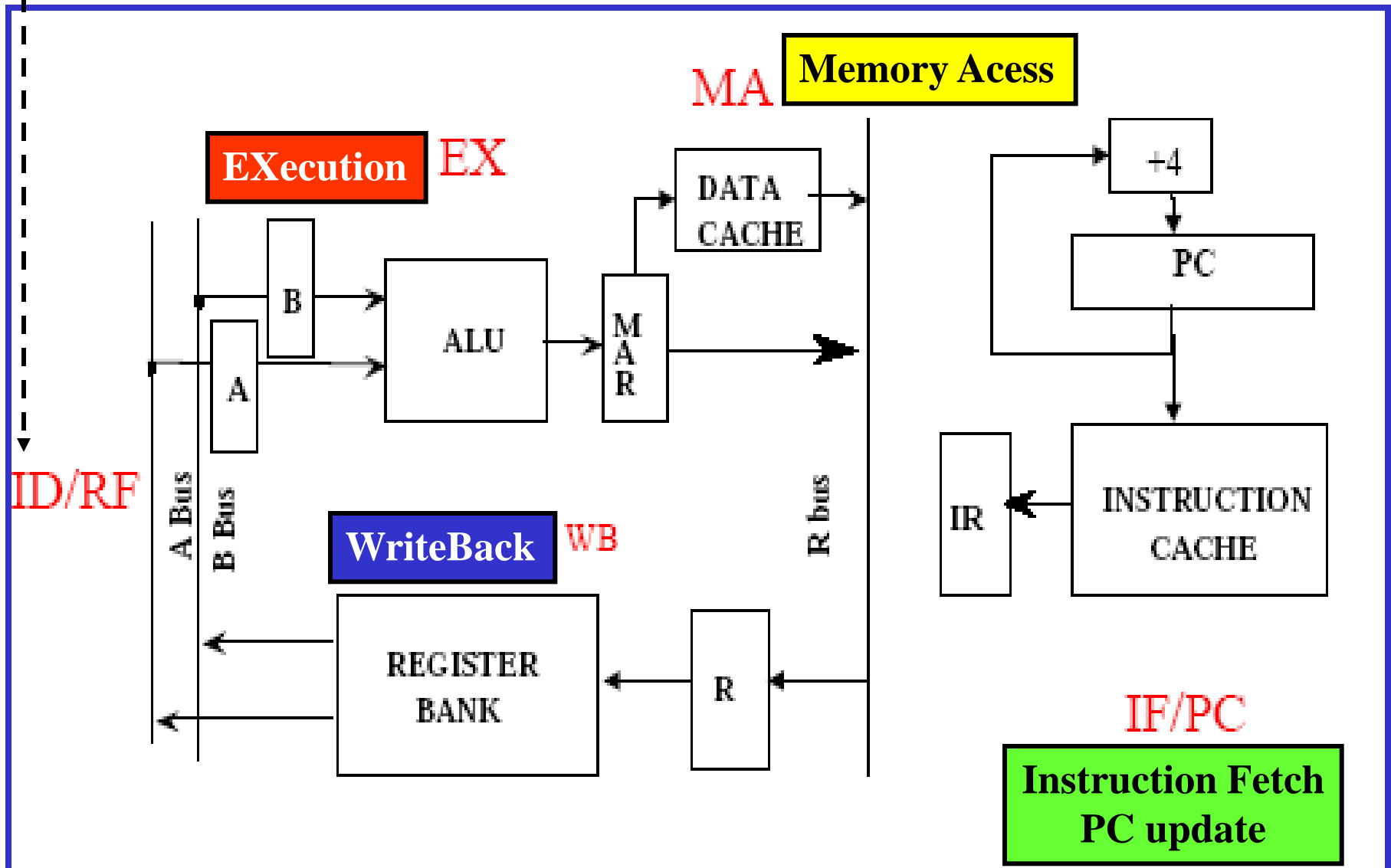
## Types : UAL, Mémoire et Branchement

<i>ALU Instructions</i>	<i>Memory instructions</i>	<i>Branch instructions</i>
Instruction Fetch	Instruction fetch	Instruction fetch
PC update	PC update	PC update
Decoding	Decoding	Decoding
Operand fetch	Address computation	Branch address computation
Execution	Memory access	Execute
Write Back	Write back	

- Integer Instructions  
IF/PC ID/RF EX WB
- FP Instructions  
IF/PC ID/RF EX1 EX2 ... WB
- Memory Instructions  
IF/PC ID/RF AC MA WB
- Branch Instructions de  
IF/PC ID/BAC/EX

# Processeur : chemin de données (Datapath)

Instruction Decoding  
Operand fetch





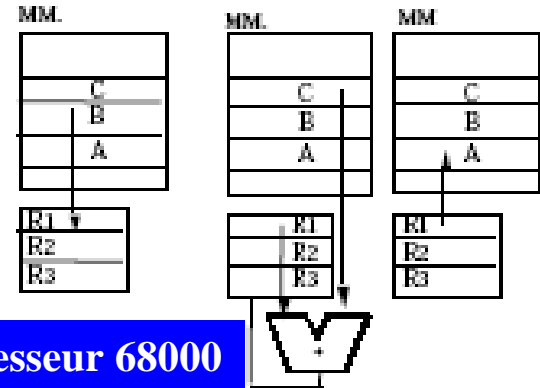
# Exemples de modèles de Processeurs : Registre-mémoire, Load-Store

## Processeur CISC

REG REG  
 REG MEM  
 REG IMM  
 MEM REG  
 MEM IMM

### REGISTER-MEMORY (2,1)

Load	R1	@B
Add	R1	@C
Store	R1	@A

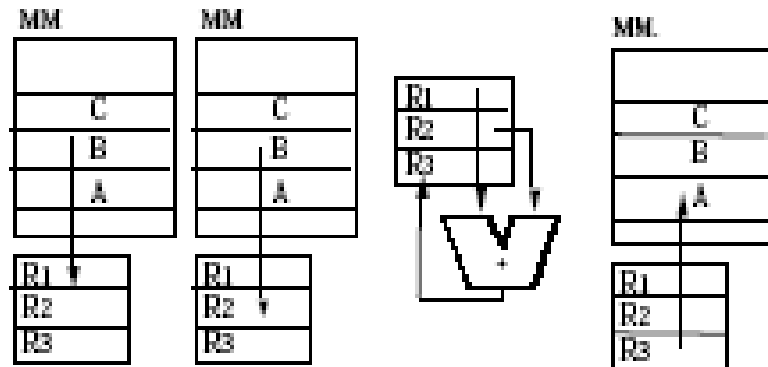


Par exemple : microprocesseur 68000

## Processeur RISC

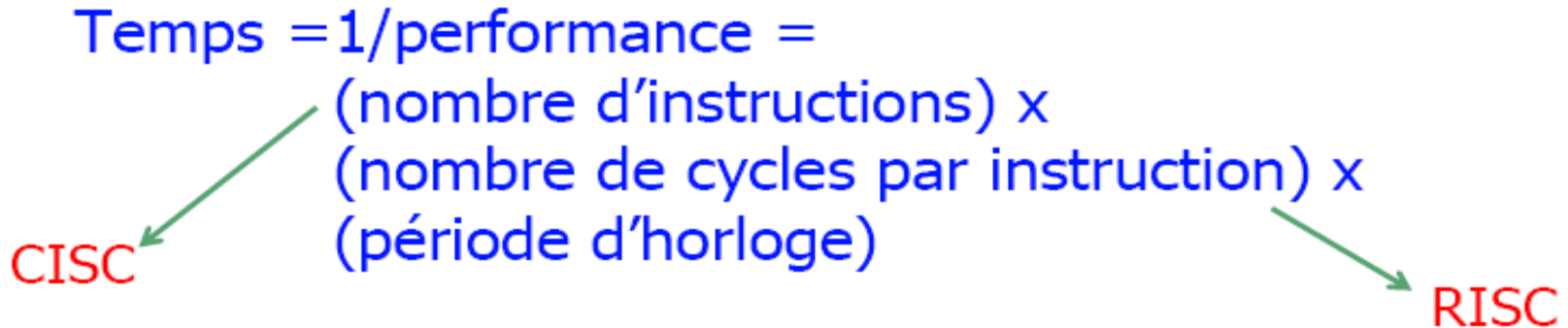
### LOAD-STORE (3,0)

Load	R1	@B
Load	R2	@C
Add	R3	R2 R1
Store	R3	@A



# Processeur RISC

- processeurs CISC (*Complex Instruction Set Computer*)
- processeurs RISC (*Reduced Instruction Set Computer*)

$$\text{Temps} = 1/\text{performance} = \frac{(\text{nombre d'instructions}) \times (\text{nombre de cycles par instruction}) \times (\text{période d'horloge})}{\text{CISC} \quad \quad \quad \text{RISC}}$$


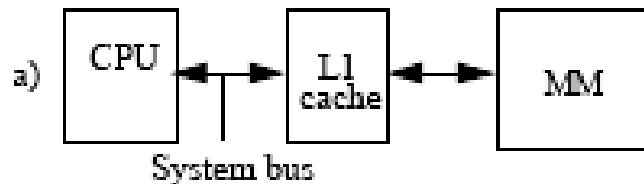
## Caractéristiques générales d'un processeur RISC :

- Pipeline d'exécution des instructions
- Instructions de longueur fixe
- Format d'instructions à 3 adresses
- Codage simple et homogène des instructions
- Accès mémoire : uniquement par les instructions Load/Store
- Modes d'adressage simples

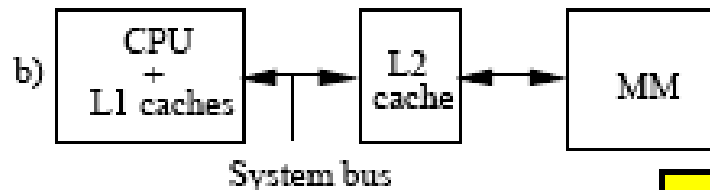
# Exemple de Processeur Pentium (Intel)

## Processeur CISC (Intel)

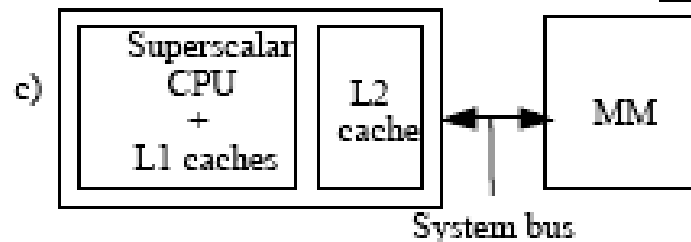
486



Pentium

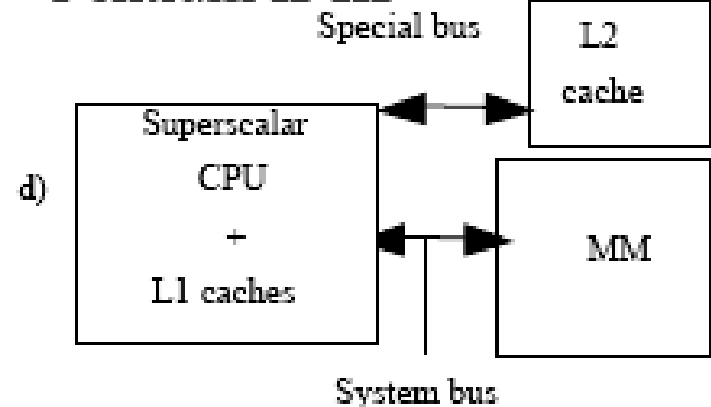


Pentium Pro

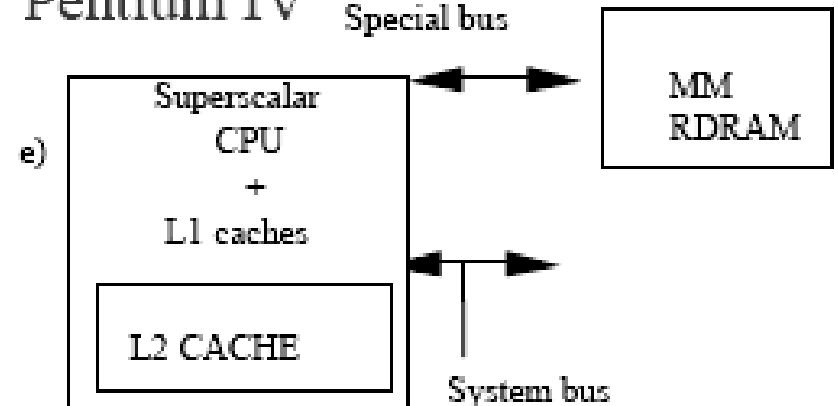


**RISC**

Pentium II-III



Pentium IV

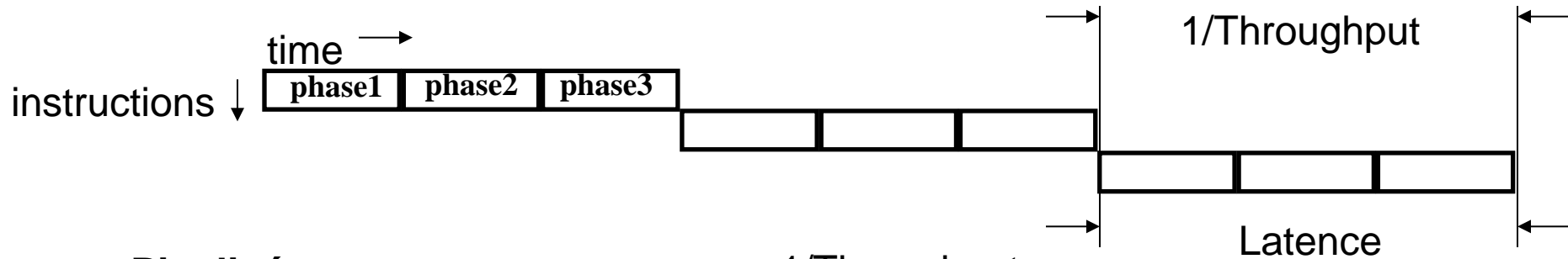


## **Cours 2 : Architecture des processeurs RISC**

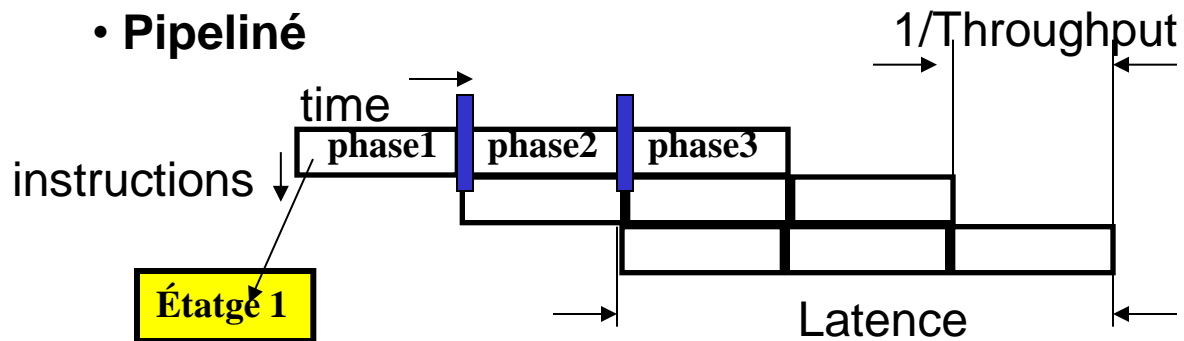
- 1. Processeurs CISC : rappels sur les caractéristiques générales**
- 2. Processeur RISC :**
  - 1. Technique du pipeline**
  - 2. Traitements des aléas**
    - 1. Aléas structurels**
    - 2. Aléas de données**
    - 3. Aléas de Contrôle**

# Modèle d'exécution en pipeline (1)

- Non-pipeliné (séquentiel)



- Pipeliné



- On dispose de  $n$  instructions à exécuter, chacune étant divisée en 5 sous-traitements nécessitant  $\tau$  unités de temps.
- $T_{\text{seq}} = 5n\tau$
- $T_{\text{par}} = 5\tau + (n-1)\tau$
- $A = \frac{T_{\text{seq}}}{T_{\text{par}}} = 5$

- Cas - Idéal :  $T_{\text{pipeline}} = \frac{T_{\text{séquentiel}}}{\text{Profondeur-Pipeline}}$

# Notion de latence d'un traitement

- Le programme comprend  $n$  opérations, l'opérateur pipeline est formé de  $k$  étages, et le temps de traversée d'un étage dure  $t$   
 $\Rightarrow$  temps total :  $kt + (n - 1)t$ .
- Une instruction décodée par unité de temps. Fin de la première instruction à la date  $kt$ . Toutes les  $t$  unités de temps, fin d'une nouvelle instruction.
- Durée totale :  $kt + (n - 1)t = (k - 1)t + nt$ .
- $(k - 1)t$  est appelé temps de latence. Temps durant lequel aucun résultat n'est disponible.

## Accélération ou speed-up : $A_k$ ou $S_k$

- le facteur d'accélération ou *speed-up* : nombre de fois plus vite qu'en séquentiel.

$$S_k = \frac{t_1}{t_k} = \frac{nkt}{(k-1+n)t} \sim k \text{ lorsque } n \text{ est grand.}$$

- le débit : fréquence de sortie du pipeline des instructions.

$$d = \frac{1}{t_{ExecInst}} = \frac{n}{(k-1+n)t} \sim \frac{1}{t}$$

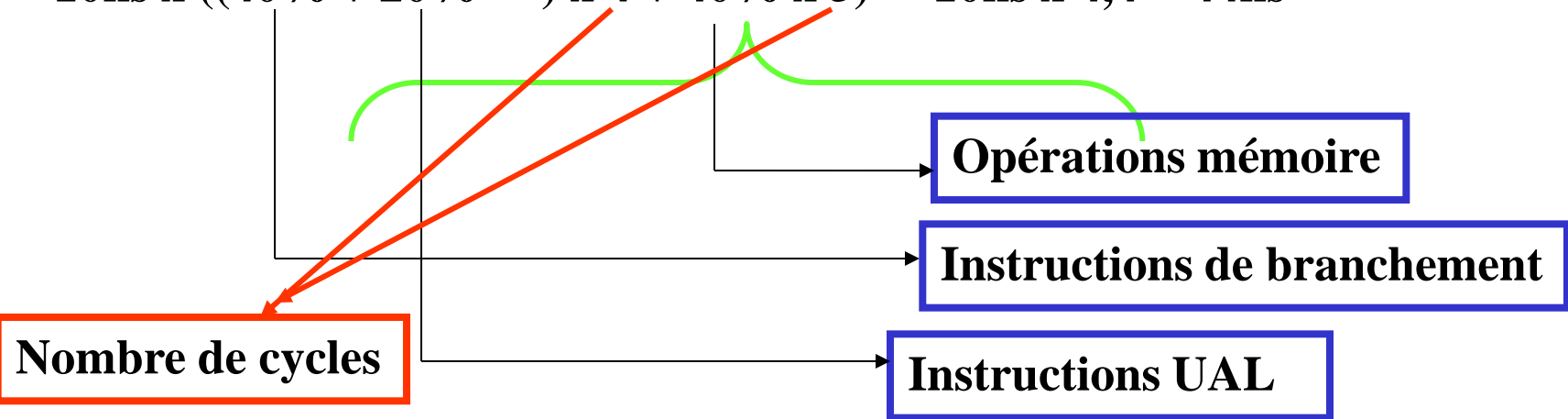
$\frac{1}{t}$  est souvent le cycle machine.



# Accélération – pipeline (1): $A_p$

Processeur séquentiel :

Temps d'exécution moyen d'une instruction = Cycle d'horloge x CPI moyen  
=  $10\text{ns} \times ((40\% + 20\%) \times 4 + 40\% \times 5) = 10\text{ns} \times 4,4 = 44\text{ns}$



Processeur pipeline

→ cycle d'horloge 11ns (10+1)

→ vitesse de l'étage le plus long

$$A_p = \frac{\text{Tps moyen d'exécution d'une instruction sans pipeline}}{\text{temps moyen d'exécution d'une instruction avec pipeline}}$$

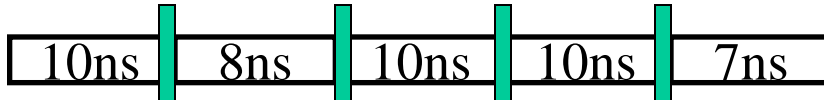
$A_p = 44\text{ns}/11\text{ns} = 4 \rightarrow$  Le processeur avec pipeline est 4 fois plus rapide que le processeur séquentiel

## Accélération – pipeline (2) :

**Processeur séquentiel : temps de cycle est égal à la somme des temps d'exécution de chaque étape :**



**Processeur pipeline : temps de cycle = temps de traversée du plus long étage + le surcoût = 10ns + 1ns = 11ns**

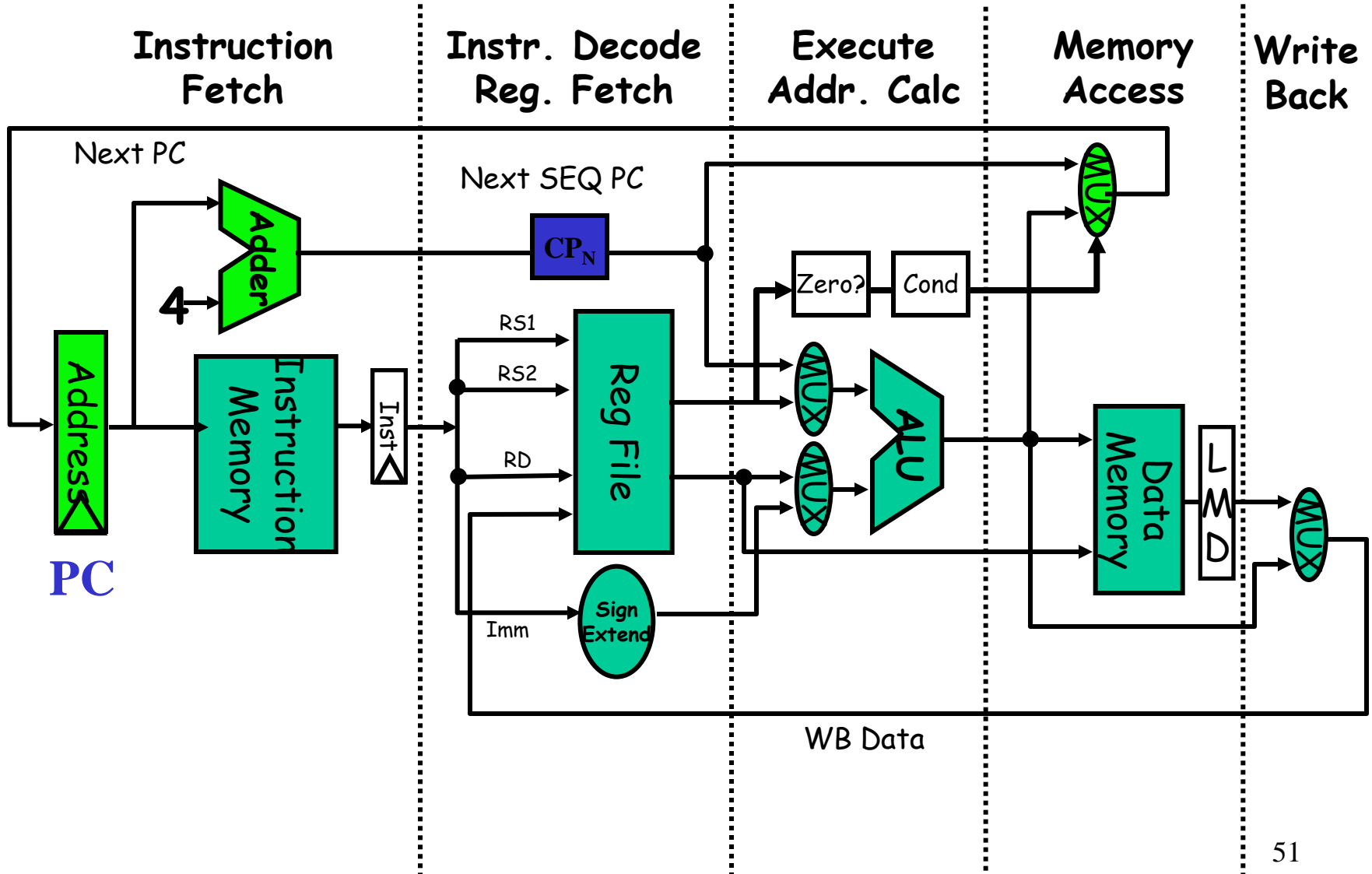


$$A_p = \frac{\text{Tps moyen d'exécution d'une instruction sans pipeline}}{\text{temps moyen d'exécution d'une instruction avec pipeline}}$$

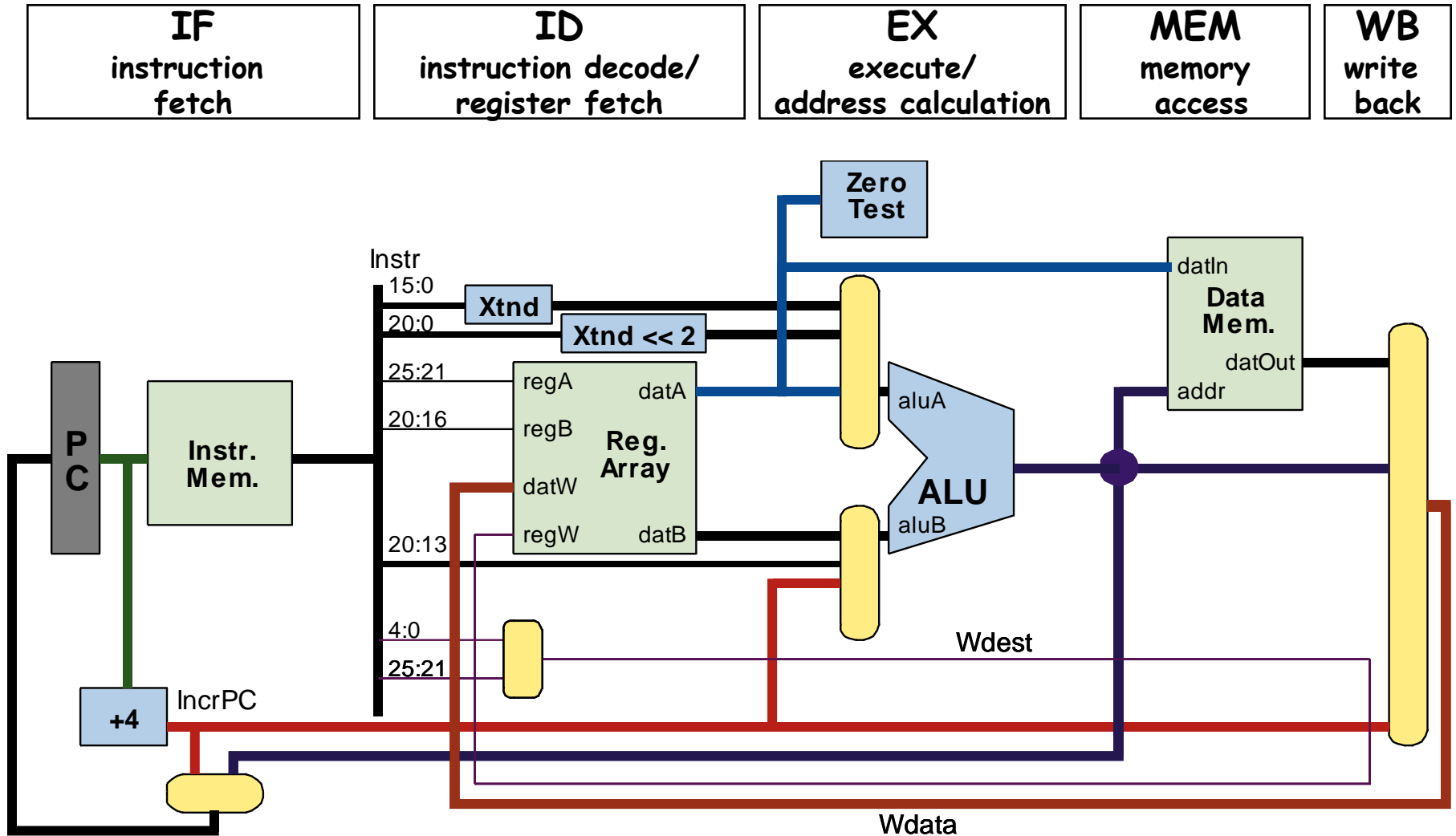
**$A_p = 4,1$  le processeur est 4,1 fois plus rapide que le processeur séquentiel  
Technique du pipeline → moyen d'améliorer le CPI, d'améliorer la fréquence d'horloge et parfois de faire les deux**

# Chemin de données (Datapath) du processeur

## DLX : 5 étapes



# Chemin de données (Datapath)



# Les différents cycles des instructions DLX

## 1. Cycle de lecture de l'Instruction : IF

$$RI \leftarrow M[CP]$$

$$CP_N \leftarrow CP + 4$$

## 2. Cycle de décodage de l'instruction/lecture registre : ID

$$A \leftarrow \text{Regs } [RI_6 - 10]$$

$$B \leftarrow \text{Regs } [RI_{11} - 16]$$

$$\text{Imm} \leftarrow ((RI_{16}))^{16} \text{## } RI_{16} - 11)$$

## 3. Cycle d'exécution/adresse effective : EX

### 3.1 Accès mémoire :

$$\text{Sortie UAL} \leftarrow A + \text{Imm}$$

Instruction UAL Registre – Registre

$$\text{Sortie UAL} \leftarrow A \text{ op } B$$

### 3.2 Instruction UAL registre-Immédiat

$$\text{Sortie UAL} \leftarrow A \text{ op Imm}$$

### 3.3 Branchement

$$\text{Sortie UAL} \leftarrow CP_N + \text{Imm}$$

$$\text{Cond.} \leftarrow (A \text{ op } 0)$$

# Les différents cycles des instructions DLX

## 4. Cycle d'accès mémoire et de fin de branchement : MEM

### 4.1 Accès mémoire :

$\text{LMD} \leftarrow \text{Mem} [\text{SortieUAL}]$  ou  $\text{Mem} [\text{SortieUAL}] \leftarrow \text{B}$

### 4.2 Branchement :

Si (cond.)  $\text{CP} \leftarrow \text{SortieUAL}$  sinon  $\text{CP} \leftarrow \text{CP}_N$

## 5. Cycle d'écriture du résultat : WB

### 5.1 Instruction UAL Registre-Registre :

$\text{Regs} [\text{RI}_{16} - 20] \leftarrow \text{SortieUAL}$

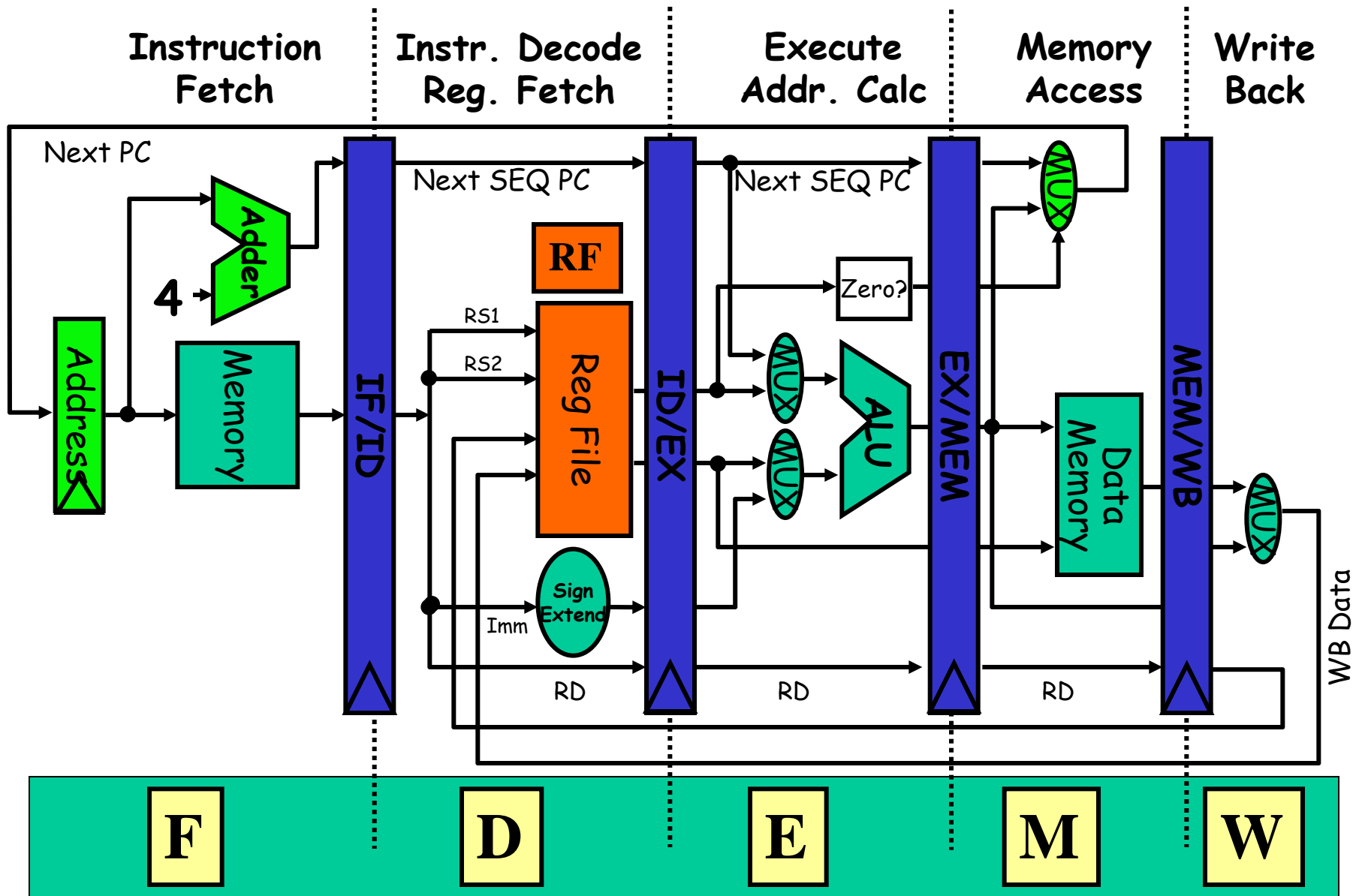
### 5.2 Instruction UAL Registre-Immédiat :

$\text{Regs} [\text{RI}_{11} - 15] \leftarrow \text{SortieUAL}$

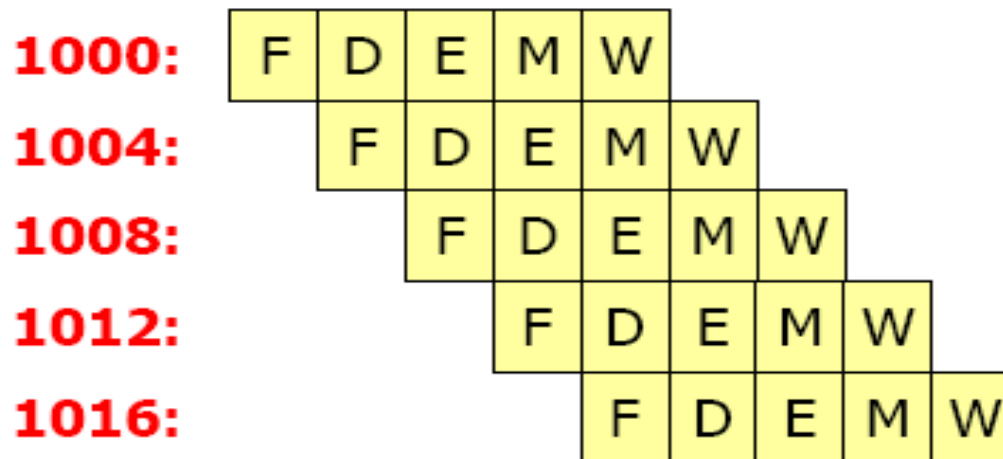
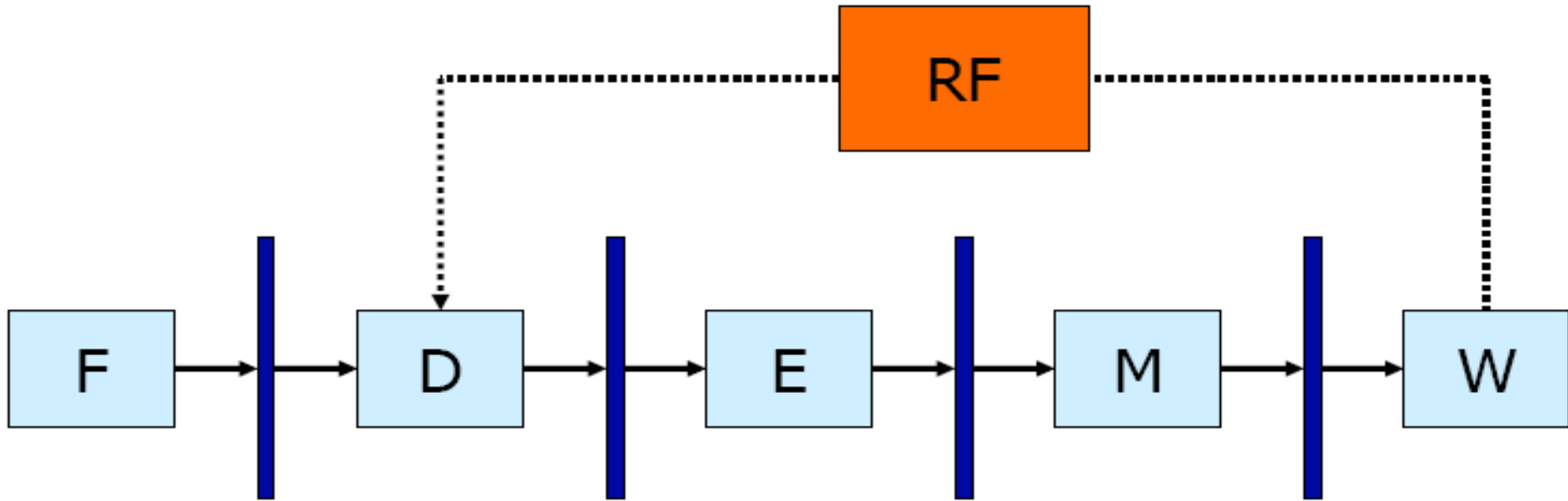
### 5.3 Instruction de chargement :

$\text{Regs} [\text{RI}_{11} - 15] \leftarrow \text{LMD}$

# Pipeline de base du DLX

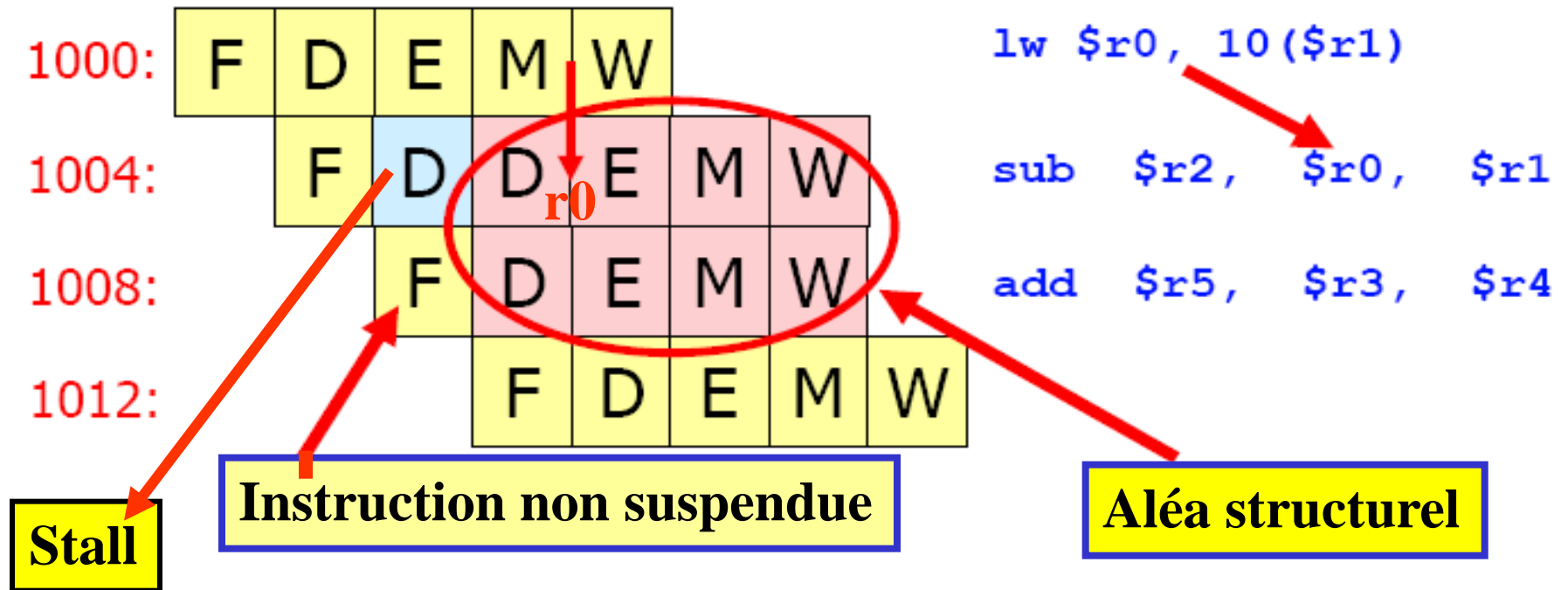


# Exécution des Instructions : évolution du Pipeline



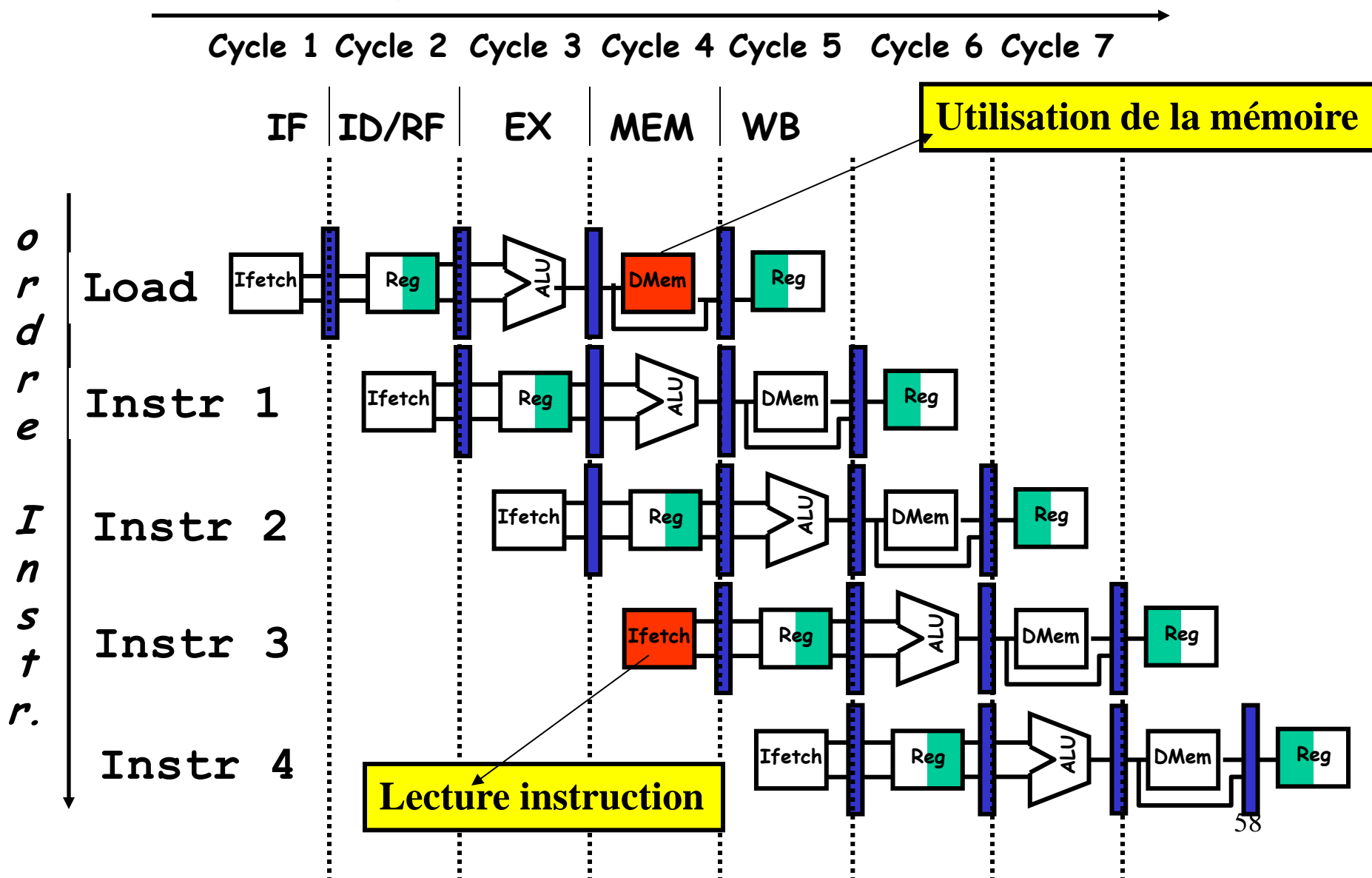


# Aléa structurel : conflit de ressources



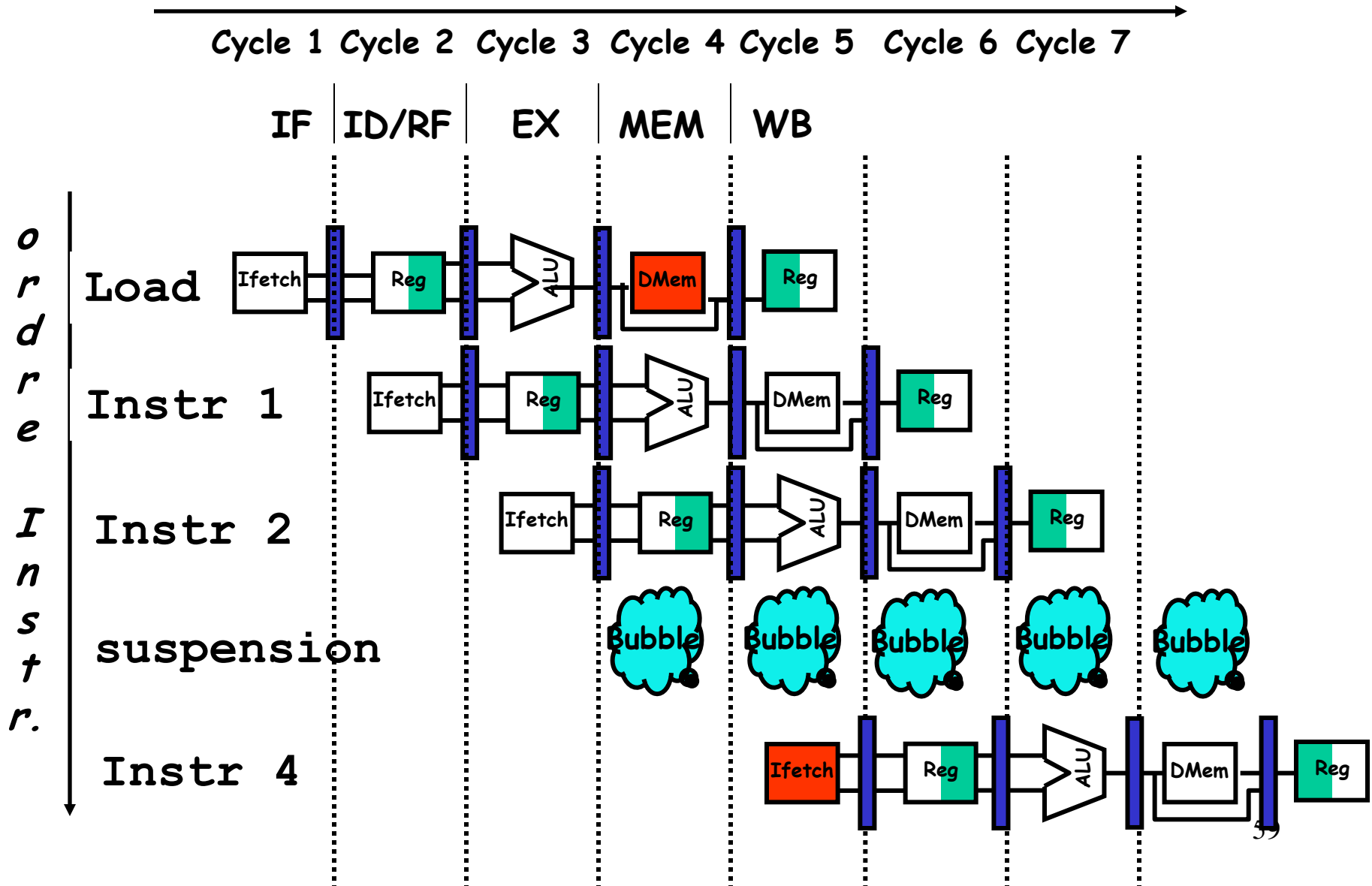
# Aléa structurel : un seul port mémoire

*Time (clock cycles)*



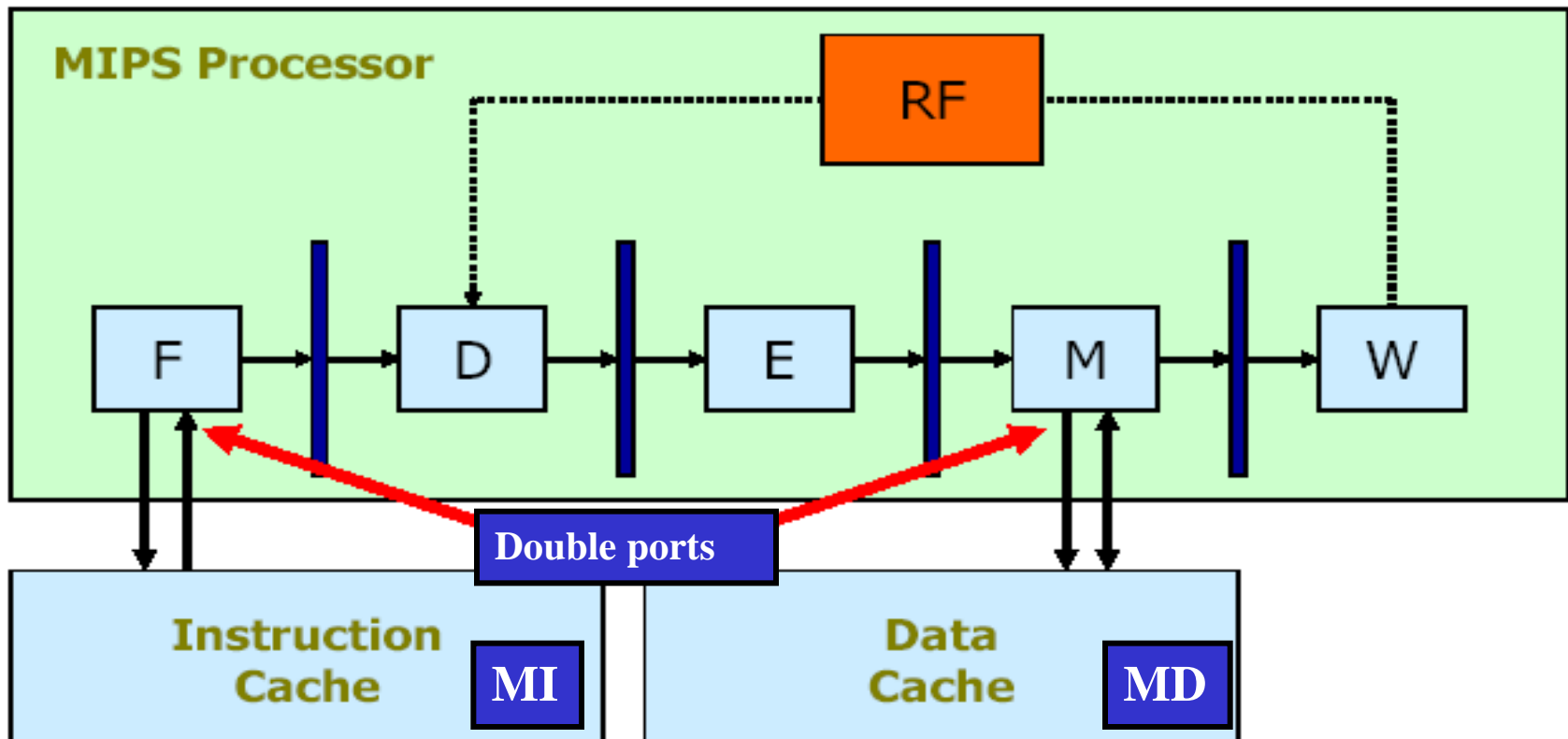
# Aléa structurel : un seul port mémoire

*Time (clock cycles)*

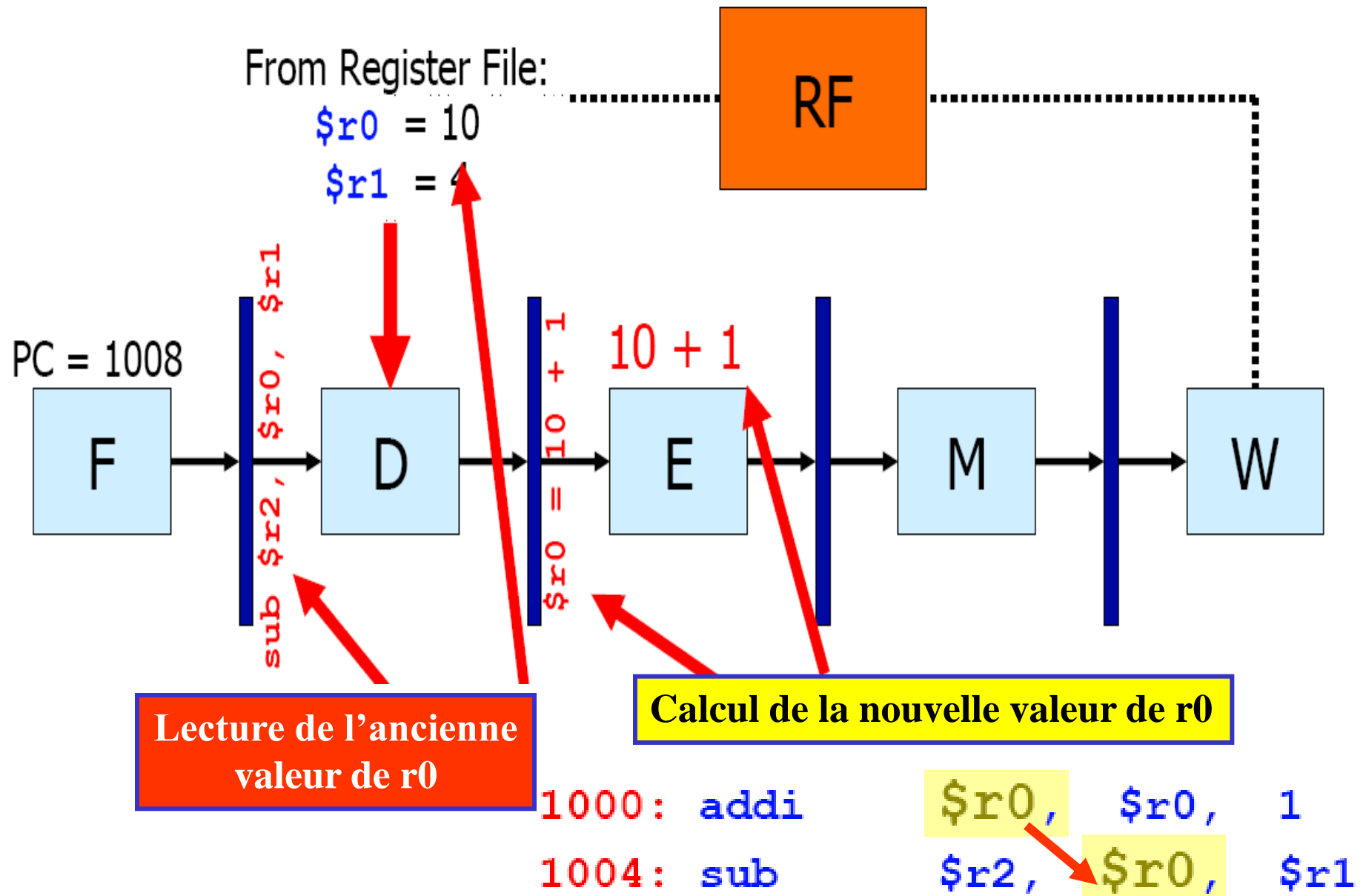


# Aléa structurel - un seul port mémoire :

**Solution → architecture Harvard : double ports : MI et MD**

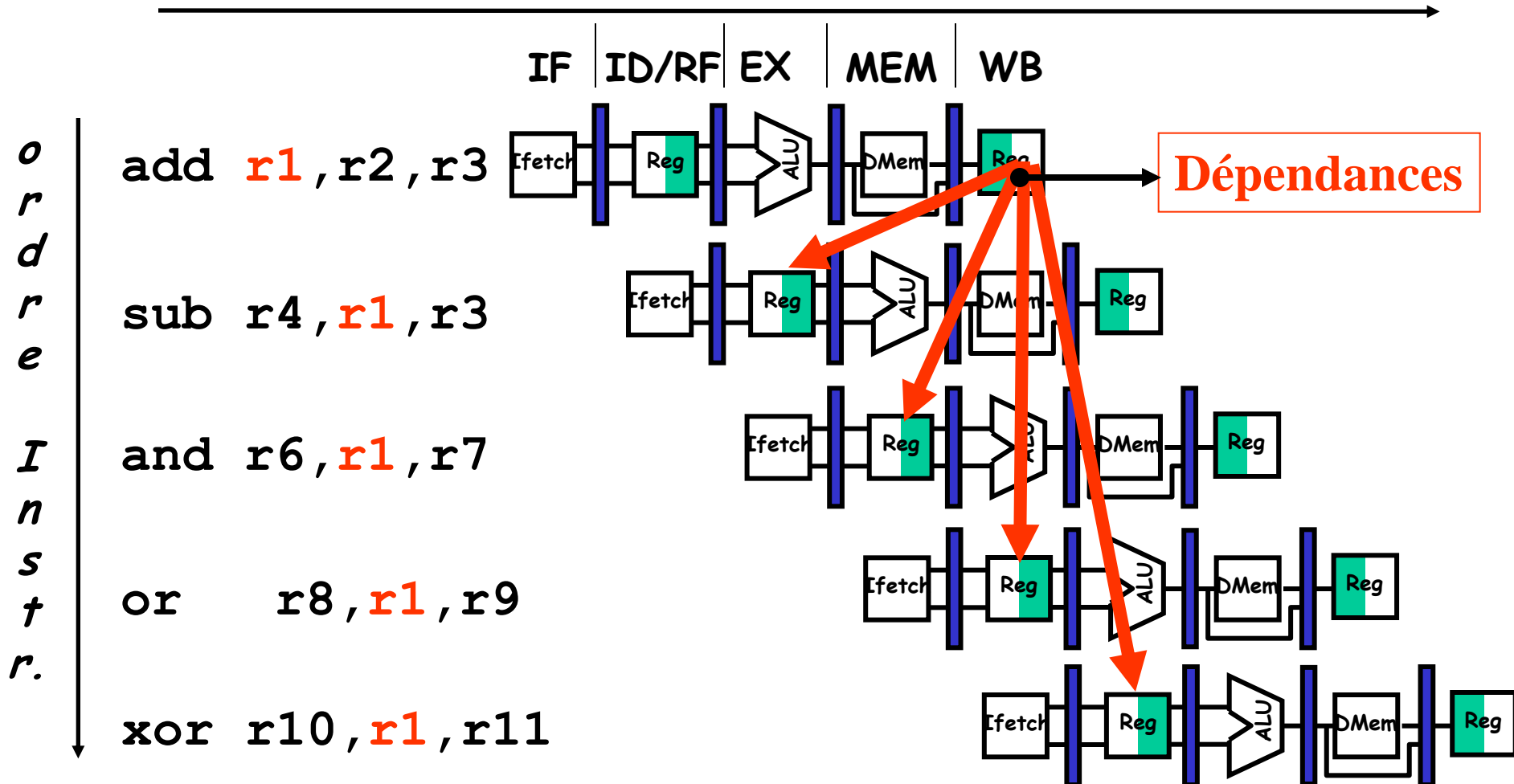


# Aléa de Données : exemple type lecture avant écriture (RAW)



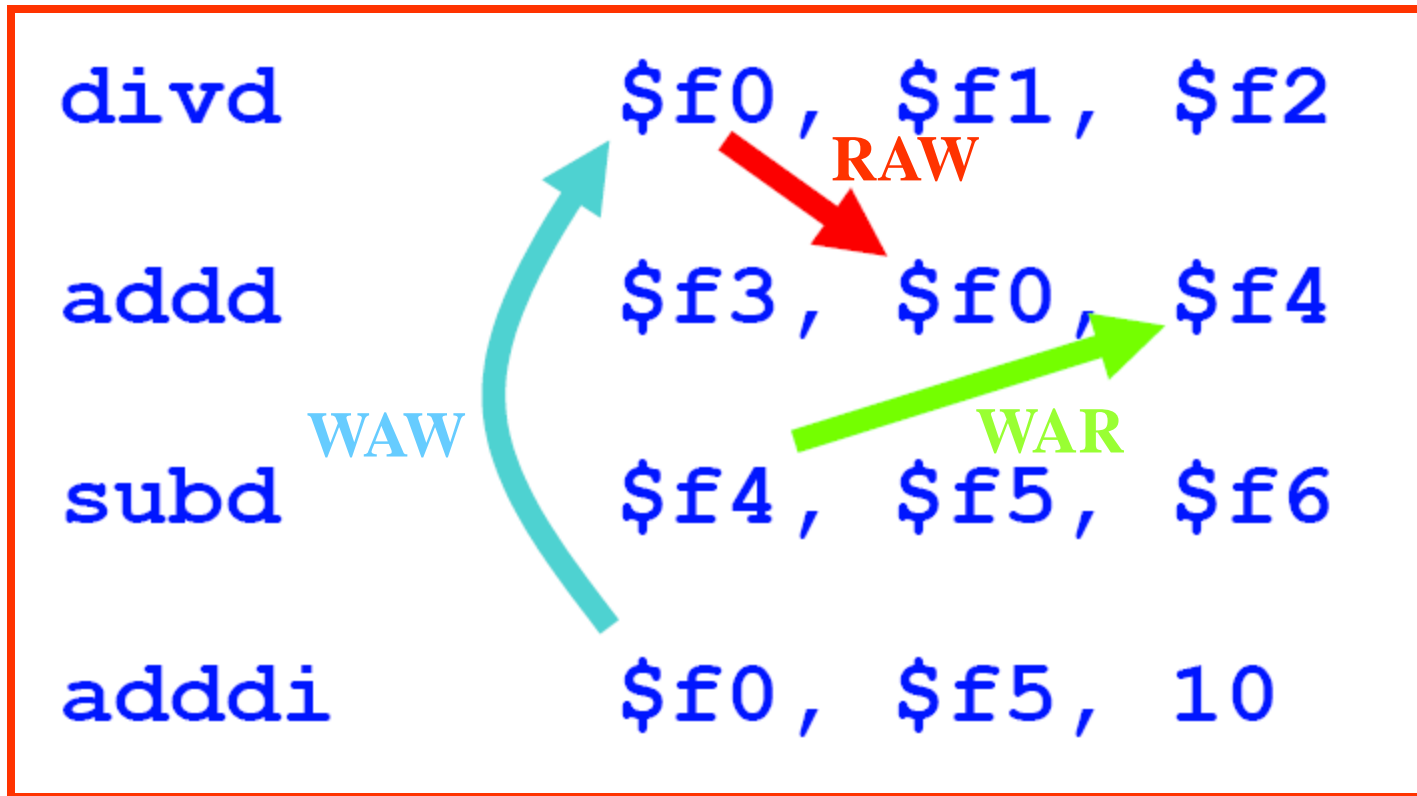
# Les aléas de données sur r1

*Time (clock cycles)*



**Lecture de r1 avant son écriture par l'instruction add**

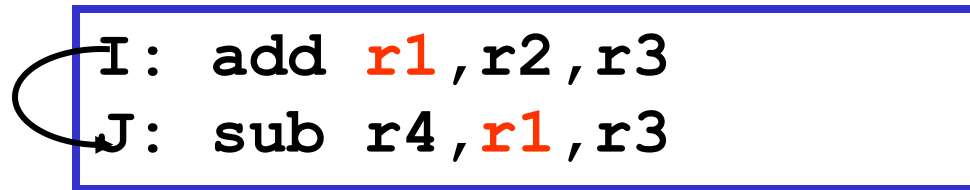
# Différentes types d'aléas de Données



<b>RAW :</b>	<b>Read After Write</b>
<b>WAR :</b>	<b>Write After Read</b>
<b>WAW :</b>	<b>Write After Write</b>
<b>RAR :</b>	<b>Read After Read (n'est pas un aléa)</b>

# Dépendance de type RAW

- Instr<sub>J</sub> présente une **dépendance de donnée** par rapport à l'Instr<sub>I</sub>. L'Instr<sub>J</sub> essaie de lire **r1** avant que l'Instr<sub>I</sub> ne l'écrive.

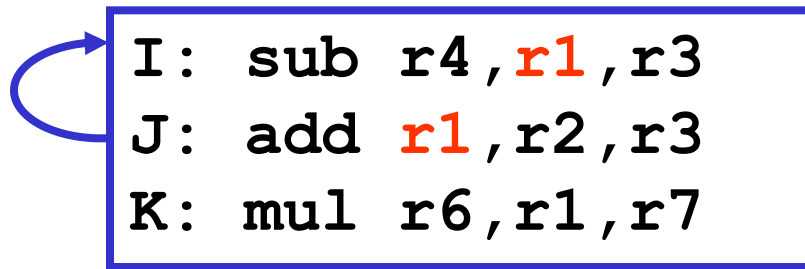


- Cette dépendance est appelée une “**dépendance vraie**” (True Dependence) en compilation.



# Dépendance de type WAR

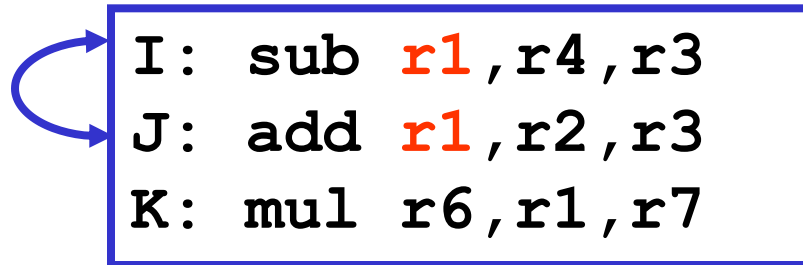
- Instr<sub>J</sub> écrit dans **r1** avant que l'Instr<sub>I</sub> ne le lit.



- Elle est appelée “**anti-dépendance**” en compilation.

# Dépendance de type WAW

- Instr<sub>J</sub> écrit **r1** *avant* que l'Instr<sub>I</sub> ne l'écrit.



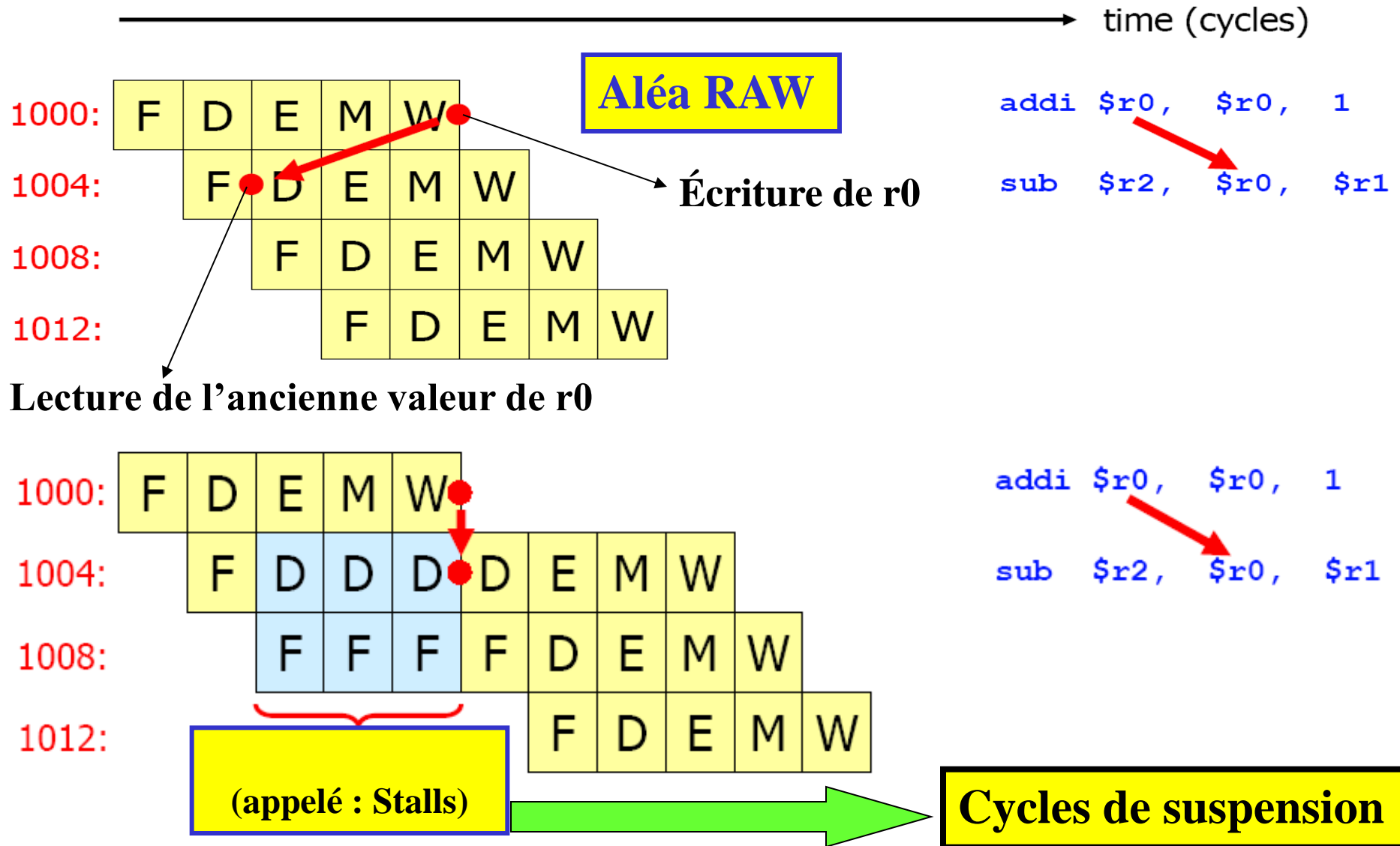
- Elle est appelée “dépendance de sortie (**output dependence**) en compilation.

# Résumé : Types de Dépendances

- **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 n'ait été utilisée (fausse dépendance).
- **Pipeline simple** : toutes les instructions sont exécutées dans l'ordre, on ne s'intéresse qu'aux dépendances **RAW**
- Si on décidait d'exécuter dans un autre ordre, il faudrait gérer les dépendances **WAR** et **WAW** pour être correct

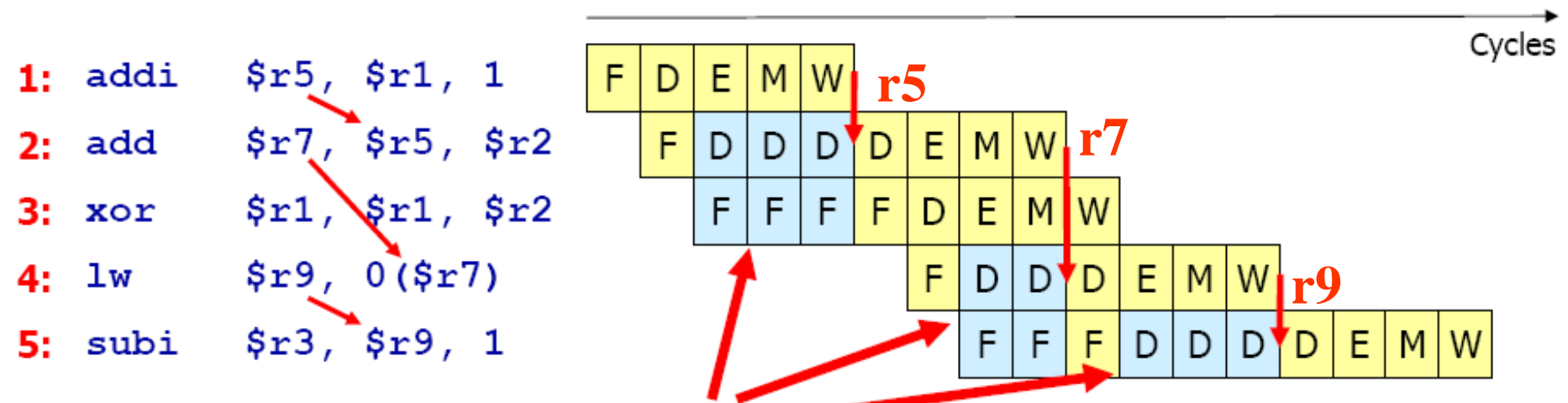
# Aléa de Données – gel du pipeline (suspension)

## Exemple 1 :



# Aléa de Données – gel du pipeline (suspension)

## Exemple 2 :



**Stalls ou NOP (No Opérations) : insérer NOP**

# Performance du pipeline avec suspension

$$\begin{aligned} A_p &= \frac{\text{temps moyen d'une instruction sans pipeline}}{\text{temps moyen d'une instruction avec pipeline}} \\ &= \frac{\text{CPI non pipeliné} \times \text{Temps de cycle non pipeliné}}{\text{CPI pipeliné} \times \text{Temps de cycle pipeliné}} \\ &= \frac{\text{CPI non pipeliné}}{\text{CPI pipeliné}} \times \frac{\text{Temps de cycle non pipeliné}}{\text{Temps de cycle pipeliné}} \end{aligned}$$

**Pipeline → Comme un moyen pour diminuer soit CPI soit le Temps de cycle**

$$\begin{aligned} \text{CPI pipeliné} &= \text{CPI idéal} + \text{Cycles de suspension du pipeline/Instruction} \\ &= 1 \swarrow + \text{Cycles de suspension du pipeline/Instruction} \end{aligned}$$

Si on néglige le surcoût de cycle lié au pipeline et si on suppose que les étages sont parfaitement équilibrés → les temps de cycle des 2 processeurs peuvent être égaux :

$$A_p = \text{CPI non pipeliné} / 1 + \text{Cycles de suspension du pipeline / instruction}$$

# Performance du pipeline avec suspension

Cas - toutes les instructions prennent le même nombre de cycles :

→ CPI non pipeliné = profondeur du pipeline ( soit  $P_p$  )

→  $A_p = P_p / (1 + \text{cycles de suspension du pipeline/instruction})$

**S'il n'y a pas de suspensions du pipeline alors  $A = P_p$**

Si le pipeline est un moyen d'amélioration du temps de cycle, alors on peut supposer que CPI non pipeliné = CPI pipeline = 1

→  $A_p = \frac{\text{CPI non pipeliné} \times \text{Cycle d'h. sans pipeline}}{\text{CPI pipeliné} \times \text{Cycle d'h. avec pipeline}}$

→  $A_p = [1 / (1 + \text{Cycle de suspension du pipeline/Instruction})] \times [\text{Cycle d'h. sans pipeline} / \text{Cycle d'h. avec pipeline}]$

# Performance du pipeline avec suspension

Cas où les étages sont parfaitement équilibrés (pas de suroût) alors le temps de cycle du processeur pipeliné est plus petit que celui du processeur séquentiel (processeur non pipeline) :

Temps de cycle pipeliné = temps de cycle non pipeliné /  $P_p$

$P_p$  = temps de cycle non pipeliné / temps de cycle pipeliné

Si CPI idéal est de 1 mais les fréquences d'horloge diffèrent :

$A_p = [1 / (1 + \text{cycles de susp. du pipeline} / \text{Instr.})] \times (\text{Cycles d'h. sans pipeline} / \text{Cycle d'h. avec pipeline})$

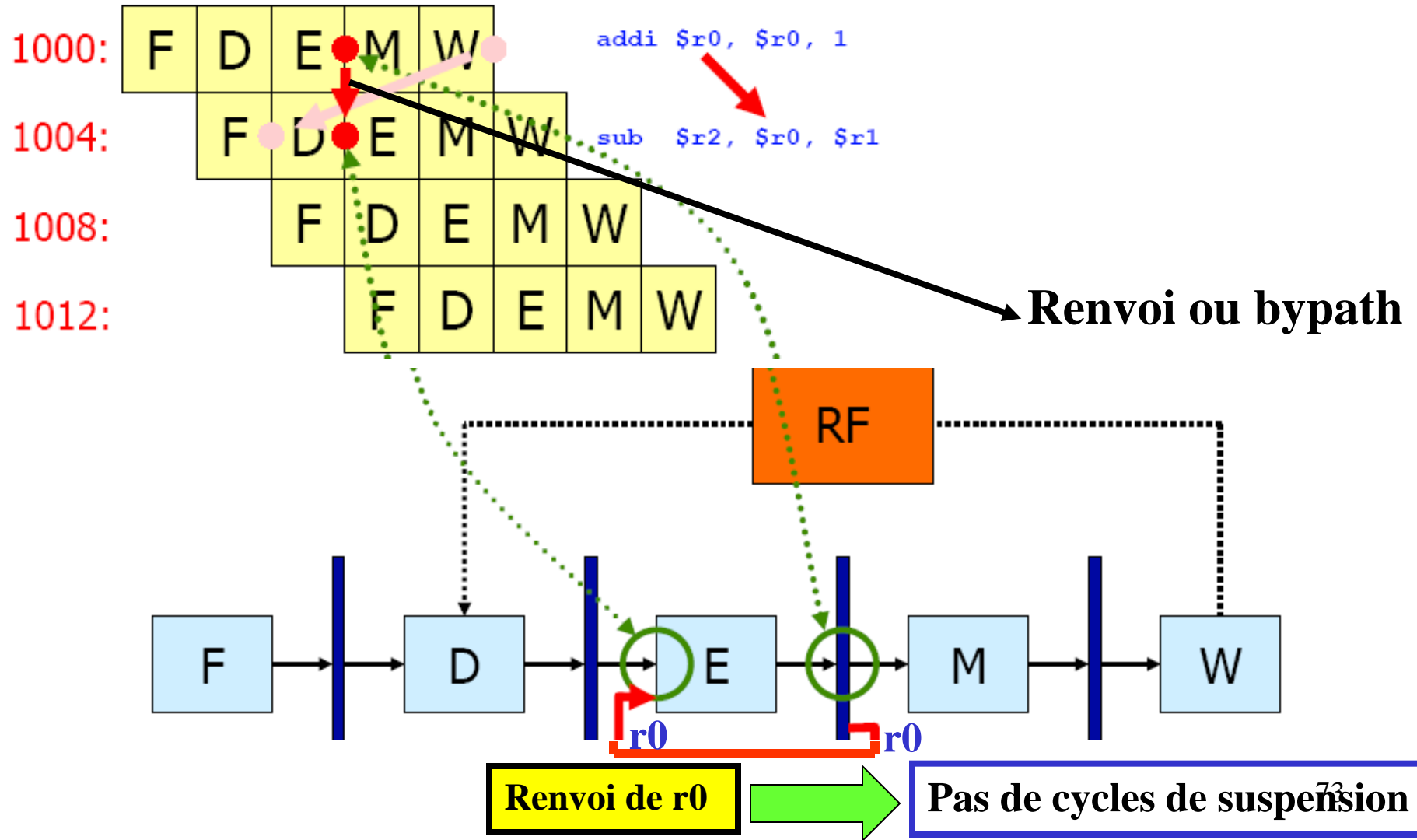
$= [1 / (1 + \text{cycles de susp. du pipeline} / \text{Instr.})] \times P_p$

Ainsi s'il n'a pas de suspensions  $A = P_p$

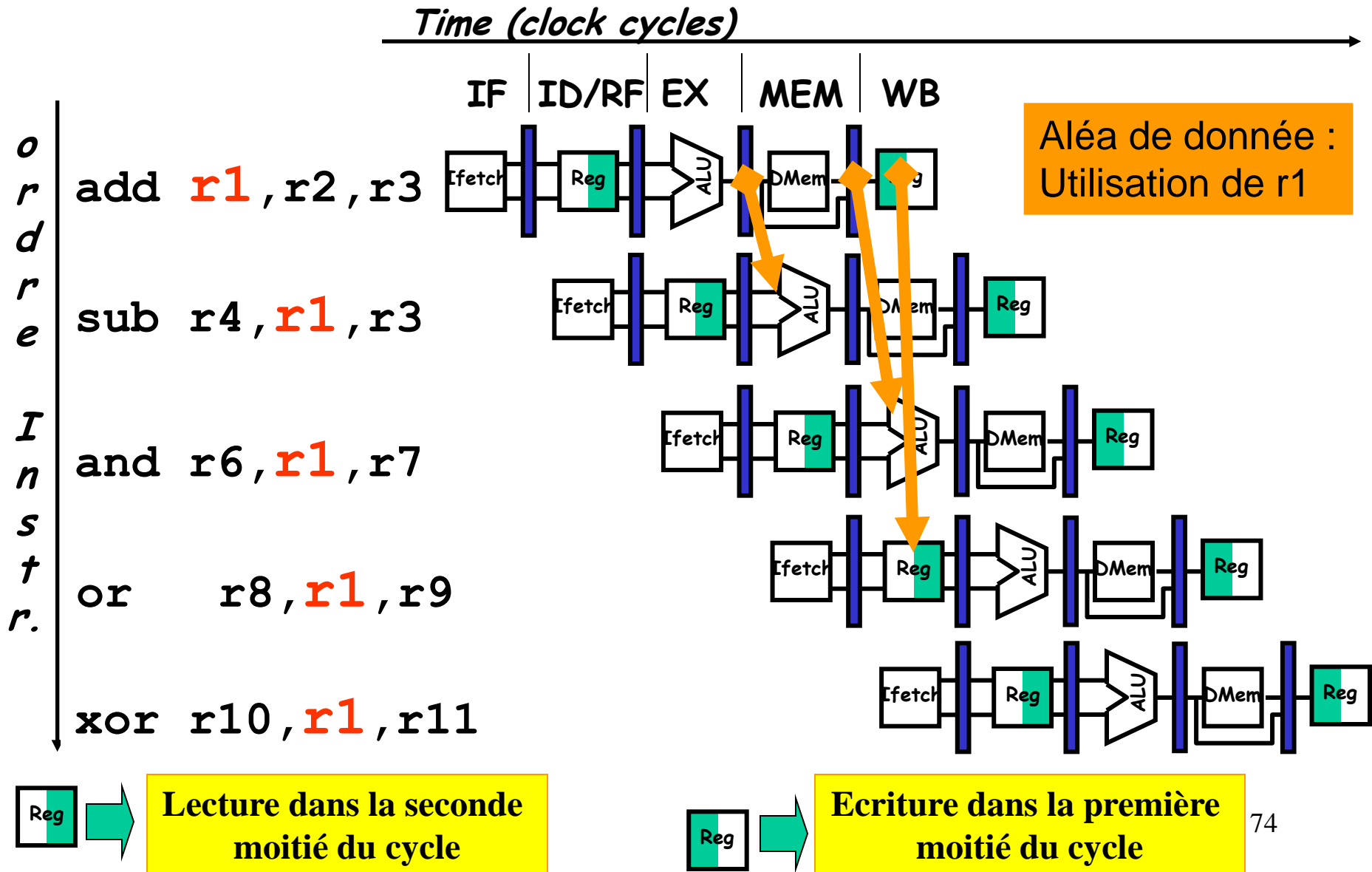


# Amélioration - Aléa de Données : création – renvoi (bypass)

- solution matérielle : Ajout de chemins supplémentaires (renvois)

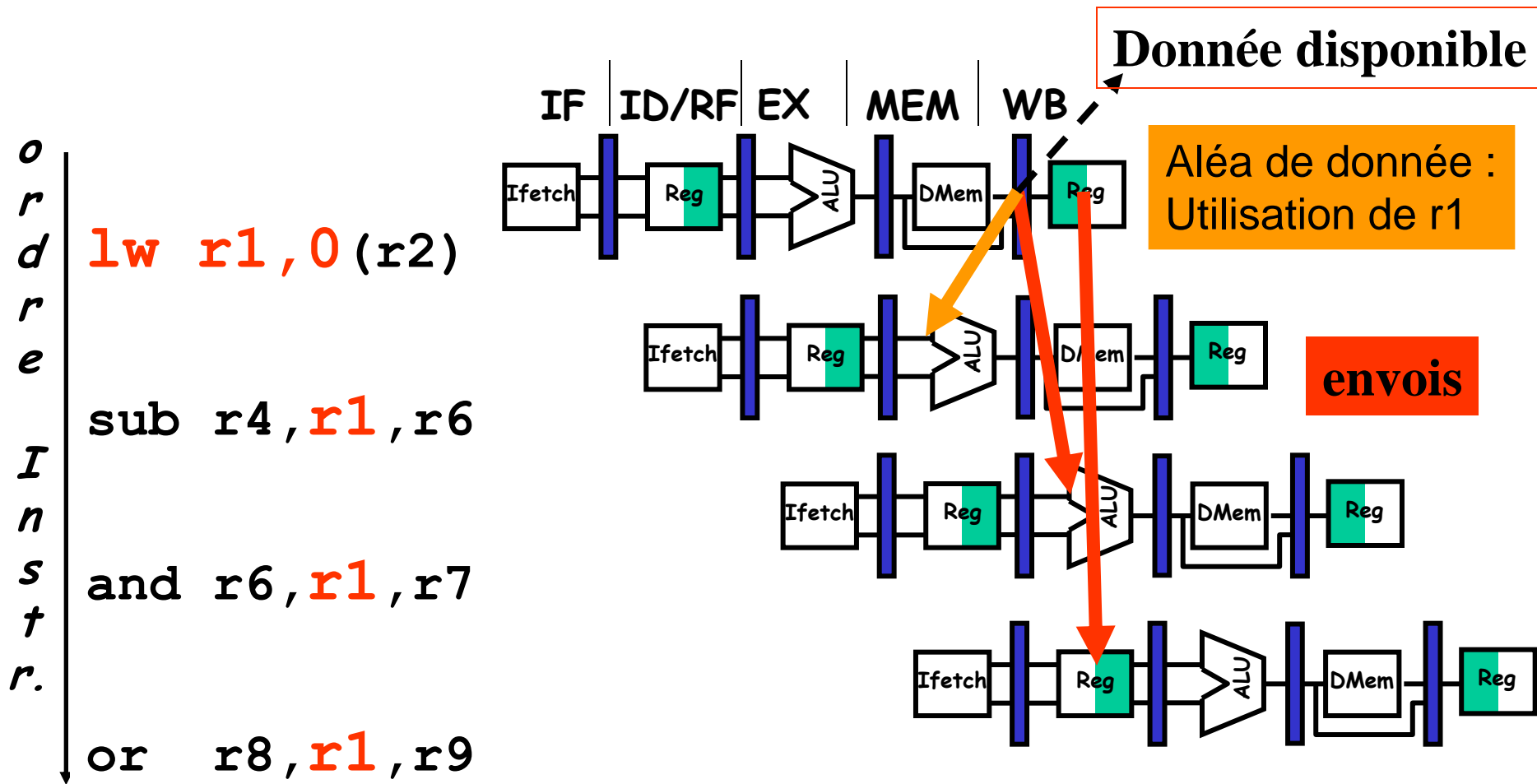


# Amélioration - Aléa de données - solution : utilisation de chemins de renvoi (Forwarding)



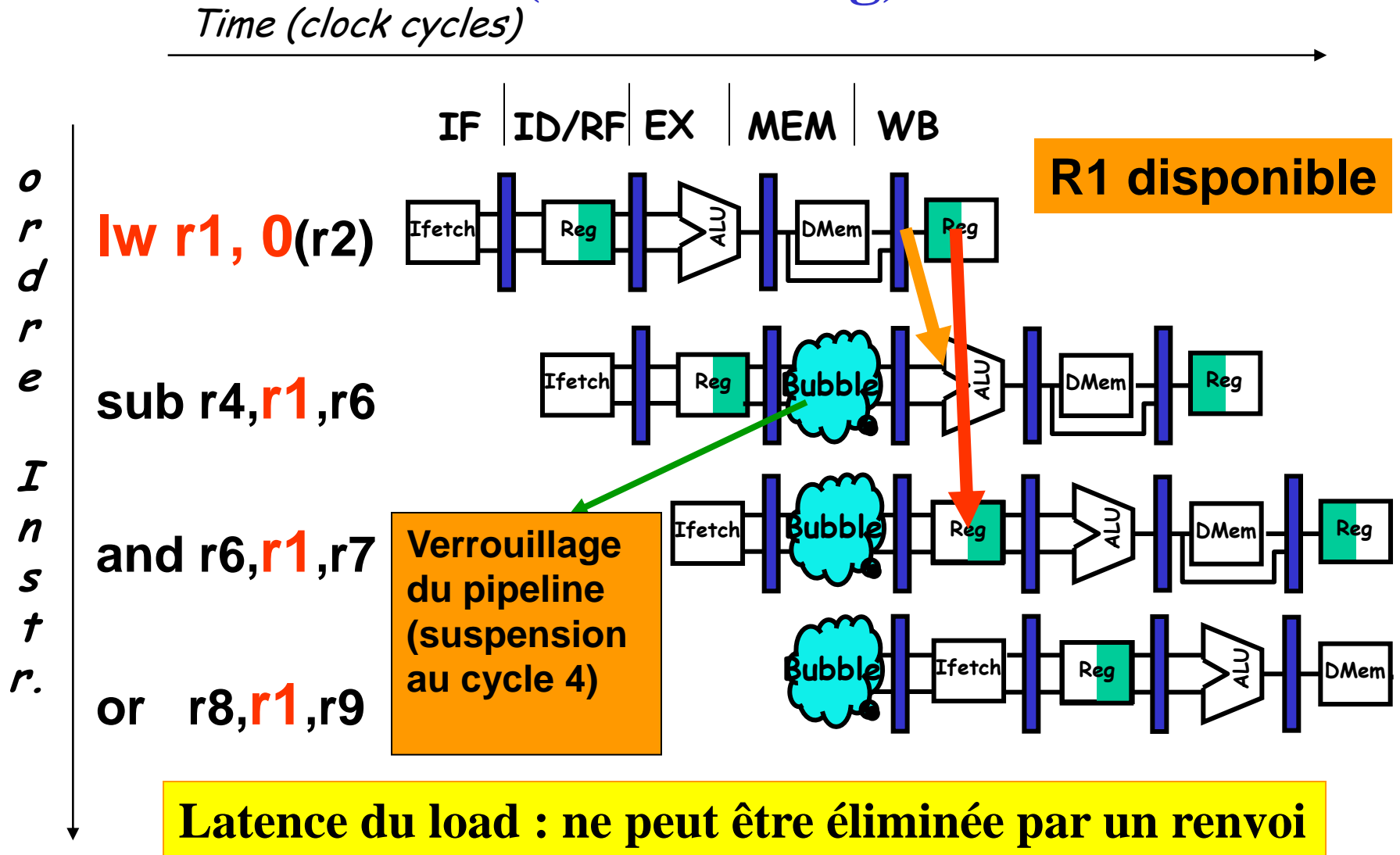
# Solution matérielle : verrouillage du pipeline + chemins de renvoi (forwarding paths)

*Time (clock cycles)*



Pour `sub` le résultat envoyé arrive trop tard (fin du cycle). Le délai ou latence du load ne peut être éliminé par le seul envoi → Ajout d'un verrouillage du pipeline

# Aléa de données - solution : verrouillage du pipeline + utilisation de chemins de renvoi (Forwarding)

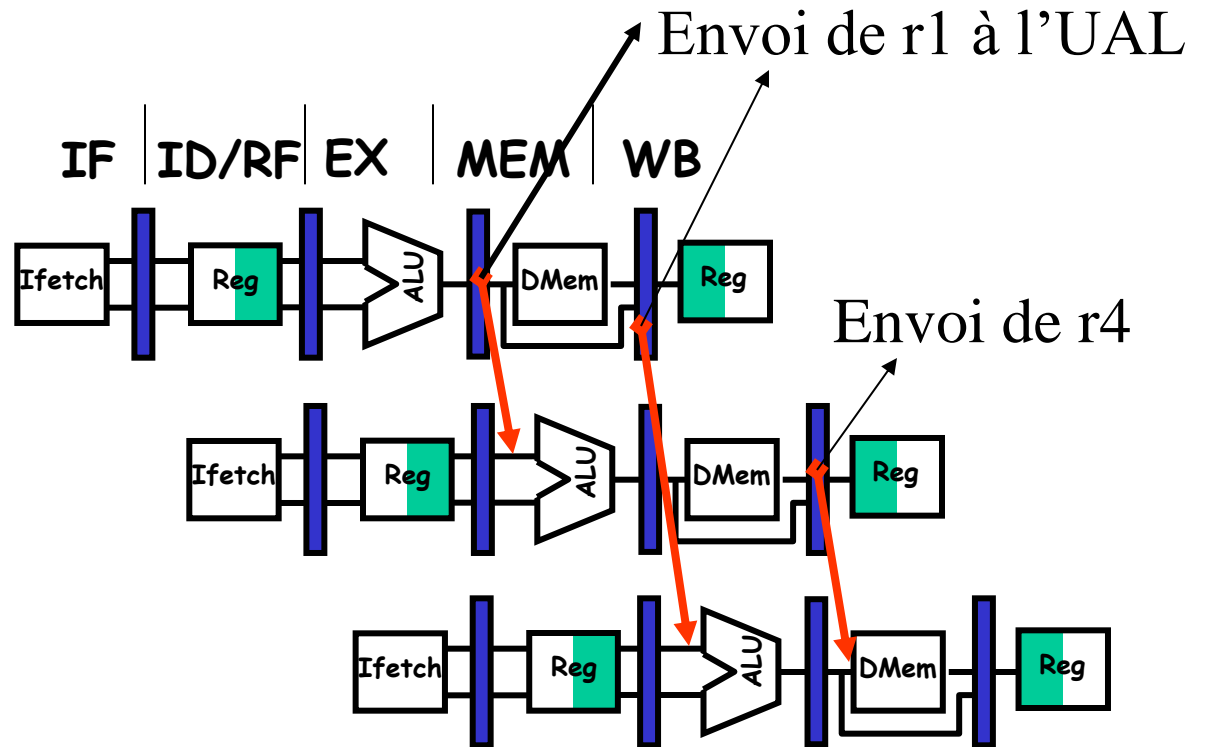


# Aléa de données - chemins de renvoi (Forwarding)

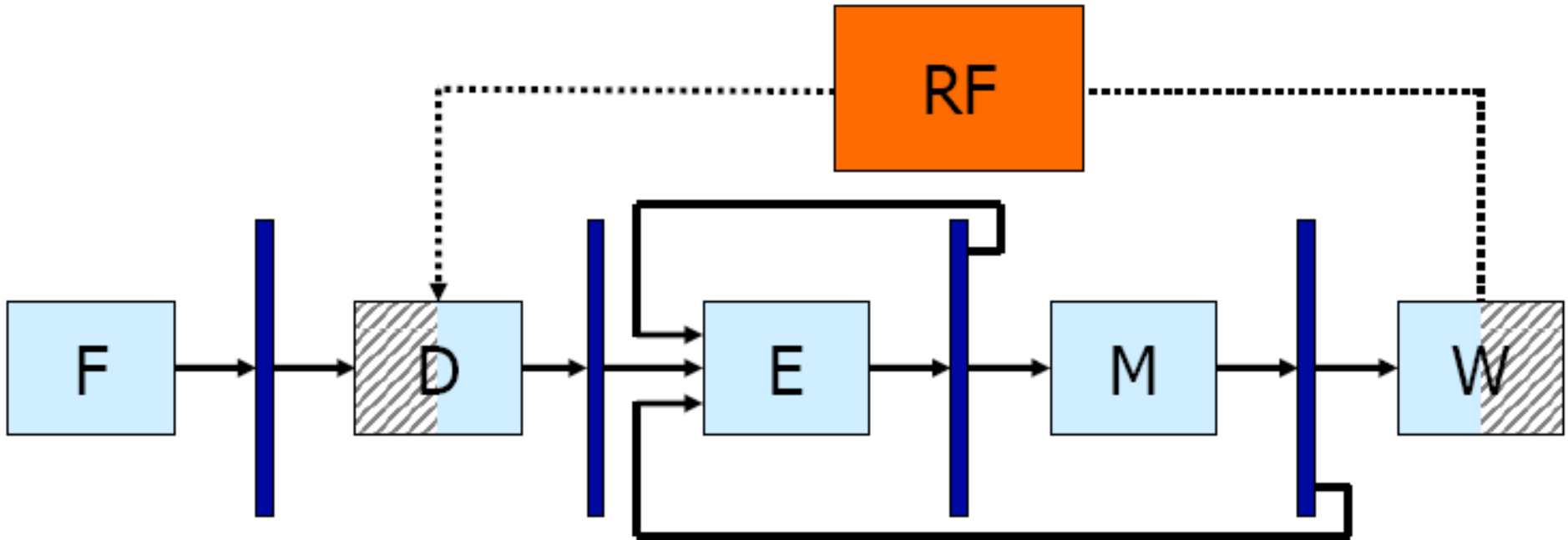
`add r1, r2, r3`

`lw r4, 0(r1)`

`sw 12(r1), r4`



# Solution matérielle : chemins de renvoi (forwarding paths)



## Forwarding paths :

- $E \rightarrow E$ ,  $M \rightarrow E$
- $W \rightarrow D$

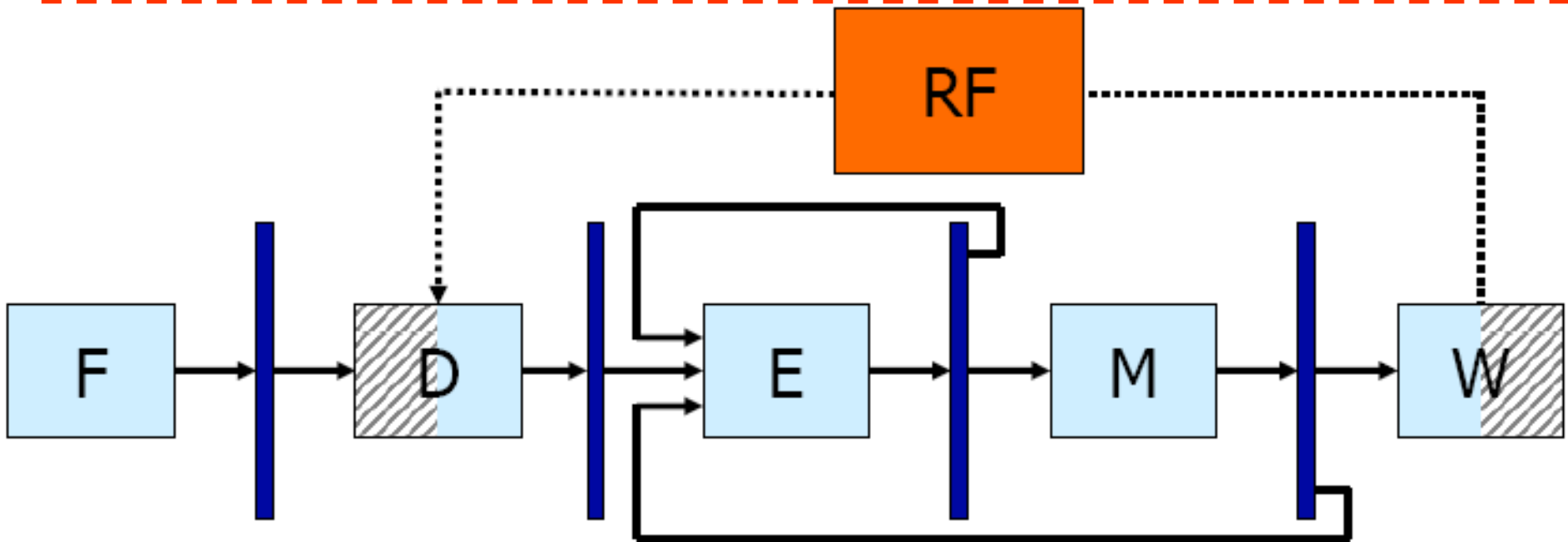
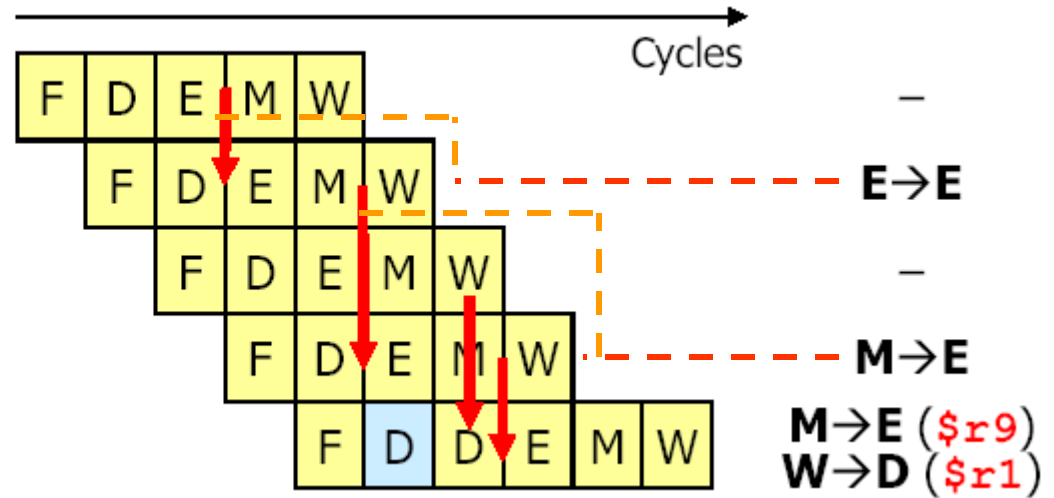
## Register-file forwarding :

- $W \rightarrow D$

- Durant W les registres sont modifiés dans la 1ère moitié du cycle
- Durant D les registres sont lus lors de la deuxième phase du cycle

# Solution matérielle : chemins de renvoi (forwarding paths)

1: addi \$r5, \$r1, 1  
2: add \$r7, \$r5, \$r2  
3: xor \$r1, \$r1, \$r2  
4: lw \$r9, 0(\$r7)  
5: sub \$r3, \$r9, \$r1



# Aléas de données – solution : réordonnancement des instructions

Le compilateur réorganise le code pour éliminer les aléas, par exemple éviter qu'un chargement (load) soit suivi par une utilisation immédiate de la valeur chargée

Soit le calcul suivant :

$a = b + c;$

$d = e - f;$

Soit les variables a, b, c, d, e, et f allouées en mémoire.

Code non optimisé :

2 blocages  
des chargements

LW Rb,b  
LW Rc,c  
ADD Ra,Rb,Rc  
SW a,Ra  
LW Re,e  
LW Rf,f  
SUB Rd,Re,Rf  
SW d,Rd

Code optimisé :

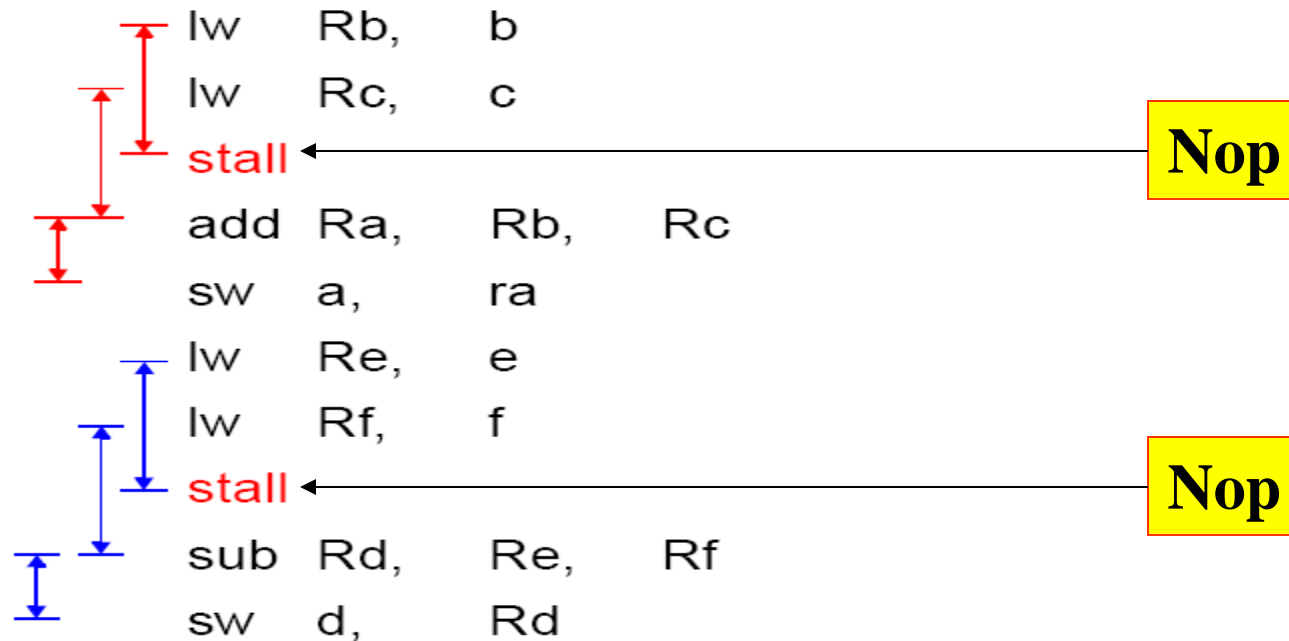
LW Rb,b  
LW Rc,c  
LW Re,e  
ADD Ra,Rb,Rc  
LW Rf,f  
SW a,Ra  
SUB Rd,Re,Rf  
SW d,Rd

Permutation de sw et lw



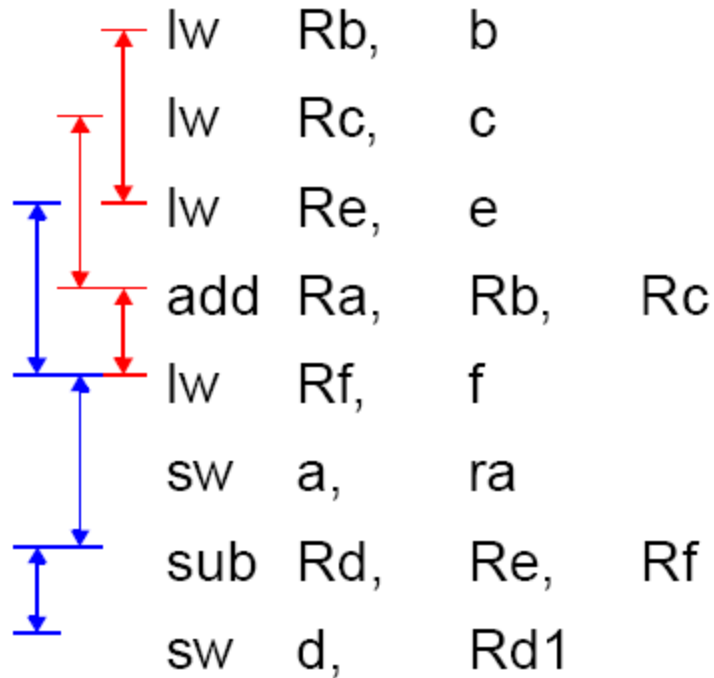
# Aléas de données – solution : réordonnancement des instructions

- $a = b + c$ ;  $d = e + f$
- -- load retardé :



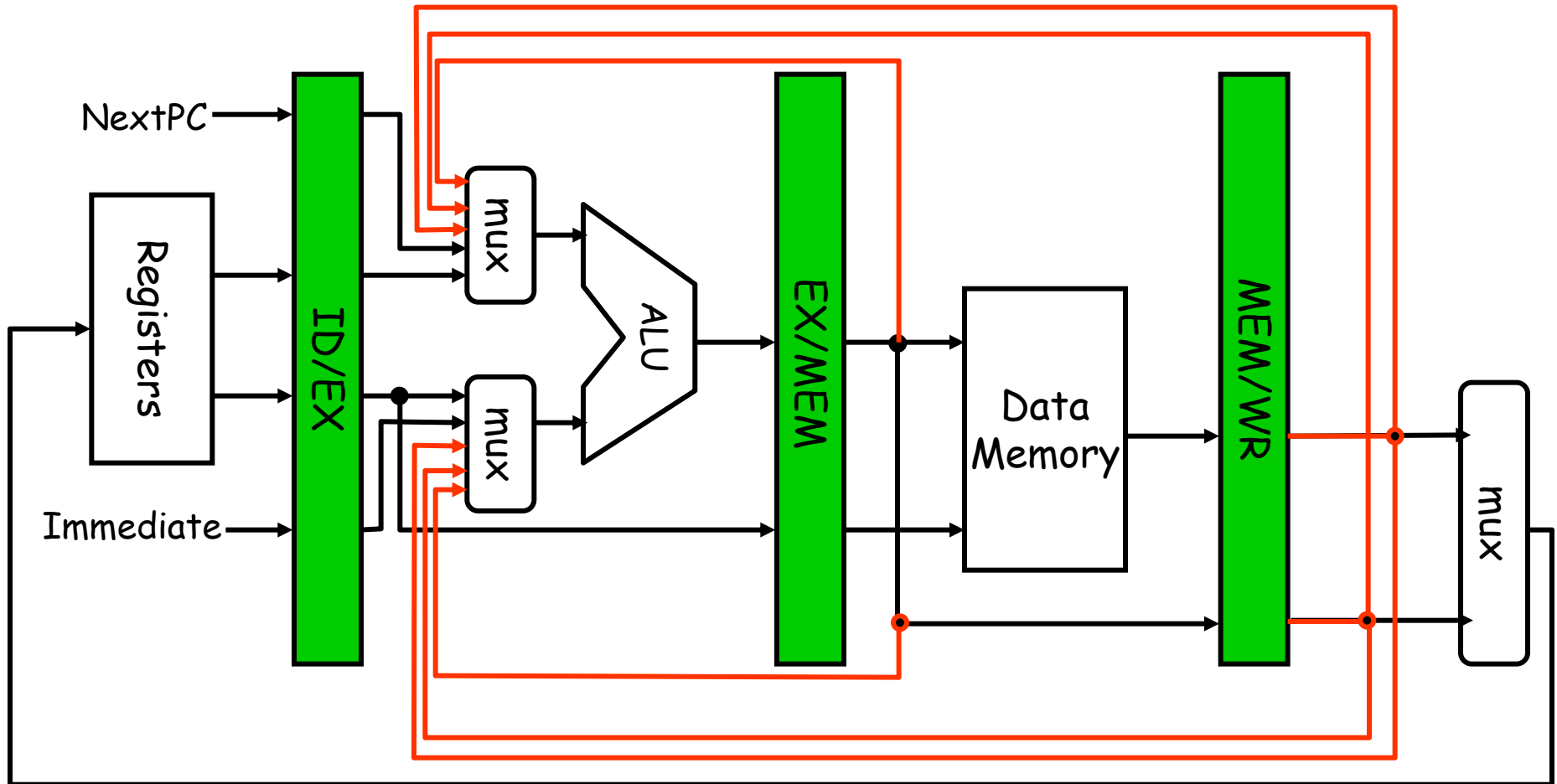
# Aléas de données – solution : réordonnancement des instructions

- Après réordonnancement :

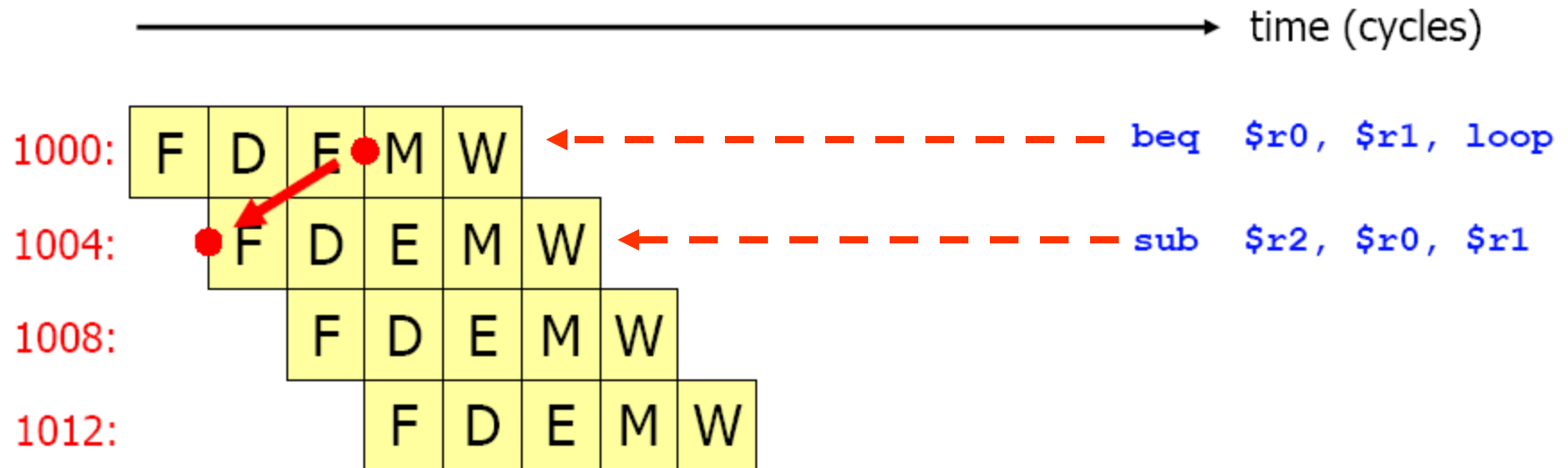


**Pas de suspensions**

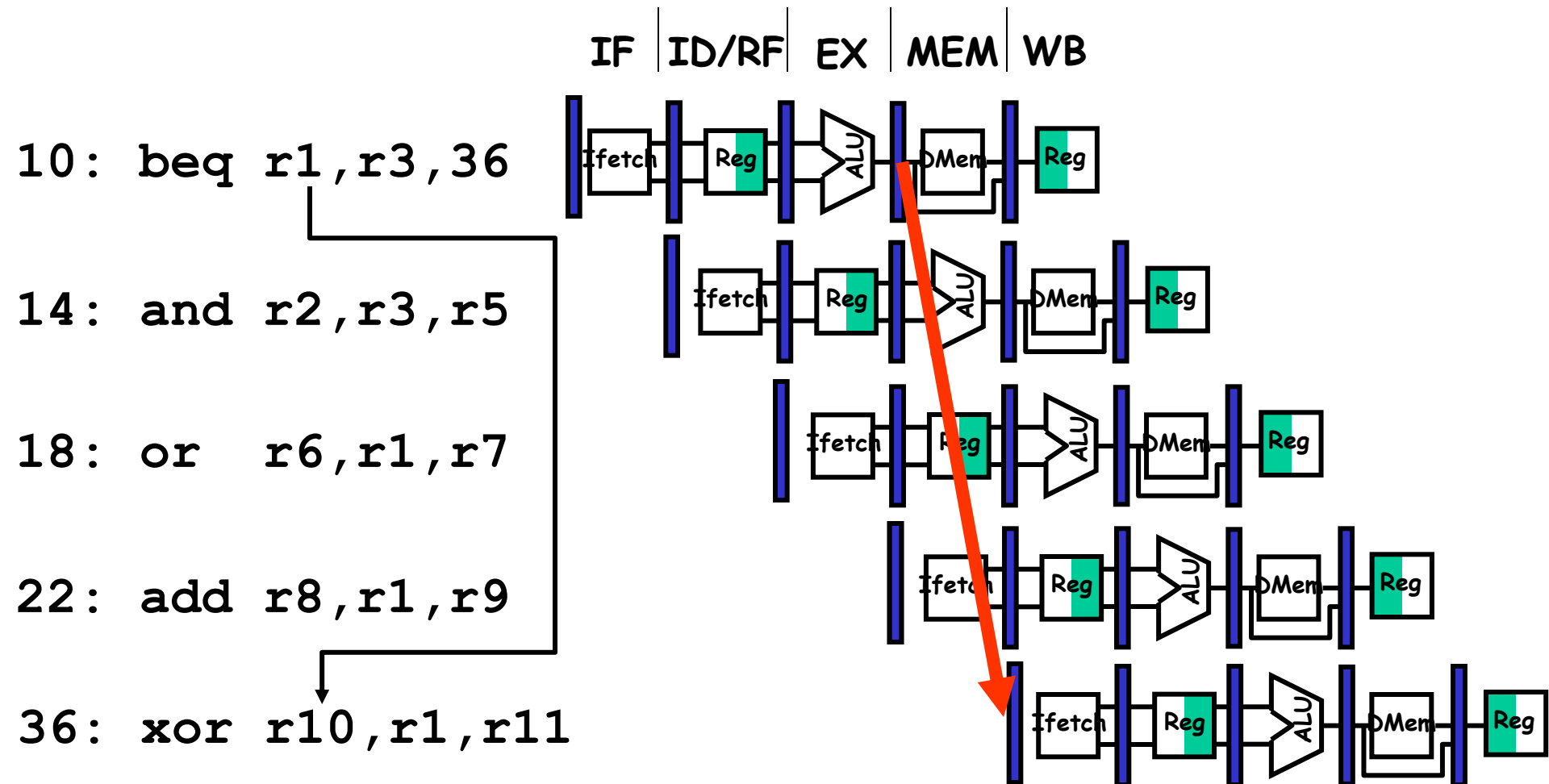
# MIPS-Datapath avec les renvois(Forwarding)



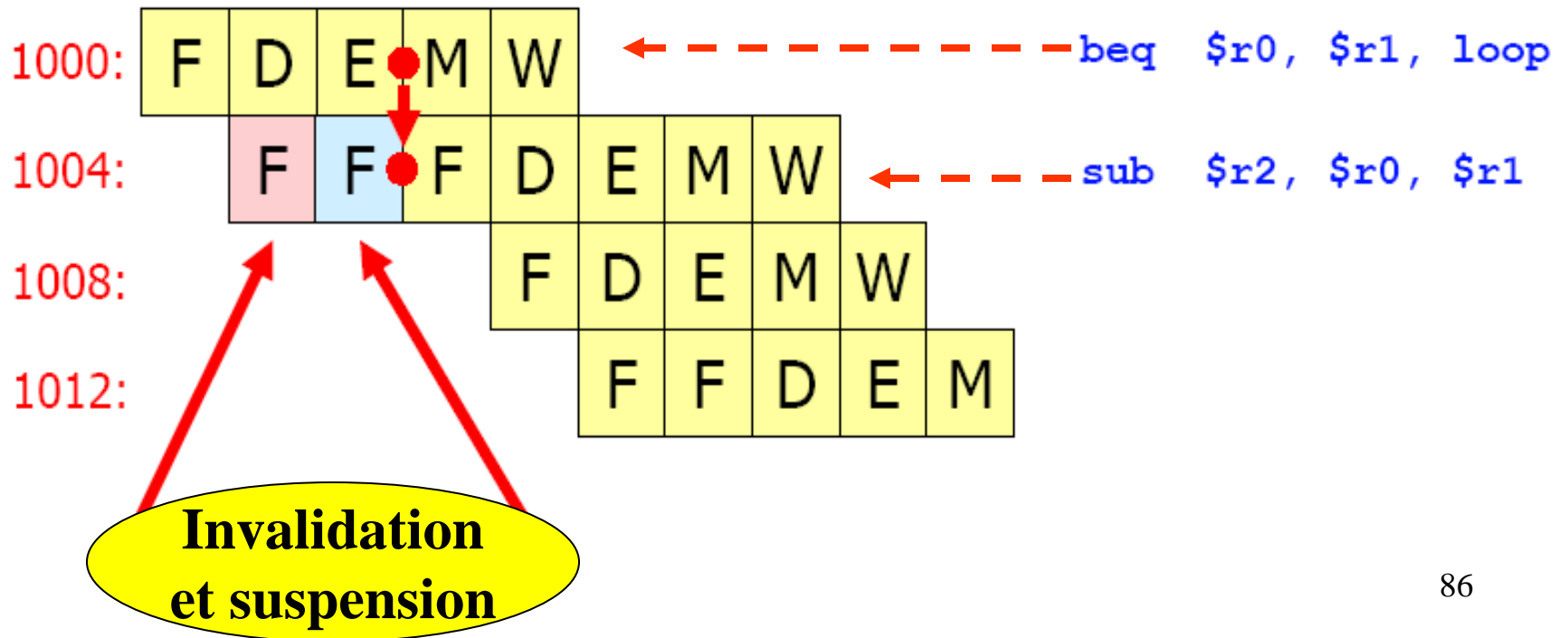
# Aléa de contrôle



# Aléa de contrôle

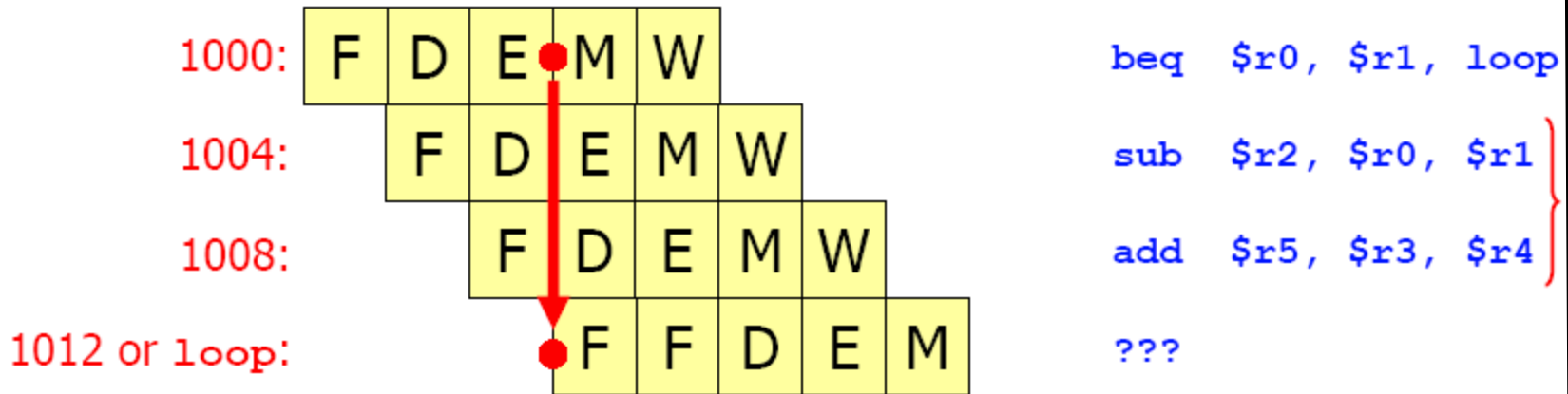


# Aléa de contrôle



# Aléa de contrôle – solutions :

## insertion de nop / réordonnancement des instructions



Instructions - délais

```

1000: sub $r2, $r0, $r7
1004: mul $r1, $r6, $r7
1008: add $r5, $r3, $r4
1012: beq $r0, $r1, loop
1016: nop
1020: nop
1024: lw $r8, 12($r9)

```

délais



```

1000: mul $r1, $r6, $r7
1004: beq $r0, $r1, loop
1008: sub $r2, $r0, $r7
1012: add $r5, $r3, $r4
1016: lw $r8, 12($r9)

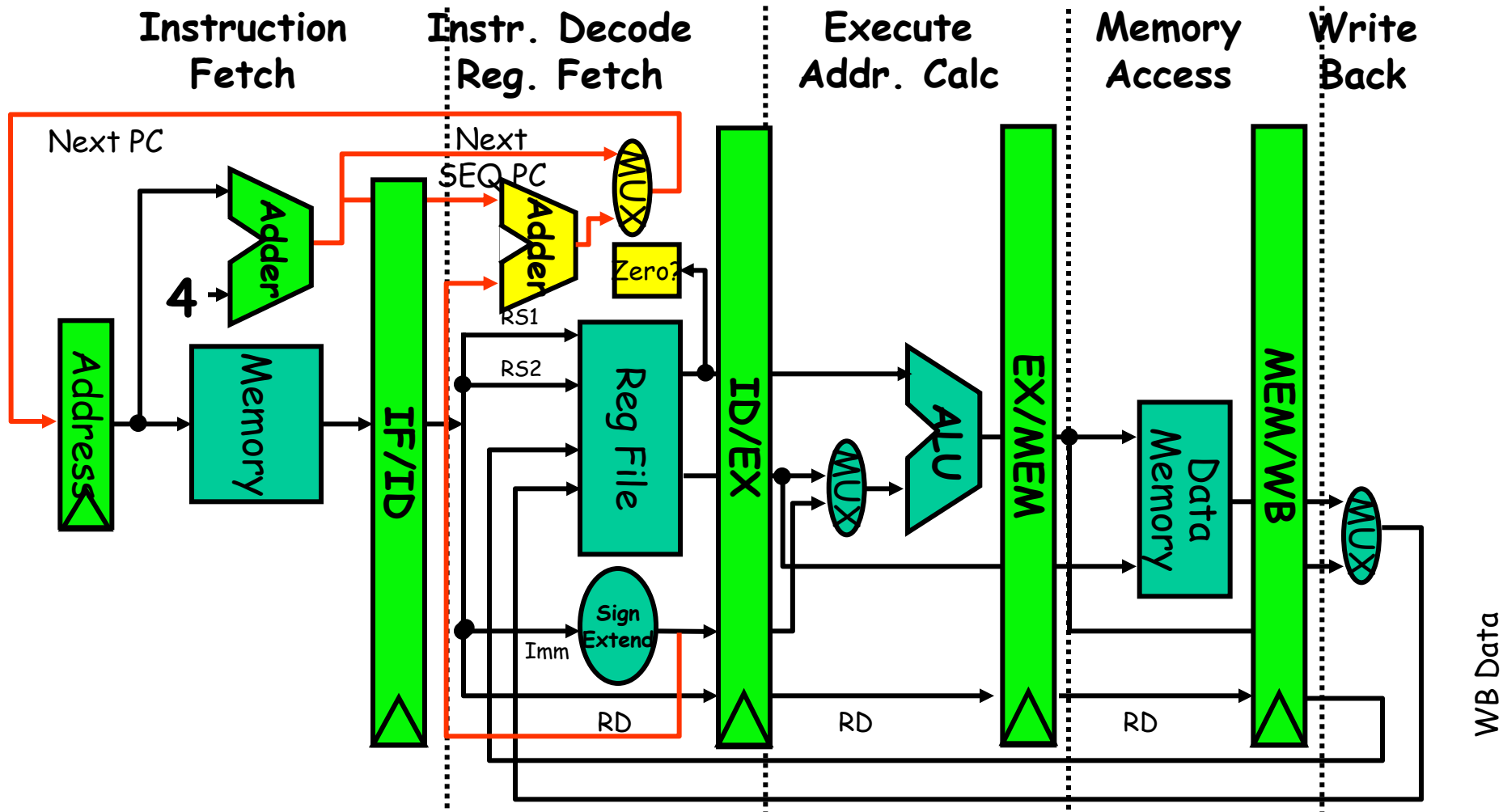
```

Aléa RAW – sur r1

délais

# Datapath du MIPS :

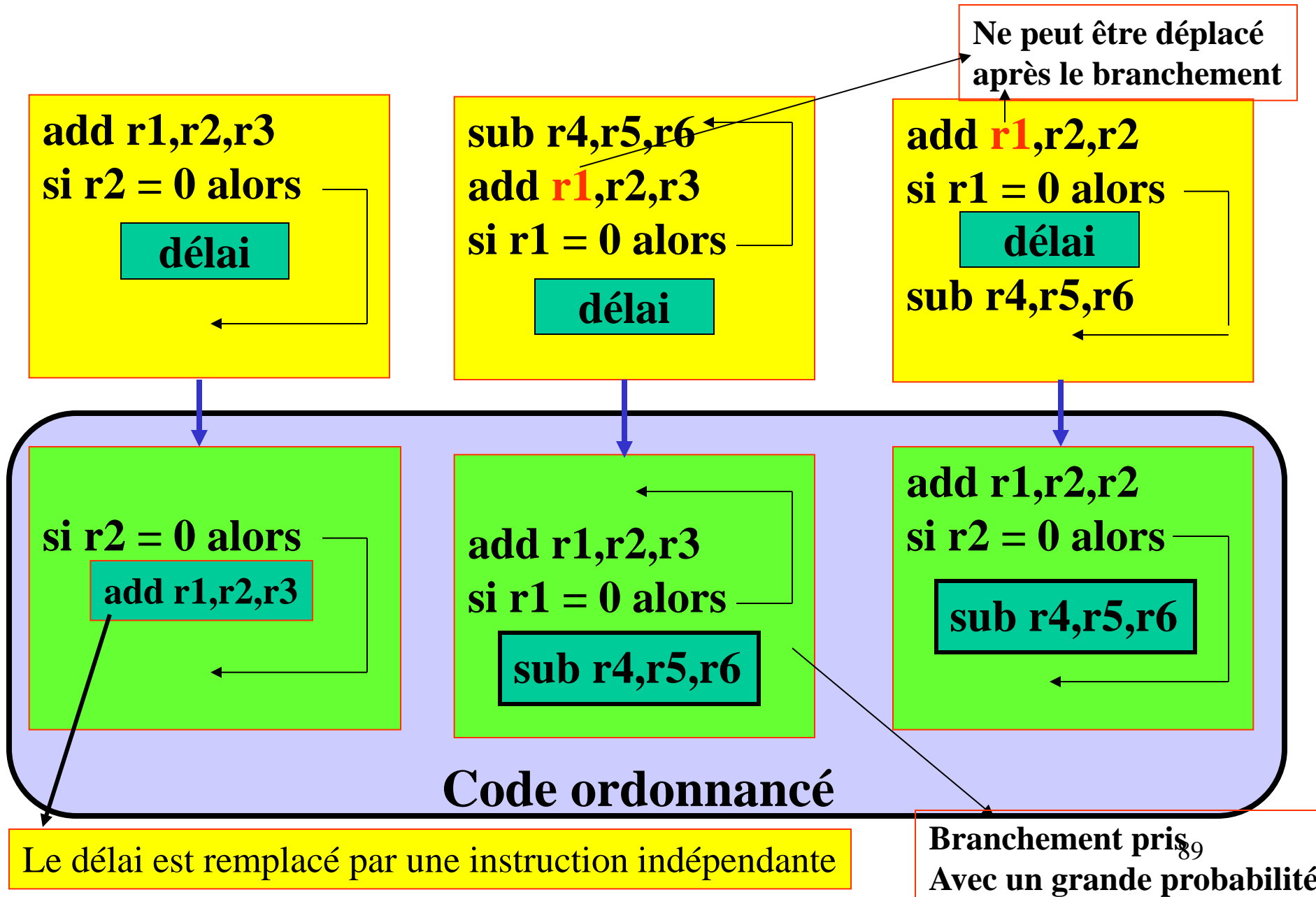
## réduction des suspensions dûs au branchement



**Transfert du test du 0 et du calcul d'adresse vers l'étage ID/RF**



# Ordonnancement du délai de branchement



# Evaluation de l'accélération : cas du Branchement

$$A_p = P_p / (1 + \text{Cycles de susp. du pipeline dus aux branchements})$$

Comme les cycles de susp. dus aux branchements sont fonction de la Fréquence des branchements

on obtient :

$$A_p = P_p / [1 + (\text{Fréq. des branchements} \times \text{Pénalité de branchement})]$$

# Parallélisme d'Instruction: analyse des dépendances

1. Les dépendances vraies (true dependence/flow dependence) : Il existe une dépendance vraie d'une instruction I1 vers une instruction I2 si I2 est placée après I1 et si I2 accède en lecture à un emplacement mémoire modifié par I1. On la note  $I1 \delta^f I2$
2. Les anti-dépendances (anti dependence) : Il existe une anti-dépendance d'une instruction I1 vers une instruction I2 si I2 est placée après I1 et si I2 accède en écriture à un emplacement mémoire qui est lu par I1. On la note  $I1 \bar{\delta} I2$ .
3. Les dépendances de sortie (output dependence) : Il existe une dépendance de sortie d'une instruction I1 vers une instruction I2 si I2 est placée après I1 et si I2 modifie un emplacement mémoire précédemment modifié par I1. On la note  $I1 \delta^o I2$ .

Code séquentiel:

I1:A=B+C

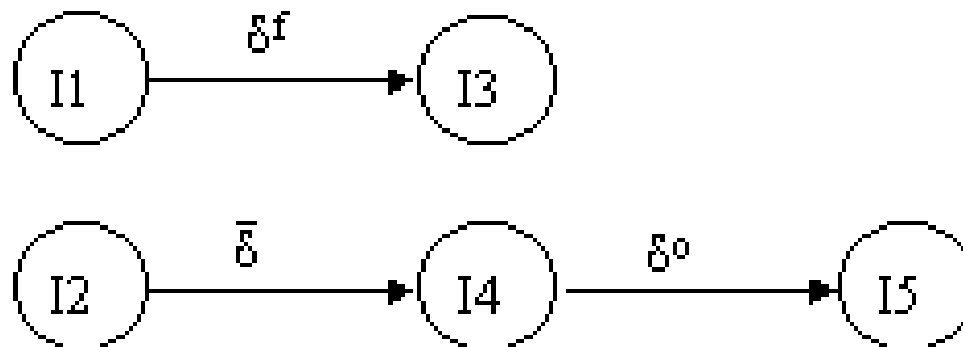
I2:D=E+F

I3:G=A+C

I4:E=H+C

I5:E=1

Le graphe de dépendance:



**Les différents types de dépendance**

# Dépendance de contrôle

1. Si la condition C conditionne l'exécution de I2, on dit qu'il y a une dépendance de contrôle de C vers I2, que l'on note  $C \delta^c I2$ .
2. Analyse des dépendances : suivant le résultat du test, Il existe :
  - une dépendance de sortie entre I1 et I2, car ces deux instructions modifient A
  - ou il existe une dépendance vraie entre I1 et I3, car I3 utilise A précédemment modifié par I1.

```
I1 :      A=B+C
C :      if (X>0) then
I2 :      A=D+E
          else
I3 :      F= A+G
          end if
```

**Dépendance de contrôle**

# Les boucles : dépendances au sein d'une boucle

- Les dépendances indépendantes des boucles (loop independent dependences), lorsque celles-ci existent sans la boucle,
- Les dépendances portées par les boucles (loop carried dependences), lorsqu'une instruction a besoin d'un emplacement mémoire modifié par l'itération précédente

```
for (i=1;i<N;i++)  
{  
    a[i]=a[i]*12;           calcul de a[i]  
    b[i]=a[i-1]*x;         b[i] à besoin du résultat a[i-1]  
                           calculé lors de l'itération précédente  
}
```

**Dépendances au sein de la boucle**

1. Analyse de dépendance des données
2. Optimisation des instructions en supprimant certaines dépendances, permettant une parallélisation des instructions
3. Ordonnancement ou organisation des instructions en fonction de l'architecture cible.

Ces opérations d'optimisation sont effectuées sur le code intermédiaire fourni par la partie frontale du compilateur.

Les nouvelles techniques de compilation permettent aux compilateurs de générer un code optimal assurant un taux d'utilisation maximal des unités fonctionnelles du microprocesseur et une accélération des programmes

# Le renommage

Le Renommage permet de renommer des variables afin de supprimer deux types de dépendances de données :

- les anti-dépendances
- les dépendances de sorties

La suppression de ces dépendances fait apparaître du parallélisme au niveau des instructions, c'est-à-dire des instructions indépendantes.

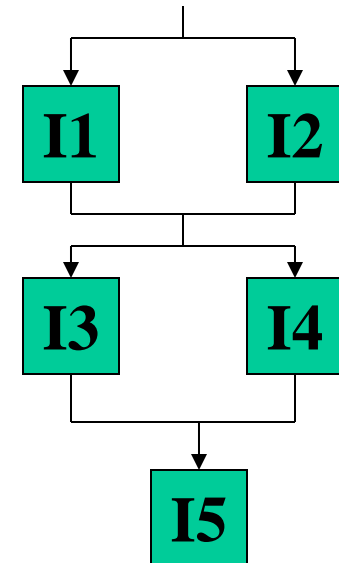
Code séquentiel:

I1:  $y = x / c$   
 I2:  $x = x + c$   
 I3:  $z = y + c$   
 I4:  $y = c - y$

dépendance vraie  
 anti-dépendance  
 dépendance de sortie

Code séquentiel après renommage:

I1:  $y' = x / c$   
 I2:  $x = x + c$   
 I3:  $z = y' + c$   
 I4:  $y = c - y'$   
 I5:  $x = x'$



## Renommage de variables

- La variable  $y$  de I1 est renommée en  $y'$  ce qui supprime la dépendance de sortie.
- Afin de ne pas modifier le résultat du programme, les variables  $y$  de I3 et I4, sont aussi renommées en  $y'$  et une instruction de compensation I5, sera ajoutée.
- Ce renommage a permis de supprimer deux dépendances.
- Le code séquentiel fait apparaître un parallélisme de 1 alors que le code séquentiel renommé fait apparaître un parallélisme de 2 car les instructions I1 et I2 sont exécutables en parallèle ainsi que I3 et I4



# Le déroulement de boucle (Loop Unrolling)

Paralléliser les opérations contenues dans une boucle, s'il n'y a pas de dépendance entre les itérations.

Pour extraire un maximum de parallélisme, la technique du déroulement de boucle, consiste à travailler sur plusieurs itérations en même temps, c'est-à-dire à dupliquer le traitement de la boucle  $d$  fois.

Si une boucle répète un traitement  $N$  fois sur des données, le nombre d'itération de la boucle sera de  $N/d$  (partie entière de la division).

Les traitements restants seront réalisés en dehors de la boucle et le nombre sera égal au reste de la division  $N/d$ .

La nouvelle boucle pourra subir des transformations comme, le renommage des registres, etc...., afin de supprimer le plus de dépendances possible.

# Le déroulement de boucle (Loop Unrolling)

Code intermédiaire  
non optimisé

```
i=0
boucle: F0=A[i]
        F1=F0*B
        F2=F1+C
        A[i]=F2
        i=i+1
        si i<100 va à boucle
```

loop unrolling de 2

Code intermédiaire  
optimisé

i=0

```
boucle: F0=A[i]
        F1=F0*B
        F2=F1+C
        A[i]=F2
```

copie 1  
1<sup>ère</sup> itération

```
F10=A[i+1]
F11=F10*B
F12=F11+C
A[i+1]=F12
```

copie 2  
2<sup>ème</sup> itération

i=i+2

si i<50 va à boucle

6 cycles/itération sur un superscalaire de degré 2

6 cycles pour 2 itérations sur un superscalaire de degré 2  
soit 3 cycles/itération

**Code intermédiaire optimisé avec un loop unrolling de 2**

Le code intermédiaire de la boucle doit subir des transformations pour que les traitements dupliqués dans la boucle soient indépendants.

Cela passe par une utilisation accrue des registres du microprocesseur, car il va falloir dupliquer tous les registres utilisés par la boucle d'origine, à savoir :

- Le registre mémorisant la valeur de  $A[i]$
- Le registre mémorisant le résultat  $A[i]*B$
- Le registre mémorisant le résultat  $A[i]*B+C$

Cette indépendance va permettre aux copies 1 et 2 de s'exécuter en parallèle.

Pour déterminer le temps d'exécution de la boucle en cycle/itération, on admet qu'une instruction s'exécute en un seul cycle. Il faut 4 cycles pour exécuter les copies, étant donné qu'elles sont parallélisées et 2 cycles pour le test de fin de boucle, ce qui donne 6 cycles pour 2 itérations, soit 3 cycles par itération sur un processeur superscalaire de degré deux.

On peut remarquer que l'exécution du code intermédiaire non optimisé, de cette boucle, sur un processeur superscalaire de degré deux, n'améliore pas son temps d'exécution, faute de parallélisme. L'accélération du programme est de  $6/3$  soit 2.

# Exercice :

Le corps de chaque itération est indépendant donc c'est une boucle parallèle.

Exécution de la boucle sans ordonnancement

		<b>Démarre au cycle</b>
Bou :	LD F0,0(R1)	1
	nop	2
	ADD F4,F0,F2	3
	nop	4
	nop	5
	SD 0(R1),F4	6
	SUBI R1,R1,#8	7
	BNEZ R1,BOU	8
	nop	9

**on a donc 9 cycles par itération.**

Bou :	LD	F0,0(R1)	1
	nop		2
	ADD	F4,F0,F2	3
	SUBI	R1,R1,#8	4
	BNEZ	R1,BOU	5
	SD	8(R1),F4	6

Le temps d'exécution a été réduit de 9 à 6 cycles. On termine une itération et on range un élément du vecteur tous les 6 cycles, le traitement à proprement dit (chargement, addition et rangement) ne prend que 3/6 cycles. Les 3 cycles restants sont dus à la gestion de la boucle (SUBI, BNEZ) et une suspension.

Le déroulage de boucle permet d'augmenter le nombre d'instructions par rapport au branchement et aux Instructions de gestion de boucle.

**Le déroulage de boucle permet d'augmenter le nombre d'instructions par rapport au branchement et aux Instructions de gestion de boucle.**

**Donc le déroulage de boucle consiste à dupliquer le corps de la boucle puis à adapter le code de terminaison de boucle.**

BOU :	LDF	0,0(R1)	2 cycles	
	ADDD	F4,F0,F2	3 cycles	<u>6 cycles</u>
	SD	0(R1),F4	1 cycle	
	LD	F6,-8(R1)		
	ADDD	F8,F6,F2		<u>6 cycles</u>
	SD	-8(R1),F8		
	LD	F10,-16(R1)		
	ADDD	F12,F10,F2		<u>6 cycles</u>
	SD	-16(R1),F12		
	LD	F14,-24(R1)	2 cycles	<u>6 cycles</u>
	ADDD	F16,F14,F2	3 cycles	
	SD	-24(R1),F16	1 cycle	
	SUBI	R1,R1,#32	1 cycle	<u>3 cycles</u>
	BNEZ	R1, BOU	2 cycles	

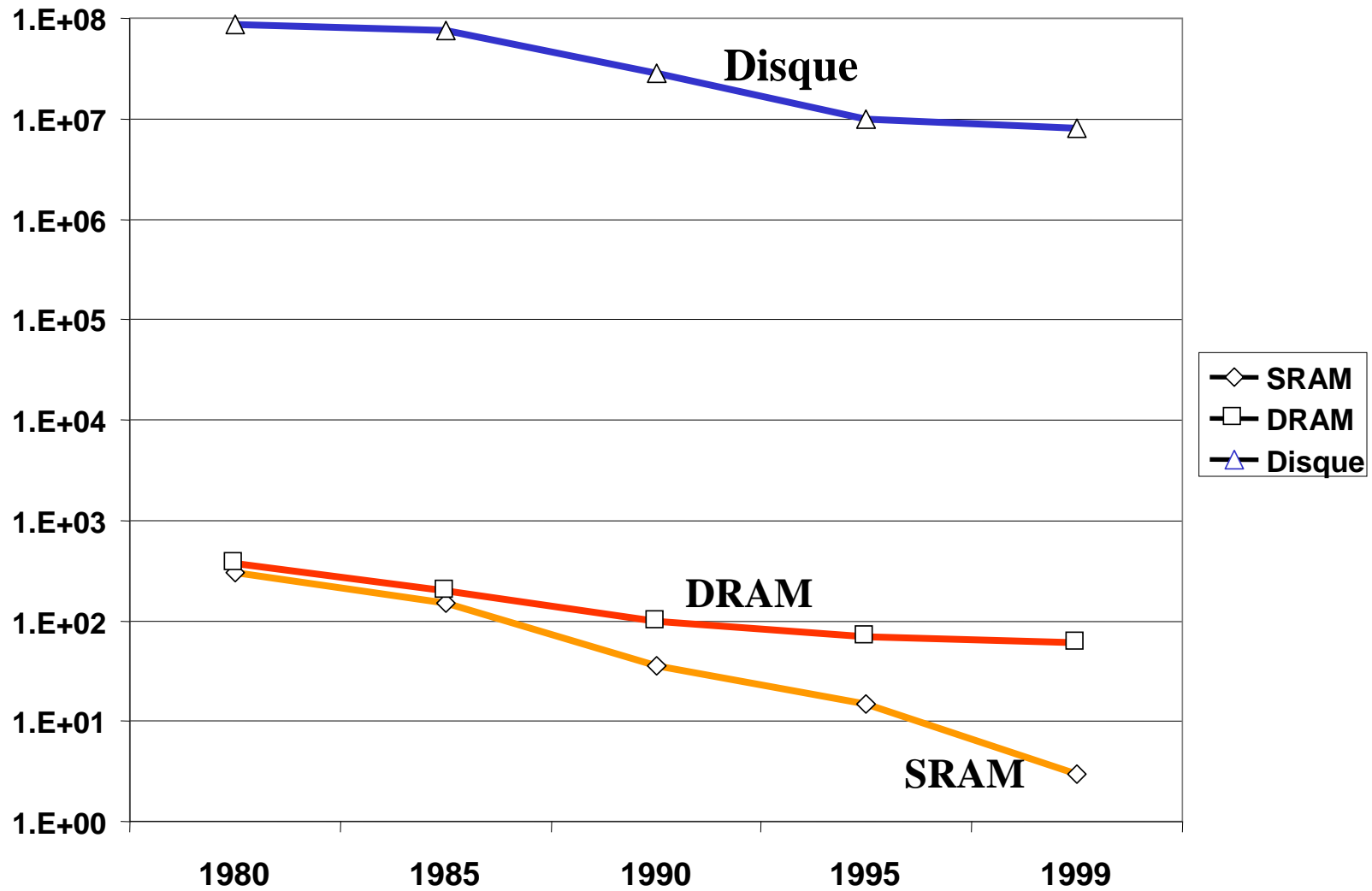
On a éliminé 3 branchements et 3 décréments de R1. Les adresses des chargements et des rangements ont été rectifiées pour permettre la fusion des instructions SUBI sur R1.

**Sans ordonnancement cette boucle s'exécute en 27 cycles ( $4 \times 6 + 3$ ).  
Soit 6.8 cycles pour chacun des quatre éléments.**

			<u>Démarre au cycle</u>
BOU :	LD	F0,0(R1)	1
	LD	F6,-8(R1)	2
	LD	F10,-16(R1)	3
	LD	F14,-24(R1)	4
	ADDD	F4,F0,F2	5
	ADDD	F8,F6,F2	6
	ADDD	F12,F10,F2	7
	ADDD	F16,F14,F2	8
	SD	0(R1),F4	9
	SD	-8(R1),F8	10
	SD	-16(R1),F12	11
	SUB	R1,R1,#32	12
	BNEZ	R1,BOU	13
	SD	8(R1),F16	14

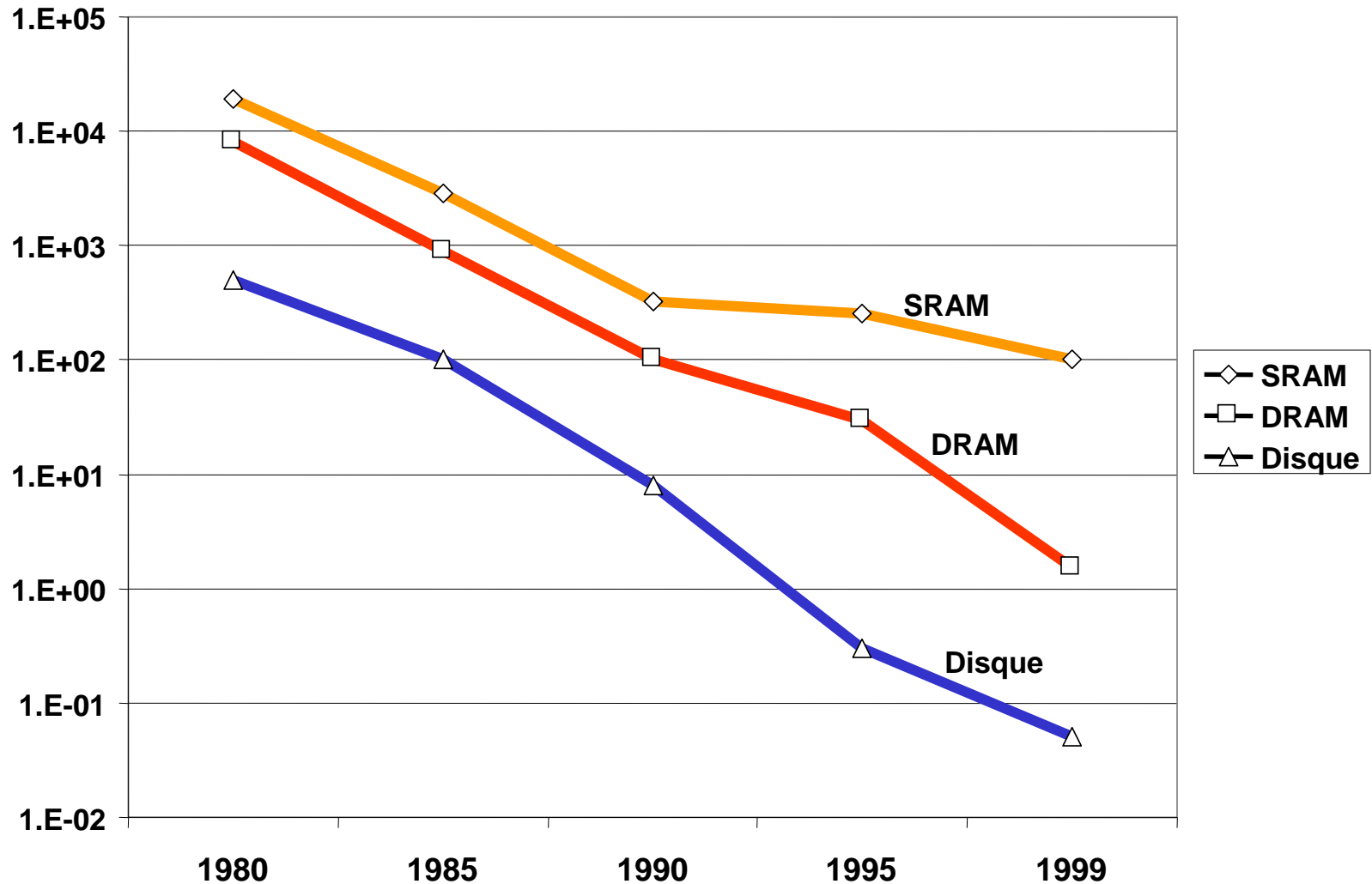
**Le temps d'exécution de la boucle déroulée est de 14 cycles (soit 3.5 cycles par élément/ à 6.8 cycles avant ordonnancement et à 6 cycles avec ordonnancement pour la boucle non déroulée.**

# Annexe - Mémoire (1) : temps d'accès (nsec)

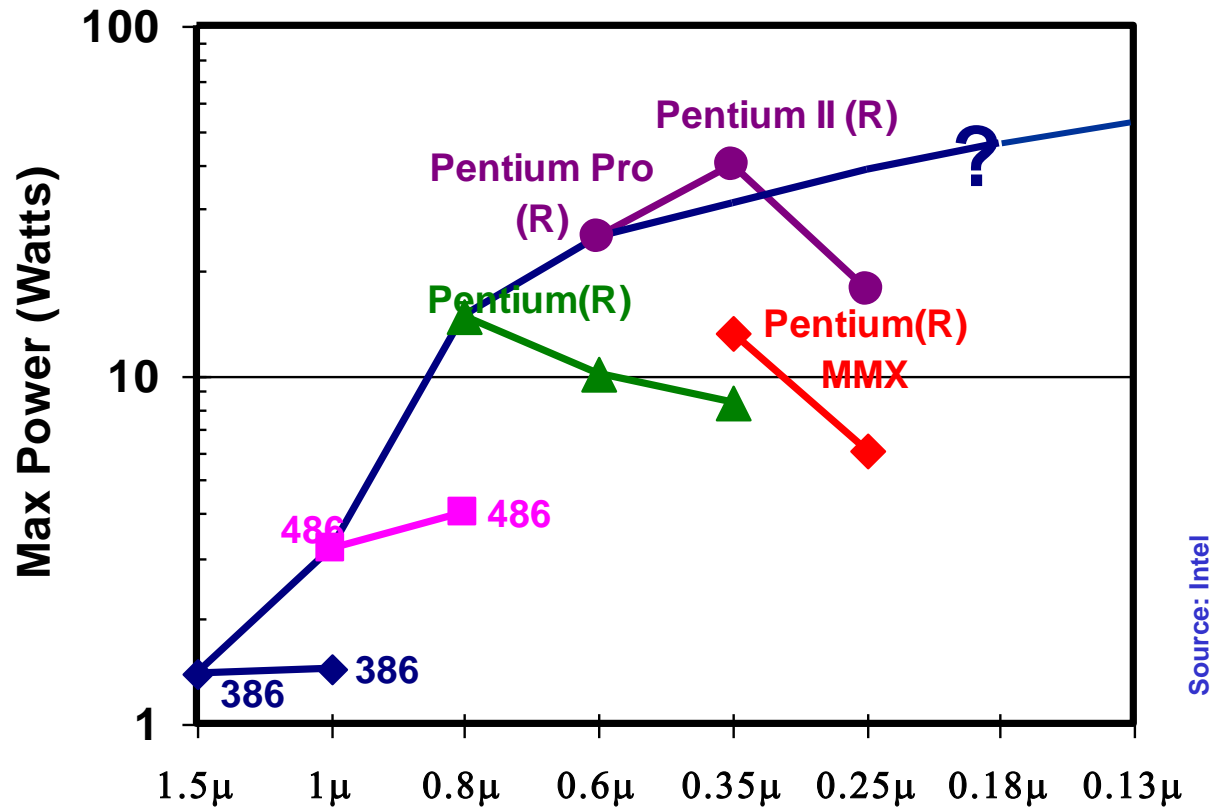




# Annexe - Mémoire (2) : Evolution des prix (\$/Mbyte)



# Annexe - Consommation d'Energie (3)



# Annexe - Récapulatif (4) : Evolution de la technologie

- Processeur
  - Densité : 30% par an
  - Horloge : 20% par an
- Mémoire
  - DRAM capacité : 60% par an (4x tout les 3 ans)
  - Vitesse : 10% par an
  - Coût/bit : 25% par an
- Disque
  - Capacité : 60% par an

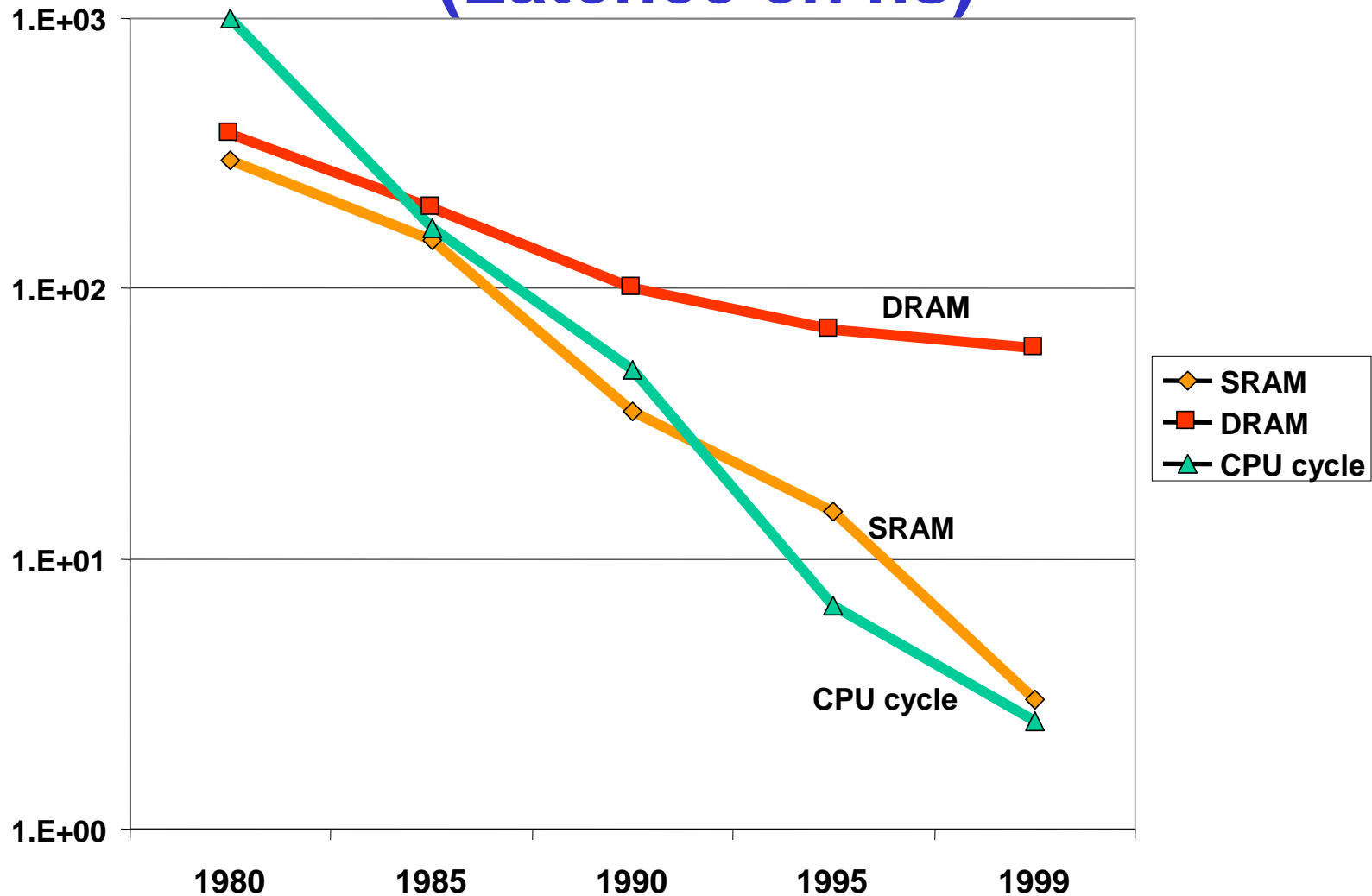
# Annexe - Récapulatif (5) : Evolution de la technologie

Year of First Shipment	1995	1998	2001	2004	2007	2010
Minimum Feature Size ( $\mu\text{m}$ )	0.35	0.25	0.18	0.13	0.10	0.07
<b>Memory</b>						
Bits/chip	64Mb	256Mb	1Gb	4Gb	16Gb	64Gb
Cost/bit (millicents)	0.017	0.007	0.003	0.001	0.005	0.0002
<b>Microprocessor logic (high volume)</b>						
Transistors/cm <sup>2</sup>	4M	7M	13M	25M	50M	90M
Memory cache (bits/cm <sup>2</sup> )	2M	6M	20M	50M	100M	300M
Cost/transistor (millicents)	1	0.5	0.2	0.1	0.05	0.02
<b>ASIC logic (low volume)</b>						
Transistors/cm <sup>2</sup>	2M	4M	7M	12M	25M	40M
Design cost/transistor (millicents)	0.3	0.1	0.05	0.03	0.02	0.01

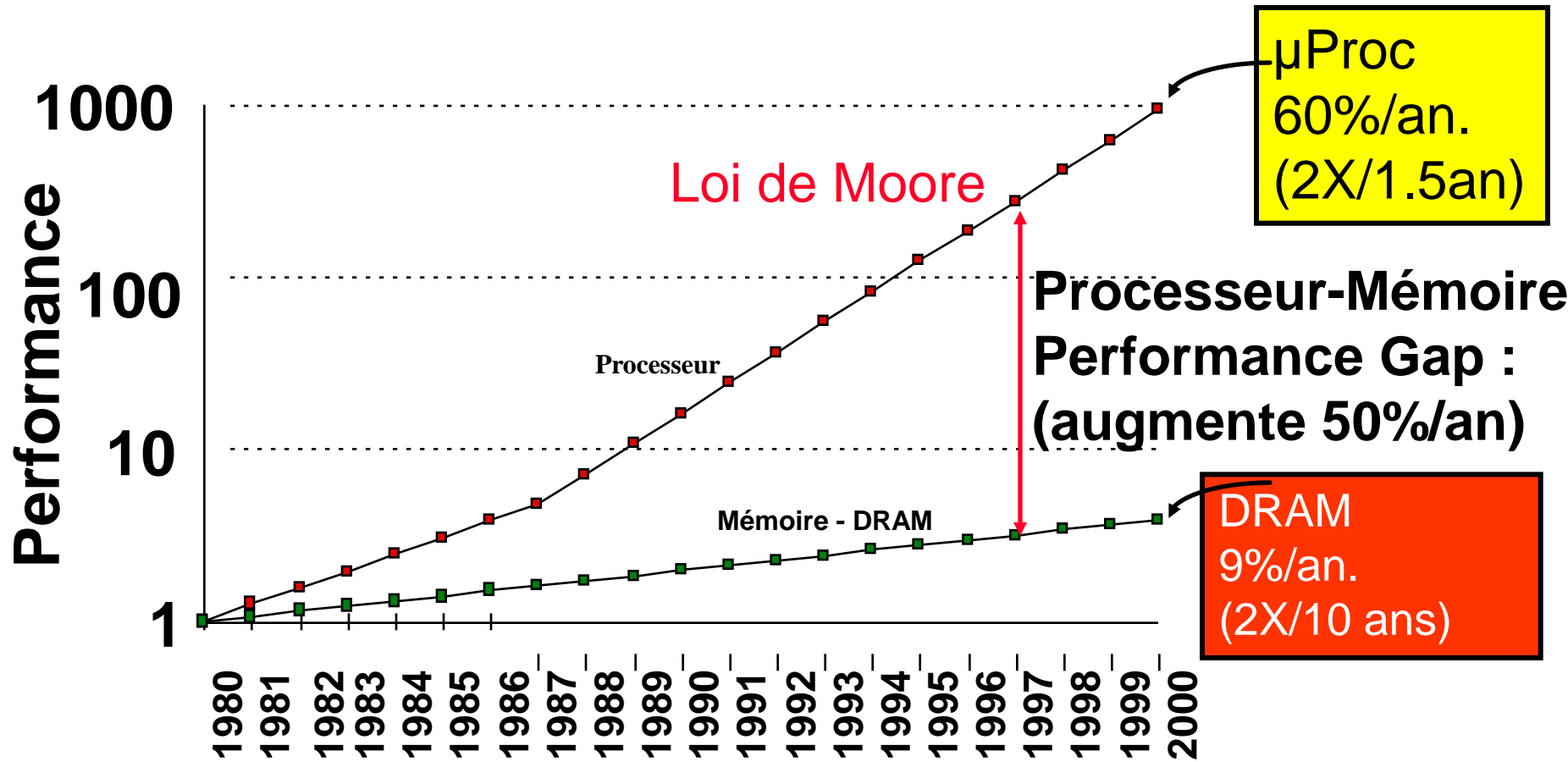
Source: National Technology Roadmap for Semiconductors, 1994.

# Annexe - Récapitulatif (5) :

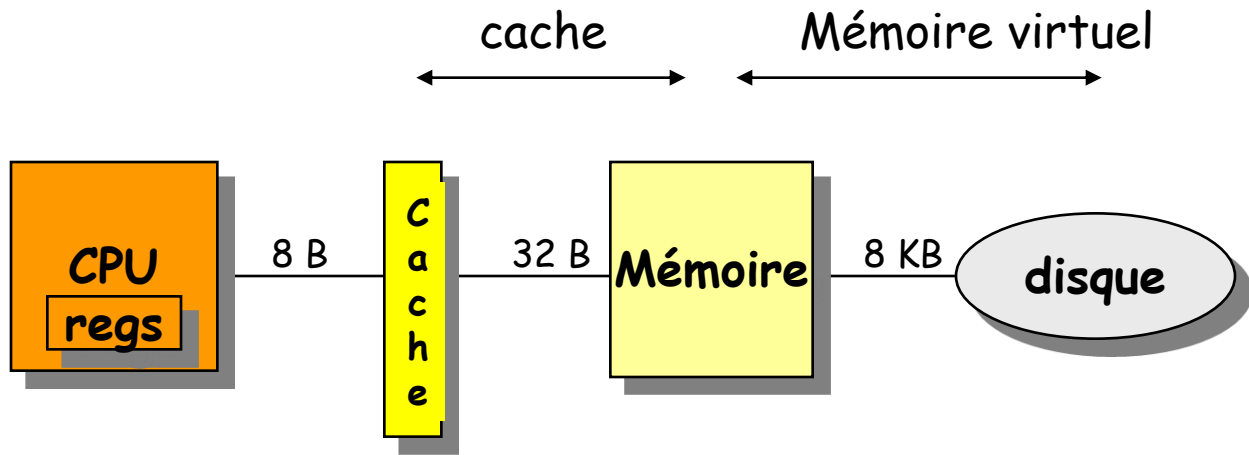
## Comparaison Processeur – Mémoire (Latence en ns)



# Annexe - Récapitulatif (6) : Loi de MOORE



# Annexe - Récapitulatif (7) : Hiérarchie Mémoire

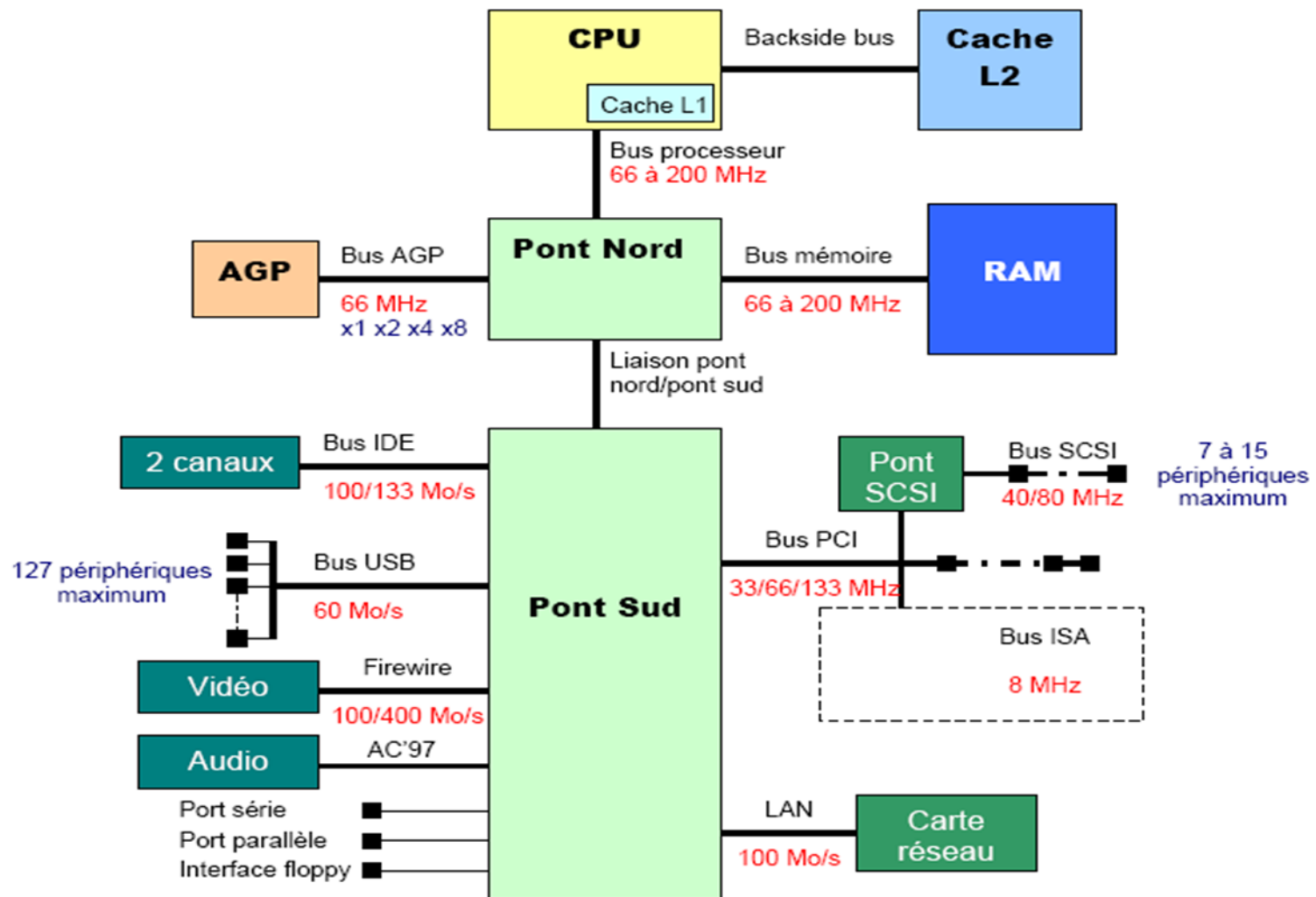


	Registres	Cache	Mémoire	Disque
Taille :	200 B	32 KB / 4MB	128 MB	20 GB
Latence :	3 ns	6 ns	60 ns	8 ms
\$/Mbyte:		\$100/MB	\$1.50/MB	\$0.05/MB
Taille-bloc:	8 B	32 B	8 KB	

large, petit, cher →

# Annexe Organisation d'un ordinateur

## Mono processeur (2) : architecture d'une carte mère





## Annexe microprocesseurs CISC (4) :



**80386 : 8 cycles d'horloge par instruction**

**80486 : 2 cycles d'horloge par instruction**

**80386 : 1/8 (nbre d'instr. par cycle d'horloge)**

**80486 : 1/2 (nbre d'instr. par cycle d'horloge)**

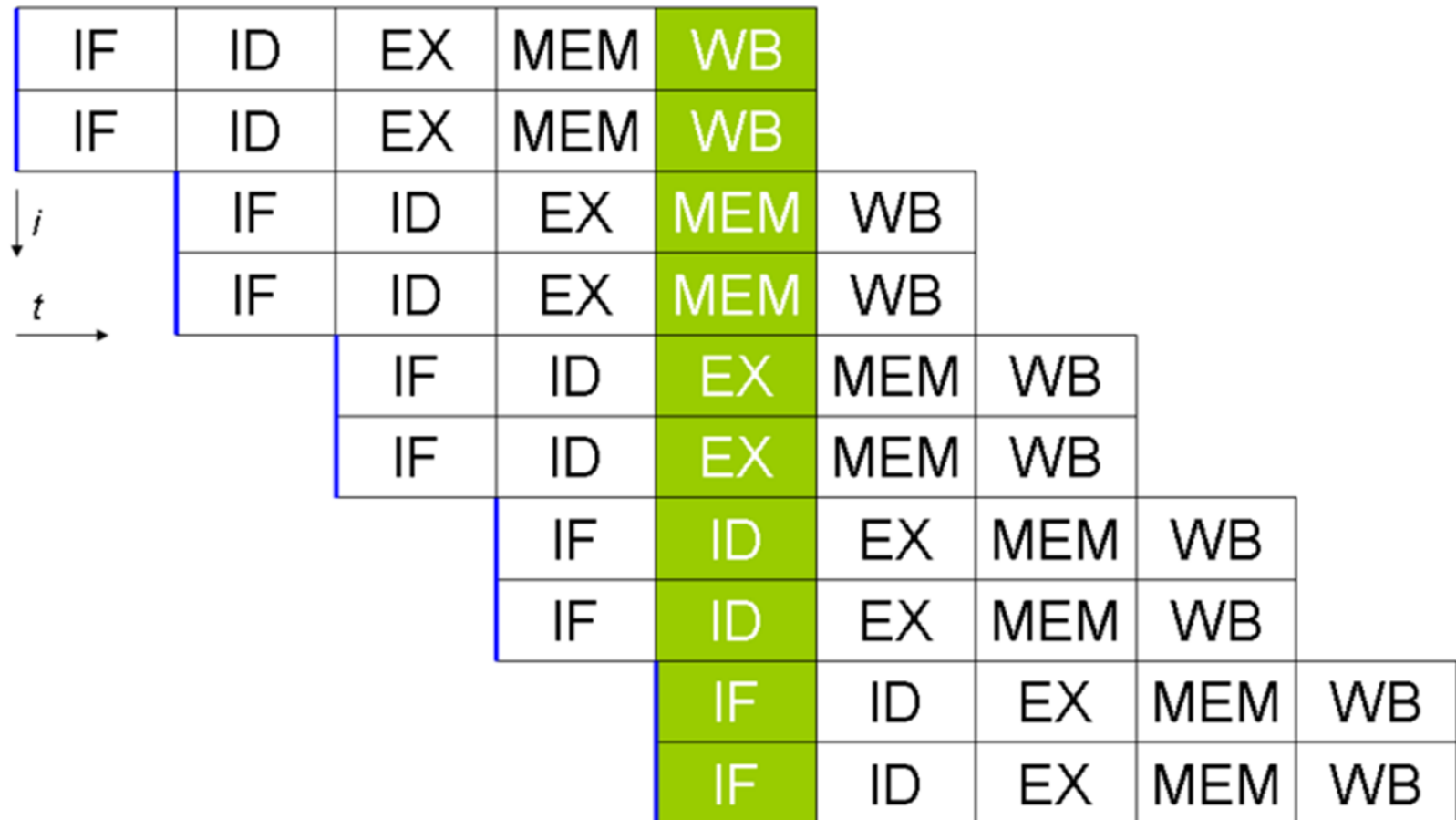
**80386 : F = 16 Mhz – P = 3w**

**80486 : F = 33 Mhz – P = 6w**

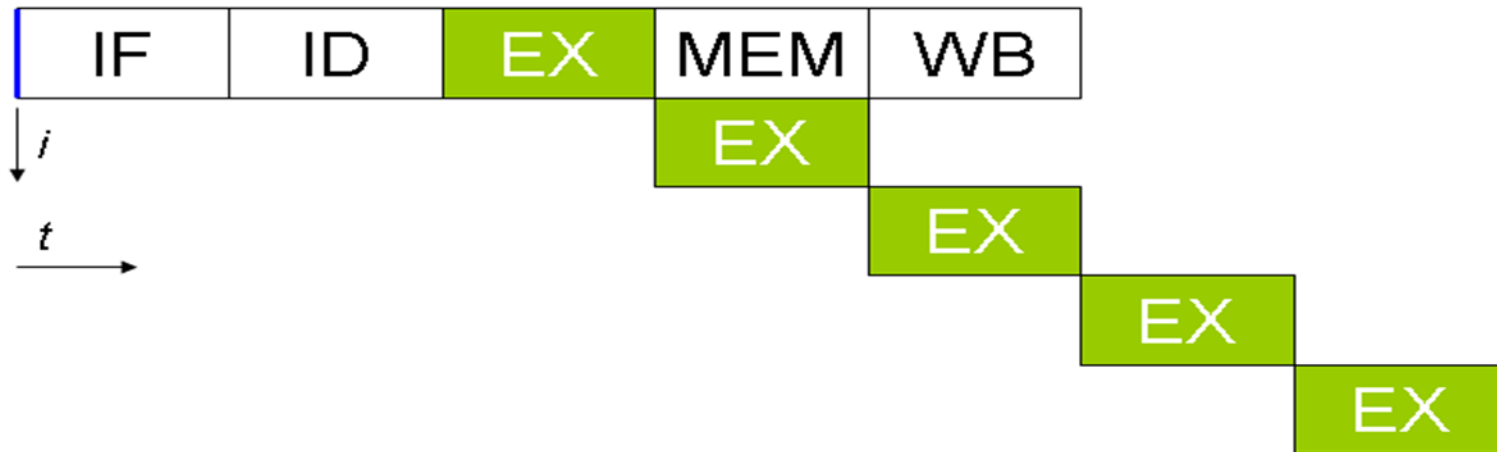
# Microprocesseurs RISC (5) : Pipeline simple



# Microprocesseurs RISC (13) : Superscalaire



# Microprocesseurs RISC (17) : Vectoriel

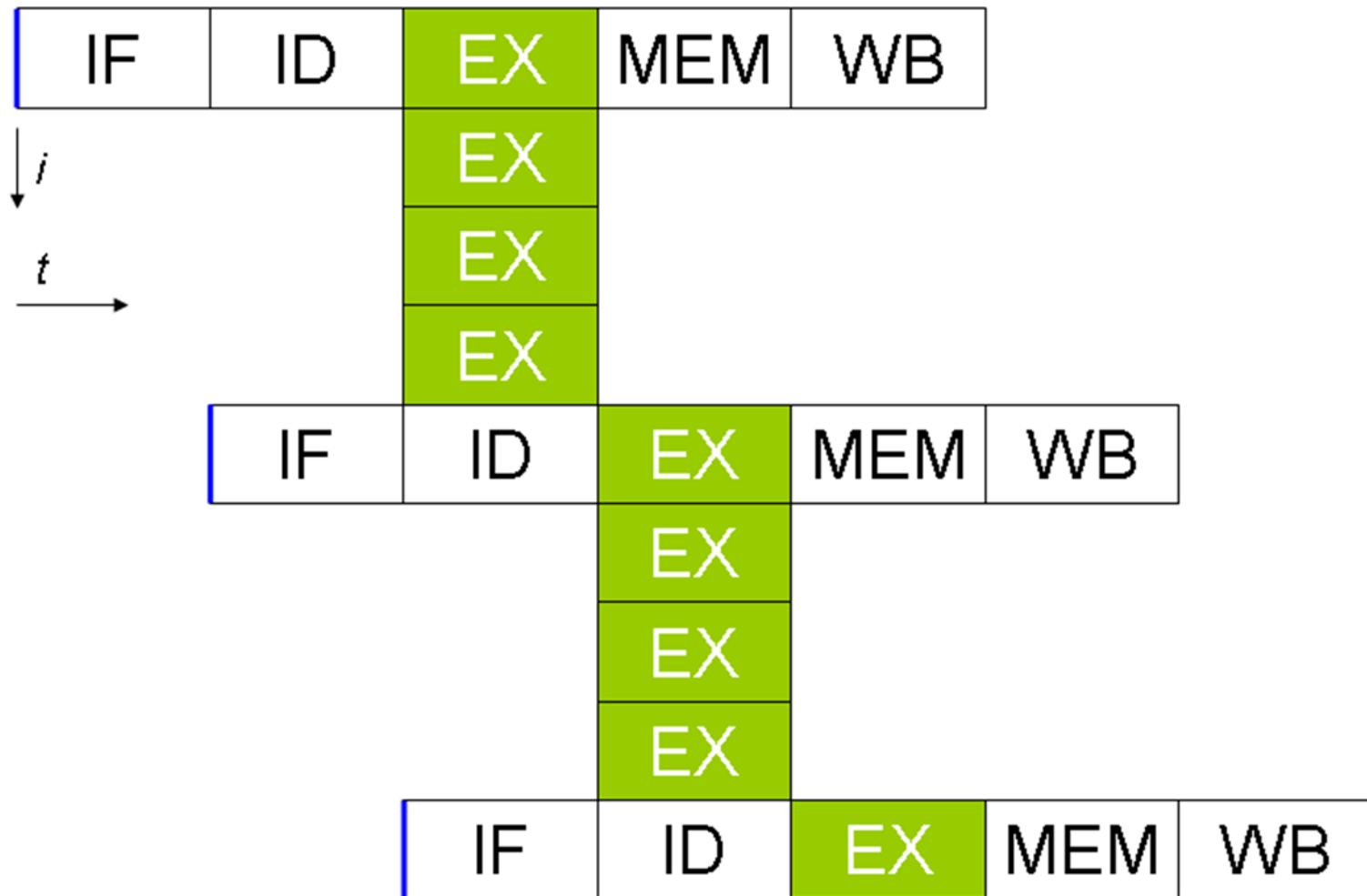


Les processeurs vectoriels utilisent une architecture **SIMD** (Single Instruction Multiple Data). Ils sont capables d'effectuer la même opération sur plusieurs données différentes en même temps.

Les processeurs actuels intègrent des unités de calcul vectoriel dédiées au multimédia :

- **MMX** (Multi-Media eXtension) intégrée au Pentium MMX (registres 64 bits, entiers)
- **SSE** (Streaming Simd Extension) apparu sur les Pentium III (128 bits, entiers ou flottants 32 bits)
- **SSE2** évolution du SSE sur le Pentium 4 (flottants 64 bits)
- **3D Now !** version MMX/SSE des processeurs AMD apparue sur les K6-II (64 bits)

# Microprocesseurs RISC (18) : VLIW



# Microprocesseurs RISC (19) : VLIW

