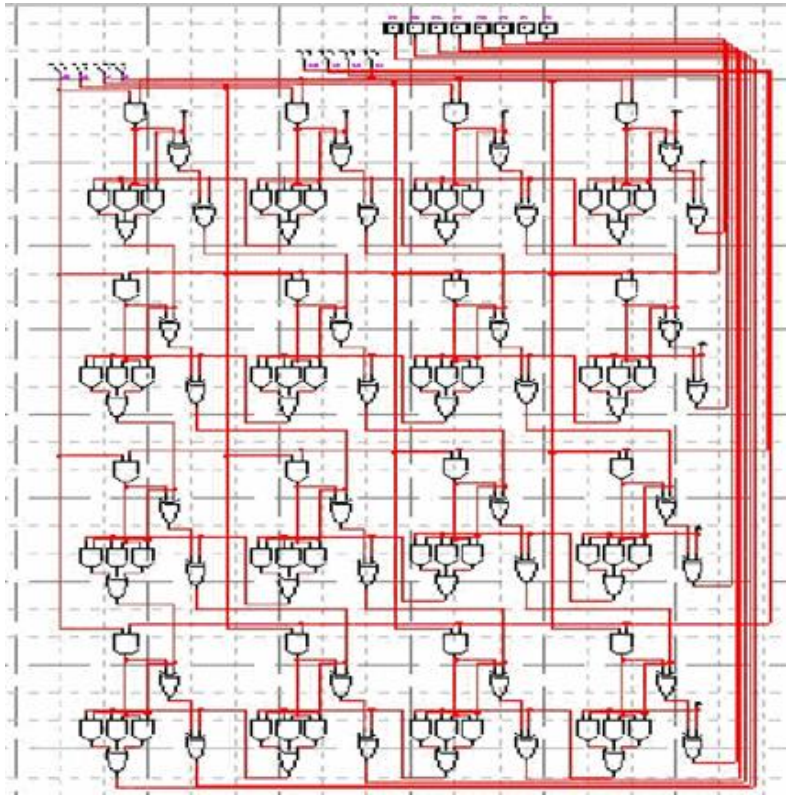
Architecture des ordinateurs à mémoire commune

- Prendre connaissance des technologies en jeu
  - Pour optimiser les programmes
    - Éviter les bévues
    - Mieux coder
    - Déterminer les options de compilation adéquate
  - Pour mieux interpréter les performances
- Références
  - *Computer architecture: a quantitative approach* par John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau
  - http://www.irisa.fr/caps/people/michaud/ARA/

Où trouve-t-on du parallélisme

- Au niveau des circuits

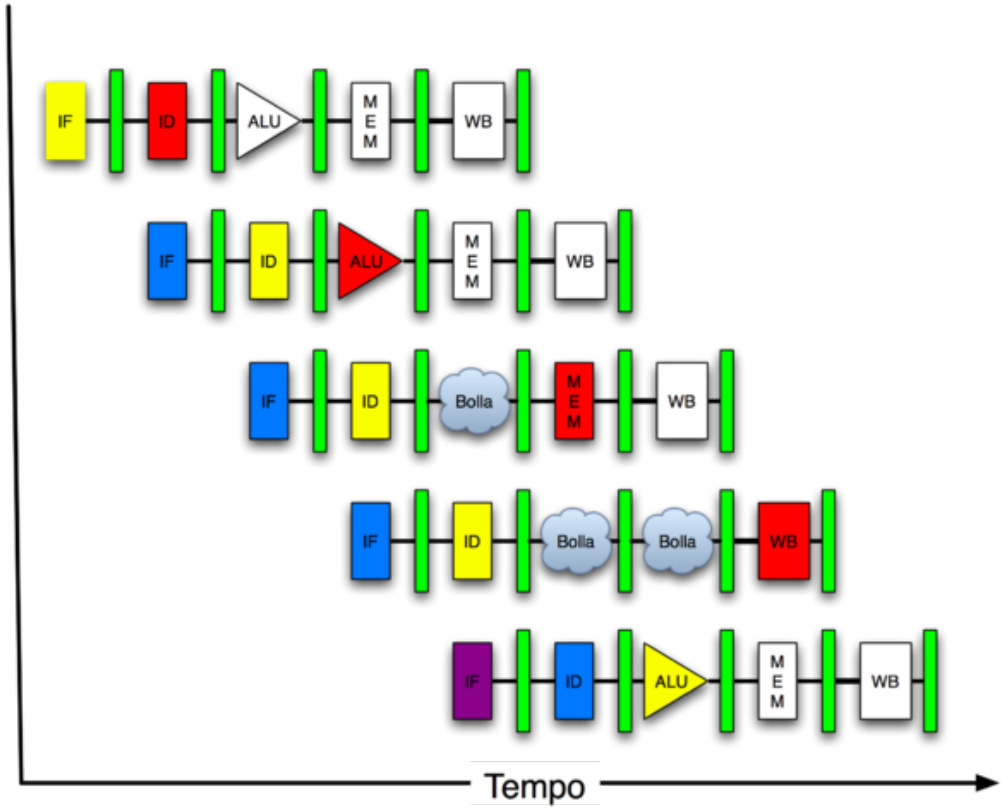
Multiplicateur 4x4



Où trouve-t-on du parallélisme

- Au niveau des circuits
- Au niveau des instructions

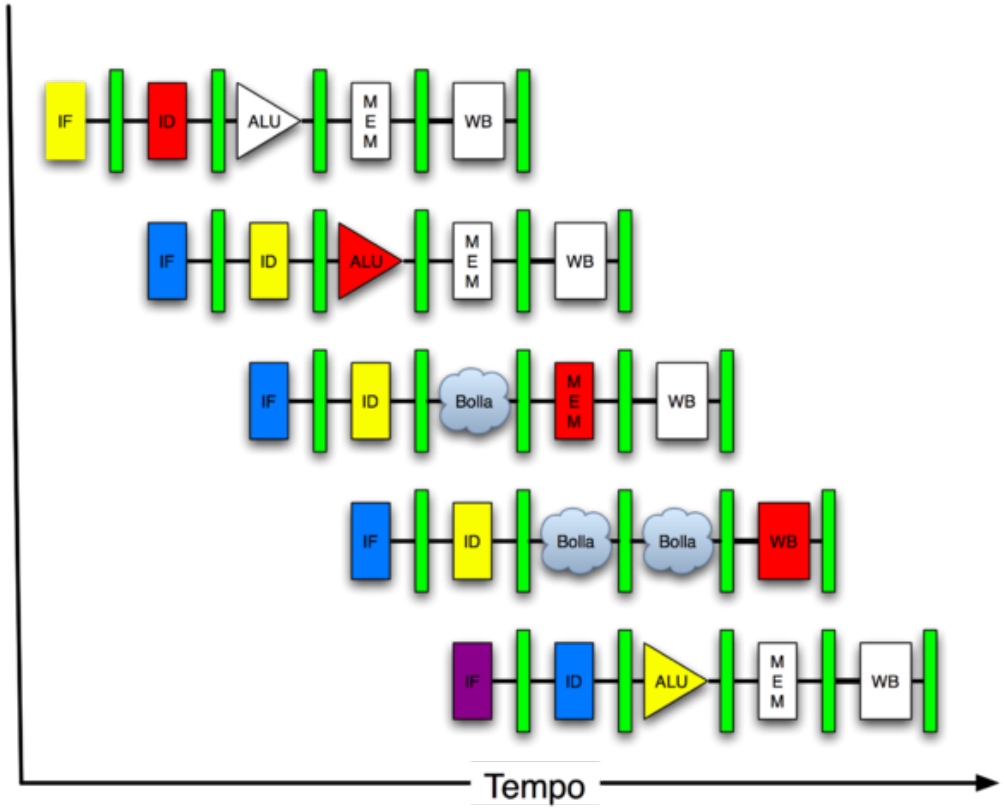
Pipeline 5 étages



Où trouve-t-on du parallélisme

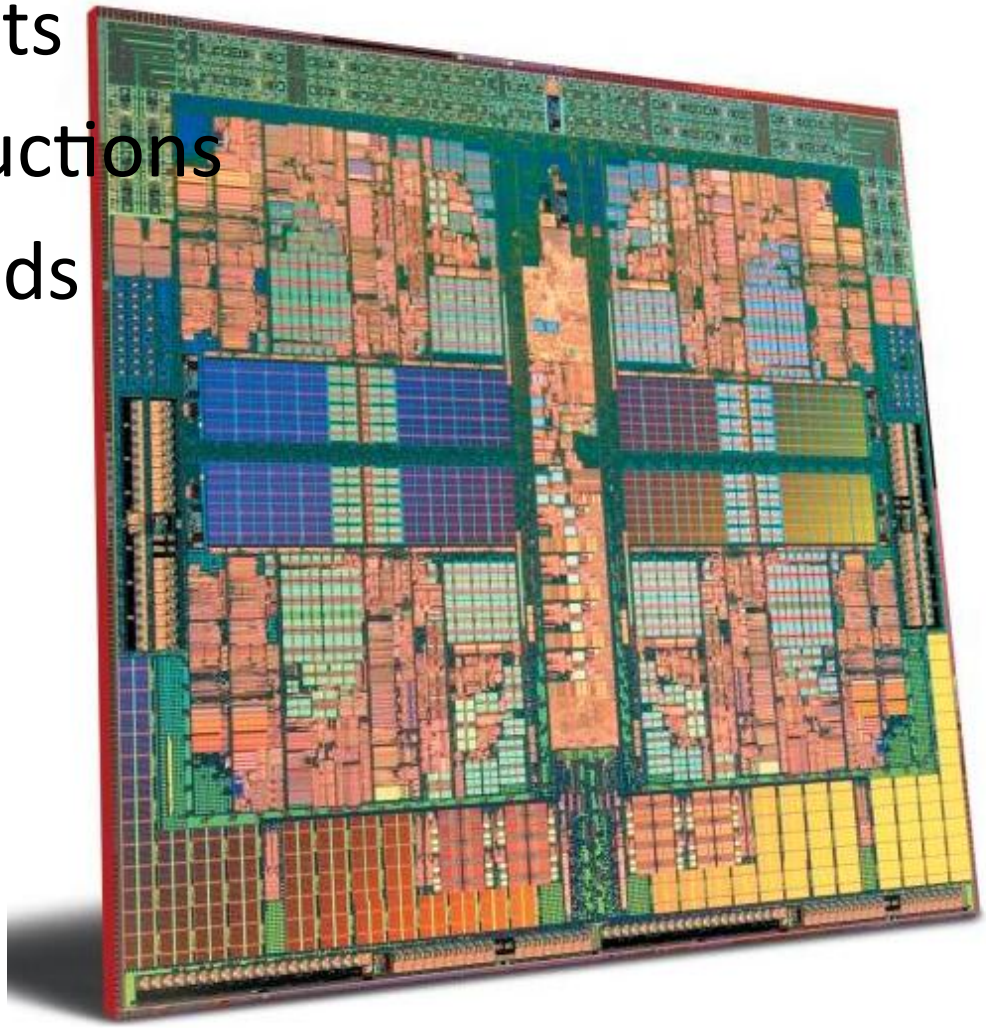
- Au niveau des circuits
- Au niveau des instructions

Pipeline 5 étages



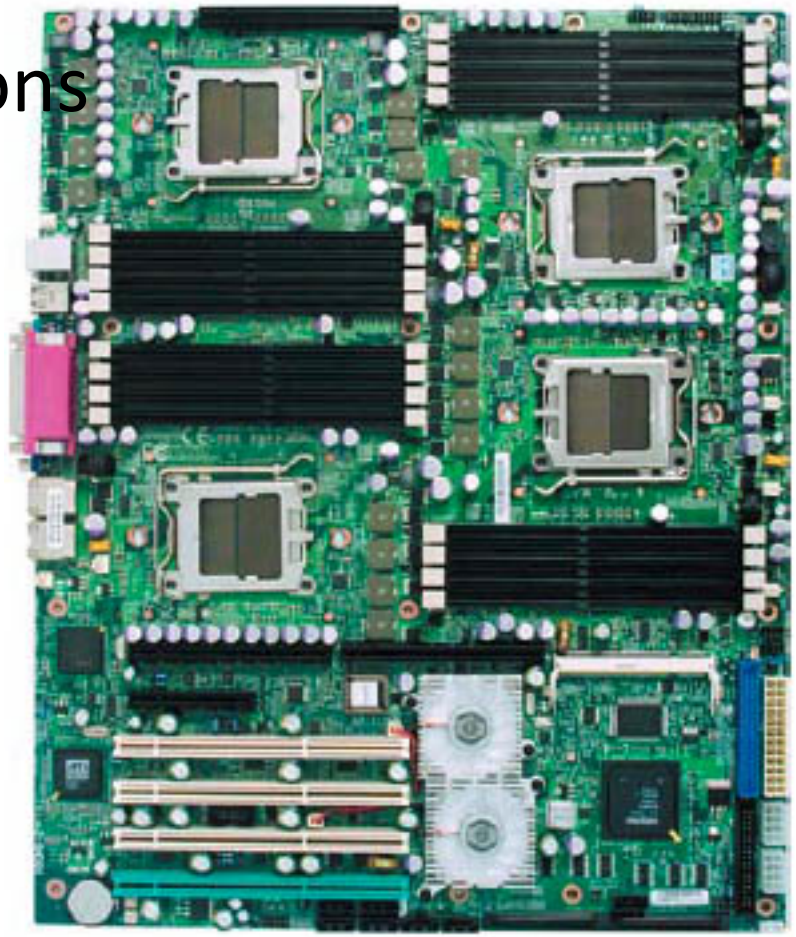
Où trouve-t-on du parallélisme

- Au niveau des circuits
- Au niveau des instructions
- Au niveau des threads
  - Multicœur



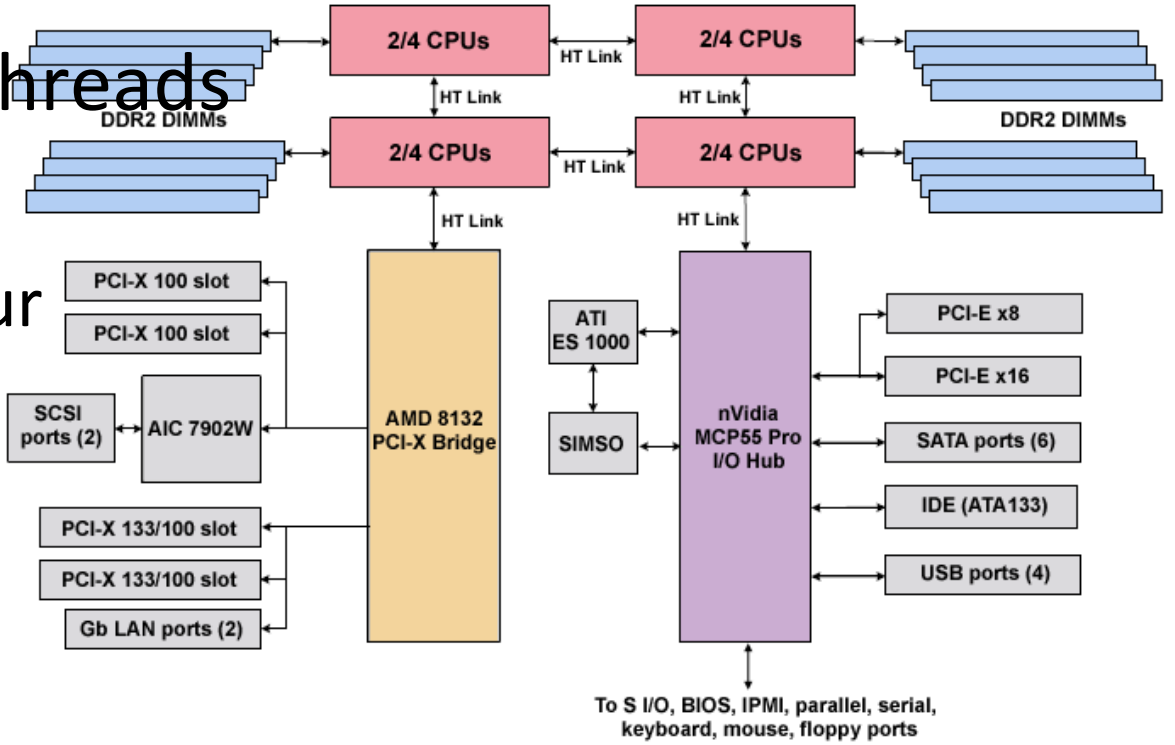
Où trouve-t-on du parallélisme

- Au niveau des circuits
- Au niveau des instructions
- Au niveau des threads
  - Multicœur
  - Multiprocesseur



Où trouve-t-on du parallélisme

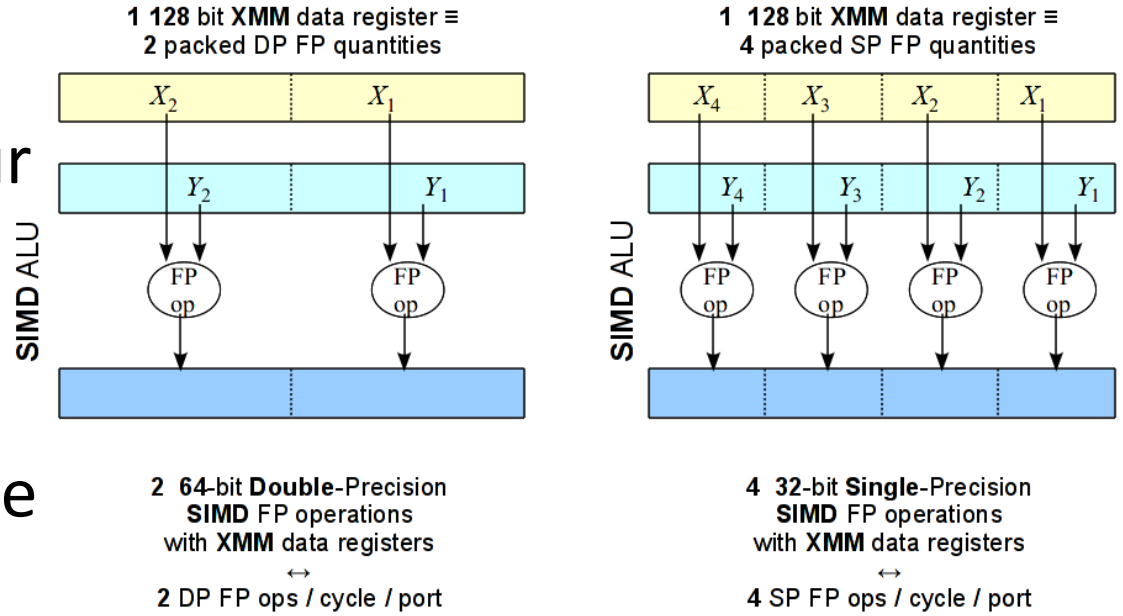
- Au niveau des circuits
- Au niveau des instructions
- Au niveau des threads
  - Multicœur
  - Multiprocesseur



SuperMicro H8QM8-2 Motherboard

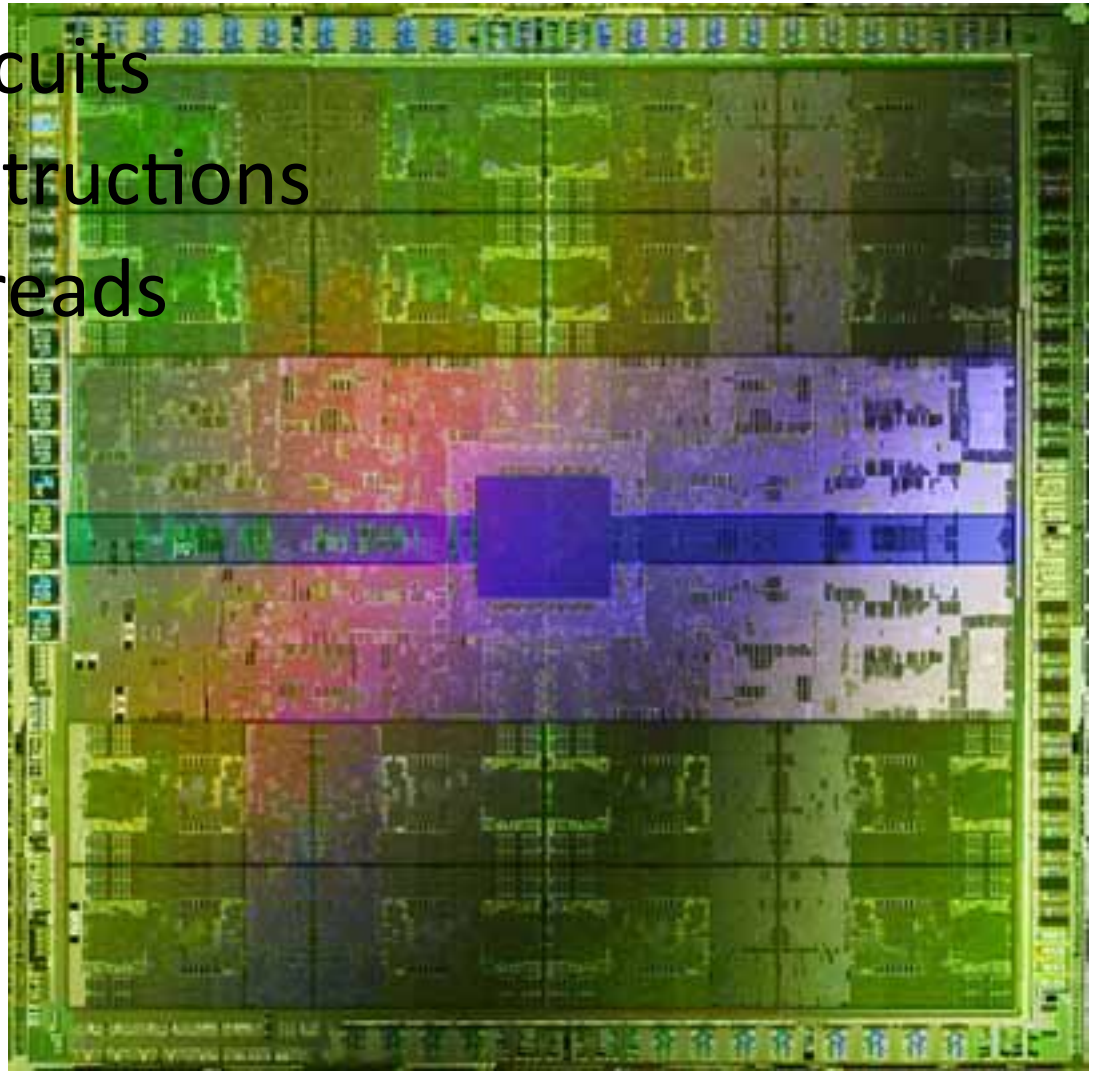
Où trouve-t-on du parallélisme

- Au niveau des circuits
- Au niveau des instructions
- Au niveau des threads
  - Multicœur
  - Multiprocesseur
- Au niveau des données
  - Unité vectorielle



Où trouve-t-on du parallélisme

- Au niveau des circuits
- Au niveau des instructions
- Au niveau des threads
  - Multicœur
  - Multiprocesseur
- Au niveau des données
  - Unité vectorielle
  - Accélérateur



Plan du Chapitre

- Instruction Level Parallelism
  - Pipeline
  - Superscalaire
  - Prédiction de branchement
  - Out-of-Order
  - Cache

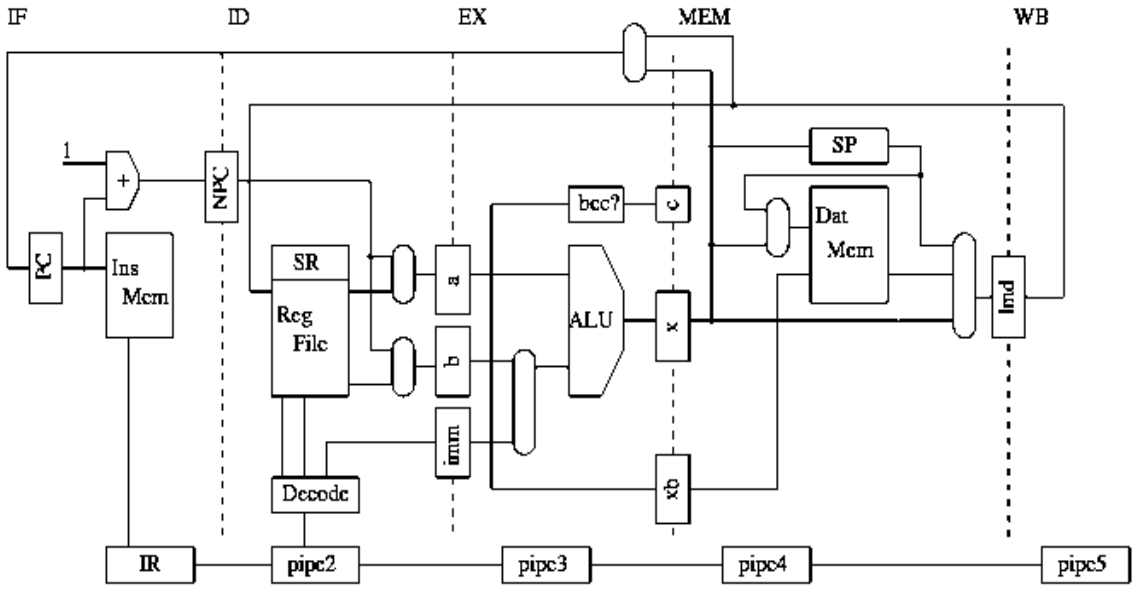
Latence

- Thread Level Parallelism
  - SMT
  - Cohérence Mémoire
  - Multi-cœur
  - organisation SMP, NUMA
- Data paralellism

Débit

Pipeline

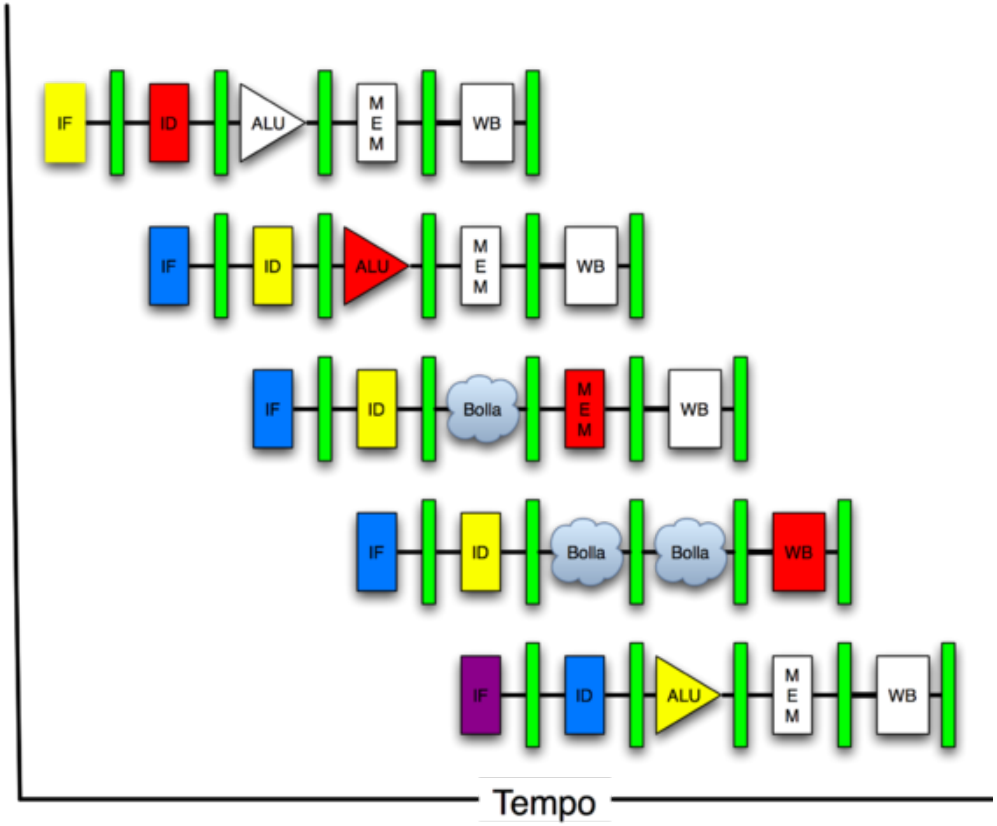
- Travail à la chaîne
  - Fetch Decode Execute Memory Write



MIPS

Pipeline

- Chaque phase est traitée par une unité fonctionnelle dédiée
- Idéalement chaque instruction progresse d'un étage à chaque top horloge
  - 5 étages => 5 instructions en parallèle



Intérêts du pipeline

- Plus d'étages => plus de parallélisme
- La durée de l'étage le plus lent détermine la fréquence maximale du pipeline
  - 333 MHz

Intérêts du pipeline

- Plus d'étages => plus de parallélisme
- La durée de l'étage le plus lent détermine la fréquence maximale du pipeline
  - 333 MHz

  - Scinder les unités fonctionnelles lentes
    - 500MHz

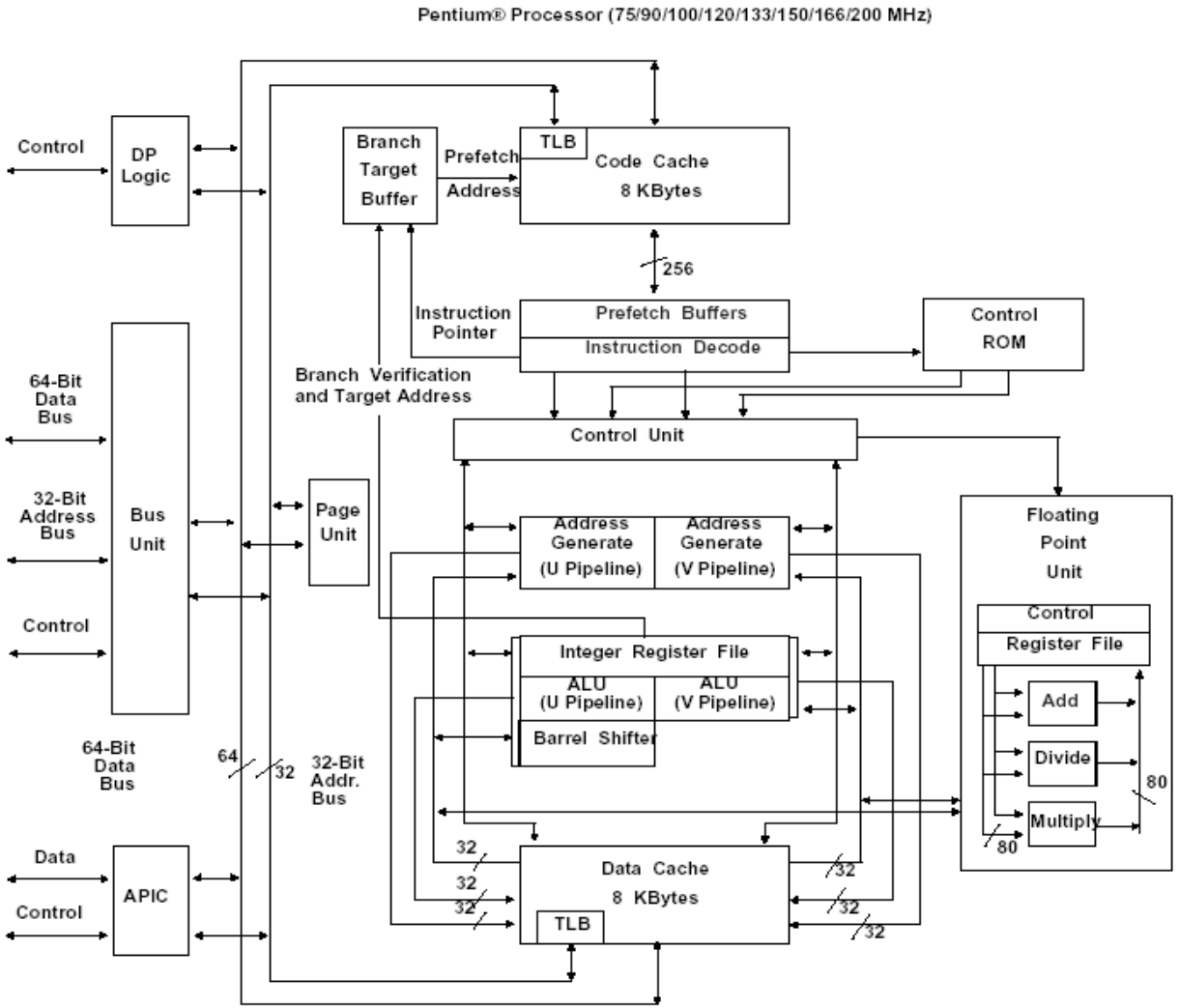
Intérêts du pipeline

- Plus d'étages => plus de parallélisme
- La durée de l'étage le plus lent détermine la fréquence maximale du pipeline
  - 333 MHz

  - Scinder les unités fonctionnelles lentes
    - 500MHz

  - Ou les dédoubler : processeur *superscalaire*
    - 500MHz

Les ALU U et V du Pentium



Pipeline
Origine des bulles

- Dépendance de données

  ADD R1, R1, 1
  MUL R2, R2, R1

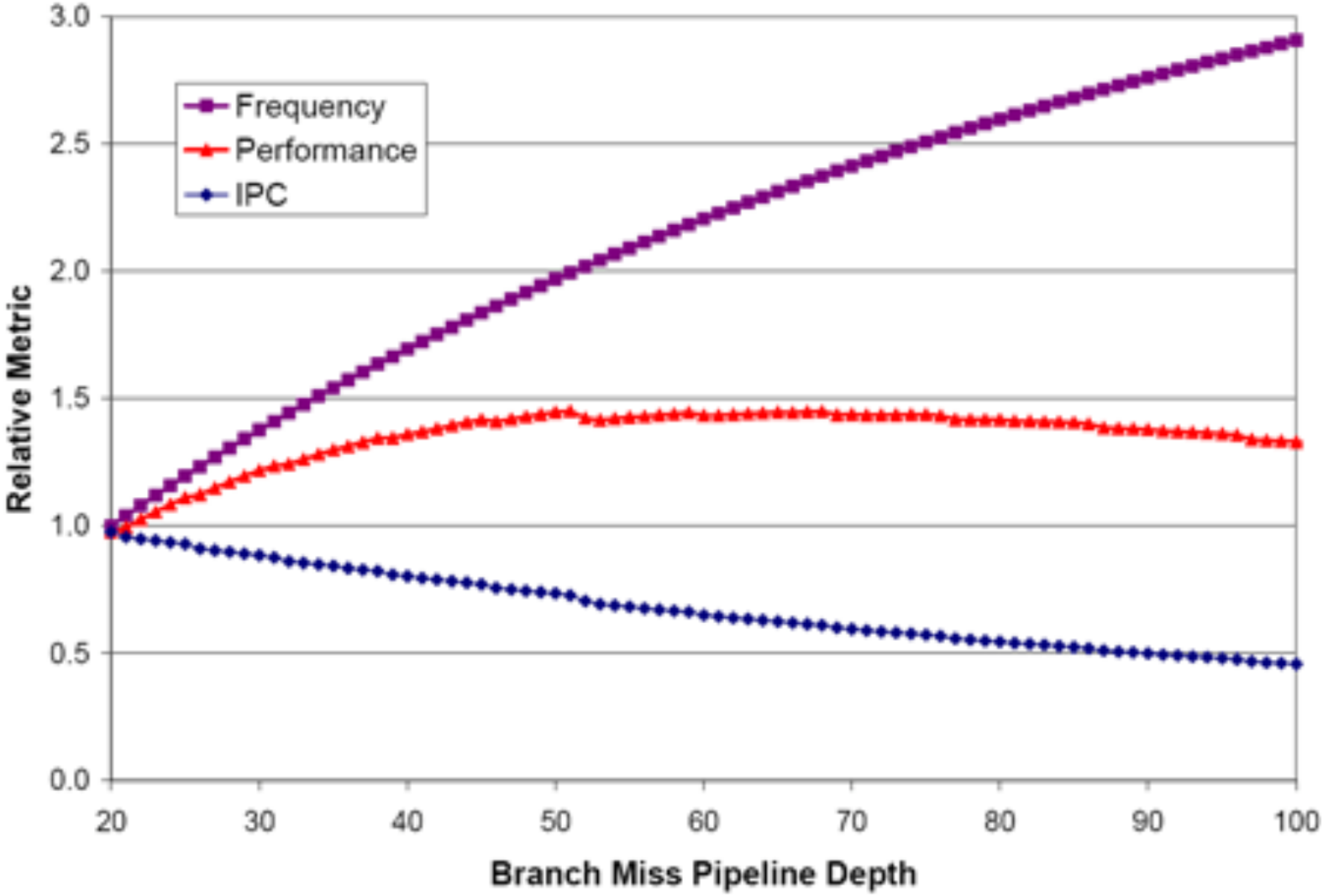
Fetch	Dec	ALU	MEM	WB
xx	MUL	ADD		
xx	MUL		ADD	
xx	MUL			ADD
yy	xx	MUL		

- Dépendance de contrôle
  - saut conditionnel et indirection (pointeur de fonction)
    - Très coûteux car bloque le pipeline, par exemple :
      - Perte de 10 cycles à chaque saut, 1 saut toutes 10 instructions

      ➔ Chute de 50% des performances

- Défaut de cache
  - Mémoire ~60ns = 240 cycles à 2,5 GHz

Influence de la profondeur du pipeline (INTEL)

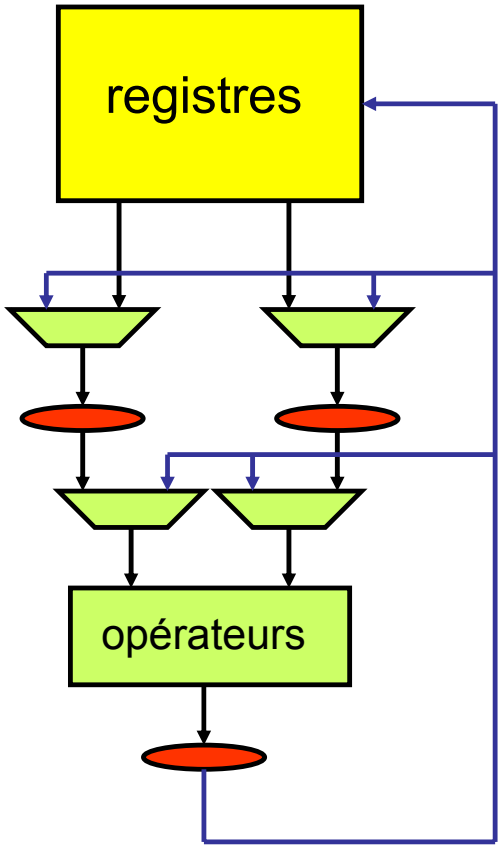


Dépendances de données

- Dépendance de données I1 ; I2
  - *Read After Write (RAW)*
    - Le résultat de I1 est utilisé par I2
    - Aussi appelée dépendance vraie
  - *Write After Read* (WAR)
    - I2 écrit dans un registre lu par I1
    - Aussi appelée anti-dépendance
  - *Write After Write* (WAW)
    - I2 écrit dans le même registre que I1
    - Aussi appelée dépendance de sortie

Dépendances de données

- Dépendance de données I1 ; I2
  - *Read After Write* (RAW)
    - Le résultat de I1 est utilisé par I2
    - Aussi appelée dépendance vraie
  - *Write After Read* (WAR)
    - I2 écrit dans un registre lu par I1
    - Aussi appelée anti-dépendance
  - *Write After Write* (WAW)
    - I2 écrit dans le même registre que I1
    - Aussi appelée dépendance de sortie



Dépendances de données

- Dépendance de données I1 ; I2
  - *Read After Write* (RAW)
    - Le résultat de I1 est utilisé par I2
    - Aussi appelée dépendance vraie
  - *Write After Read* (WAR)
    - I2 écrit dans un registre lu par I1
    - Aussi appelée anti-dépendance
    - **Problématique pour superscalaire**
  - *Write After Write* (WAW)
    - I2 écrit dans le même registre que I1
    - Aussi appelée dépendance de sortie
    - **Problématique pour superscalaire**

Compilation
Exécution Out-of-Order

Compilation
Renommage de registres

Optimisation du pipeline OoO

- Éviter les bulles en modifiant l'ordre d'exécution des instructions
  - statiquement par le compilateur
  - dynamiquement au sein du processeur
- Barrières mémoire empêchent le réordonnancement

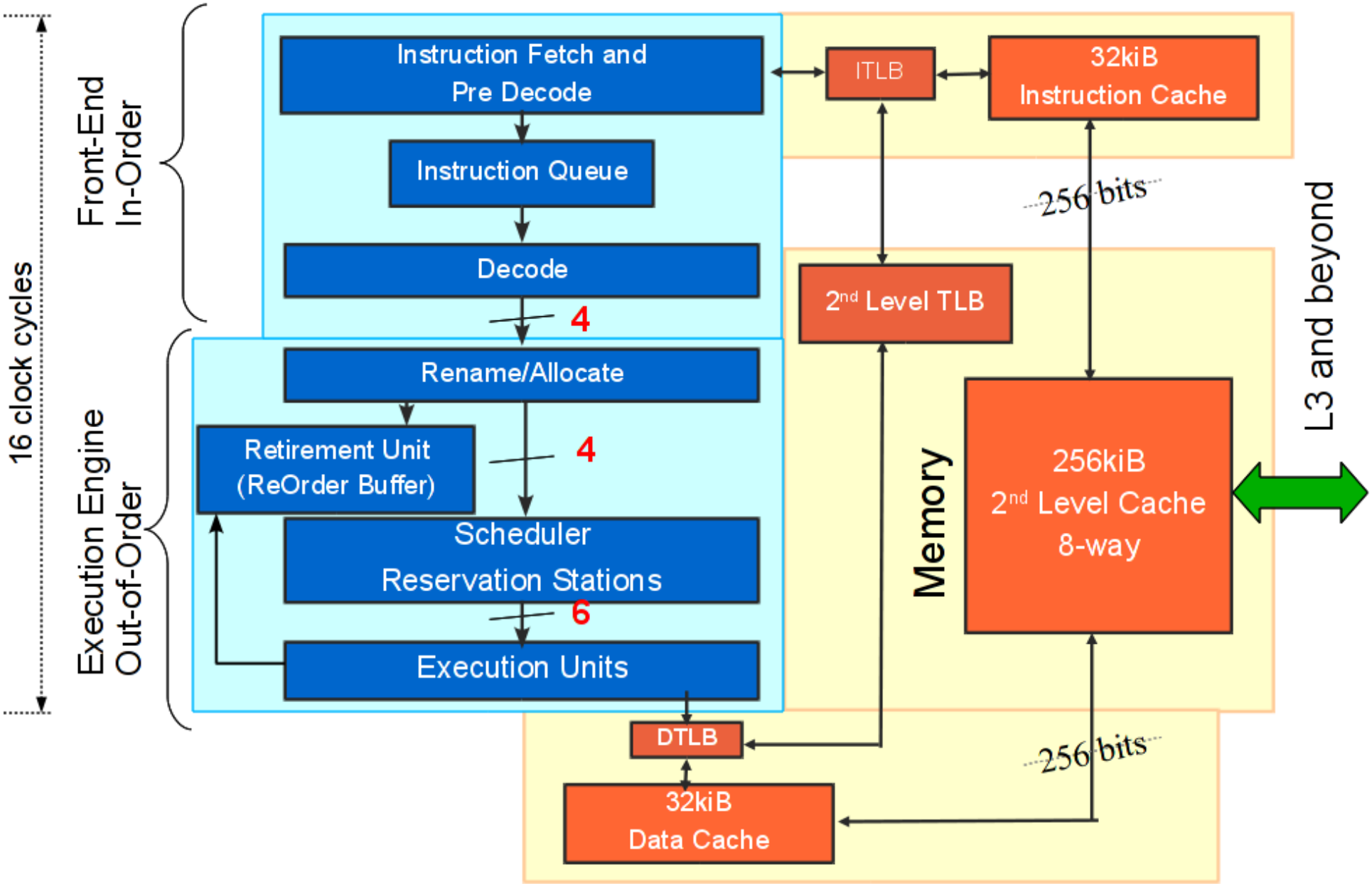
```
#define memory_barrier() __asm__ __volatile__("mfence":::"memory")
#define memory_read() __asm__ __volatile__("lfence":::"memory")
#define memory_write() __asm__ __volatile__("sfence":::"memory")
```

Dépendances de données
Renommage

- Pour éviter les pseudo-dépendances
  - À la compilation
    - Static Single Assignation (SSA)
  - Dynamiquement par le processeur
    - # registres physiques > # registres logiques

f = f * x; **x = x + 1 ;**	**allouer f1** **f1 = f0 * x0** **libérer f0**	**allouer x1** **x1 = x0 + 1** **libérer x0**

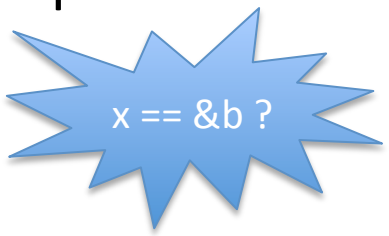
Nehalem Core Pipeline



Dépendances de données
Aliasing

- Utilisation de pointeurs ou indirection

```
*x = a;
*y = b;
```




```
void somme (int *a, int *b, int*c)
{
for (int i=0; i < 10; i++)
        a[i] = b[i] + c[i]
 }
```

- Solutions

C99

```
void somme (restricted int *a,
                restricted int *b,
                restricted int *c);
```

- Dépendance potentielle
  - Déterminée à la volée
    - Introduction de bulles
    - Évaluation spéculative

Optimisation du pipeline
Évaluation spéculative

- Pari sur le paramètre de l'instruction
- Exécution de l'instruction (des instructions)
- Vérification
  - Obtention de la valeur d'un test
  - Rejoue de l'instruction
- Annulation des instructions invalides
  - Retarder les écritures pour les annuler
- Applications
  - Aliasing
  - Branchement conditionnel
  - Indirection (Retour de fonction, pointeur de fonction, switch)

Optimisations des branchements

Éviter les branchements ;-)

- Éviter les tests
  - Impair = (x % 2 == 1) ? 1 : 0; → impair = x & 1
- Fusionner les boucles
  - For (i= x; i < y; i++) A();

    For (i= x; i < y; i++) B(); → For (i= x; i < y; i++) {A();B();}
- Dérouler les boucles
  - For (i= 0; i < 3; i++) A(); → A(); A(); A();
  - gcc -O3 -funroll-loops
  - Augmente la taille du code

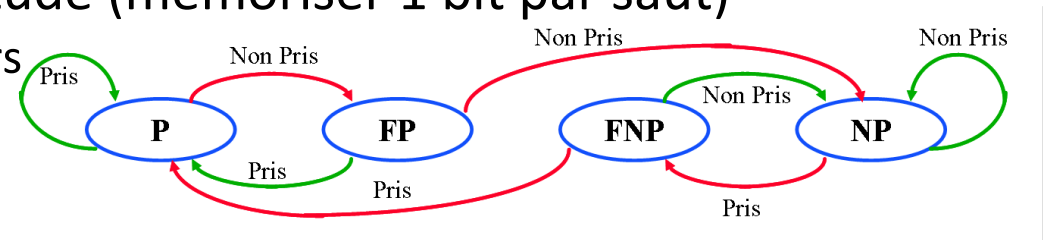
Optimisations des branchements

Prédire le branchement

- Optimiser l'apport de l'évaluation spéculative

```
for (i= 0; i < 100; i++) // 101 tests
  for(j=0; j< 10; j++)  // 1100 tests
    …
```

- Stratégies de prédiction simples :
  - stratégie toujours
    - on se trompe à chaque fin de boucle : 1 + 100 erreurs
  - stratégie jamais
    - complément de toujours 99 + 1000
  - stratégie comme la dernière fois (mémoriser 1 bit par saut)
    - 1 + 200
  - stratégie comme d'habitude (mémoriser 1 bit par saut)
    - équivalente ici à toujours



Optimisations des branchements

Prédire le branchement

- Exemple (par C. Michaud)

```
int MIN = x[0];
for (i=1; i<1000; i++)
        If (x[i] < MIN)
                MIN = x[i];
```

- Branchement de boucle:
  - non pris à la dernière itération seulement
- Branchement de test:
  - pris en moyenne moins de 6,5 fois
    - somme des inverses de 1 à 1000
    - En fait c'est moins

Optimisations des branchements

Prédire le branchement

- Exemple (par C. Michaud)

```
int MIN = x[0];
for (i=1; i<1000; i++)
        If (x[i] < MIN)
                MIN = x[i];
```

- Branchement de boucle:
  - non pris à la dernière itération seulement
- Branchement de test:
  - pris en moyenne moins de 6,5 fois
    - somme des inverses de 1 à 1000
    - En fait c'est moins

- stratégie toujours
  - 1000 erreurs
- stratégie jamais
  - 993,5 erreurs
- Stratégie comme la dernière fois
  - 14 = 1 + 2 x 6,5
- Stratégie comme d'habitude
  - 7,5 = 1 + 6,5

Prédire le branchement
Branch Target Buffer

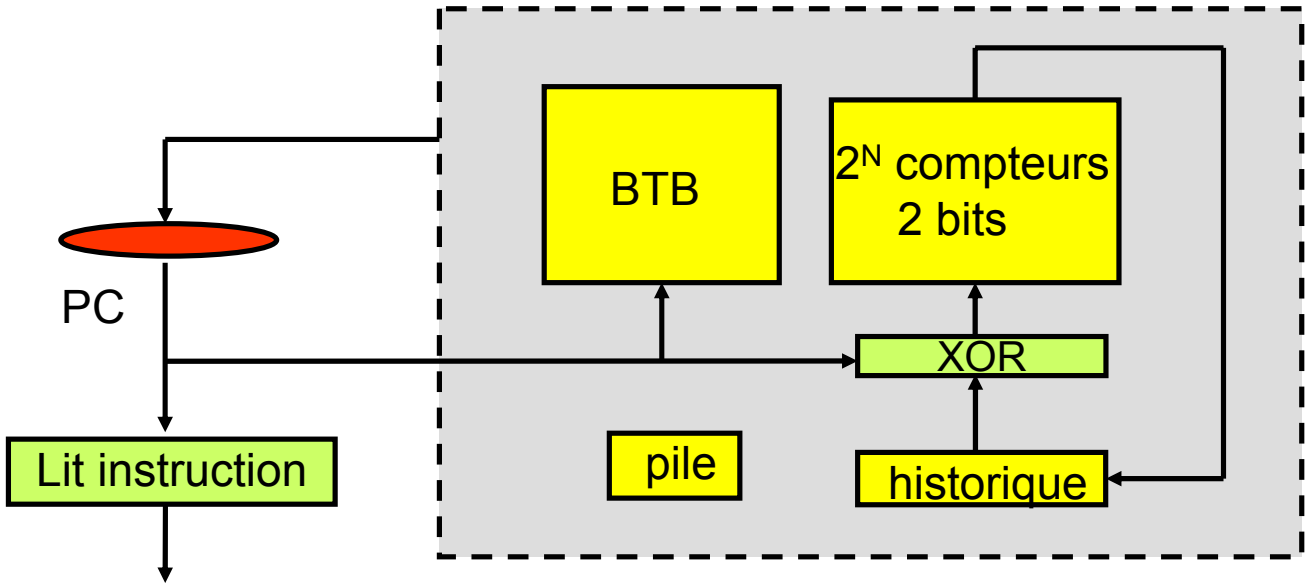
- Comme un cache, sauf qu'on y stocke des informations sur les instructions de contrôle
  - Tag = bits de poids fort de l'adresse de l'instruction de contrôle
  - Mémorise état de l'automate, la destination du saut, le type de contrôle

- Petite pile pour prévoir les adresses de retour
  - Lorsque le BTB détecte un CALL, empiler l'adresse de retour
  - Lorsqu'on exécute un retour, dépiler l'adresse et sauter spéculativement

- Permet de prévoir 90% des sauts

Branch Target Buffer
+ global history

- addresse du saut + historique global (+ local au saut ou compteur) → prédicteur 2 bits
- Historique = direction des N derniers branchements conditionnels dans un registre à décalage (non pris → 0, pris → 1)
- Table de hachage : Accès compteur 2 bits via PC xor historique
- 96% de réussite

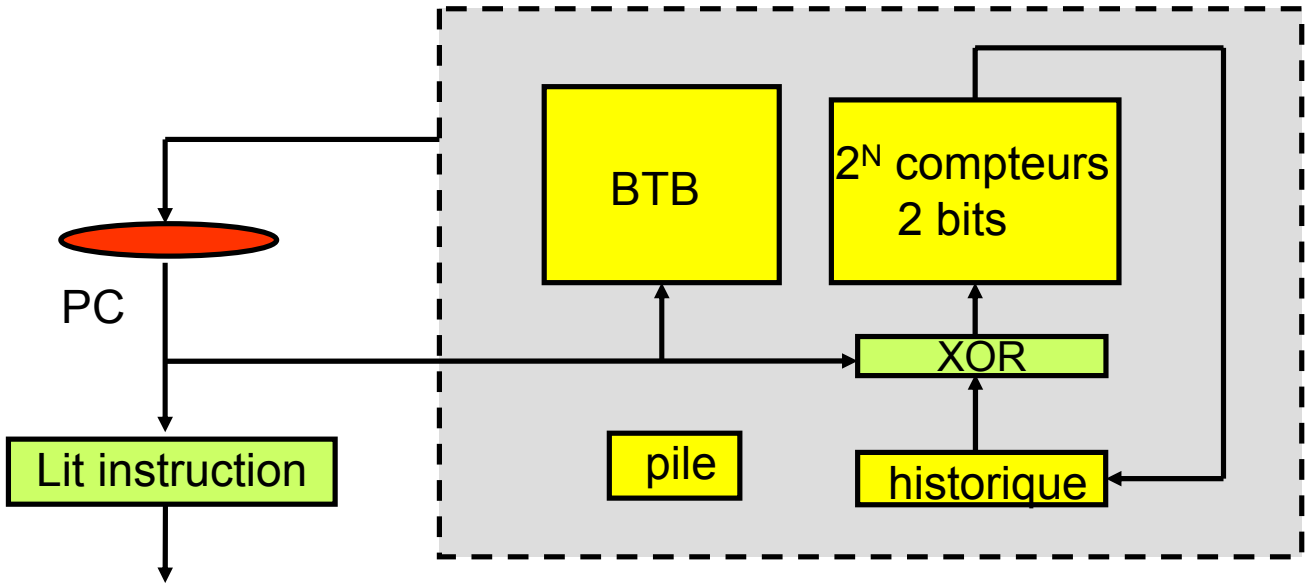


Branch Target Buffer
+ global history

```
for (i=0; i<10; i++)
  for (j=0; J<10; j++)
    x = x + a[i][j];
```

Prévisible à partir de

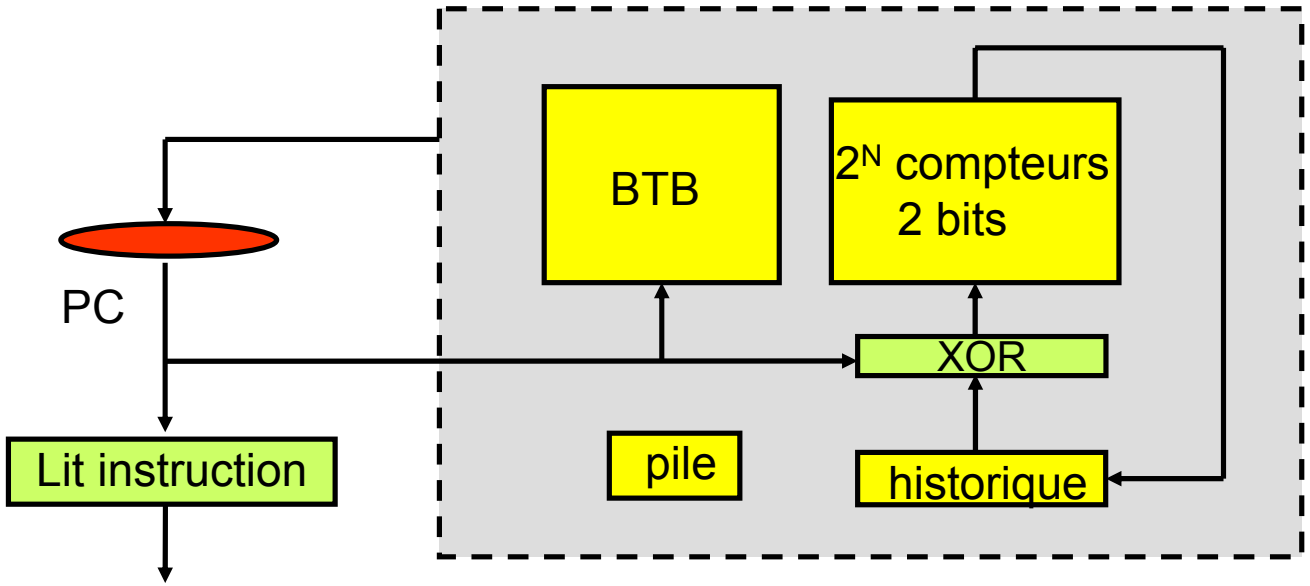
- 11 bits d'historique
- 2048 compteurs 2 bits



Branch Target Buffer
+ global history

Une dizaine de sauts pistés sur core 2

→ Il faut faire la chasse aux sauts conditionnels



Cache

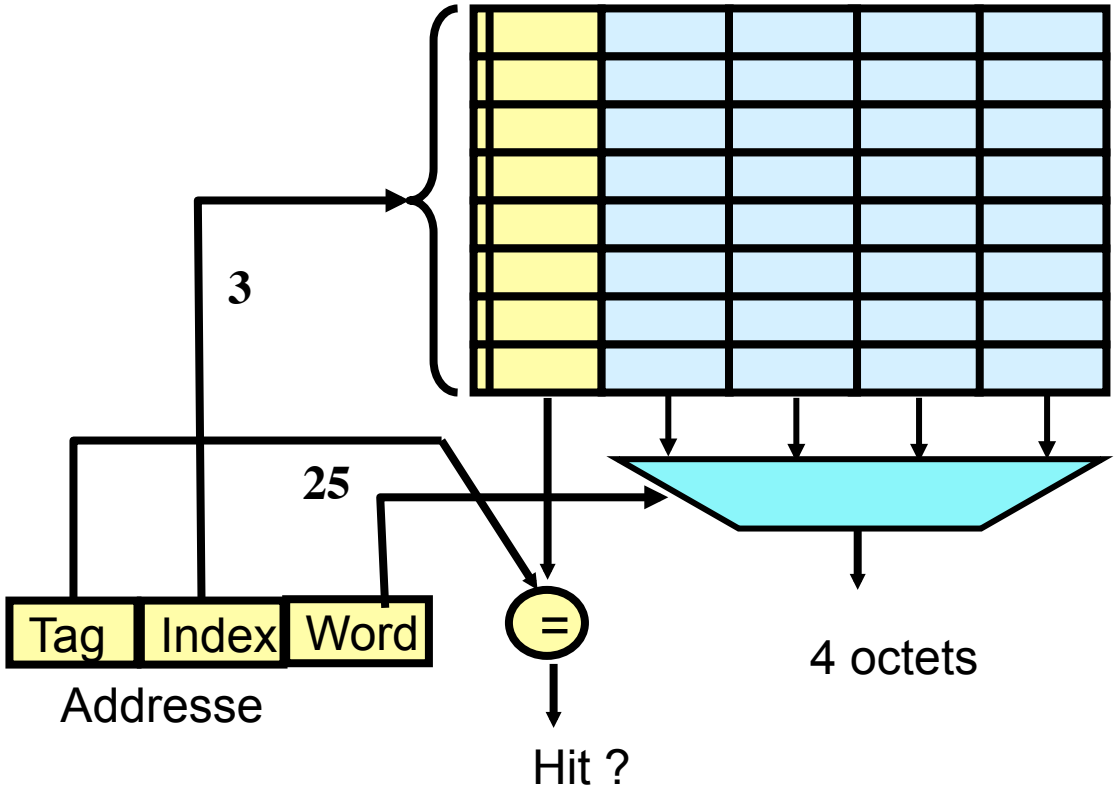
- Observations
  - Plus une mémoire est petite plus son temps de réponse peut être faible
  - Localité spatio-temporelle des programmes « *90% de l'exécution se déroule dans 10% du code »*
- Contient une copie d'une faible partie de la mémoire
  - Cache hit / Cache miss
  - Nécessite l'éviction de données, la synchronisation avec la mémoire
  - Il faut éviter les aller-retour Mémoire/Cache (ping-pong)

Cache
Performances mesurées

	Speed (GHz)	L1	L2	L3	Mem (ns)
• **Intel Xeon X5670**	2.93GHz	4	10	56	87
• **Intel Xeon X5570**	2.80GHz	4	9	47	81
• **AMD Opteron 6174**	2.20GHz	3	16	57	98
• **AMD Opteron 2435**	2.60GHz	3	16	56	113

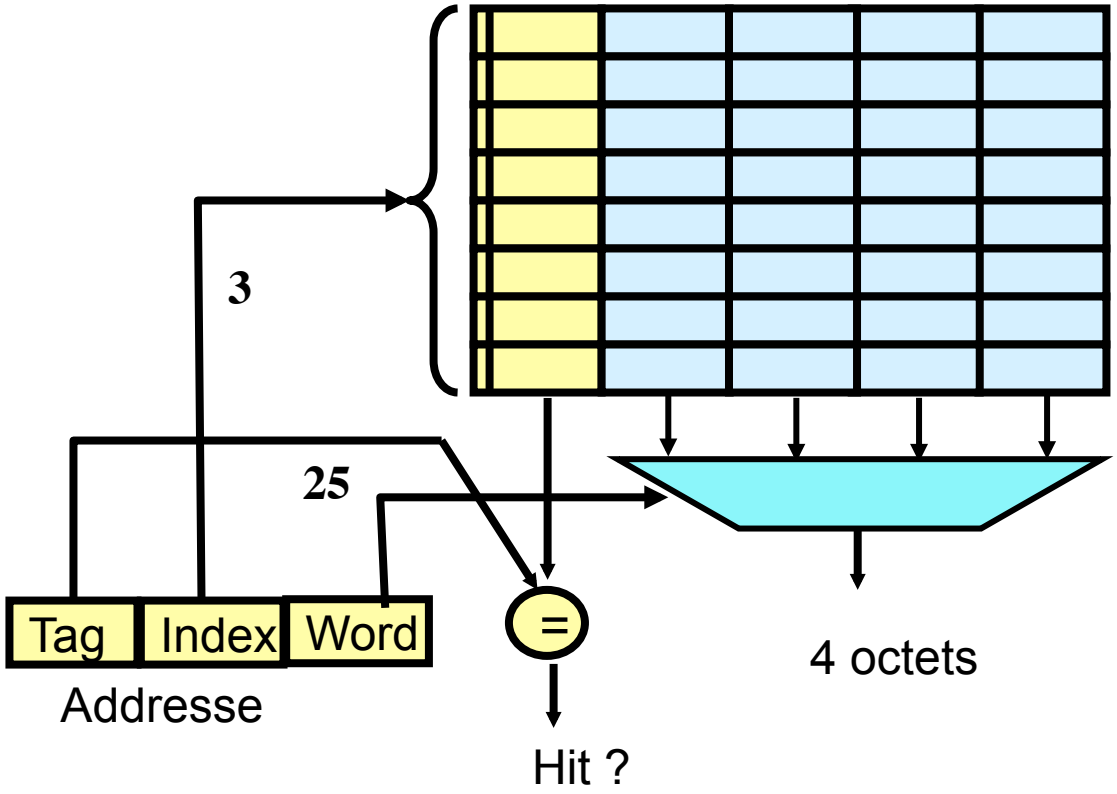
Cache direct

- Ex. 8 entrées de 4x4 octets, adresse de 32 bits



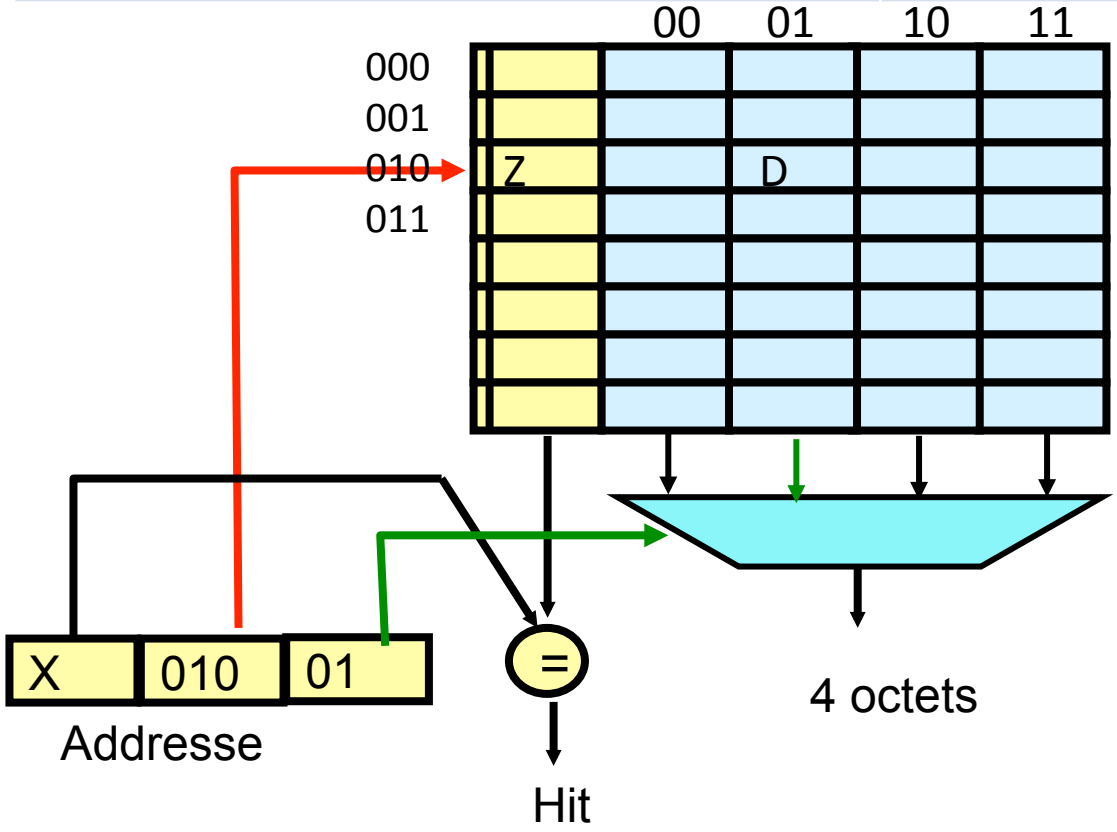
Cache direct

Tag = étiquette à mémoriser / comparer 25 bits	Index dans le cache 3 bits	Choix du Mot 2 bits
$X=x_{24}...x_1$	010	01

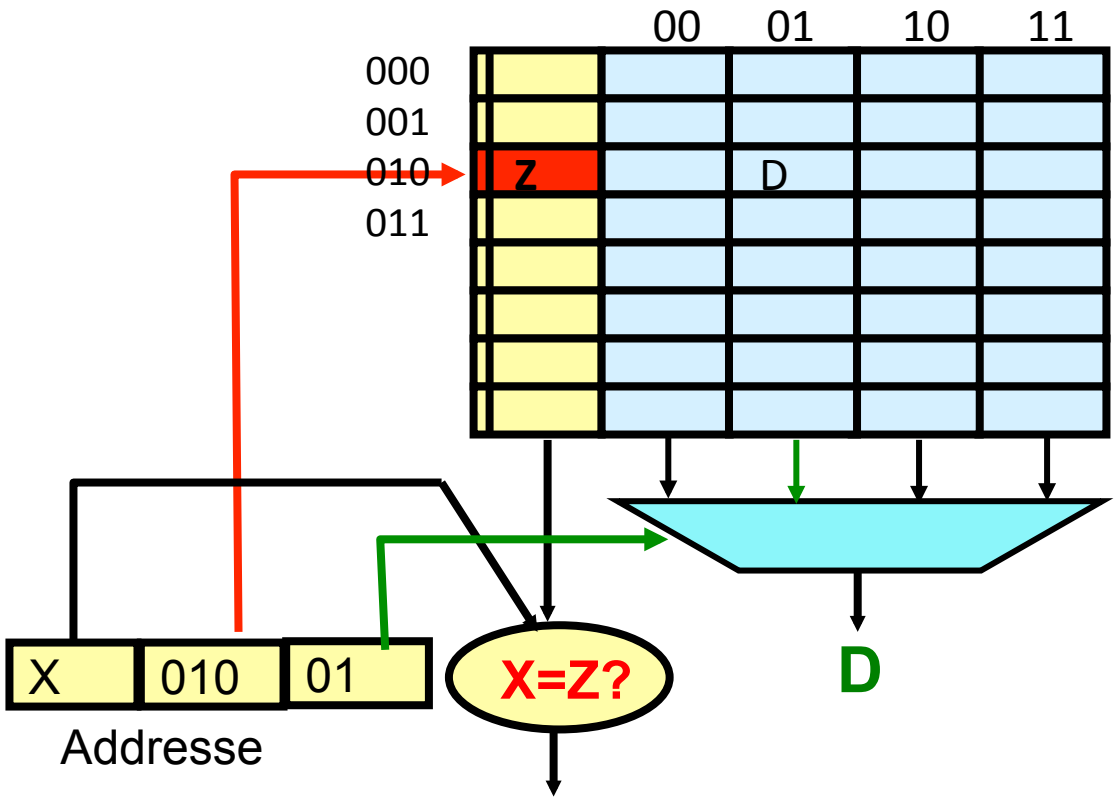


Cache direct

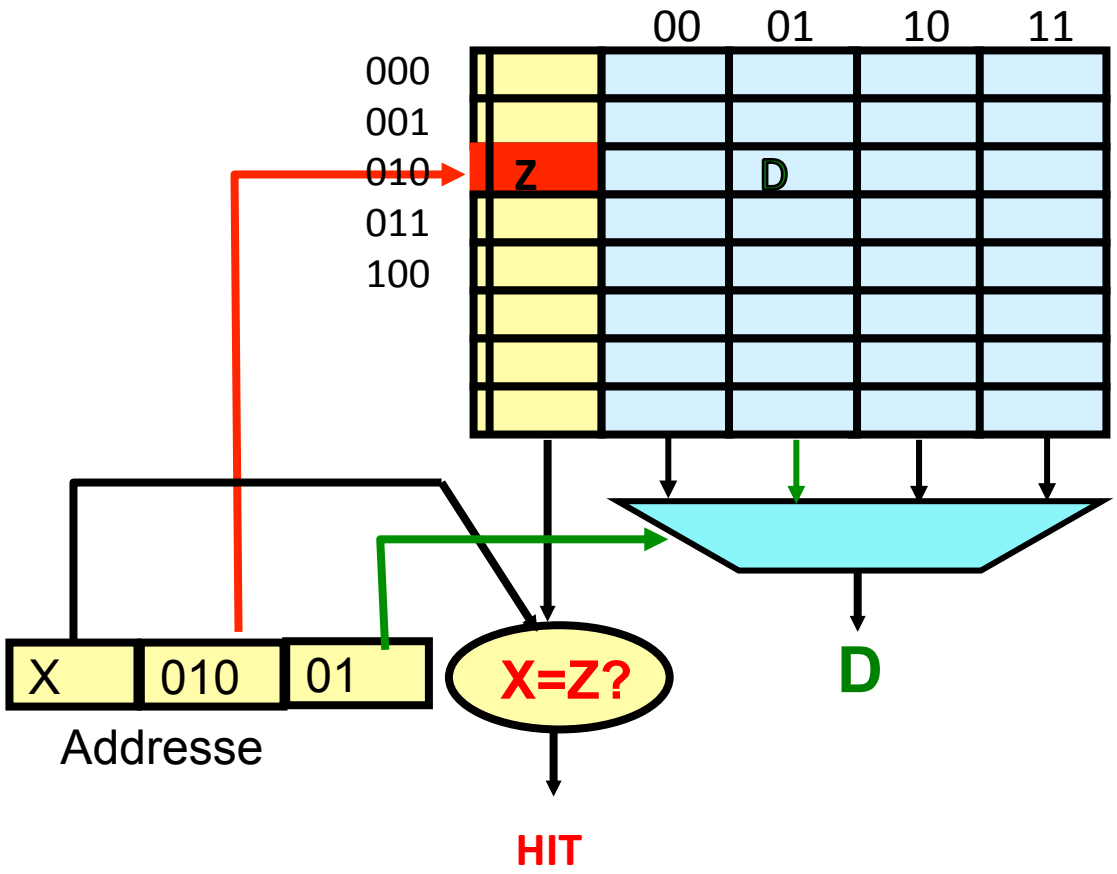
Tag = étiquette a mémoriser / comparer 25 bits	Index dans le cache 3 bits	Choix du Mot 2 bits
$X=x_{24}...x_1$	010	01



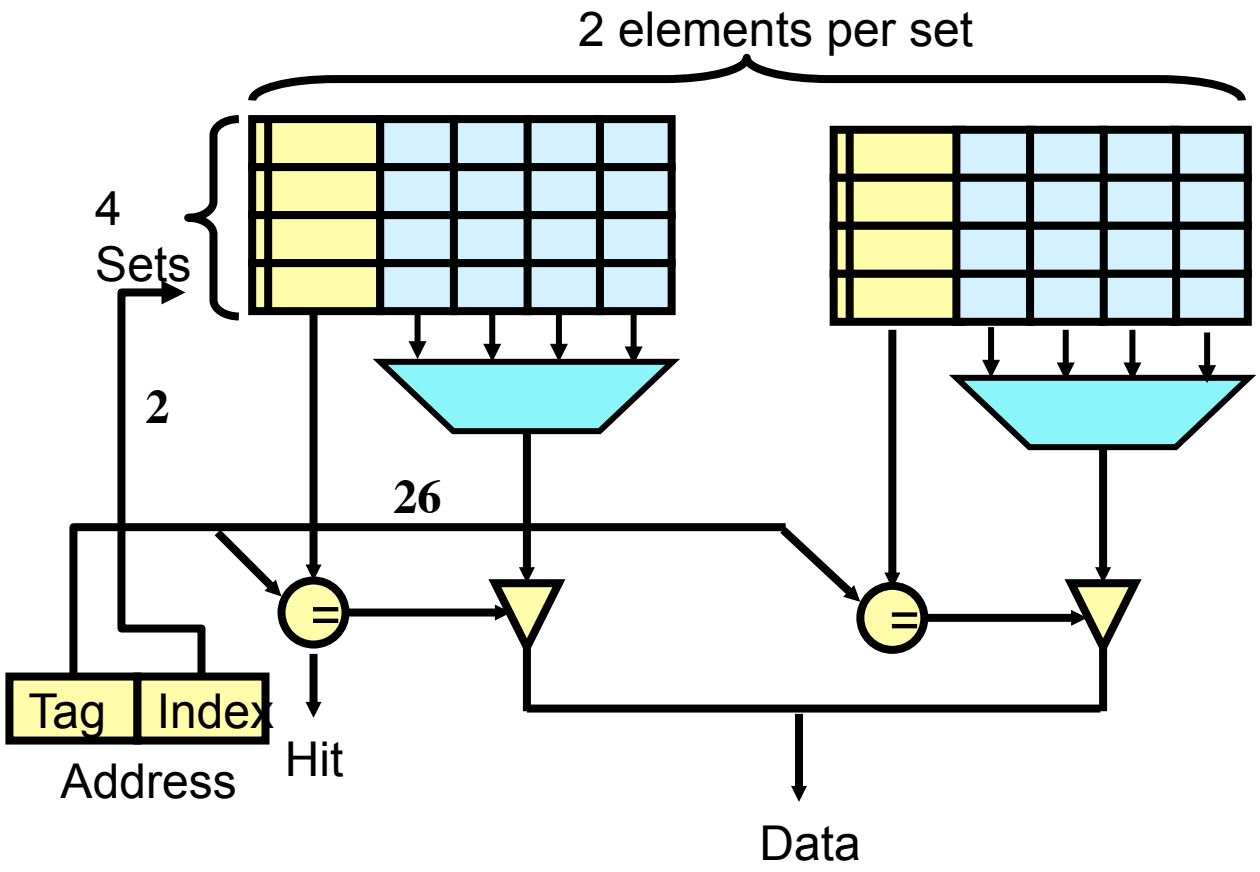
Cache direct



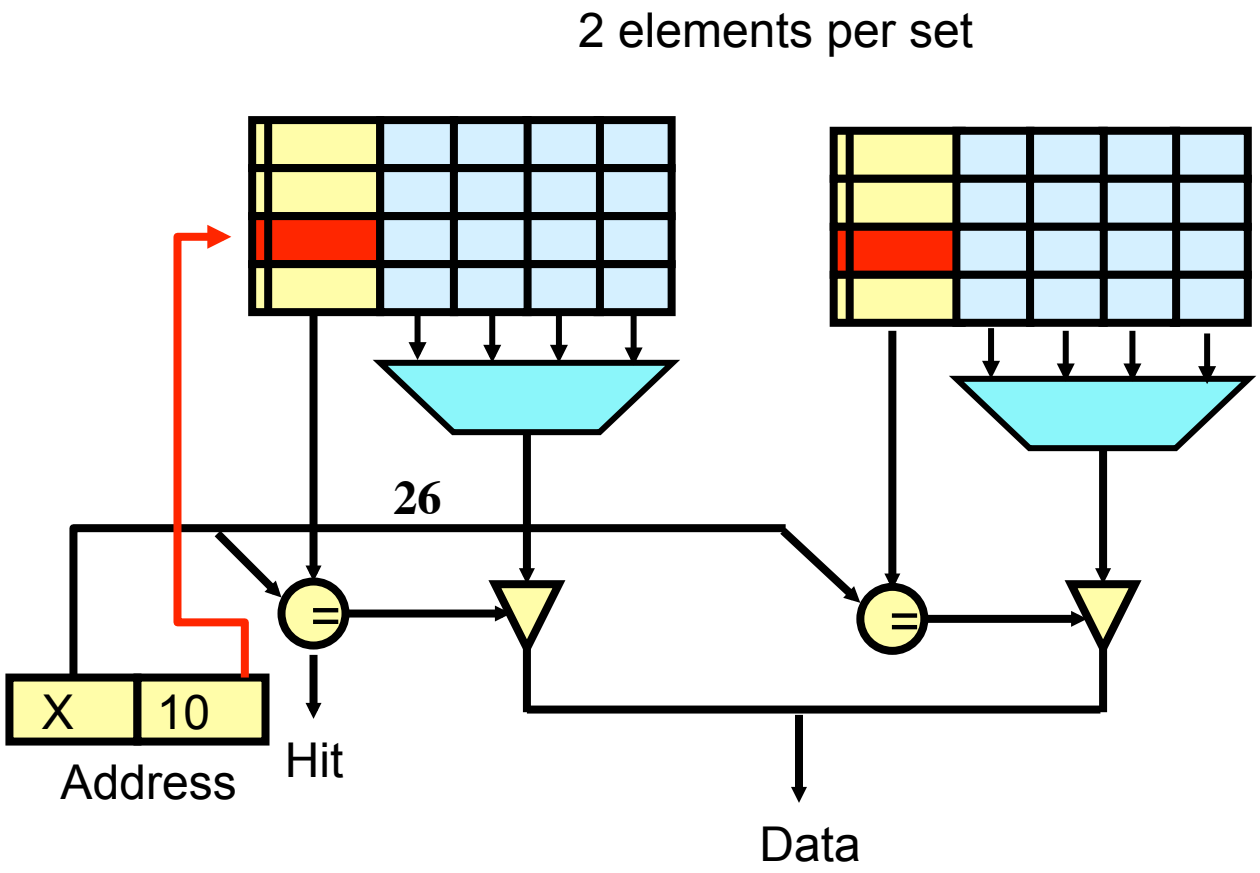
Cache direct



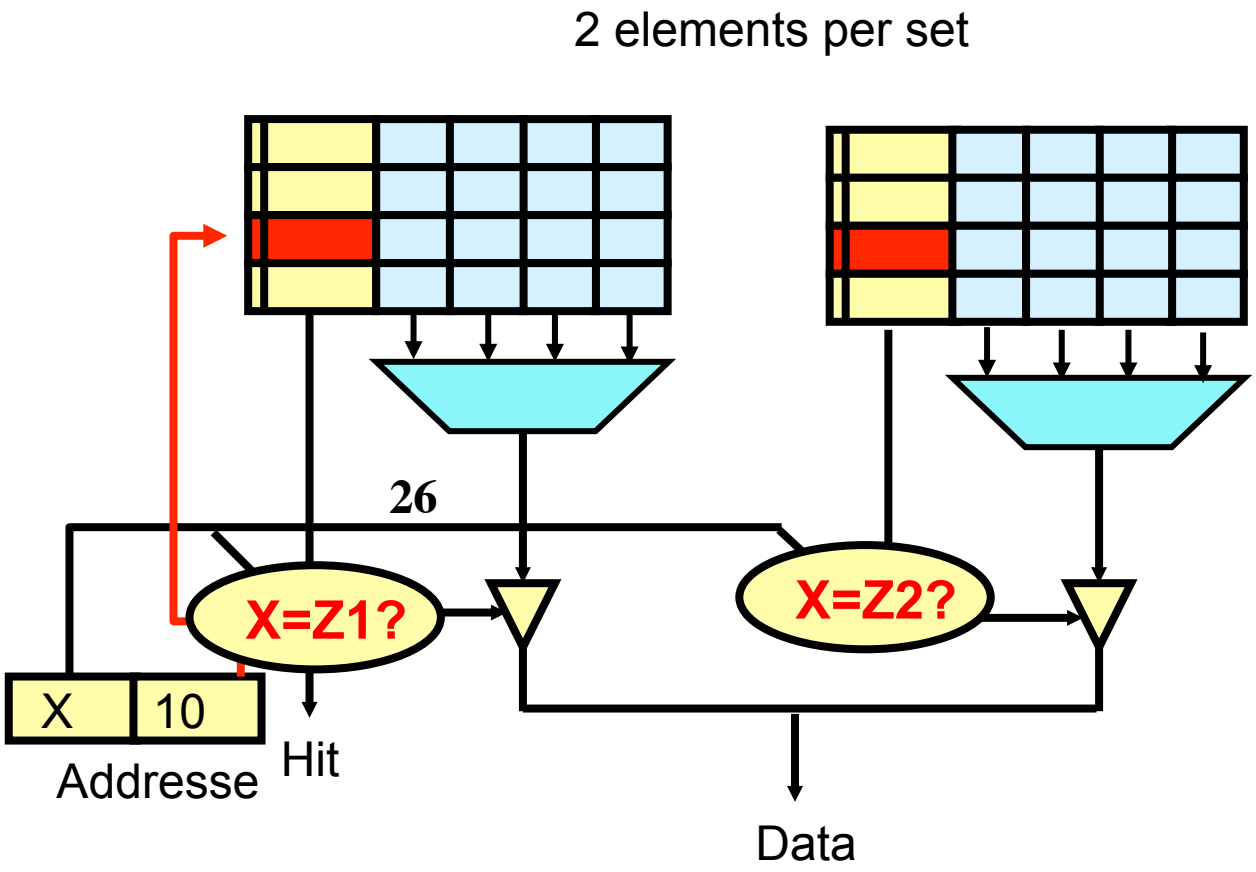
Cache 2 associatif



Cache 2 associatif

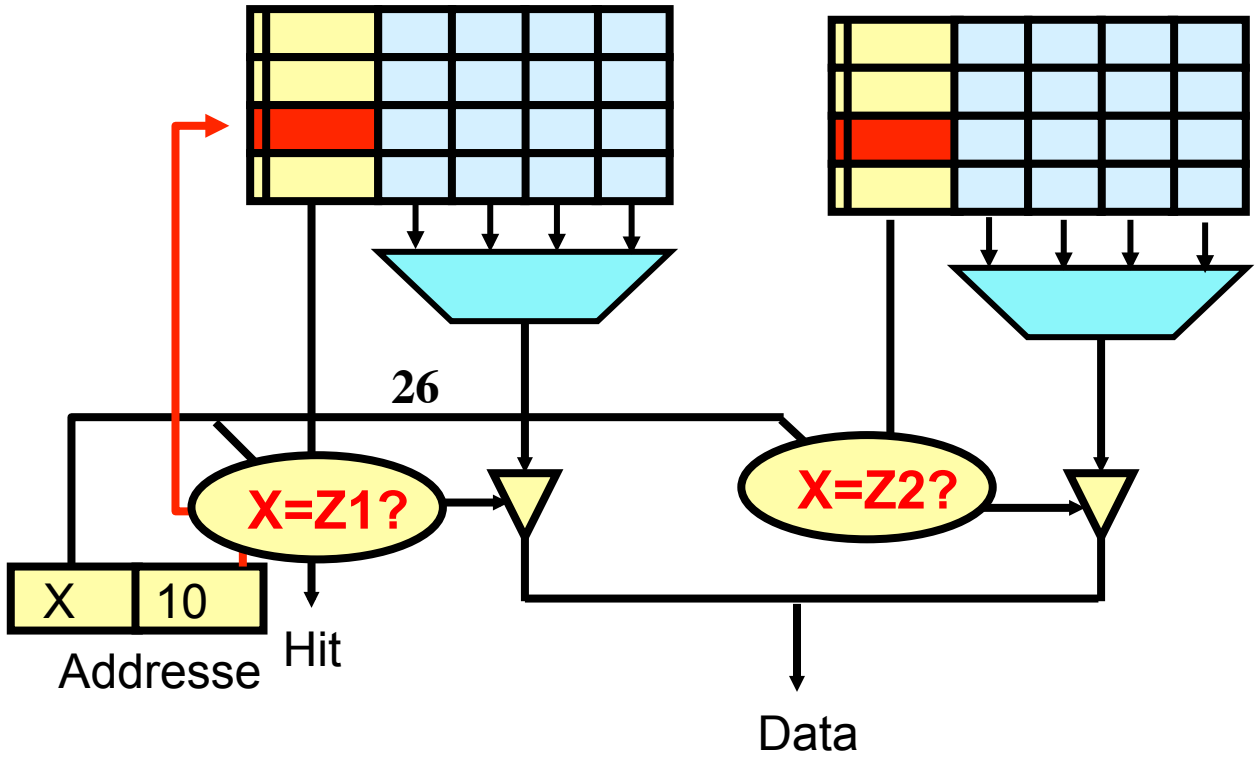


Cache 2 associatif

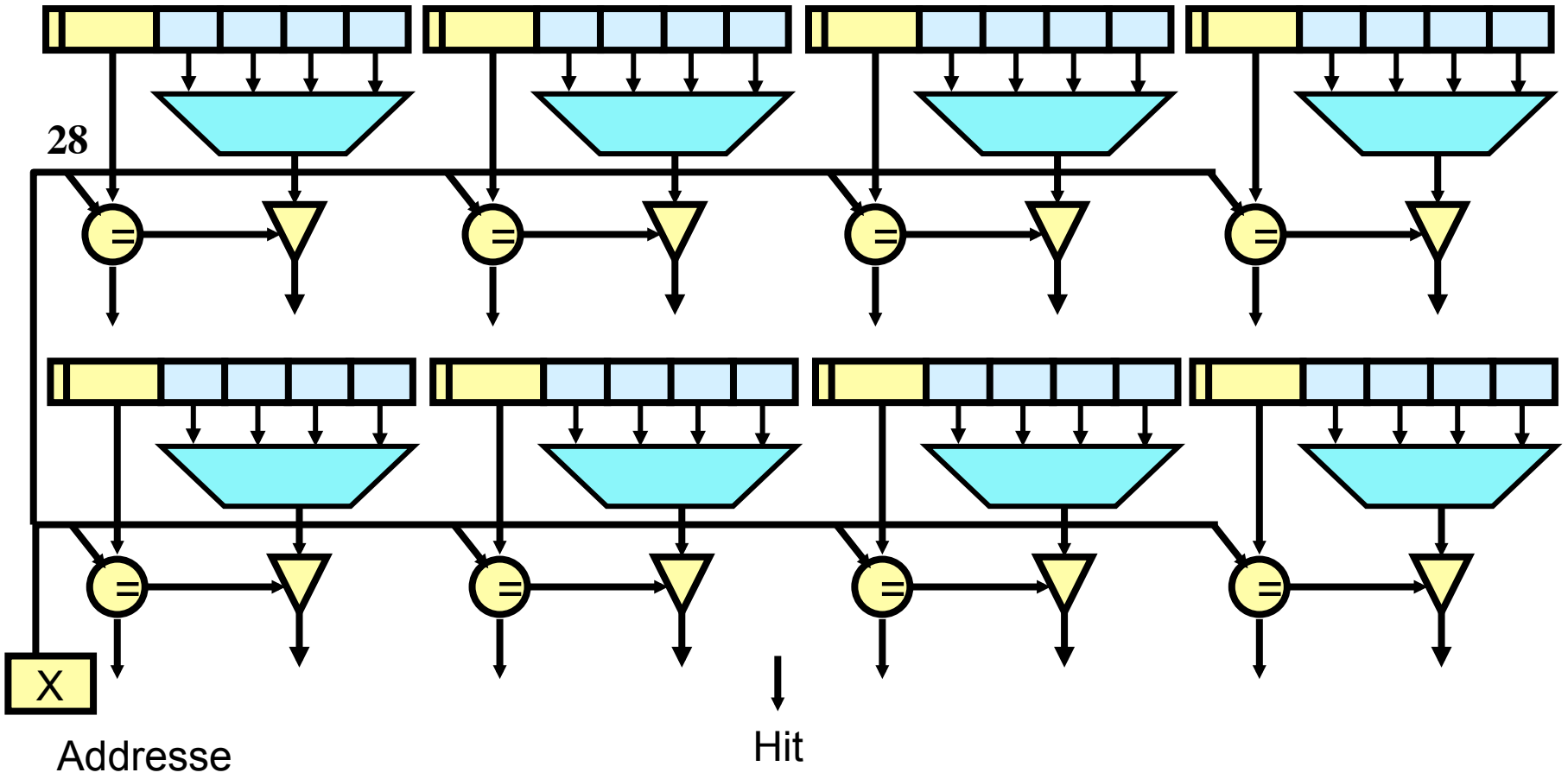


Cache 2 associatif

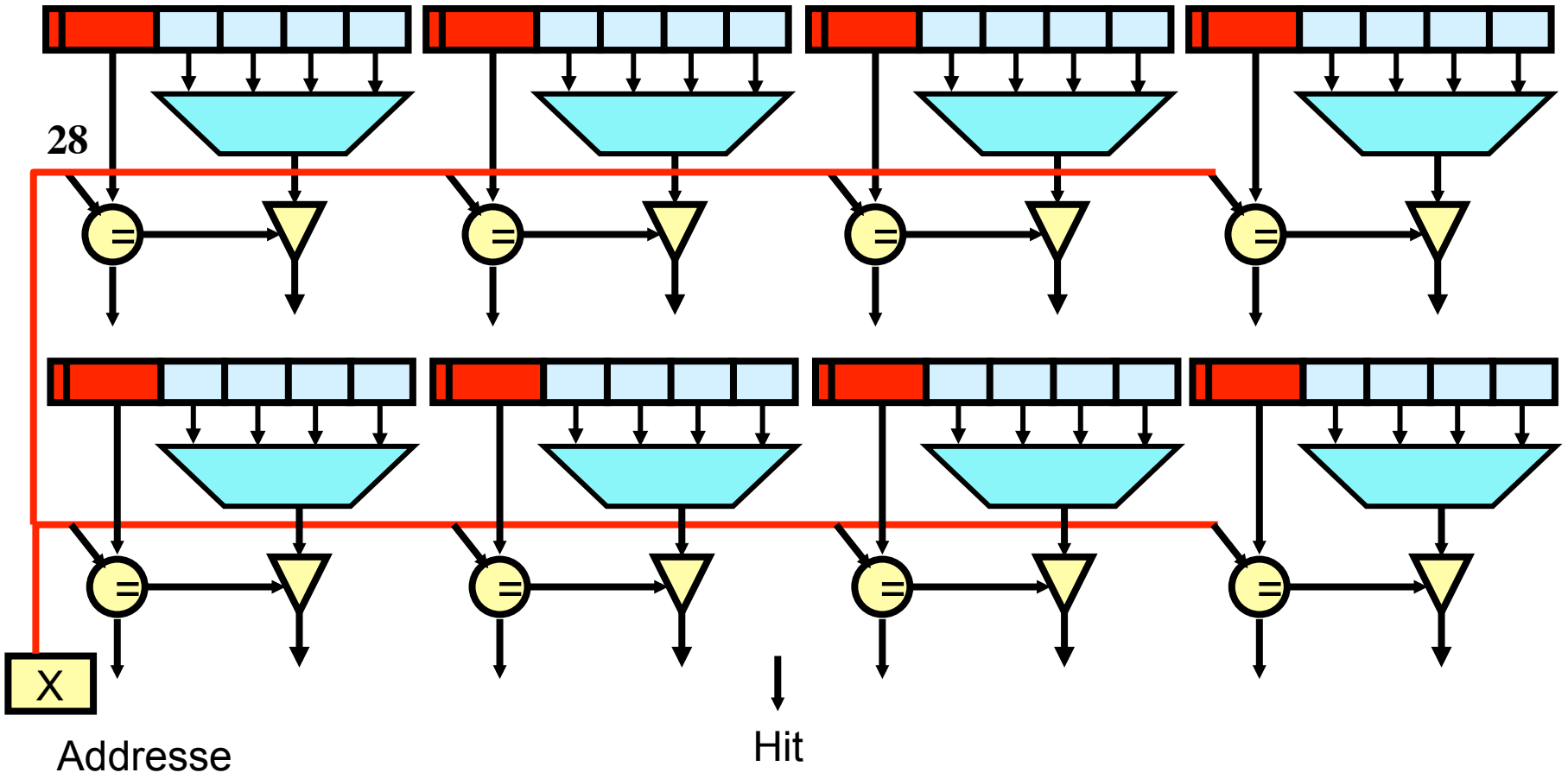
- Algo de type LRU pour l'éviction



Cache associatif



Cache associatif

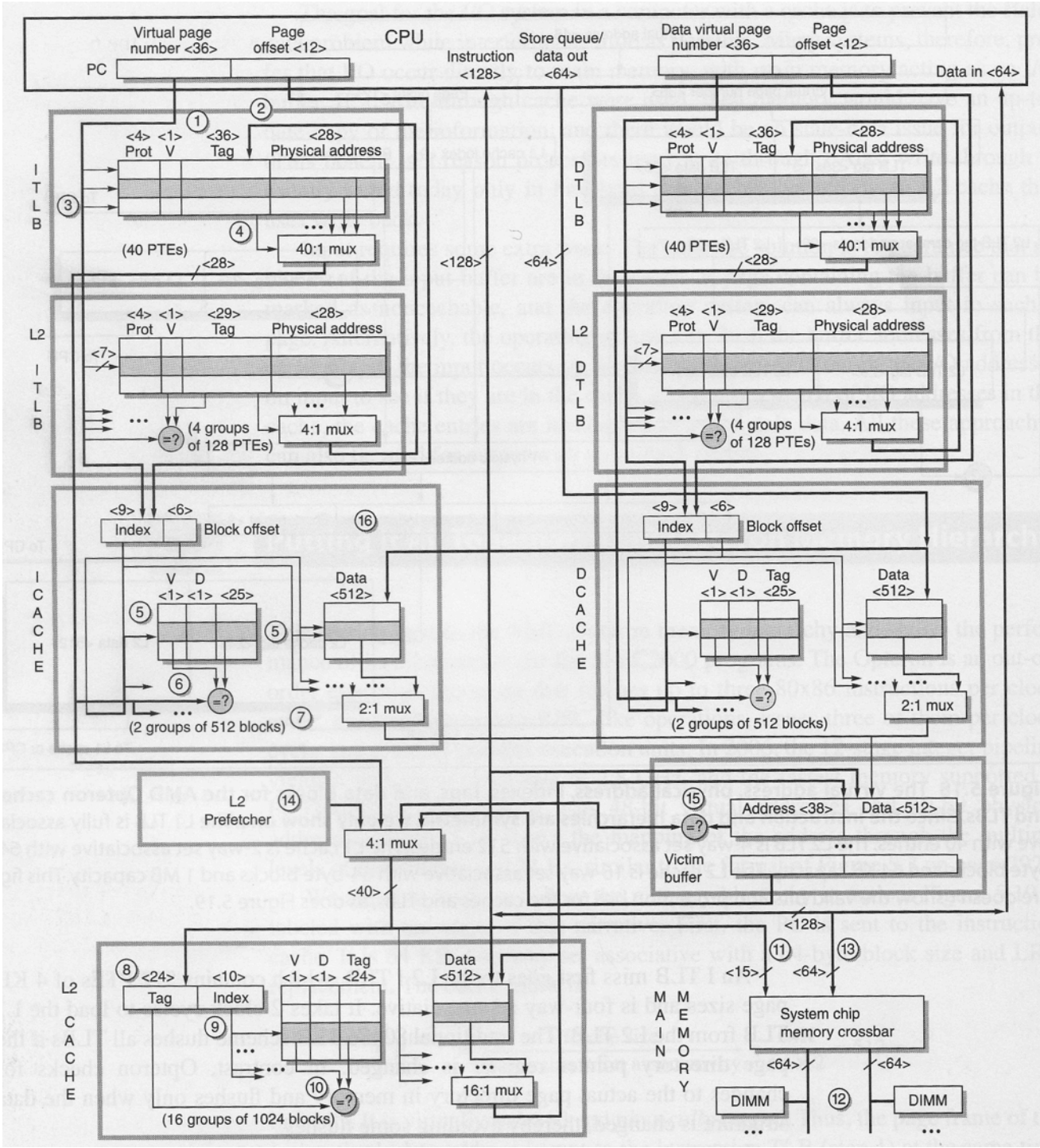


Cache

- Unité d'information stockée = 1 ligne de cache
  - 64 octets en général

- Exemple d'utilisation des caches
  - Caches associatifs :TLB
  - Cache k-associatif : L1, L2, L3, Branch Target Buffer

Un cache 2-associatif de taille T est aussi performant qu'un cache direct de taille 2T


CPU
PC
Virtual page number <36>
Page offset <12>
Instruction <128>
Store queue/ data out <64>
Data virtual page number <36>
Page offset <12>
Data in <64>
ITLB
DTLB
<4> Prot
<1> V
<36> Tag
<28> Physical address
(40 PTEs)
40:1 mux
L2 ITLB
L2 DTLB
<29> Tag
<7>
(4 groups of 128 PTEs)
4:1 mux
ICACHE
DCACHE
<9> Index
<6> Block offset
V <1>
D <1>
Tag <25>
Data <512>
2:1 mux
(2 groups of 512 blocks)
L2 Prefetcher
Address <38>
Victim buffer
<40>
<128>
<15>
<64>
L2 CACHE
<24> Tag
<10> Index
Tag <24>
16:1 mux
(16 groups of 1024 blocks)
MAIN MEMORY
System chip memory crossbar
DIMM

Opteron

Cache
Occupation du bus

- *techniques d'écritures lors d'un cache-miss*
  - write-allocate --> la ligne est chargée dans le cache avant d'être modifiée *: il faut lire avant d'écrire !!!*
  - write-no allocate --> on modifie directement la donnée en mémoire

- *2 techniques d'écritures si ligne présente*
  - write-through --> écriture simultanée on écrit toujours à la fois dans le cache et la mémoire (buffer)
  - write-back --> on marque la ligne de cache comme modifiée et on la sauve en mémoire que sur obligation (au moment du remplacement)

→ couple write-back+ write-allocate minimise l'utilisation du bus

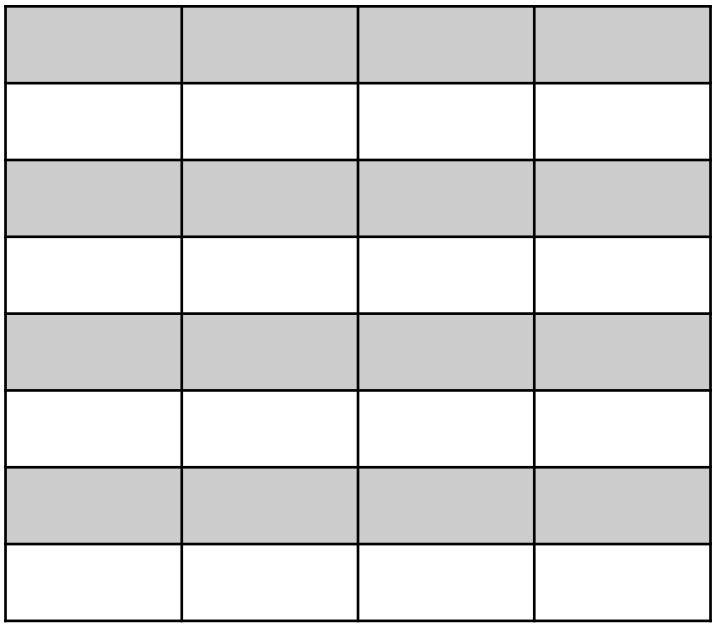
→ write-no allocate utile pour memcpy

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        C[i][j] = A[i][j] + B[i][j];
```



Cache direct 4 mots / lignes

J = 0

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        C[i][j] = A[i][j] + B[i][j];
```

0,0	0,1	0,2	0,3

J = 0

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        C[i][j] = A[i][j] + B[i][j];
```

0,0	0,1	0,2	0,3
0,4	0,5	0,6	0,7

J = 4

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        C[i][j] = A[i][j] + B[i][j];
```

0,0	0,1	0,2	0,3
0,4	0,5	0,6	0,7
0,8	0,9	0,10	0,11

J = 8

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        C[i][j] = A[i][j] + B[i][j];
```

0,0	0,1	0,2	0,3
0,4	0,5	0,6	0,7
0,8	0,9	0,10	0,11
0,12	0,13	0,14	0,15

J = 12

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

 for (i=0;i<N;i++)
      for (j=0;j<N;j++)
         C[i][j] = A[i][j] + B[i][j];
```

Tellement régulier que le cache peut anticiper les demandes

0,0	0,1	0,2	0,3
0,4	0,5	0,6	0,7
0,8	0,9	0,10	0,11
0,12	0,13	0,14	0,15

J = 12

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        C[i][j] = A[i][j] + B[i][j];
```

0,0	0,1	0,2	0,3
0,4	0,5	0,6	0,7
0,8	0,9	0,10	0,11
0,12	0,13	0,14	0,15
0,16	0,17	0,18	0,19

J = 12

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        C[i][j] = A[i][j] + B[i][j];
```

0,0	0,1	0,2	0,3
0,4	0,5	0,6	0,7
0,8	0,9	0,10	0,11
0,12	0,13	0,14	0,15
0,16	0,17	0,18	0,19
0,20	0,21	0,22	0,23

J = 16

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        C[i][j] = A[i][j] + B[i][j];
```

0,24	0,25	0,26	0,27
0,0	0,1	0,2	0,3
0,4	0,5	0,6	0,7
0,8	0,9	0,10	0,11
0,12	0,13	0,14	0,15
0,16	0,17	0,18	0,19
0,20	0,21	0,22	0,23

J = 20

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        C[i][j] = A[i][j] + B[i][j];
```

0,24	0,25	0,26	0,27
0,28	0,29	0,30	0,31
0,0	0,1	0,2	0,3
0,4	0,5	0,6	0,7
0,8	0,9	0,10	0,11
0,12	0,13	0,14	0,15
0,16	0,17	0,18	0,19
0,20	0,21	0,22	0,23

J = 24

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        C[i][j] = A[i][j] + B[i][j];
```

0,24	0,25	0,26	0,27
0,28	0,29	0,30	0,31
0,30	0,31	0,32	0,33
0,4	0,5	0,6	0,7
0,8	0,9	0,10	0,11
0,12	0,13	0,14	0,15
0,16	0,17	0,18	0,19
0,20	0,21	0,22	0,23

Le nombre d'accès à la mémoire est optimal.

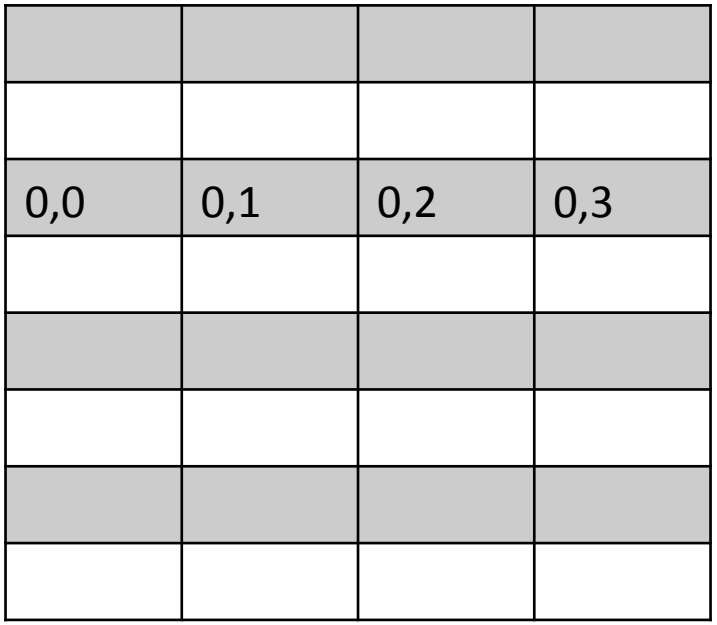
J = 28

Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (j=0;j<N;j++)
    for (i=0;i<N;i++)
        C[i][j] = A[i][j] + B[i][j];
```

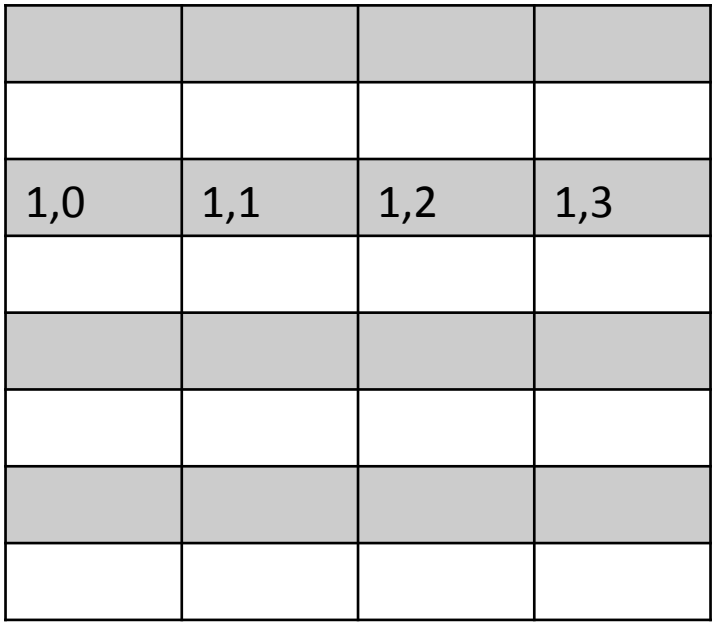


Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (j=0;j<N;j++)
    for (i=0;i<N;i++)
        C[i][j] = A[i][j] + B[i][j];
```

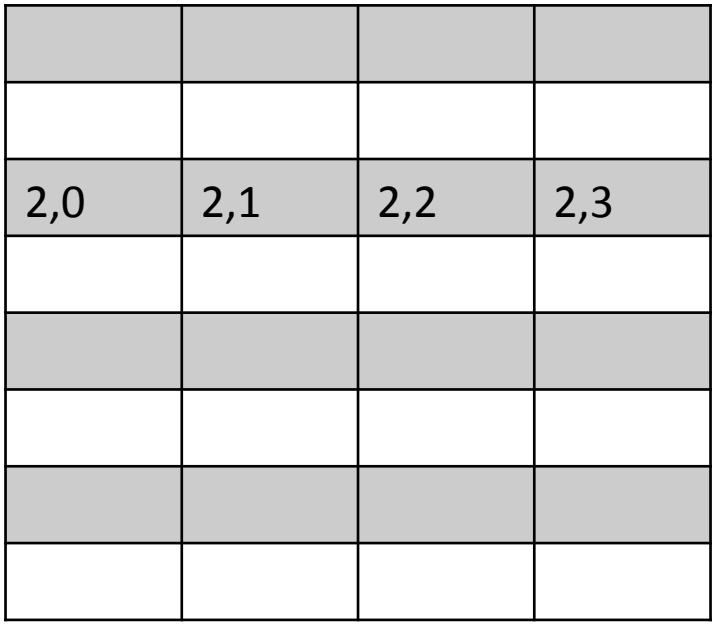


Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

for (j=0;j<N;j++)
    for (i=0;i<N;i++)
        C[i][j] = A[i][j] + B[i][j];
```



Cache
Optimisation des programmes

- Parcours séquentiel d'un tableau

```
#define N 8192
typedef long matrix[N][N];

 for (j=0;j<N;j++)
     for (i=0;i<N;i++)
        C[i][j] = A[i][j] + B[i][j];
```

Contre productif :

- 1 défaut de cache par lecture
- Et rapidement 1 défaut de TLB par lecture

2,0	2,1	2,2	2,3

Application
jeu de la vie

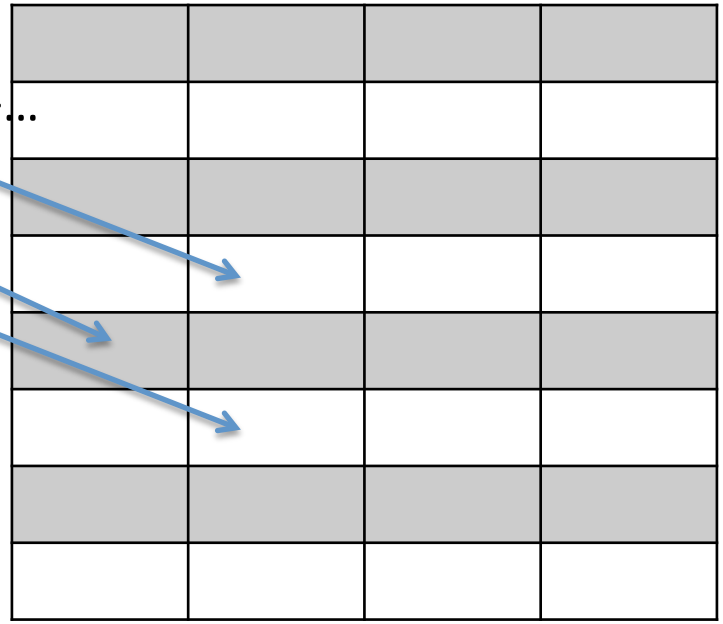
- Stockage et parcours minimisant le nombre de défaut de cache
- typedef long matrix[N][N];

- voisin[i][j] = A[i][j-1] + A[i-1][j] + A[i+1][j] …

Application
jeu de la vie

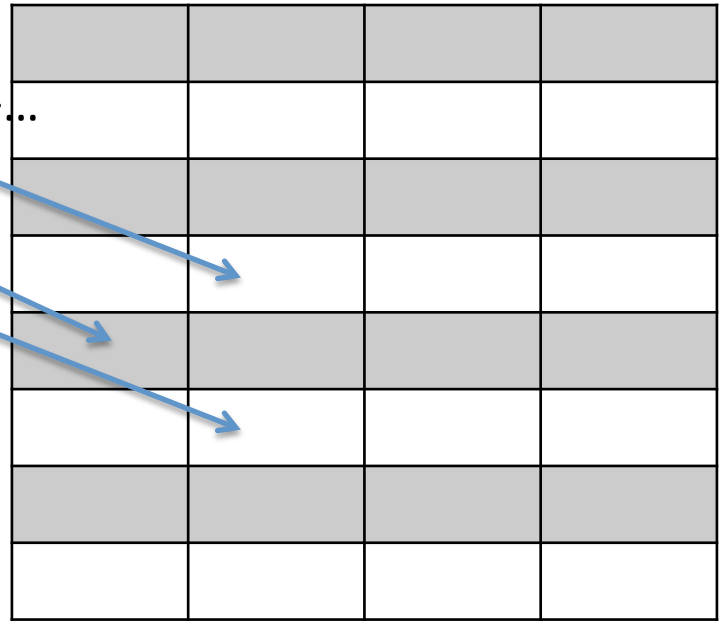
- Stockage et parcours minimisant le nombre de défaut de cache
- typedef long matrix[N][N];
- voisin[i][j] = A[i+1][j] + A[i][j-1] + A[i-1][j] +…

Bonne occupation du cache



Application
jeu de la vie

- Stockage et parcours minimisant le nombre de défaut de cache
- typedef long matrix[N][N];
- voisin[i][j] = A[i+1][j] + A[i][j-1] + A[i-1][j] +…

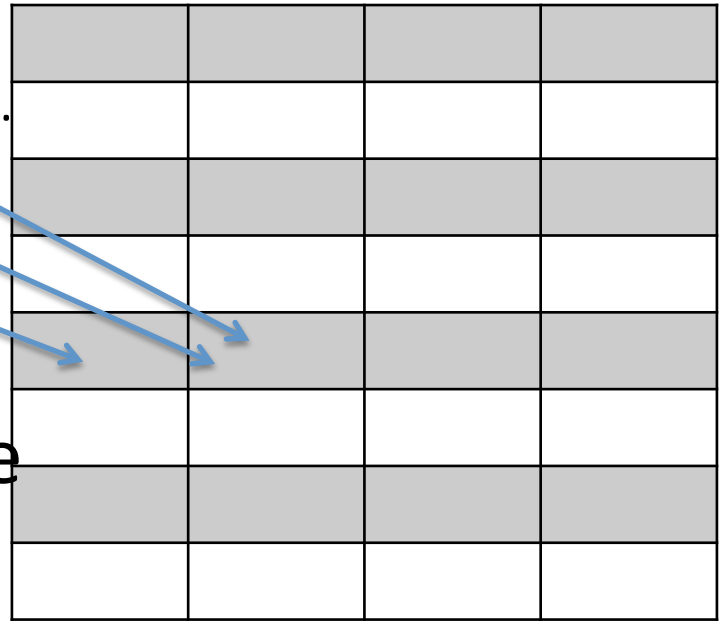


Bonne occupation du cache

- si N = 4…

Application
jeu de la vie

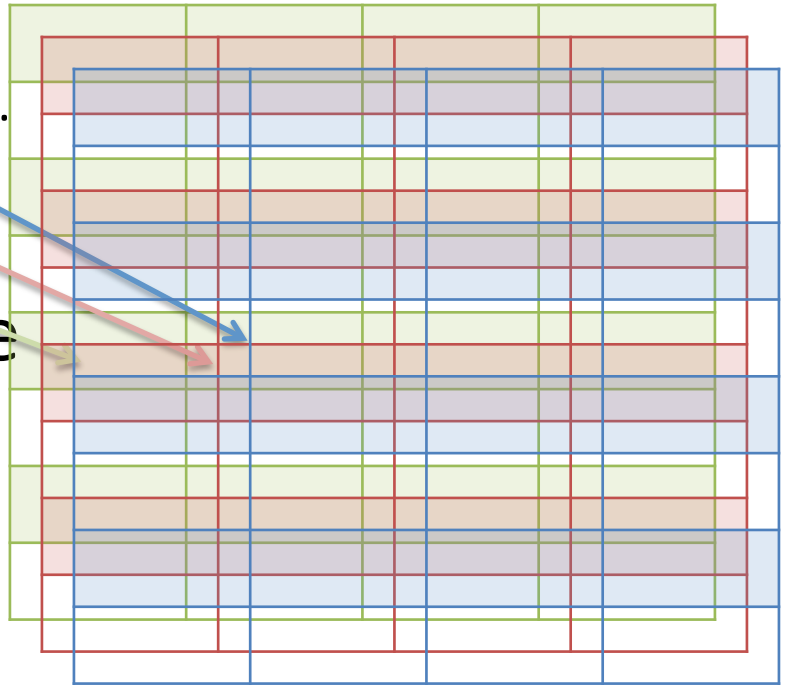
- Stockage et parcours minimisant le nombre de défaut de cache
- typedef long matrix[N][N];
- voisin[i][j] = A[i][j-1] + A[i-1][j] + A[i+1][j] …



- Lorsque N = taille du cache

Application
jeu de la vie

- Stockage et parcours minimisant le nombre de défaut de cache
- typedef long matrix[N][N];
- voisin[i][j] = A[i][j-1] + A[i-1][j] + A[i+1][j] …



- Lorsque N = taille du cache
  - Sauf si associativité > 3
  - et politique LRU

Optimisation
Cache

- Minimiser le nombre de défaut de cache
  - Réutiliser les données chargées dans le cache
    - Adapter les structures de données
  - Limiter le nombre de tableaux utilisés

- Techniques de pavage (tiling)

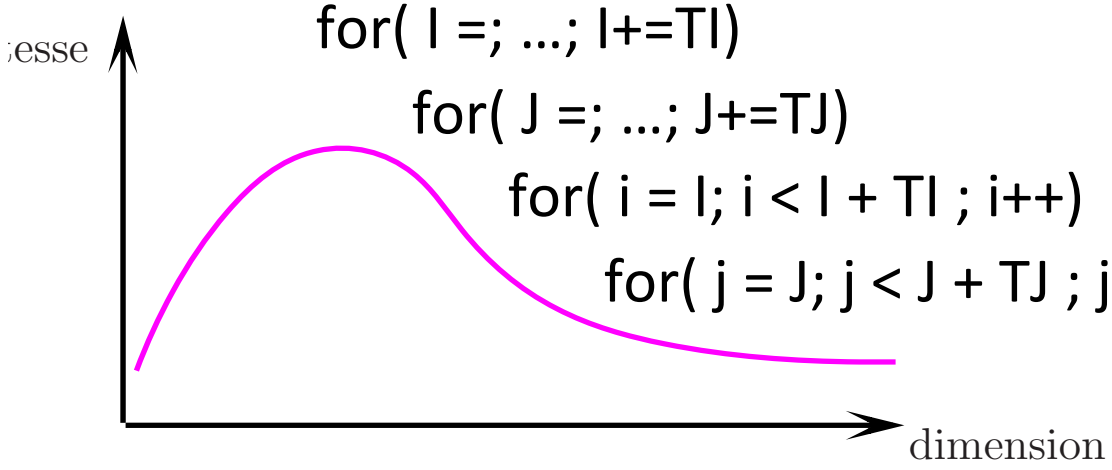
```
for( I =; …; I+=TI)
    for( J =; …; J+=TJ)
        for( i = I; i < I + TI ; i++)
            for( j = J; j < J + TJ ; j++)
```

Optimisation
Cache

- Minimiser le nombre de défaut de cache
  - Réutiliser les données chargées dans le cache
    - Adapter les structures de données
  - Limiter le nombre de tableaux utilisés

- Techniques de pavage (tiling)

```
for( I =; ...; I+=TI)
    for( J =; ...; J+=TJ)
        for( i = I; i < I + TI ; i++)
            for( j = J; j < J + TJ ; j++)
```



Optimisation cache

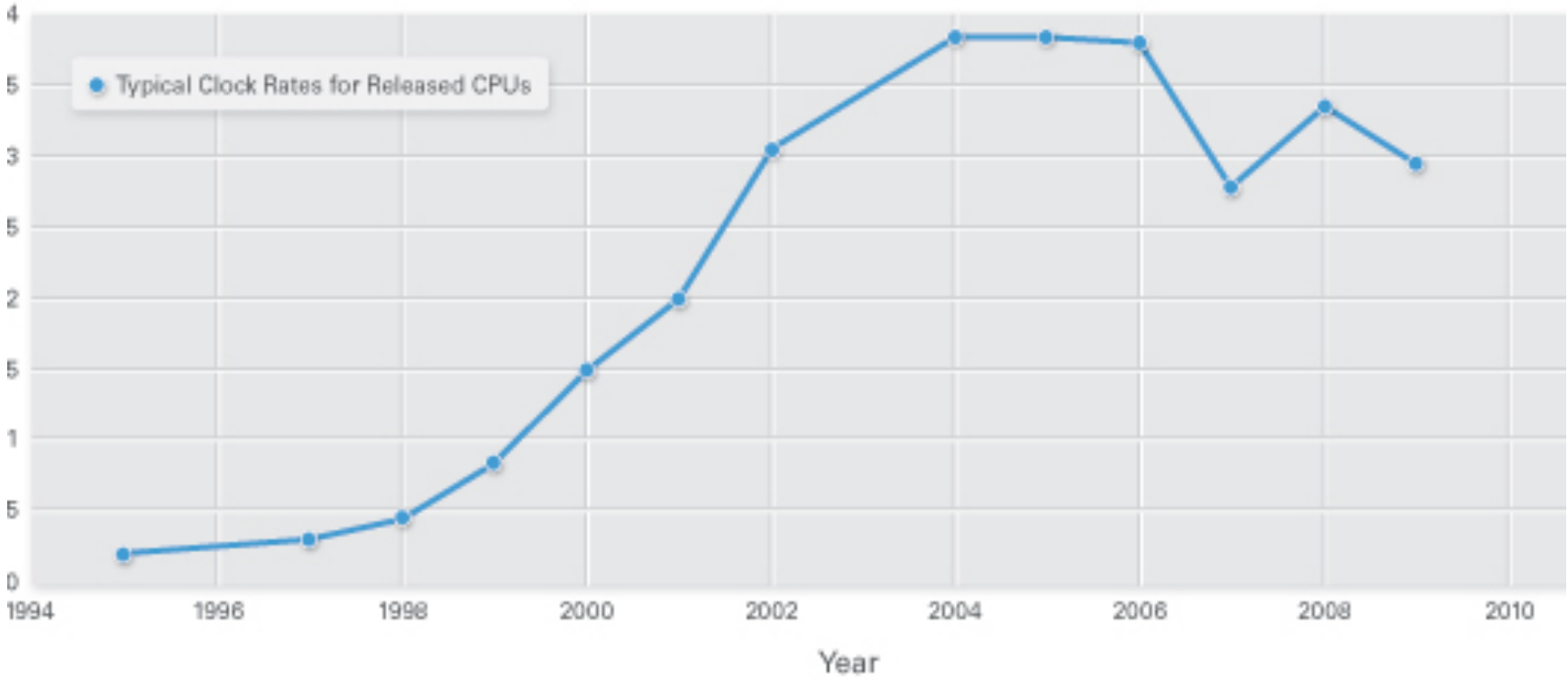
- Deux styles de programmation adaptés aux caches :
  - algorithmes *cache conscious,* tenant compte de la taille du cache
  - algorithmes *cache oblivious,* pensés pour optimiser la localité du travail (Dived & Conquer)

- Les scientifiques utilisent des bibliothèques spécialisées caches-conscious les "basic linear algebra system" BLAS.
    - BLAS1 : vecteur mémoire O(N) pour O(N) opérations
    - BLAS2 : vecteur matrice mémoire O(N^2) pour O(N^2) opérations
    - BLAS3 : matrice matrice mémoire O(N^2) pour O(N^3) opérations
  - Les BLAS de niveau 3 permettent de mieux exploiter les caches : on peut obtenir des facteurs 10 en performance.
  - Ramener le problème à des opérations matrice / matrice quitte à faire plus d'opérations.
  - Facilite la *portabilité des performances*

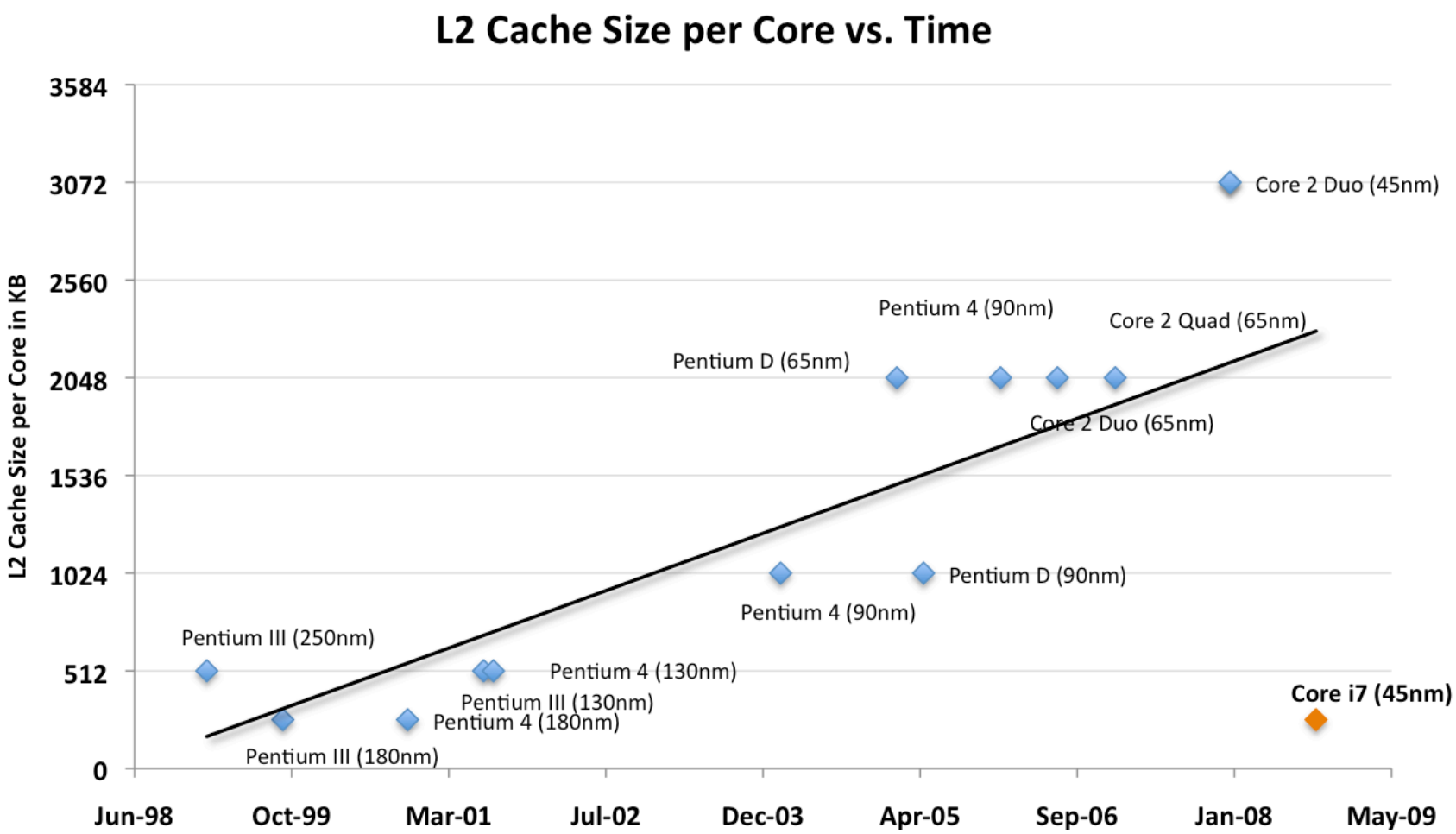
Historique Pentium

486, Pentium	5	Pentium M	14
PIII	10-12	Core	12
P-Pro	12	Core-2	14
PIV	20 31 (45)	Nehalem	16

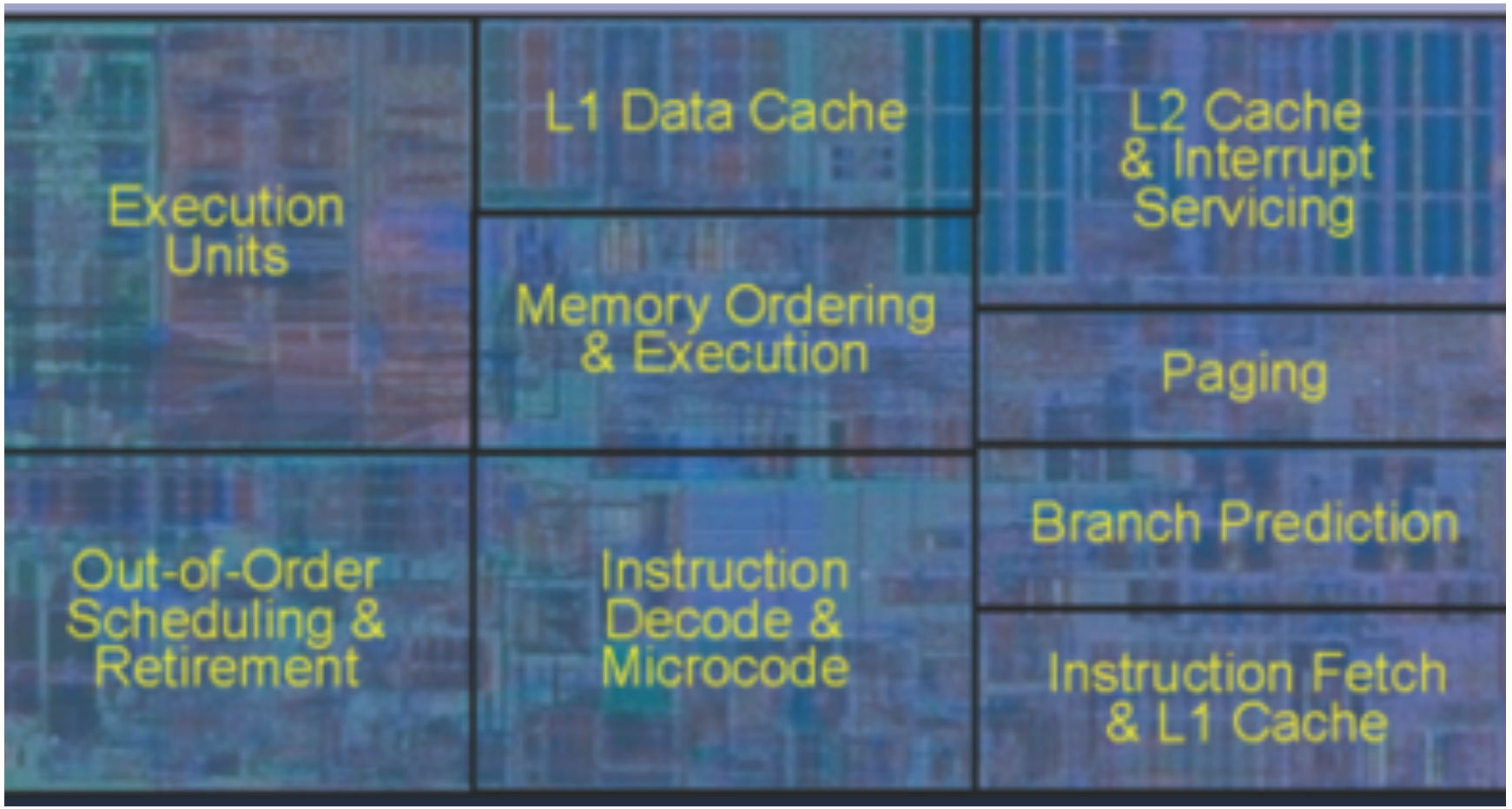
1989	Pipeline	486
1993	MMX, Superscalaire	Pentium, Pentium MMX
1995	OoO, renomage, évaluation spéculative	Pentium-pro
2000	Hyper-threading	Pentium 4
2005	Multicore	Pentium D



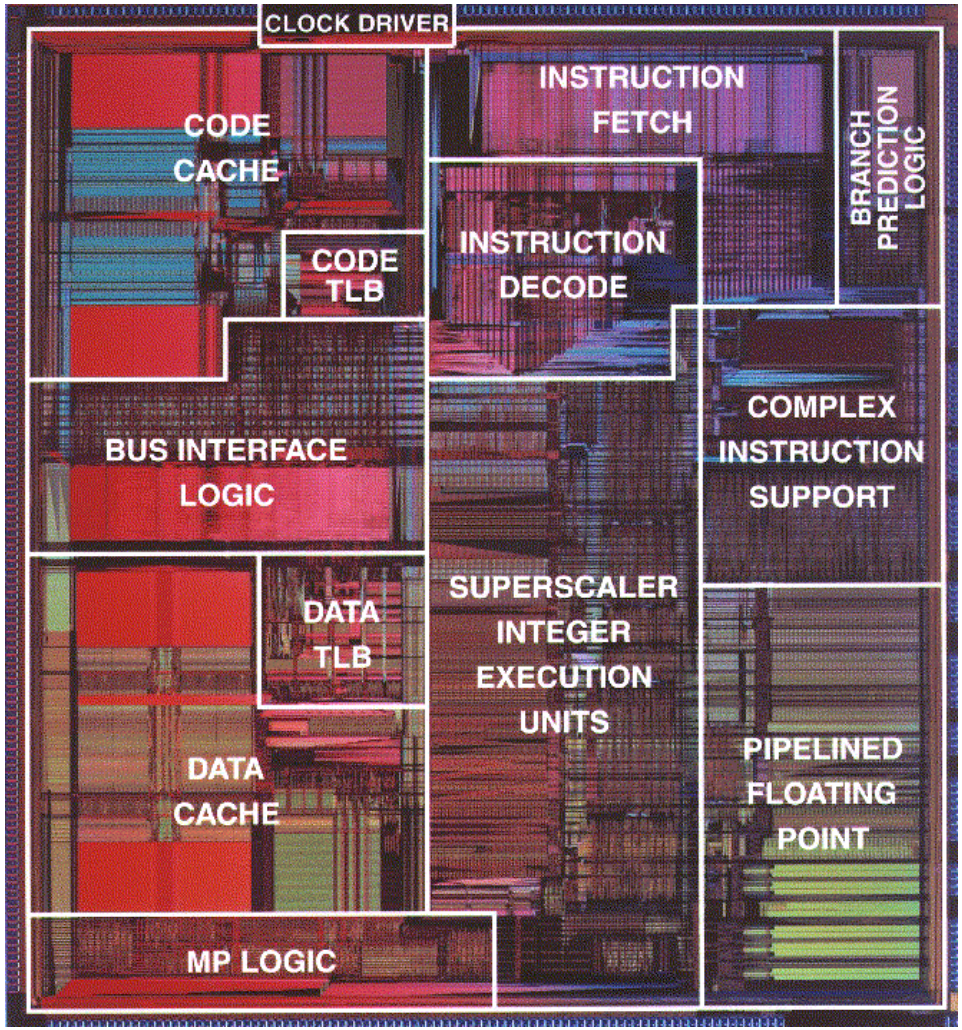
Historique

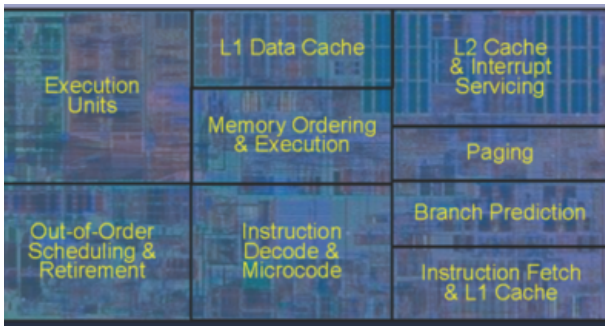

L2 Cache Size per Core vs. Time
L2 Cache Size per Core in KB
3584
3072
2560
2048
1536
1024
512
0
Jun-98
Oct-99
Mar-01
Jul-02
Dec-03
Apr-05
Sep-06
Jan-08
May-09
Core 2 Duo (45nm)
Pentium 4 (90nm)
Core 2 Quad (65nm)
Pentium D (65nm)
Core 2 Duo (65nm)
Pentium D (90nm)
Pentium 4 (90nm)
Pentium III (250nm)
Pentium 4 (130nm)
Pentium III (130nm)
Pentium 4 (180nm)
Pentium III (180nm)
Core i7 (45nm)

1 cœur du Nehalem

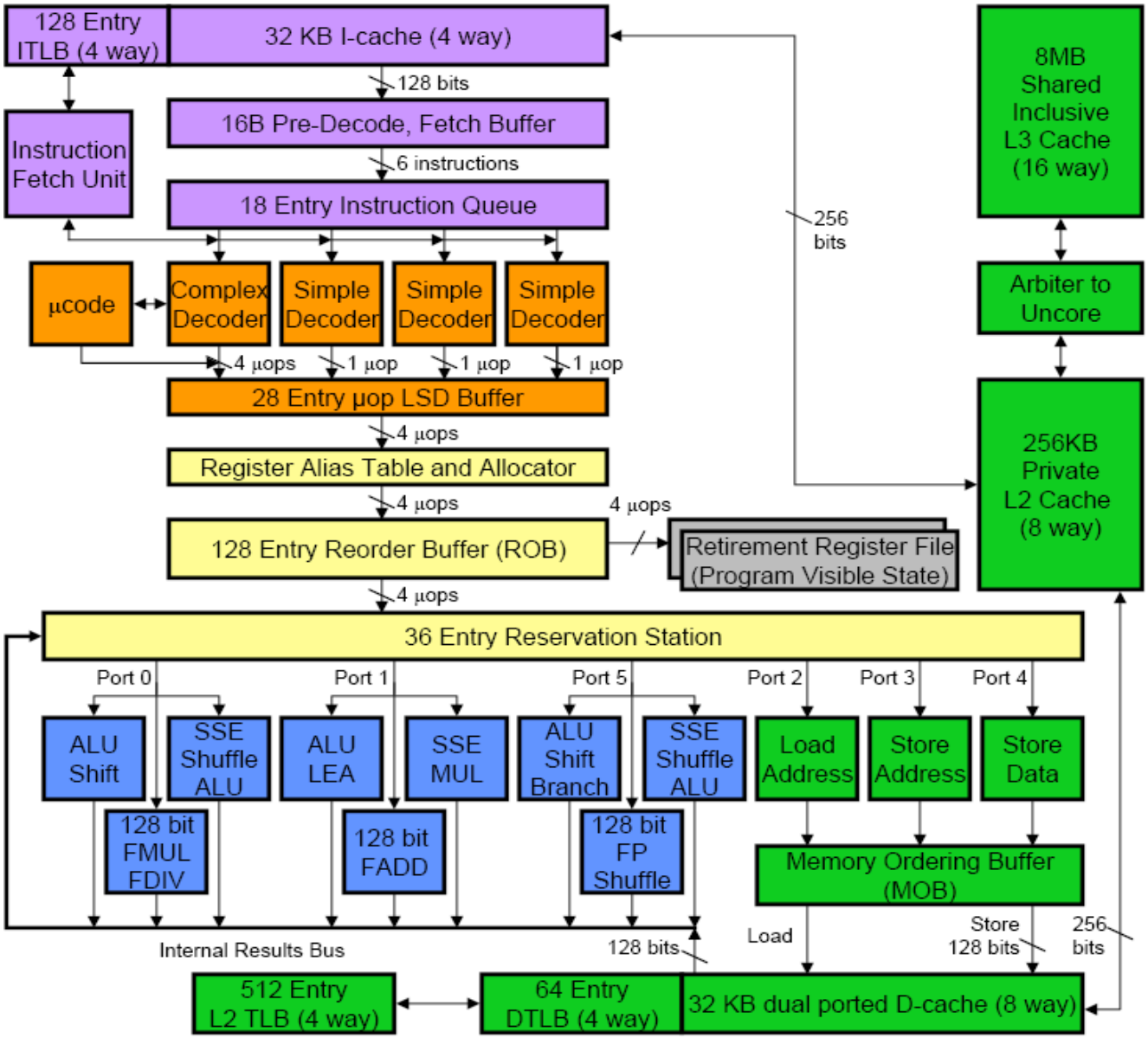


Pentium vs Nehalem



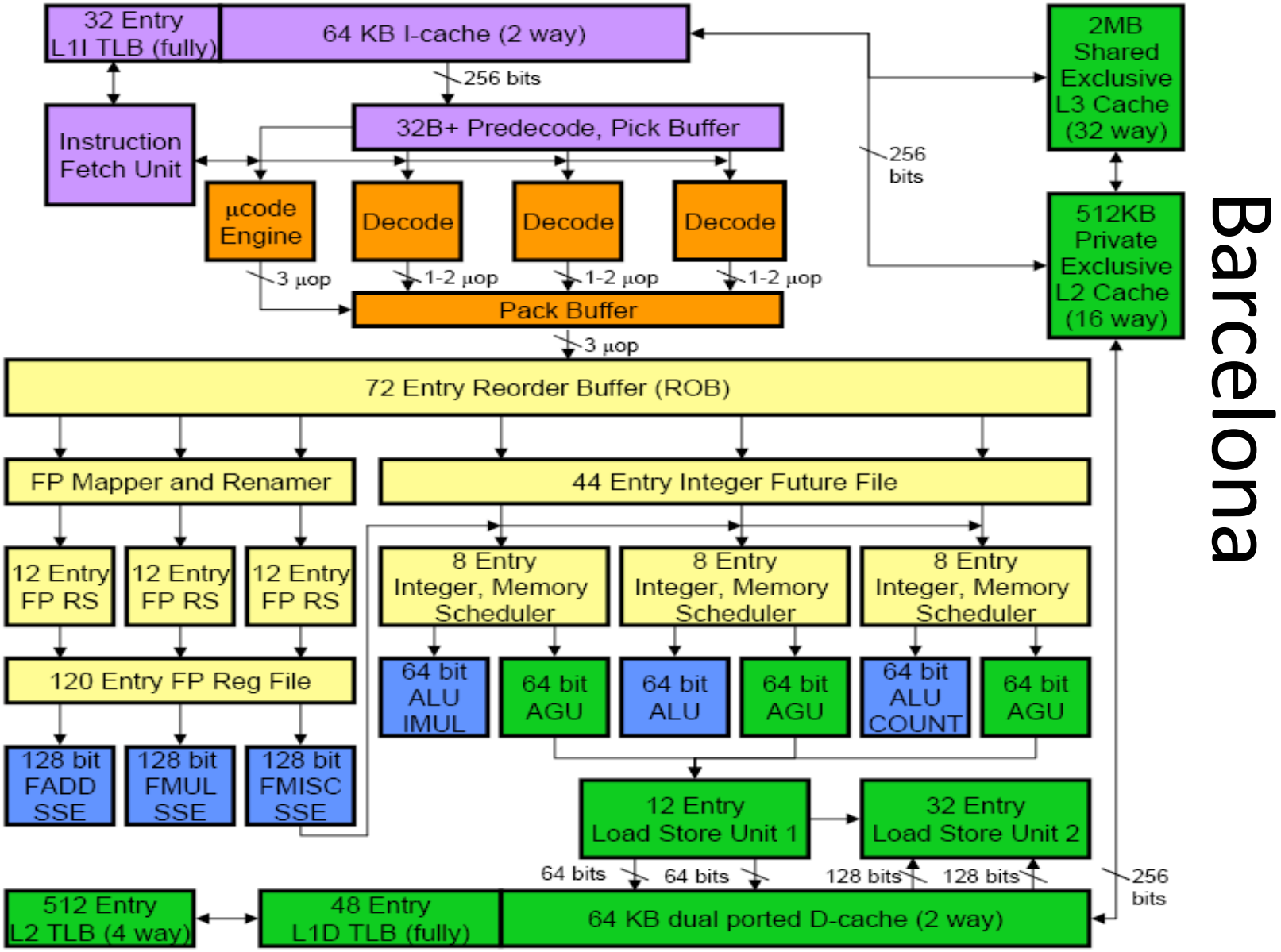


~1/10 du pentium



Nehalem

128 µinst en même temps


32 Entry L1I TLB (fully)
64 KB I-cache (2 way)
256 bits
2MB Shared Exclusive L3 Cache (32 way)
Instruction Fetch Unit
32B+ Predecode, Pick Buffer
256 bits
µcode Engine
Decode
Decode
Decode
512KB Private Exclusive L2 Cache (16 way)
3 µop
1-2 µop
1-2 µop
1-2 µop
Pack Buffer
3 µop
72 Entry Reorder Buffer (ROB)
FP Mapper and Renamer
44 Entry Integer Future File
12 Entry FP RS
12 Entry FP RS
12 Entry FP RS
8 Entry Integer, Memory Scheduler
8 Entry Integer, Memory Scheduler
8 Entry Integer, Memory Scheduler
120 Entry FP Reg File
64 bit ALU IMUL
64 bit AGU
64 bit ALU
64 bit AGU
64 bit ALU COUNT
64 bit AGU
128 bit FADD SSE
128 bit FMUL SSE
128 bit FMISC SSE
12 Entry Load Store Unit 1
32 Entry Load Store Unit 2
64 bits
64 bits
128 bits
128 bits
256 bits
512 Entry L2 TLB (4 way)
48 Entry L1D TLB (fully)
64 KB dual ported D-cache (2 way)

Barcelona