

Architecture de Ordinateurs

Module 5

# Optimisations architecturales

Dr. Yannick HERVE

V 1.5

# Plan

- Mémoires caches L2/L2/L3
- RISC
- Architecture Harvard
- Pipeline (scalaire)
- Superscalaire
- VLIW
- Vectorielle
- Multithread/Hyperthreading
- Multi Cœurs/MultiProcesseurs

# Note préliminaire

- La fréquence est passée de qq kHz à des GHz
  - Problèmes : température/gestion horloge globale
  - Presque plus de capacité d'évolution
- Taille puces ↗ ➔ rendement de fabrication ↘
- Améliorations ? Optimisations architecturales
  - Optimiser utilisation du matériel : pipeline
  - Parallélisation dans toutes les opérations
    - Interne / Externe : multi cœurs et/ou multi-processeurs
- Objectif : Nombre d'instruction par cycle (IPC) ↗

# Note préliminaire

- La fréquence est passée de qq kHz à des GHz
  - Problèmes : température/gestion horloge globale
  - Entre le 4004 et le Core i7 2600K :
    - La fréquence a été multipliée par 4600
    - La puissance en MIPS a été multipliée par 1,4 million
- Ta
- Ar
  - Ce n'est pas la fréquence seule qui fait la puissance
  - Optimiser utilisation du matériel : pipeline
  - Parallélisation dans toutes les opérations
    - Interne / Externe : multi cœurs et/ou multi-processeurs
- Objectif : Nombre d'instruction par cycle ↗

# Mémoires Caches

(voir cours sur les mémoires)

- Ou Antémémoire
- Inventé par Maurice Vincent Wilkes en 1965
  - Nom original : Slave memory
- Apparue en 1968 sur le mainframe IBM 360/85
- Dans toutes les architectures modernes
  - L1 : privé (pour un seul cœur)
  - L2 : privé ou partagé
  - L3 : partagé

# RISC : naissance

- Augmenter performances
  - Augmenter complexité jeu instructions (CISC)
    - Moins d'instructions, programmes courts
    - Taille et durée variable → microcode complexe
  - Machines trop complexes → échec
- Autre piste (1979, 1985) [80% code=20% jeu instructions]
  - Nombre d'instructions restreint (RISC)
  - Gestion mémoire simplifiée : registre ⇔ mémoire
  - Pas de microprogramme
    - le code machine est la microcommande

# RISC : qualité

- Nombre d'instructions limité, inst de taille fixe
- Pas de microprogramme
- Modes d'adressage limités
  - Registre pour source/résultat ALU/FPU
  - Adressage direct registre/mémoire
- Besoin : beaucoup de registres
  - Eventuellement en bancs.
    - CISC, premier x86 : 8 registres
    - RISC, SPARC : entre 72 et 640 (généralement 160)

M=mémoire  
R=Registre

# CISC/RIC : exemple

## Processeur CISC

ADD M1,M2,M3

Suite de microcommandes

Mais aussi :

ADD R1,R2,M3

ADD R1,M2,M3

ADD R1,R2,R2

...

## Processeur RISC

LD R1 M1

LD R2 M2

ADD R2 R2 R3

ST M3 R3

CodeOp = microcommande

LD/ST avec registres

Opérations sur les registres



# Comparaison : avantages

- CISC
  - Moins d'instructions pour une fonction
  - Microprogrammation : correction jeu d'instructions
  - Instructions très complexes/très rapides
- RISC
  - Architecture globalement moins complexe
  - Consommation d'énergie moindre
  - Interruptions plus rapides
  - Compilateur plus simple et efficace

# Comparaison : inconvénients

- CISC
  - Surface importante de silicium (optimisation archi ?)
  - Consommation importante
  - Compilateurs complexes
  - Temps développement/validation importants
- RISC
  - Taille du programme plus importante (20 % à 50%)
  - Instruction à taille fixe pas toujours optimisée
  - Accès multiples à la mémoire pénalisés ( chaînes,...)

# RISC : machines récentes

- Vue utilisateur : CISC
  - Prétraitement instructions
  - Fabrication des microcommandes
- Vue interne : RISC
  - Et autres optimisations

# Architecture Harvard

- Nom : Université Harvard, Mark I en 1944 (relais)
- Mémoire commune instructions/données
  - Goulot étranglement
- ➔ Séparation bus/mémoire données/instructions
  - Accès simultané
- Optimisation
  - Séparation données entrantes/données sortantes
  - Utilisation : DSP, Microcontrôleur

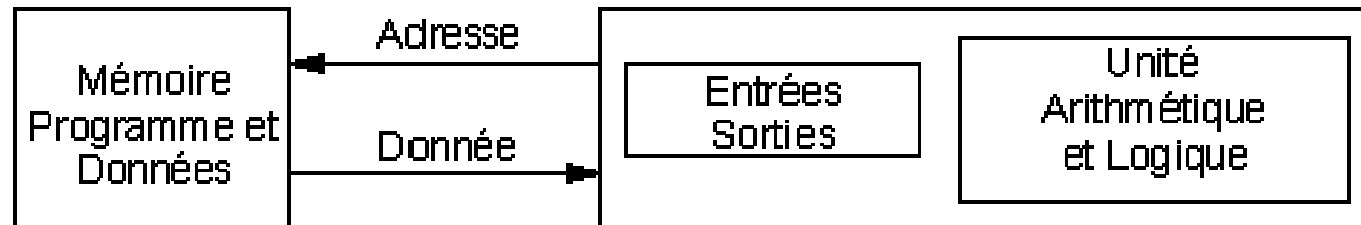
# Harvard : détails

- Mémoire différentes : taille, mot, timing ...
- Programmes rémanents : ROM (Flash)
- Le fetch est simplifié (tous les mots sont des inst)
  - R/W data en même temps possible

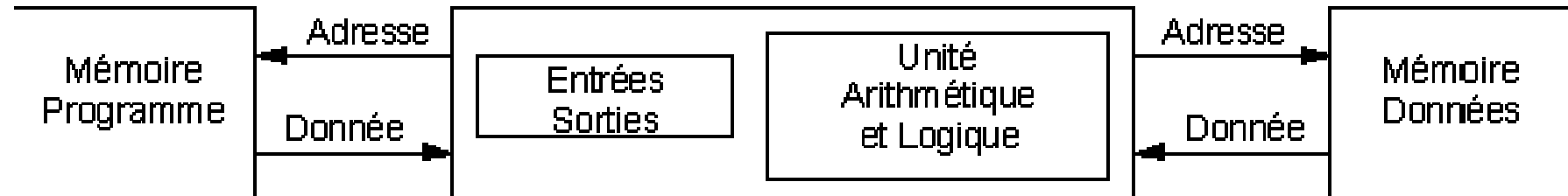
## **Actuellement**

- Harvard modifié : deux modes possibles
  - Cache D+cache I : Harvard pur / RAM : Von Neumann
    - X86, Power ISA, ARM ...
  - Données figées en mémoire instructions (inst. spéciales)

### Architecture de type Von Neuman



### Architecture de type Harvard



**1 à N bus**

# Archi/Instructions

RISC	<b>ARM7</b>	<b>ARM9</b>
CISC	<b>Pentium</b>	<b>SHARC (DSP)</b>
	von Neumann	Harvard

# Mode réel/mode protégé

Pas optimisation au sens propre

Evolution pour la sécurité

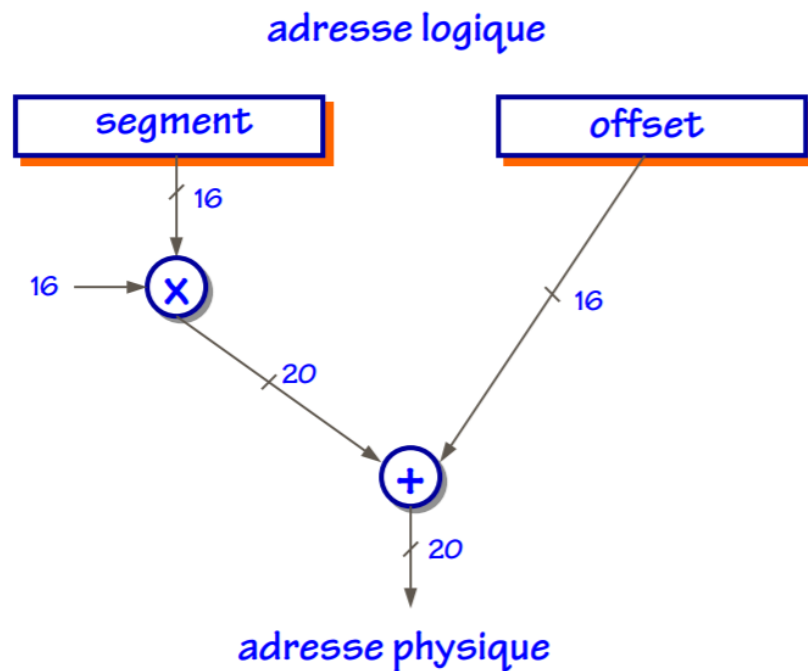


Note : le 80286 avait un  
mode d'émulation  
du 8086

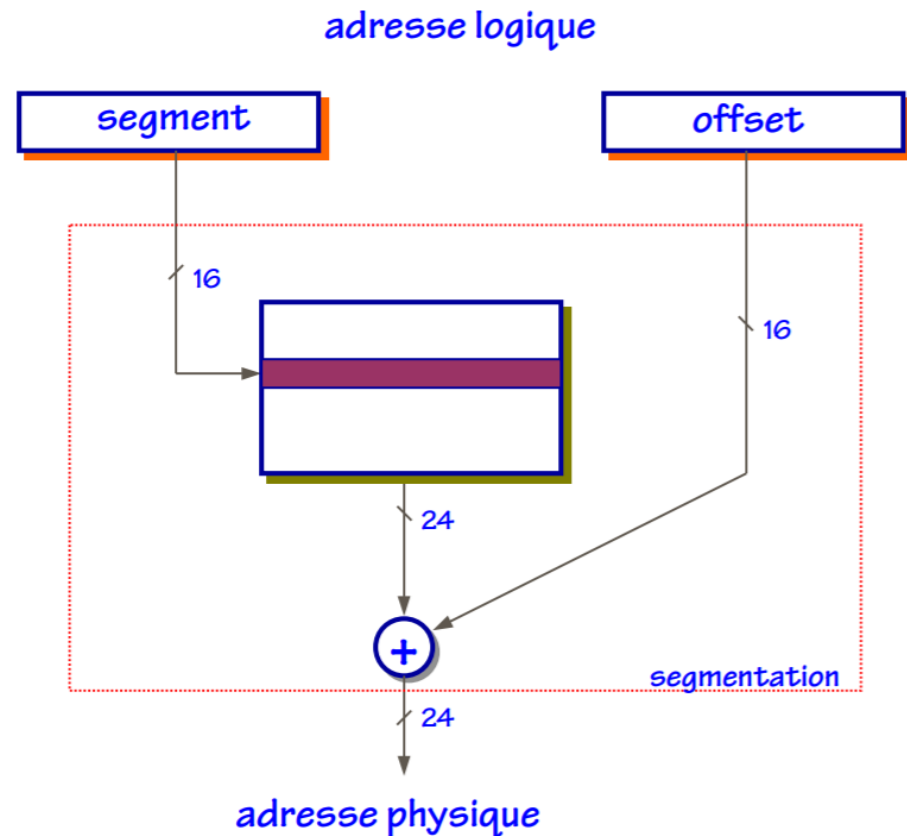
## Besoin et naissance

- Mode réel : tous les programmes accèdent à toute la mémoire ➔ Sécurité ?
- Famille Intel x86 : 80286 mode protégé (4 niveaux)
  - Mode réel
    - Les adresses sont calculées Segment+offset
  - Mode protégé
    - Les adresses sont calculées en passant par des descripteurs : Segment + adresse\_pointée\_par\_offset
    - Démarrage : le système en mode protégé pour l'OS
  - Passage de protégé à réel lent et TRES sécurisé

# Calculs d'adresse : 80286

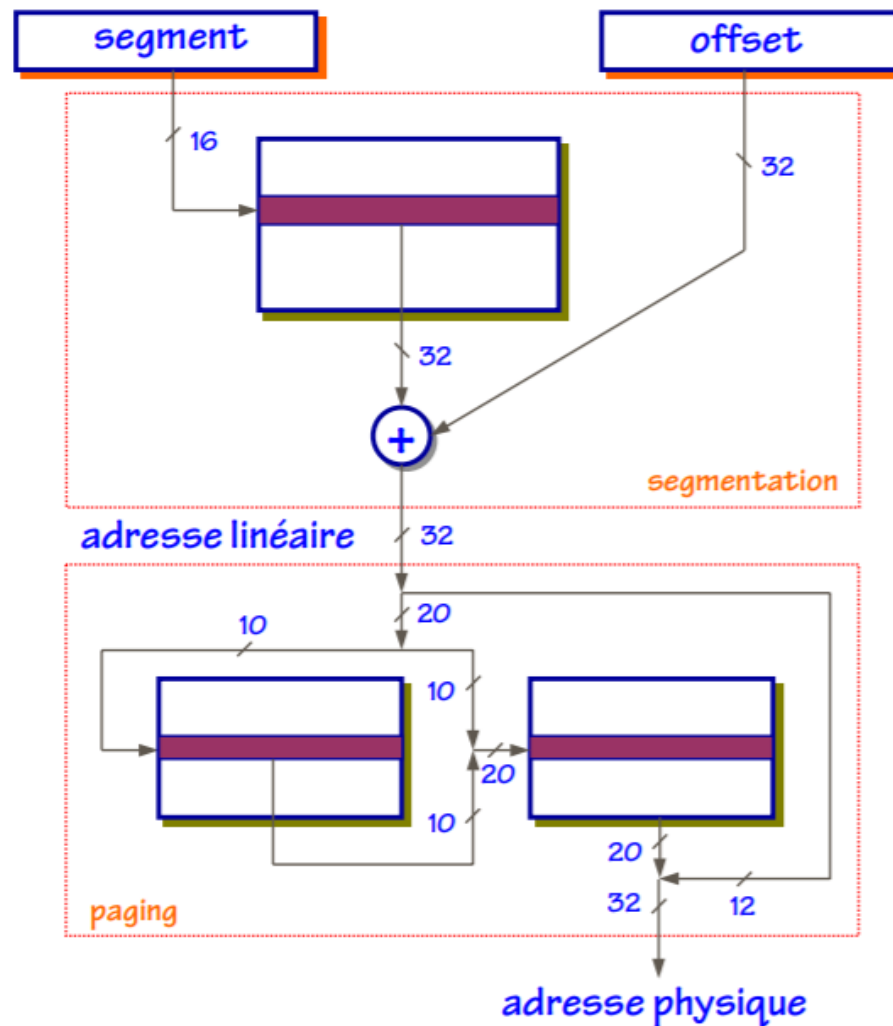


Mode réel  
(8086)



Mode protégé  
avec vérification de droit  
(privilège)

# Mode protégé : 386/486



# Descripteurs

- Contient les infos pour accès et vérification
  - Où, qui, quoi
  - Table globale GDT et table locale LDT (au prog.)
- Table GDT :  $8192 * 8$  octets ( = 64 Ko)
  - Base, limite, type, attributs
  - Stockée dans un segment accessible seulement par l'OS

# Erreurs détectées (par le hard)

- Un programme essaye d'accéder ailleurs
  - En dehors de ses zones définies
  - Segment inexistant ( ➔ indice gestion cache)
- Type de segment ex : programme accède à segment pile
- Lire plutôt qu'exécuter (violer définition-16 possibles)
  - Read Only
  - Read/Write
  - Read Only, expand down
  - Execute Only
  - ...

# Passer en mode protégé

- Par exemple appel routine système
  - Changement d'un bit dans un registre de contrôle
  - Vérification par le hard à chaque cycle machine
    - ➔ Très coûteux
- Rester en mode protégé (coûteux)
  - Préparation de contexte
    - Définir un segment code, un segment données (pile)
    - Configurer registre GDTR (Global Description Table Register)
    - Bit PM dans CR0
    - Charger registre de segment
    - Interruptions désactivées ...

# Evolution 32 → 64 bits

- Mode 32 bits x86 conservé (*Legacy Mode*)
- x86-64 : mode protégé en 64 bits (*Long mode*)
  - Mode de compatibilité prog 16/32
  - Taille de page de 4 Ko ou 2 Mo
  - Mémoire virtuelle adressée par 64 bits à plat
    - CS, DS, ES et SS sont traités comme s'ils valaient zéro.
    - FS et GS ont un traitement particulier.
  - Désactivation du mode virtuel 8086
  - Commutation de contexte confiée au logiciel
  - Nouvelle manière de changer le niveau de privilège

Retour aux optimisations  
en vue de performances



# Pipeline

- Introduit par le 8086 d'Intel début des 80's
- Inspiré des chaînes de montage
- 3 étages sur le 8086/15 en général/31 sur P4

## Pipeline RISC (classique)

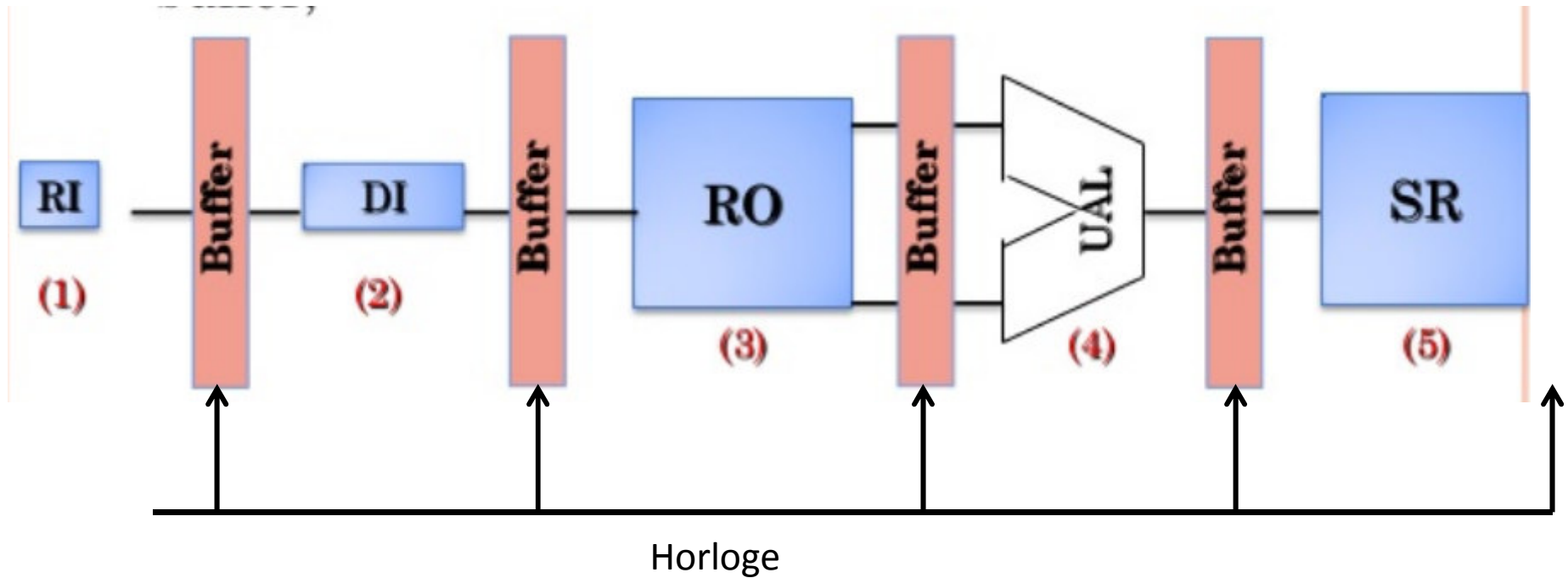
- Recherche instruction
- Décodage instruction
- Recherche opérandes
- Exécution opération
- Sauvegarde résultats

# Pipeline : profondeur

- Intel
  - i3, i5, i7 : 14
  - Core 2 Duo et Mobile : 14 et 12
  - P4 Prescott : 31 / P4 (architecture Prescott) : 20
  - Intel P3 : 10
- Processeurs AMD
  - K10 : 16 / Athlon 64 : 12 / AMD Athlon XP : 10
- Processeurs de type RISC
  - Sun UltraSparc IV : 14 / IBM Power PC 970 : 16



# Pipeline : mise en oeuvre matérielle

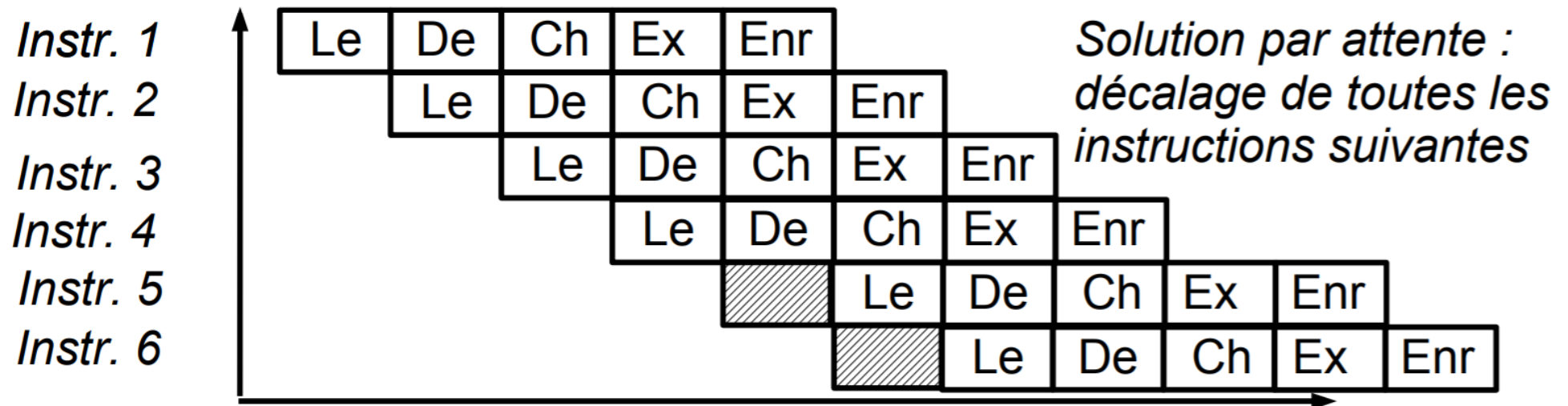
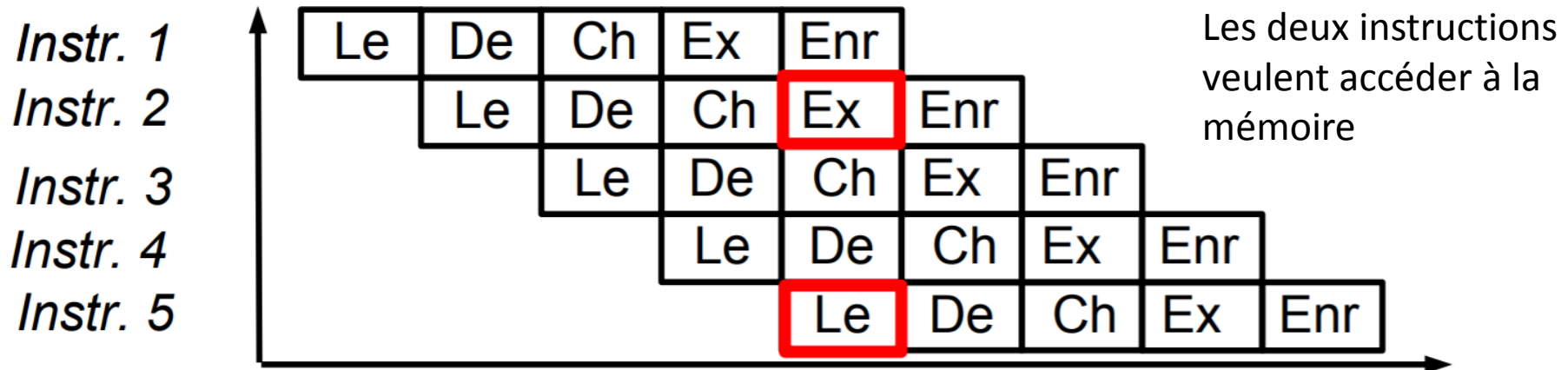


# Pipeline : difficultés (aléas)

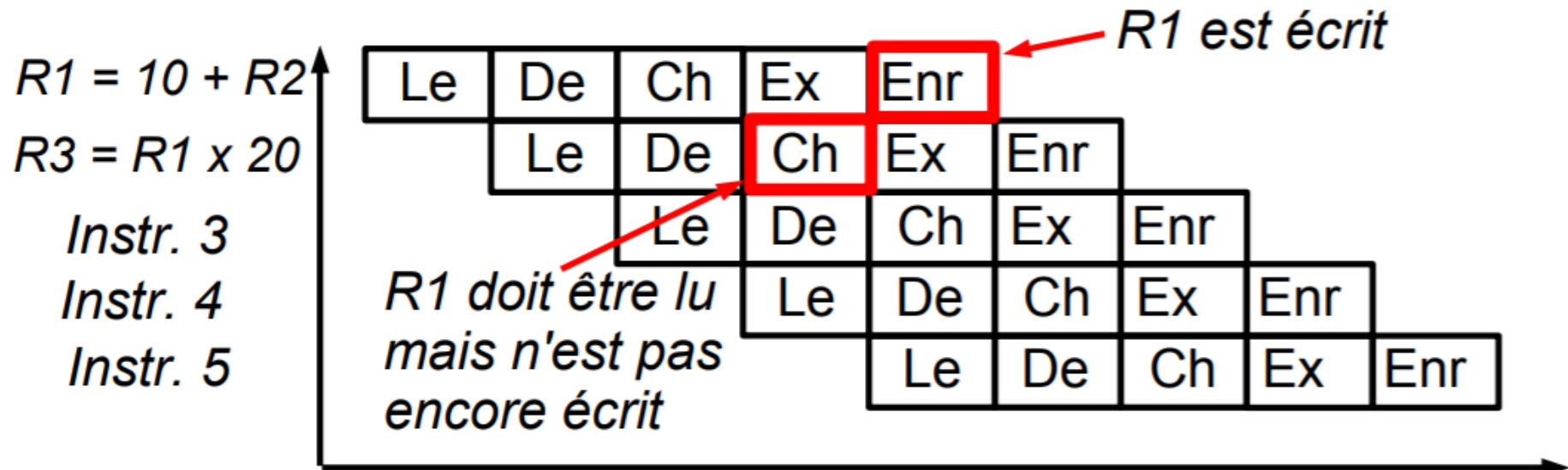
- **Aléa structurel** : deux instructions ont besoin de la même ressource (conflit de dépendance)
- **Aléa de données** : une instruction produit un résultat et une instruction déjà dans le pipeline doit l'utiliser alors qu'il n'est pas encore disponible
- **Aléa de contrôle** : à chaque fois instruction de branchement.  
Il faut attendre l'adresse de destination pour charger l'instruction suivante.  
Les instructions qui suivent le saut déjà dans le pipeline seront pour rien (?),  
il faudra alors vider le pipeline et le réamorcer (latence).  
Optimisation 1 (soft) : on peut spécifier après le branchement des instructions qui seront toujours exécutées.  
Optimisation 2 (hard) : **prédiction de branchement**, deviner le comportement le plus probable. Fiabilité de **90** à 95 %.
- **Sauvegarde contexte** : il faut tout sauver

Difficultés augmentent avec la profondeur du pipeline

# Aléa d'accès à la mémoire 📖



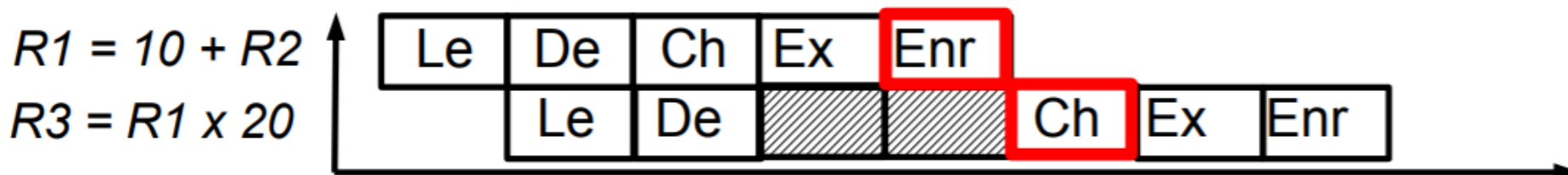
# Aléa d'accès aux registres



## Solutions

- Attendre : pas efficace
- Court-circuiter pipeline
- Changer l'ordre d'exécution (compilateur ou hardware)

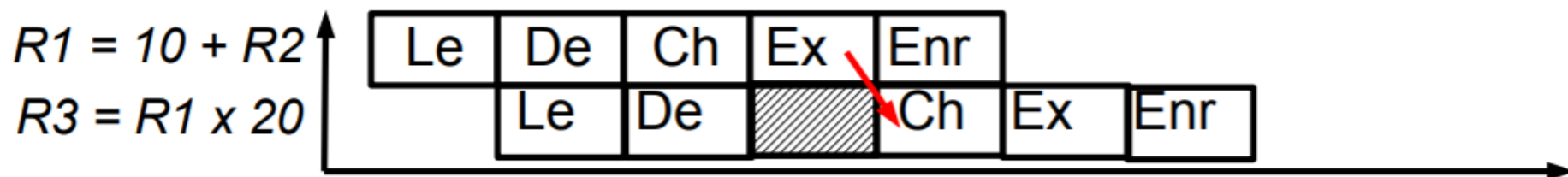
### ◆ Suspension du pipeline



- ◆ La deuxième instruction est suspendue tant que R1 n'est pas écrit

### ◆ Court-circuit du pipeline

- ◆ Après l'étape *EX* de la première instruction, on connaît la valeur de R1 : on la réinjecte directement dans l'UAL sans attendre son écriture au niveau du banc de registre



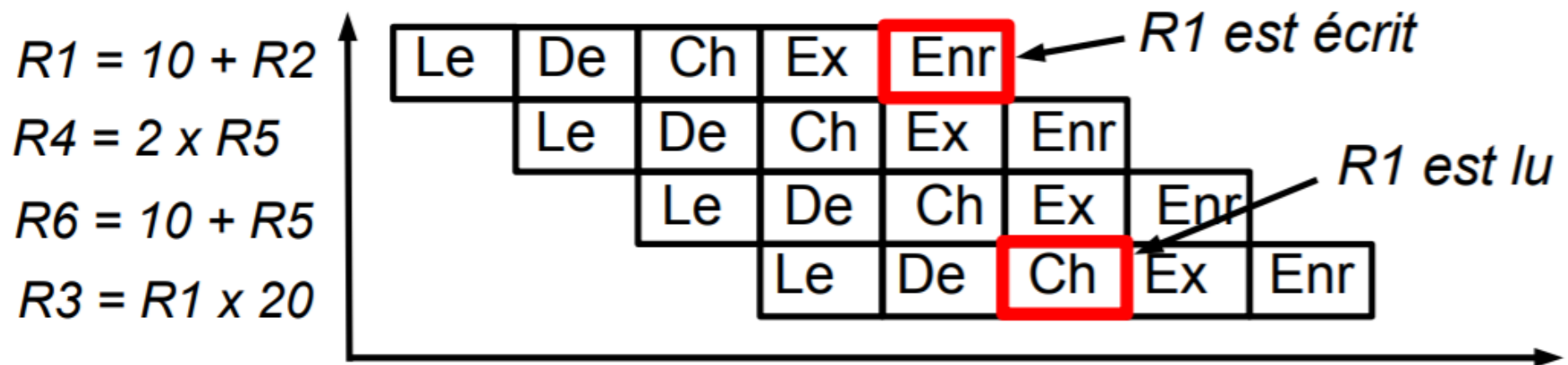


◆ **R1** = 10 + R2  
 R3 = **R1** x 20  
 R4 = 2 x R5  
 R6 = 10 + R5



◆ **R1** = 10 + R2  
 R4 = 2 x R5  
 R6 = 10 + R5  
 R3 = **R1** x 20

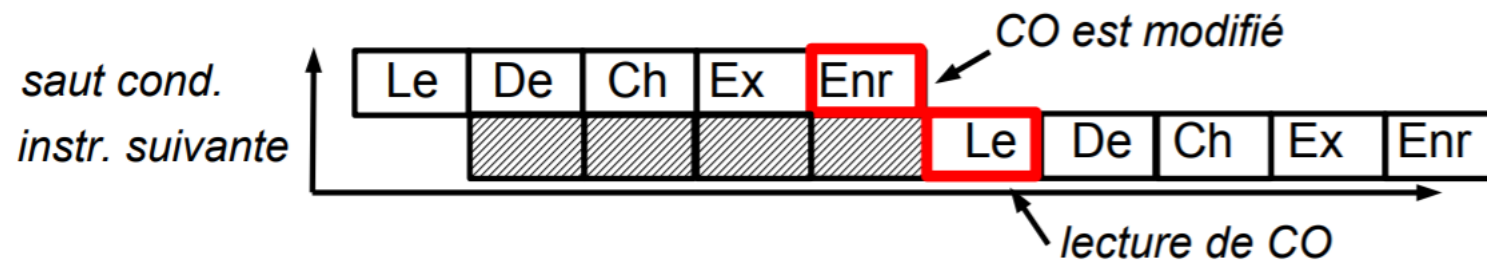
◆ Aléa de données : réordonnancement



# Aléa de contrôle

◆ `if (R1 > 30)`  
    `then R3 = 10 + R1`  
    `else R3 = 20 + R1`

◆ Solution avec attente



## ***Prédiction de branchement***

- ◆ Aléas de contrôle : prédictions de branchement pour en limiter les conséquences
  - ◆ Indispensable pour efficacité du pipeline
  - ◆ A l'aide de tables statistiques dynamiques
    - ◆ Prédit le résultat d'un test
    - ◆ On commence ensuite l'instruction suivante prédite
- ◆ Problème si prédiction erronée
  - ◆ On a commencé des calculs inutiles
  - ◆ Vidage du pipeline pour reprendre dans un état correct
    - ◆ Très coûteux en temps
    - ◆ Très pénalisant pour des pipelines profonds

# Prédiction : matériel

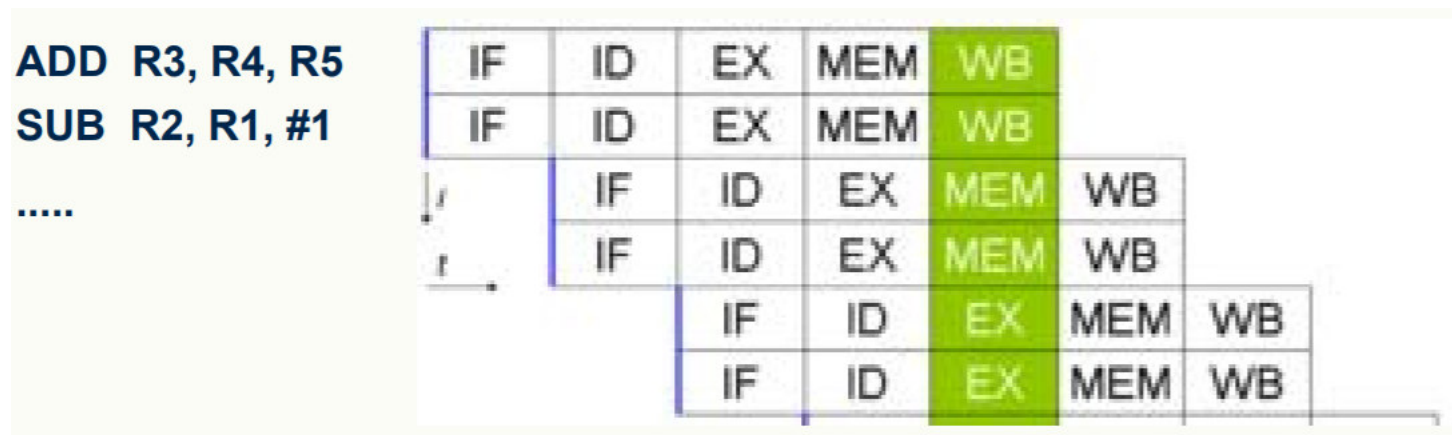
- Tampon de branches cibles (BTB)
  - Contient les adresses des branches
- Table des historiques de branchement (BHT)
  - Mémorisation des occurrences pour prédictions
  - Plusieurs algorithmes possibles (IP)
  - Exemple : 2 bits par branche
    - 00 : jamais / 01 : parfois / 10 : souvent / 11 : toujours
- Mise à jour dynamique de BTB et BHT
- Augmenter taille BTB/BHT : trouver compromis

# Pipeline : généralisation

- On utilise ce principe partout où c'est possible
  - Processeur : dans toutes ses sous-ensembles
  - Dans ALU/FPU
  - Mémoire synchrones : SDRAM
- Gestion très complexe
  - Synchronisation
  - Exceptions
  - Sauvegarde contexte
- L'optimisation se fait en « hard »

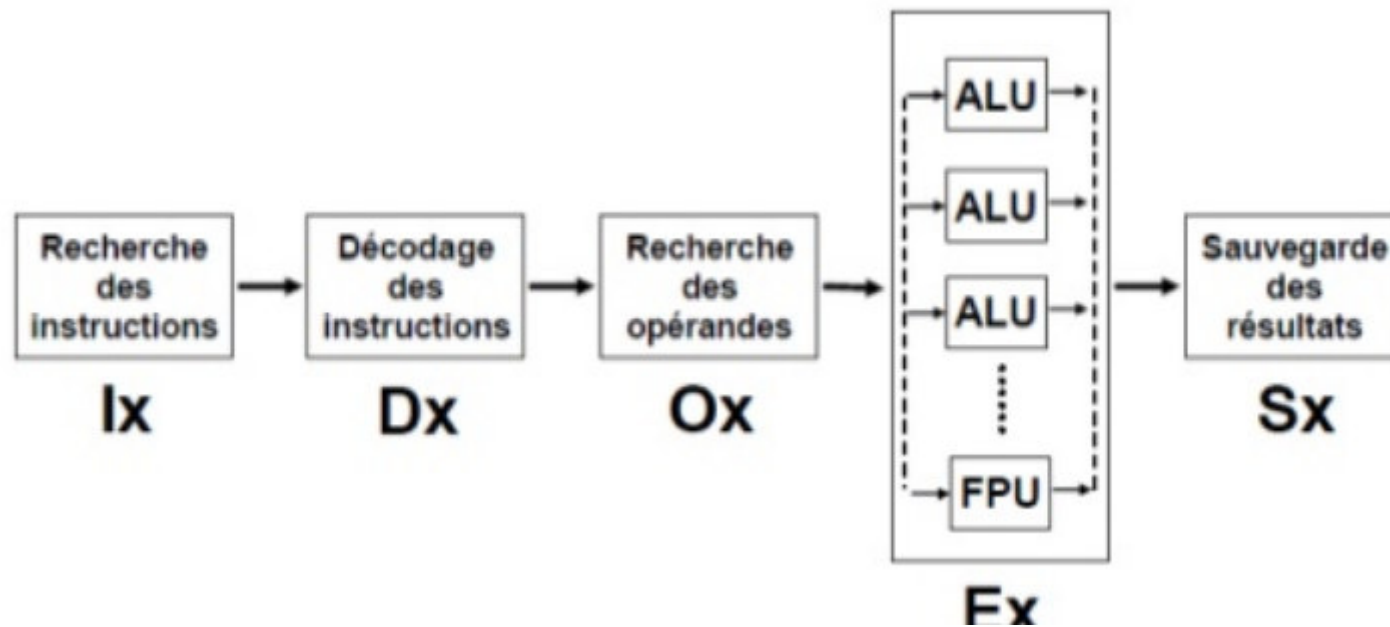
# Architecture superscalaire

- Augmenter le nombre d'instructions par cycle  
Plusieurs pipeline de traitement en parallèle

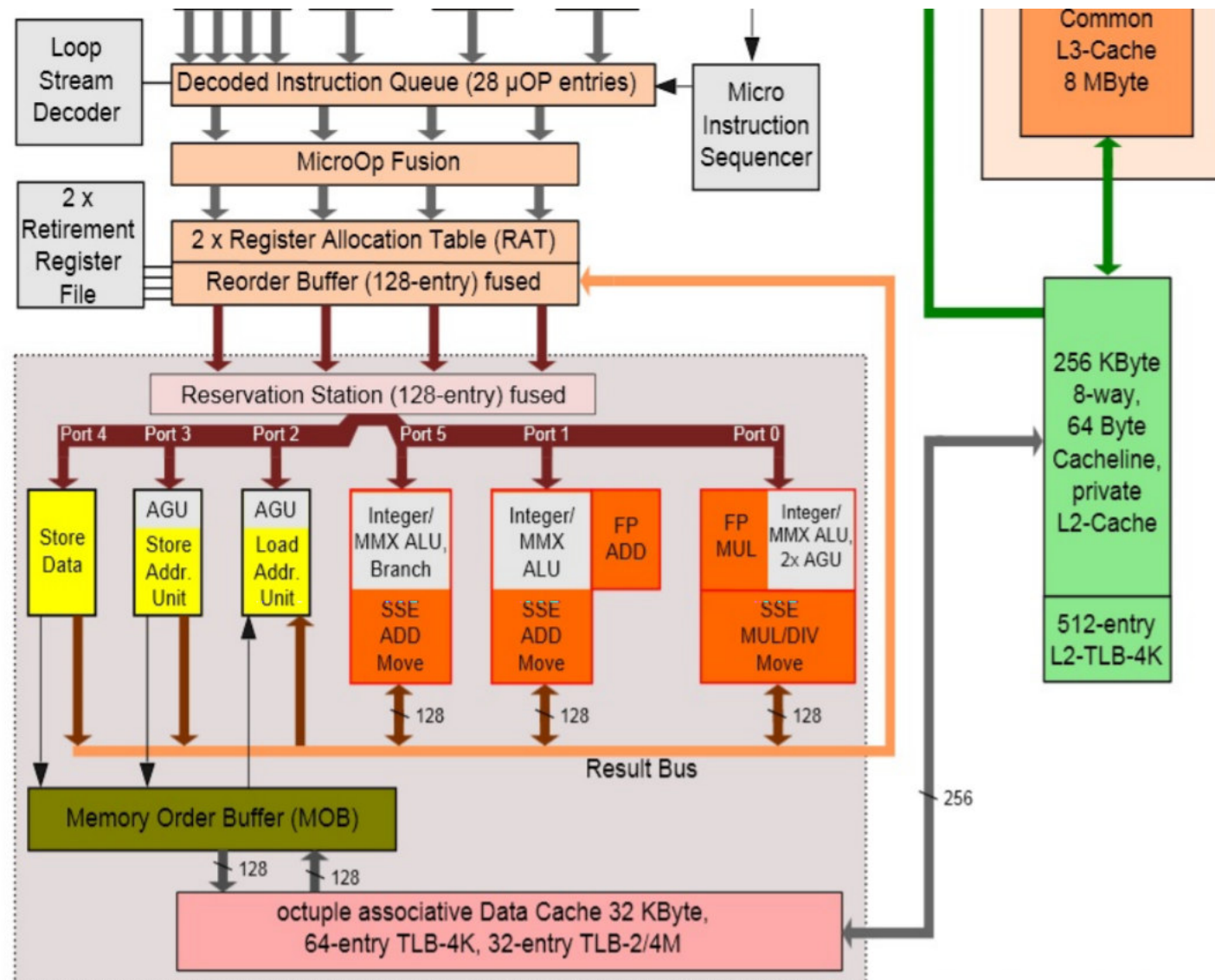


- Introduction en 1985
- Depuis mi-2000 : processeurs superscalaires

## Super scalaire simple

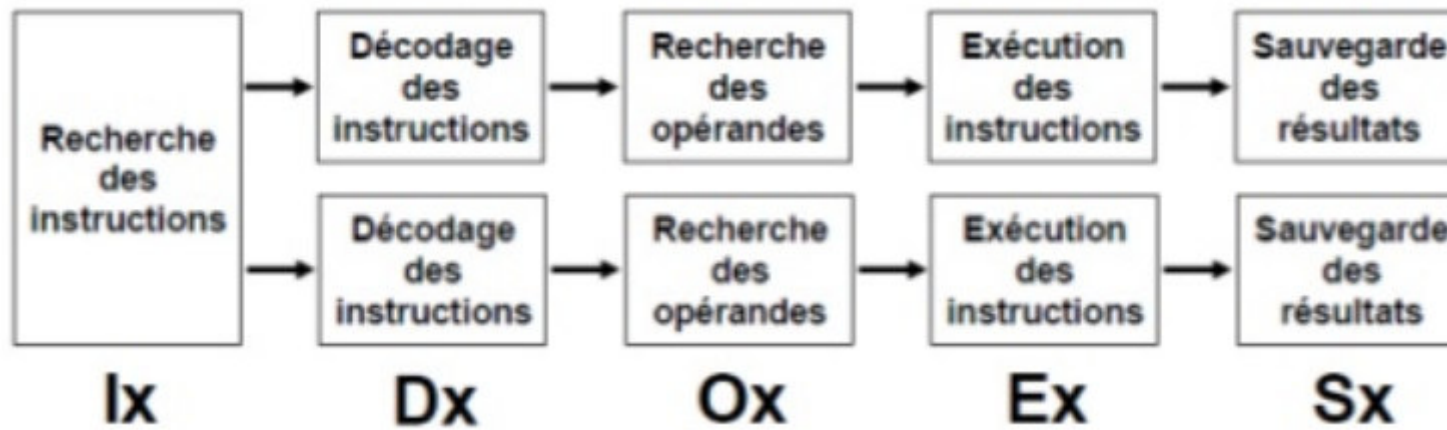


# Partie opérative Intel Nehalem





Super scalaire moins simple



# Pipeline/Superscalaire : Problèmes

Avantage : évolution de Von Neumann

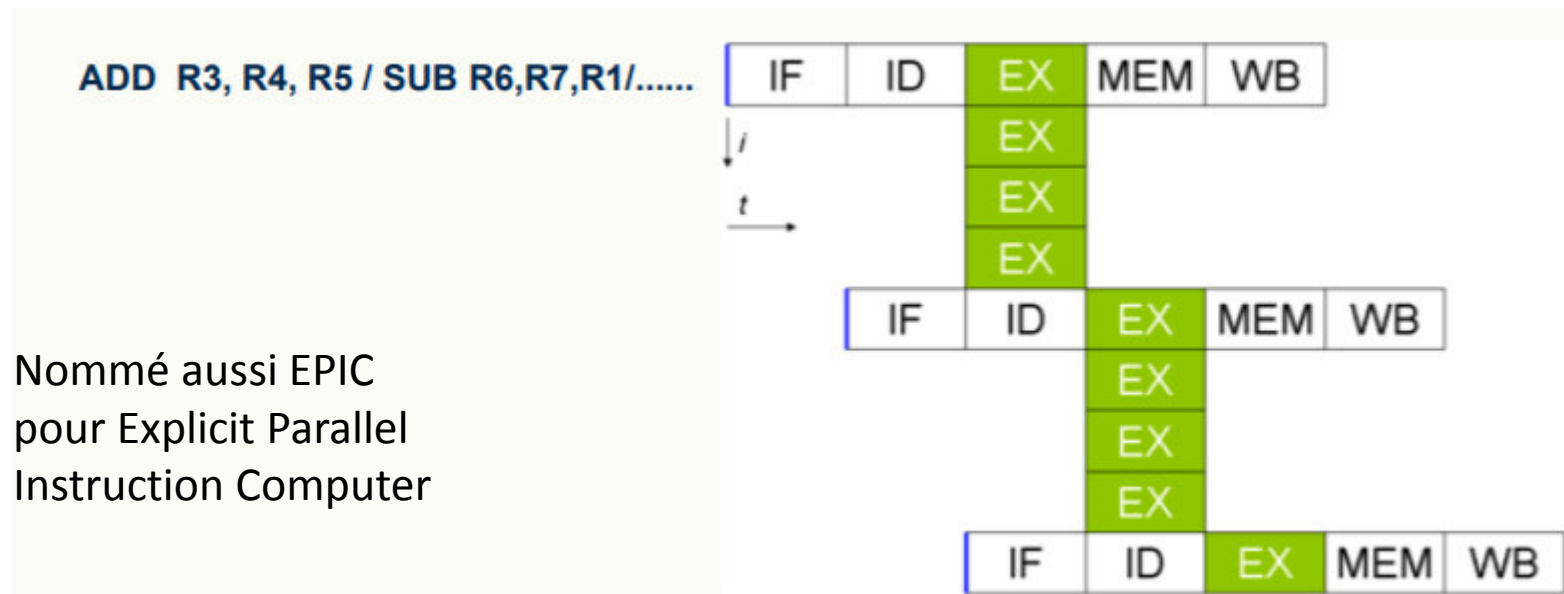
Même « style » de programmation/compilateurs

Mais difficultés

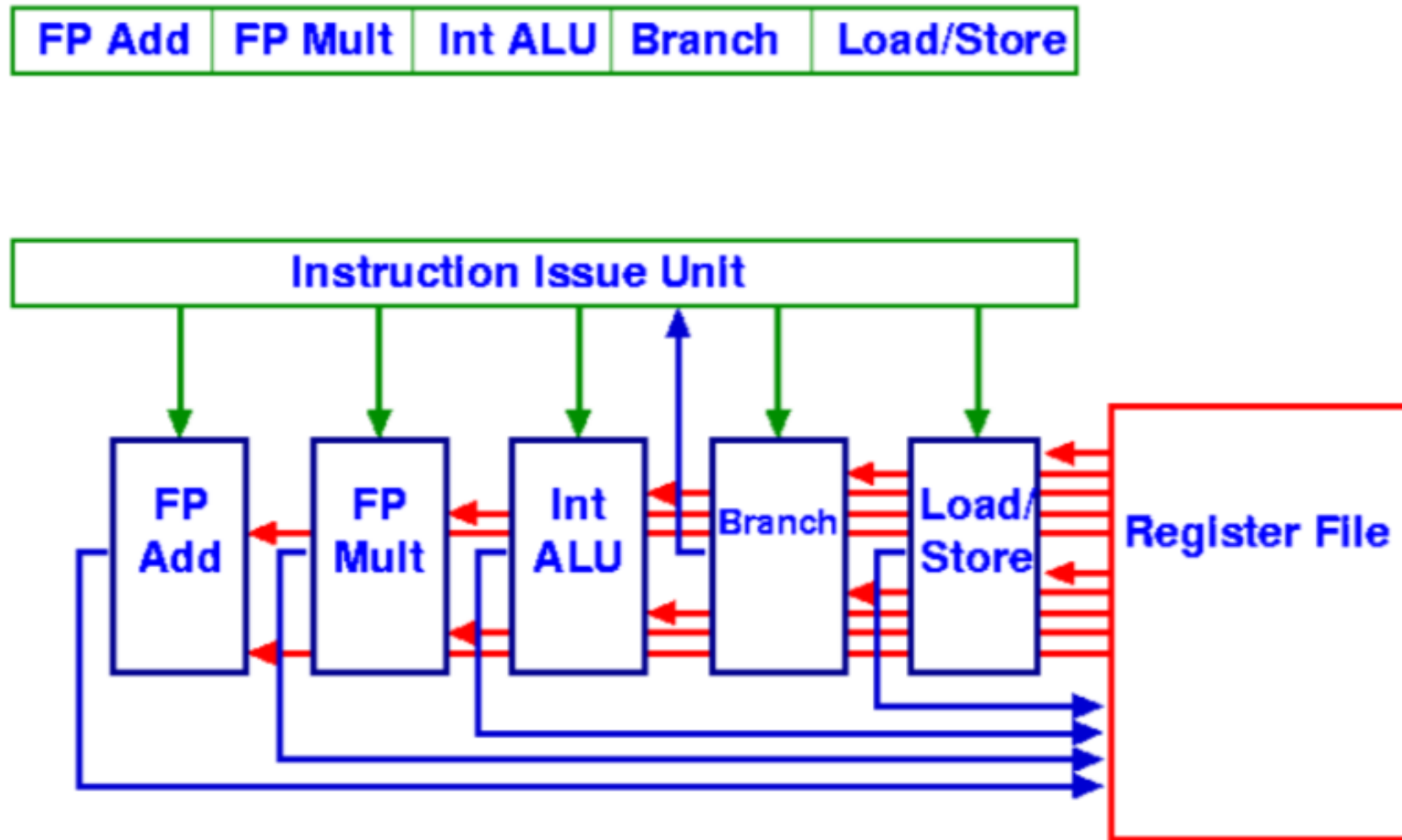
- Prévention des conflits
  - Prédiction de branchement (90% de cas)
  - Traitement des interruptions
    - Changement de contexte
- ➔ Une tentative de solution : Architecture VLIW

# Architecture VLIW

- Parallélisme intrinsèque dans l'instruction
  - Plusieurs opérations codées dans des champs
  - Instruction longues (Very Long Instruction Word)
  - Fabriquées par le compilateur

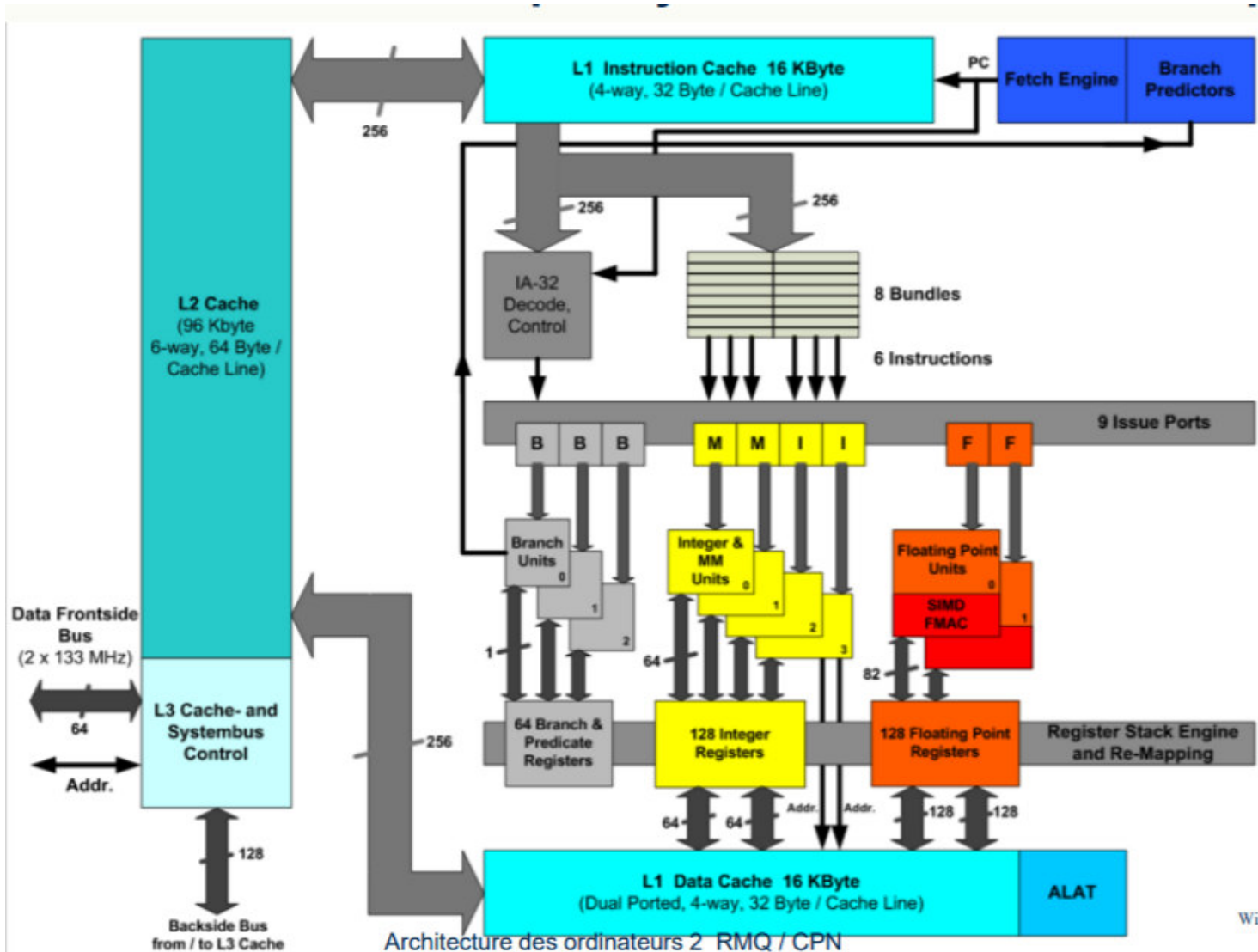


# Implication architecturale



L'architecture interne est superscalaire

# Exemple : HP-Itanium



# Intel + HP : IA-64

- Instruction : liasse de 128 bit
  - Trois instruction de 41 bits + masque de 5 bits
  - Instruction :
    - 3\*7 bits : noms de registre (128 registres possibles)
    - 6 bits de registre de prédicat
    - 13 bits : code opération
  - Masque : instructions en parallèle ? + chaînage
- Travail préparatoire du compilateur

# IA-64 : plus de branchement

**Si I égale 0**

**Alors** instruction 1

**Sinon** instruction 2

**RISC classique**

COMPARE I à 0

Si faux saute à l'étiquette SINON

ALORS: instruction 1

saute à l'étiquette SUITE

SINON: instruction 2

SUITE:

**EPIC**

P1 et P2 registres de prédiction

IA-64 a 64 registres de prédiction

COMPARE I à 0

commence à décoder instruction 1

prédicat positionné → P1

commence à décoder instruction 2

prédicat positionné → P2

si I égal 0, P1 = vrai, P2 = faux

Calculer/délivrer résultats des instructions

dont le prédicat pointe registre = vrai (P1)

# Synthèse



	CISC	RISC	Superscalar	VLIW
<b>Instruction size</b>	variable size	fixed size	fixed size	fixed size (but large)
<b>Instruction format</b>	variable format	fixed format	fixed format	fixed format
<b>Registers</b>	few, some special	many GP	GP and rename (RUU)	many, many GP
<b>Memory reference</b>	embedded in many instr's	load/store	load/store	load/store
<b>Key Issues</b>	decode complexity	data forwarding, hazards	hardware dependency resolution	code scheduling, (compiler)
<b>Instruction flow</b>		<pre> IF ID EX M WB   IF ID EX M WB     IF ID EX M WB </pre>	<pre> IF ID EX M WB IF ID EX M WB   IF ID EX M WB   IF ID EX M WB </pre>	<pre> IF ID EX M WB IF ID EX M WB   IF ID EX M WB   IF ID EX M WB </pre>



# Architecture vectorielle

- Même calcul (SIMD) sur données structurées
  - Parallélisme et pipeline
- Architecture plus spécialisée
  - Simulation : météo, calcul matriciel
  - Graphiques
- Exemple :  $R = A * B$  avec A et B des vecteurs
  - Une seule instruction « vectorielle »
  - Parallélisation / boucles automatiques ...
- Pas de dépendance ➔ pipeline optimal

# Ressources vectorielles

5.1	15.0	5.1	20.0	Vecteur 1
+				
10.0	12.5	5.1	5.1	Vecteur 2
=				
15.1	17.5	10.2	25.1	Vecteur résultat

Matériel adapté

- registres vectoriels
- mise en parallèle d'opérateurs
- utilisation du pipeline

Instructions spécialisées

Préparation par le compilateur  
(vectorisation)

Exemple basique : déroulage de boucles

```
int i;  
for (i = 0; i < 100; ++i)  
{  
    a[i] = b[i] * 7;  
}
```



```
int i;  
for (i = 0; i < 100; i+=4)  
{  
    a[i] = b[i] * 7;  
    a[i+1] = b[i+1] * 7;  
    a[i+2] = b[i+2] * 7;  
    a[i+3] = b[i+3] * 7;  
}
```

Exécutées en //

# Exemple d'instructions vectorielles

Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e., $R1 + i \times R2$ ).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index).
SVI	(R1+V2),V1	Store V1 to vector whose elements are at $R1 + V2(i)$ (i.e., V2 is an index).
CVI	V1,R1	Create an index vector by storing the values $0, 1 \times R1, 2 \times R1, \dots, 63 \times R1$ into V1.
S--VV.D	V1,V2	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
S--VS.D	V1,F0	
POP	R1,VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR,R1	Move contents of R1 to vector-length register VL.
MFC1	R1,VLR	Move the contents of vector-length register VL to R1.
MVTM	VM,F0	Move contents of F0 to vector-mask register VM.
MVFM	F0,VM	Move contents of vector-mask register VM to F0.

Avec registres  
vectoriels

VLR et VM: dans le co-processeur

# Pour approfondir

<http://wcours.gel.ulaval.ca/2016/a/GIF3000/default/5chronologie/arch-06-vectorielles.pdf>



# Multi/Super/Hyper Threading

# Définitions

**Thread** ou **fil** (d'exécution) ou **tâche**

(terme par ISO/CEI2382-7:2000)

Exécution d'une suite d'instructions.

Exécution possible en parallèle apparent

Pratique : Partie indépendante du code pouvant être exécutée seule

# (Temporal) Multithreading


- Le système d'exploitation exécute les différents threads en temps partagé sur le processeur
  - Thread après thread
  - Ou entrelacement : Changement de contexte
- L'utilisateur « voit » du calcul parallèle alors qu'il est multiplexé temporellement

# Evolution

- (Temporal) Multithreading
  - exécutions non simultanées des threads
  - Un seul à la fois (les uns à la suite des autres)
  - «illusion de parallélisme »
- (Temporal) Superthreading [terme non général]
  - exécutions des threads simultanément
  - entrelacement (time-slicing)
- Simultaneous multithreading (SMT) [Hyper intel]
  - exécution simultanée - ressources propres/partagées



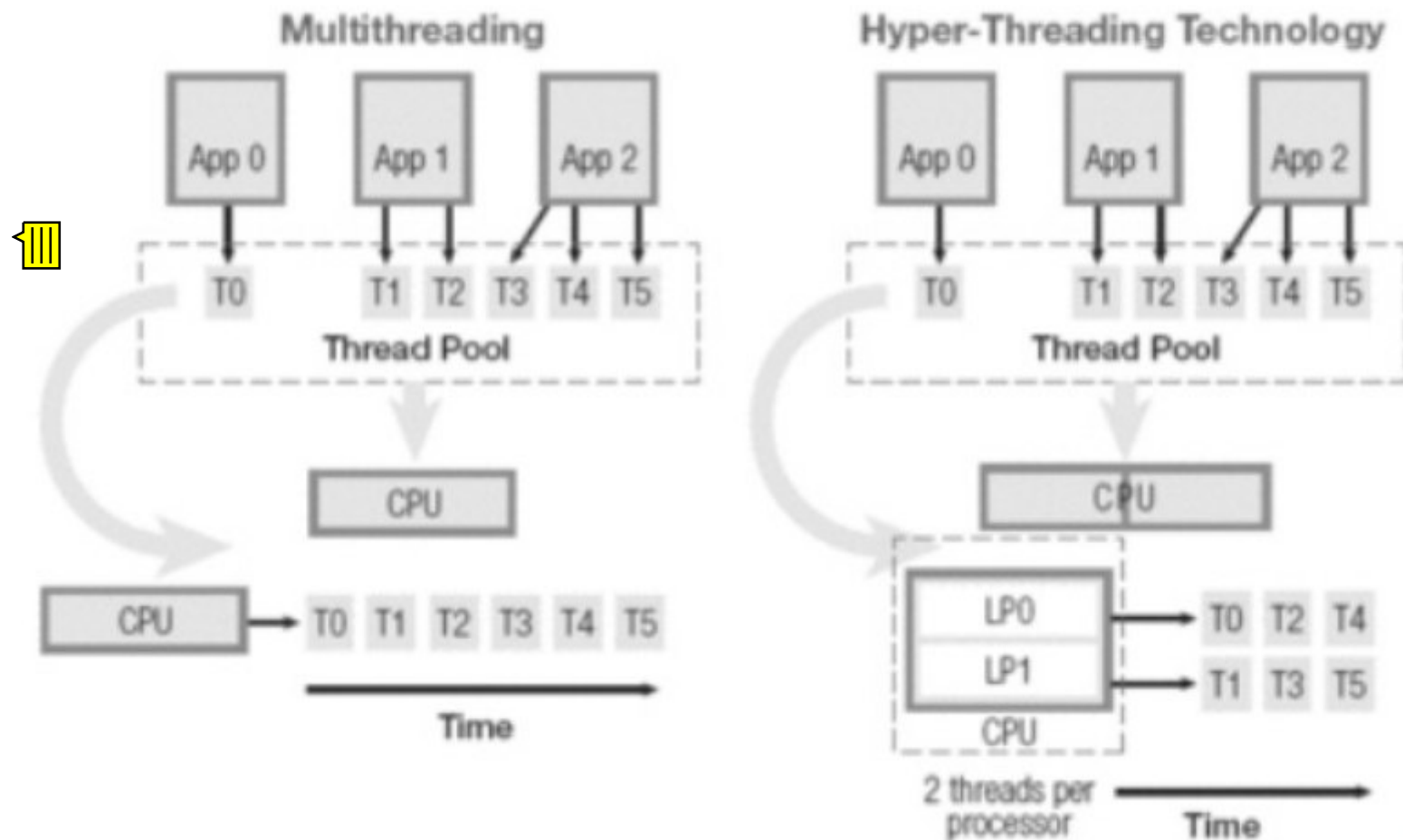
# SMT 2 voies ou Hyperthreading

- Duplication dans la processeur (\*2)
  - Registres
  - Registre de contrôle
  - Contrôleur d'interruption 
- Partage
  - Pipeline
  - Registre
  - Cache L1
- Cœur physique (1) vs coeur logique (2)

# Ordonnancement

- Les appli sont découpés en threads
  - Thread pool (liste des threads à exécuter)
- L'OS va distribuer les threads
  - Mono-core, mono thread : une à la suite des autres
  - Multi-core, mono Thread : équilibrage de charge
  - Mono-core, HT : deux par deux, gestion de files
  - Mutli-core, HT : équilibrage de charge

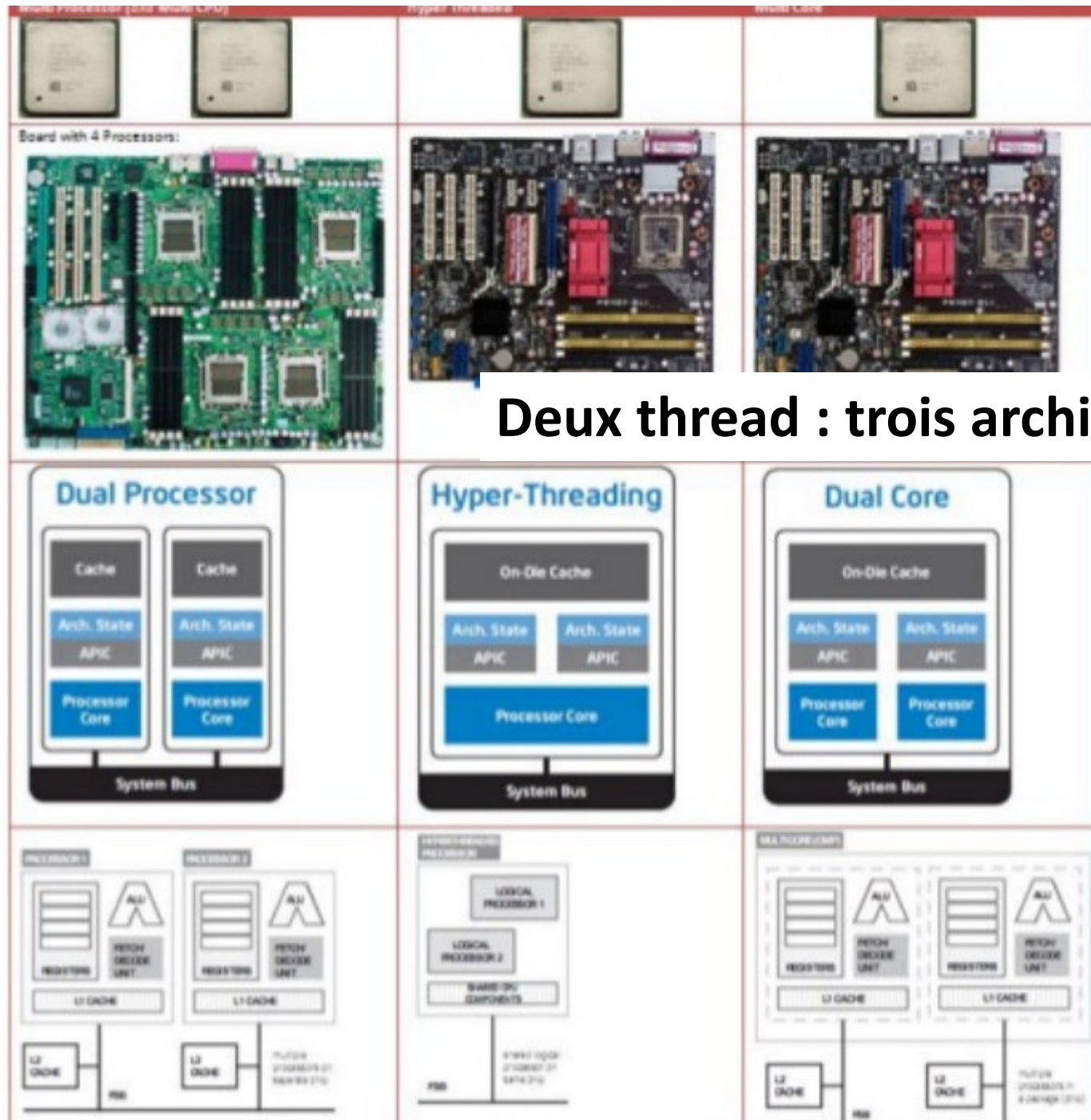
# MULTITHREADING VS HYPER-THREADING



Plusieurs processeurs logiques dans un processeur physique  
Première fois : Pentium 4 à 3,6 GHz

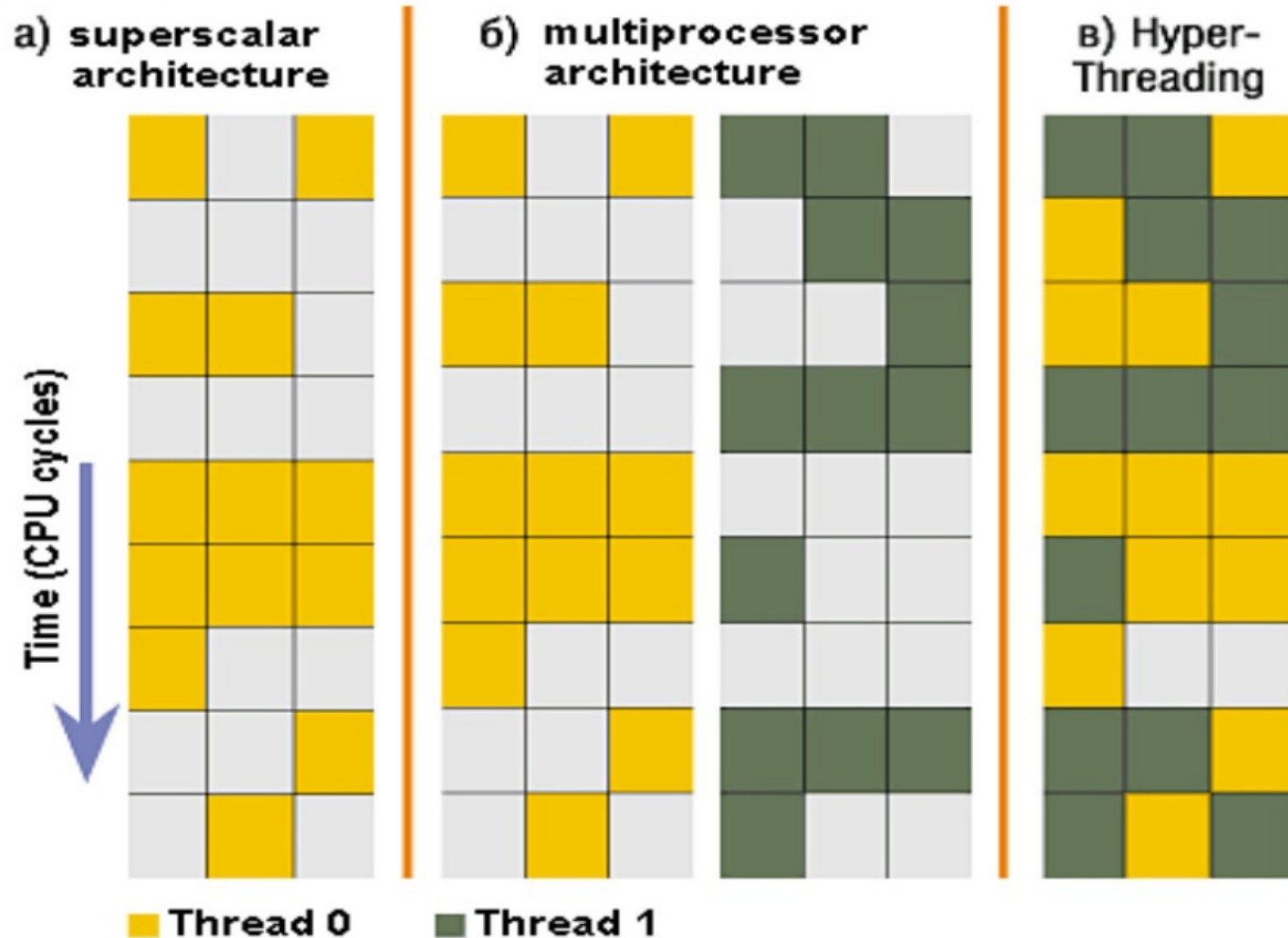
# Revendications

- Selon Intel,
  - première mise en œuvre : 5 % de surface en plus
  - gain en performance entre 15 et 30 %
  - jusqu'à 30 % Pentium 4 sans vs avec
- Cas possible (rare) : moins performant
  - Mauvaise gestion du pipeline
  - Saturation croisée du cache
- Mai 2005 : Thread malveillant possible
  - Privilèges limités mais vol d'infos d'un autre thread



Deux thread : trois architectures

# Revendications : comparaison



# Réalité

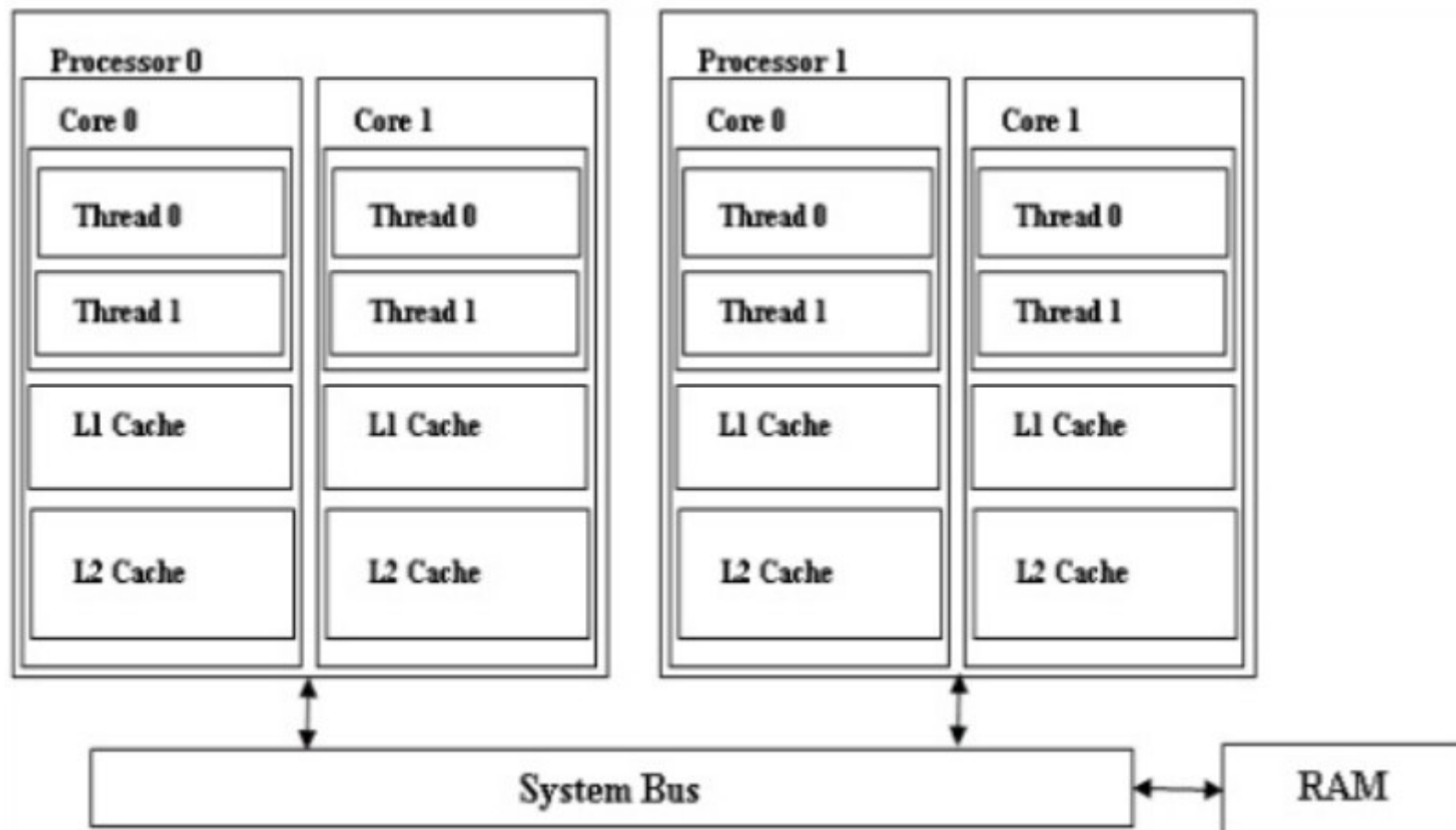
- Les performances effective dépendront
  - Des relations entre les threads et de leur longueur
  - De la gestion des caches
    - Taille et localité des données
  - De l'efficacité de l'OS
  - De la charge globale de calcul
  - De la charge des différents bus
  - ...
- Performances non modélisables (prédictibles)

Multi-core



# Multi-core (cœurs)

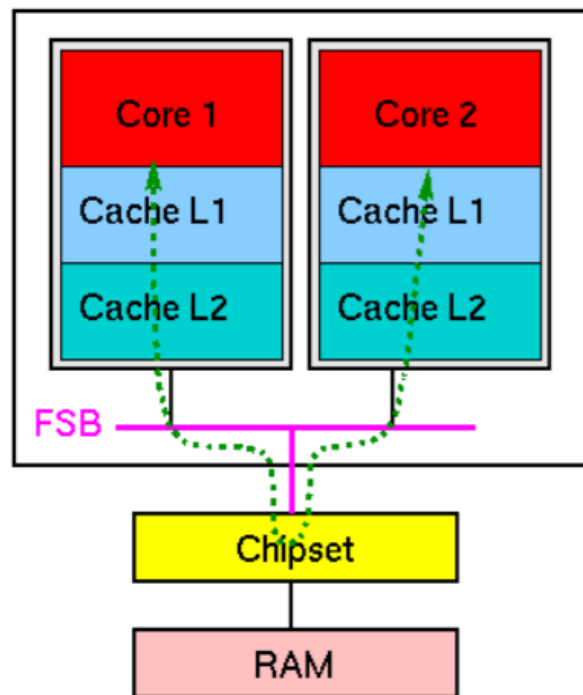
- Plusieurs CPU en parallèle dans un même processeur
- Traitement d'un ou de deux threads par CPU  
2/4/8 coeurs (> 64 ?)
- Multi-core homogène ou hétérogène
- Attention : 8 cœurs physiques à 2 threads
  - De 8 à 16 threads « simultanés »
  - Différent de 16 cœurs physiques



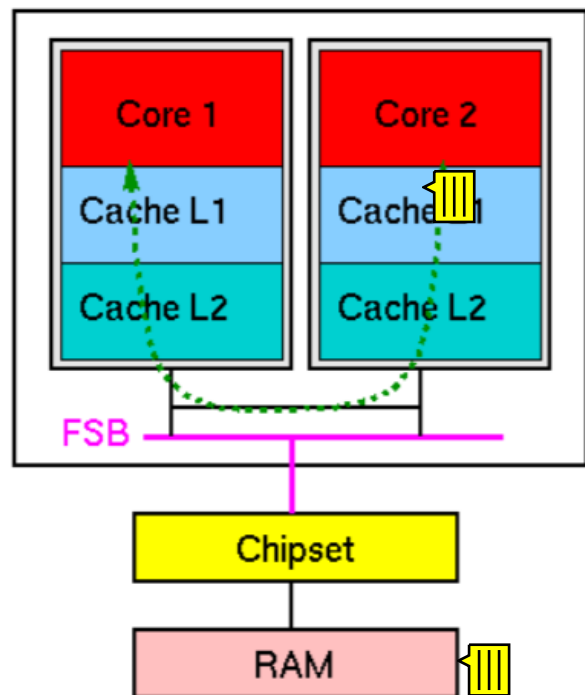
# Multi-core simple → complexe

- Deux dies (CIs) dans le même chip  
(Pentium D)
  - Echanges par le chipset → goulot
- 2 processeurs indépendants sur le même die  
(Athlon 64X2)
  - Ressources propres
- 2 ou plusieurs processeurs sur le même die  
(CORE2 duo)
  - Ressources partagées : cache L2 (L3) ...
  - Echanges optimisés

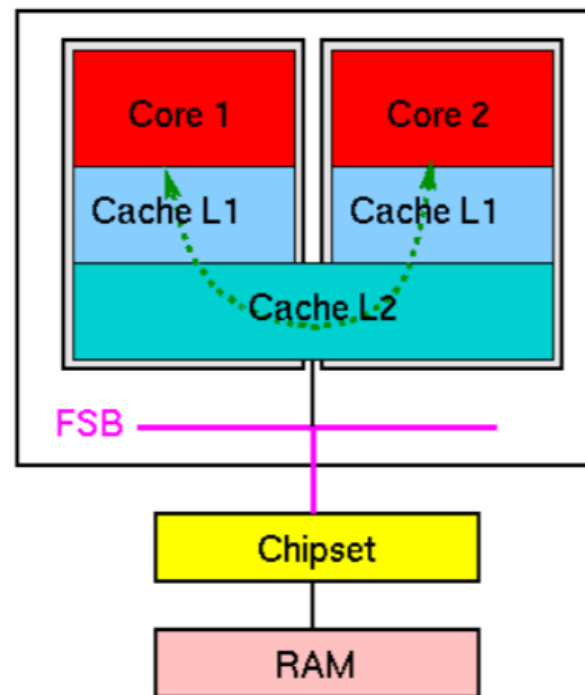
*Pentium D*




*Athlon X2*



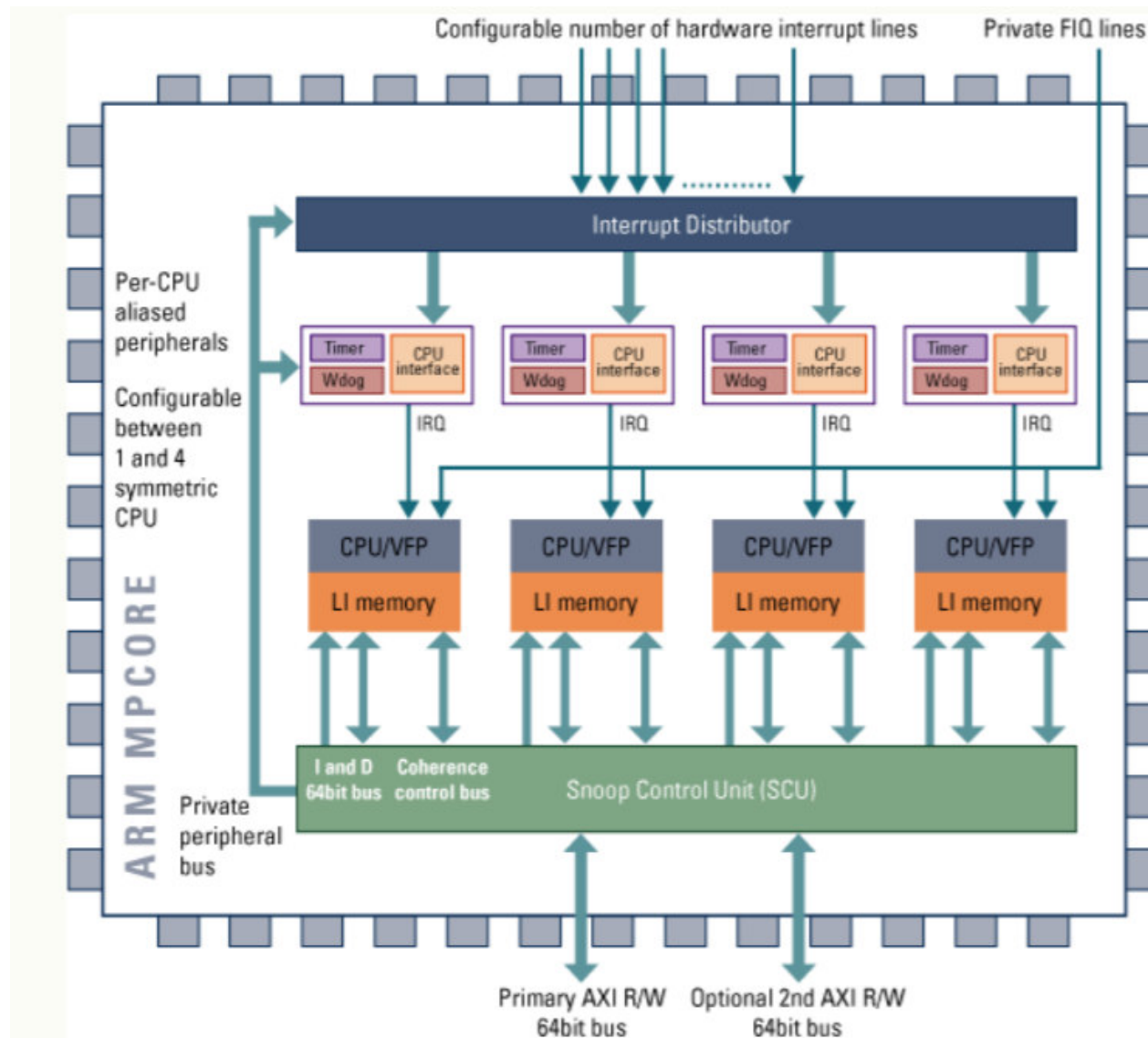
*Core 2 Duo*



# Multi-core : exemples

- Multi-cœurs homogènes
  - Intel Core2 : 2 ou 4 cœurs 
  - Sun UltraSPARCT1 : 8 cœurs, 32 threads
  - ARM11 MPcore : 2 ou 4 cœurs
- Multi-cœurs hétérogènes
  - IBM CellBe 1 PowerPC + 8 CPU spécialisés
  - AMD Fusion

# ARM11-MPCore



# Multiprocesseur

- Massively Parallel Processing (MPP)
  - plusieurs processeurs communiquants
  - chaque processeur → mémoire, OS et application
- Symetric Multi Processing (SMP)
  - plusieurs coeurs + une mémoire partagée (cache)  
une seule instance de l'OS et de l'application
- Non-Uniform Memory Access (NUMA)  
plusieurs coeurs + mémoire locale mémoire  
partagée (cache)



# GPU : Graphical Processor Unit


- Idée ancienne: ajouter coprocesseur graphiques
- 2000s : accélérateurs de calculs
  - ➔ instructions spécifiques graphiques, pipelinées et vectorisées

Réflexion : implémenter des instructions générales pour traiter le graphique ?

- 2006 : Premier GPU graphique/général haute performance : NVIDIA GT 80 chip/GeForce 8800
- Voir cours : « Processeurs spécialisés »



# Mesure de performances

- Utilisation de **benchmarks** (tests standards)
  - Logiciel réalisant un certain calcul/traitement
  - Le même traitement sur tous les processeurs 
    - ➔ comparaison temps d'exécution (+autres performances)
- 2 benchmarks pour calculs
  - Spec Int 2000 : calculs sur des entiers
  - Spec Fp 2000 : calculs sur des flottants
- Très nombreux benchmarks : spécialisation

# Exemples de benchmarks

- Performances de calculs purs  
(scalaire, vecteurs, matrice ...)
- Gestion des applicatifs  
(Excel, Word, Base de données ...)
- Performances graphiques  
(notamment pour le jeu)
- Gestion du multi-core, hypertreading ...
- I/O, consommation, gestion mémoire ..

Comme pour les tests de pollution pour les voitures les processeurs peuvent détecter les benchmarks