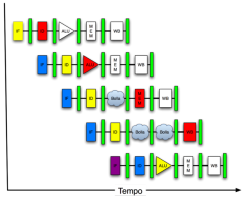
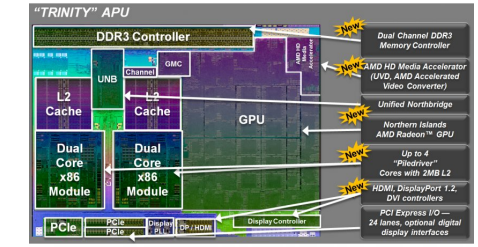
Architecture des ordinateurs à mémoire commune

- Prendre connaissance des technologies en jeu
  - Pour optimiser les programmes
    - Éviter les bévues
    - Mieux coder
    - Déterminer les options de compilation adéquates
  - 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/alf/downloads/michaud/ESIR2_2013.pdf

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

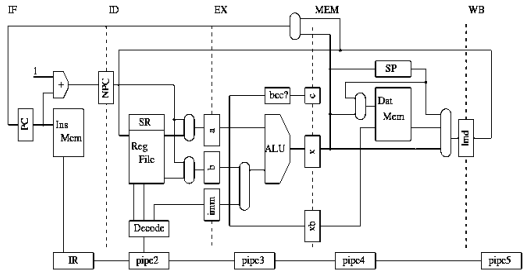
Orienté latence

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

Favorise le débit

Pipeline

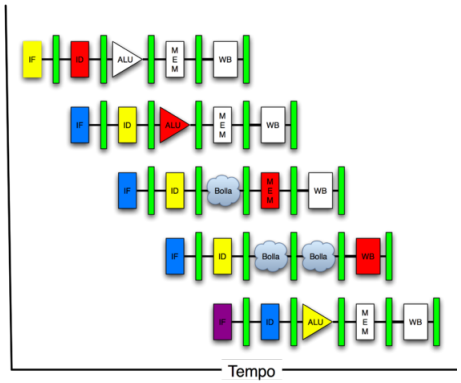
- Travail à la chaîne
  - Fetch Decode Execute Memory Write



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
  - Débit de 1 instruction / cycle
- Exemple:
  - n instructions à exécuter
  - Temps d'exécution moyen *k* cycles / instruction
  - Coût avec pipeline : 4 + n cycles
  - Coût sans pipeline : *k*.n



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

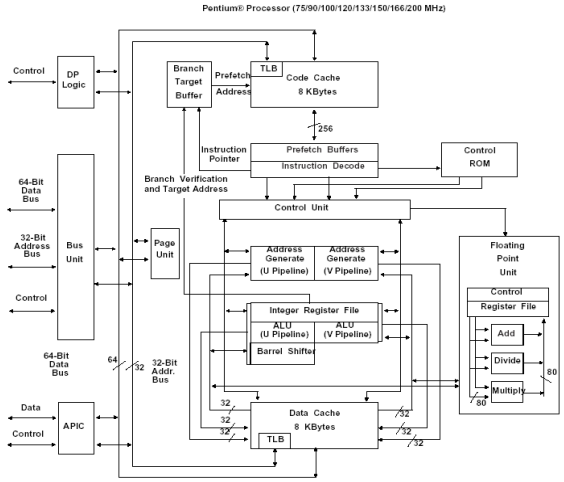
Intérêts du pipeline

- 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

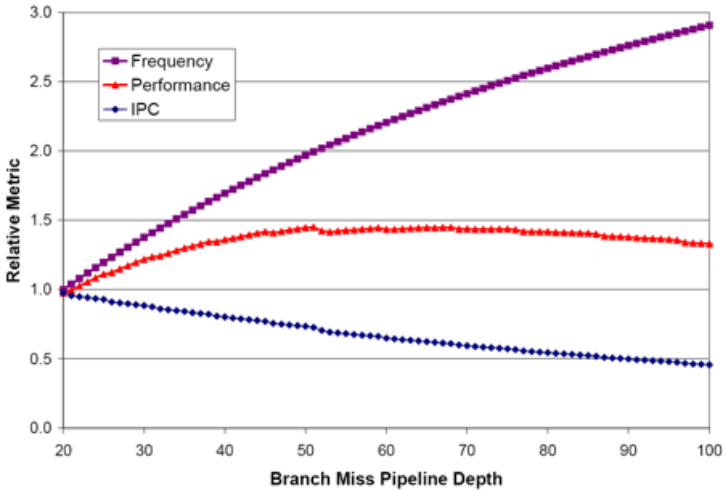
- Dépendance de données
  ADD R1, R1, 1
  MUL R2, R2, R1

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

Les opérations flottantes peuvent nécessiter plusieurs cycles et ne sont pas toutes pipelinées

- Dépendance de contrôle
  - Saut conditionnel et indirection (pointeur de fonction)
  - Très coûteux car bloque le pipeline
    - Exemple : perte de 10 cycles à chaque saut
    - 1 saut toutes 100 instructions => perte de 10% des performances
- Défaut de cache
  - Mémoire ~60ns = 240 cycles à 2,5 GHz

Influence de la profondeur du pipeline sur la fréquence et la performance [Intel]

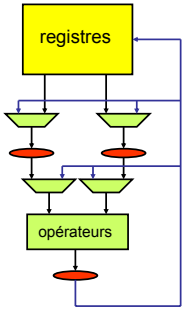


Dépendances de données

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

Dépendances de données

- *Read After Write*
  - Le résultat de I1 est utilisé par I2
- Optimisations possibles
  - Réinjecter les valeurs calculées au plus tôt dans le pipeline
  - Réarranger les instructions
    - Pour insérer des instructions indépendantes
    - À la compilation
    - À l'exécution : exécution des instructions dans le désordre



Dépendances de données

- *Pseudo-dépendances*
  - *Write After Read*
    - I2 écrit dans un registre lu par I1
  - *Write After Write*
    - I2 écrit dans le même registre que I1
- Facilement éliminées en utilisant plus de mémoire
  - À la compilation
  - À l'exécution : renommage de registres

Dépendances de données Renommage

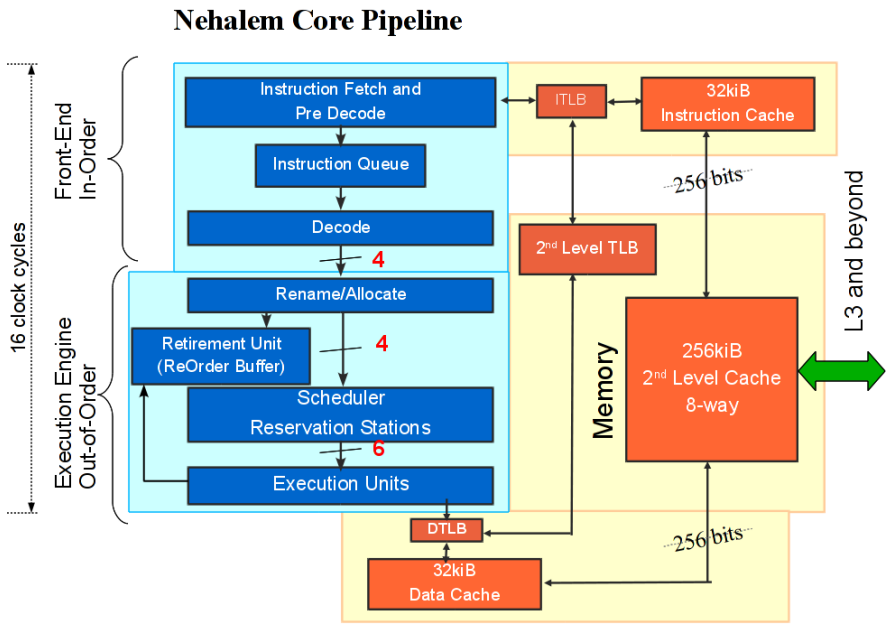
- Pour éviter les pseudo-dépendances
  - À la compilation
    - Static Single Assignation (SSA)
  - Dynamiquement par le processeur
    - # registres physiques > # registres logiques

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

Optimisation du pipeline OoO

- Éviter les bulles en modifiant l'ordre d'exécution des instructions
  - dynamiquement au sein du processeur
  - statiquement par le compilateur
- 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")
```



Transformation de code
Exemple de l' *inlining de fonction*

- Rôle complémentaire compilateur / OoO
  - Le compilateur optimise le code sur une fenêtre large
    - Exposer le plus de parallélisme d'instruction possible
  - Le processeur optimise l'exécution localement, à la volée
    - Petite fenêtre d'optimisation

```
int add( int x, int y)
{
      return x+ y;
}
for (int i=0; i < 100; i++)
       a[i] = add(b[i],c[i]);
```

```
for (int i=0; i < 100; i++)
       a[i] = b[i] + c[i];
```

Transformation de code
Exemple du *Pipeline logiciel*

- Rôle complémentaire compilateur / OoO
  - Le compilateur optimise le code sur une fenêtre large
    - Exposer le plus de parallélisme d'instruction possible
  - Le processeur optimise l'exécution localement, à la volée
    - Petite fenêtre d'optimisation

```
for (int i=0; i < 100; i++)
       a[i] = b[i] + c[i];
```

```
int x = b[0] + c[0] ;
int y = b[1] ;
int z = c[1] ;
for (int i=0; i < 100; i++)
{
      a[i] = x ;
      x = y + z;
      y = b[i+2];
      z = c[i+2];
}
```

Transformation de code
nids de boucles

- Les boucles consomment l'essentiel du temps
- La régularité des boucles rend leur optimisation possible
- Modifier le programme initial pour faire apparaître le parallélisme
  - Modifier l'ordre des opérations
  - Modifier l'utilisation de la mémoire
  - Sans modifier les résultats finaux
- Transformation de boucle
  - Fusion / éclatement
  - Permutation
- Obtenir une boucle interne où tous les indices peuvent être traités en parallèle

```
for
 ….
     for
          forall i
               Seq_i
```

- Dérouler, mélanger les instructions de plusieurs indices de la boucle interne pour optimiser le rendement du pipeline

Algorithme d'*Allen & Kennedy* source de nombreux travaux théoriques et appliqués

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)

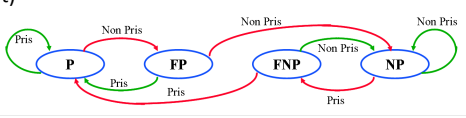
Optimisations des branchements
Prédire le branchement

- Tirer partie de l'évaluation spéculative
- Stratégies de prédiction simples :
  - stratégie toujours sauter
  - stratégie jamais sauter
- Stratégies de prédiction avec historique :
  - stratégie comme la dernière fois (mémoriser 1 bit par saut)
  - stratégie comme d'habitude (mémoriser 2 bits par saut)

- Exemple

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

Nombre de mauvaises prédictions :



Optimisations des branchements
Prédire le branchement

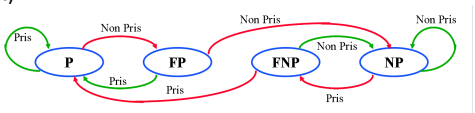
- Tirer partie de l'évaluation spéculative
- Stratégies de prédiction simples :
  - stratégie toujours sauter
  - stratégie jamais sauter
- Stratégies de prédiction avec historique :
  - stratégie comme la dernière fois (mémoriser 1 bit par saut)
  - stratégie comme d'habitude (mémoriser 2 bits par saut)

- Exemple

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

Nombre de mauvaises prédictions :

- *Jamais sauter* se trompe à chaque fin de boucle : 1 + 100 erreurs
- *Toujours:* complément de toujours 99 + 1000
- *Comme la dernière fois* : 1 + 200
- *Comme d'habitude*
  - équivalente ici à jamais



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
    - Majoré par la somme des inverses de 1 à 1000

Optimisations des branchements

Prédire le branchement

- Exemple (C. Michaud)

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

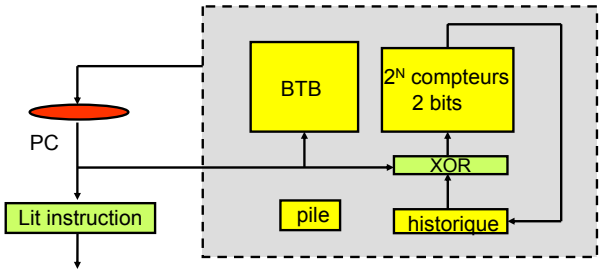
- Branchement de boucle:
  - non pris à la dernière itération seulement
- Branchement de test:
  - pris en moyenne moins de 6,5 fois
    - Majoré par la somme des inverses de 1 à 1000
    - Le $k^{ième}$ entier est a 1/k le plus petit
- stratégie toujours sauter
  - 6,5 erreurs
- stratégie jamais sauter
  - 993,5 erreurs
- Stratégie comme la dernière fois
  - 14 = 1 + 2 x 6,5
- Stratégie comme d'habitude
  - 7,5 = 1 + 6,5

Implémentation de la prédiction de branchement

- Mémorisation de l'adresse cible du saut
  - Table de hachage indexée par l'adresse de l'instructions
  - Branch Target Buffer
- Stratégie *comme d'habitude* :
  - Mémorisation de l'état de l'automate
  - Table indexée par les bits poids faibles de l'adresse de l'instruction
  - Branch History Table
- Retour de fonctions:
  - Petite pile pour prévoir les adresses de retour
  - Lors d'un *CALL* empiler l'adresse de retour
  - Lorsqu'on exécute un retour, dépiler l'adresse et sauter spéculativement

Prédiction à 2 niveaux d'historique

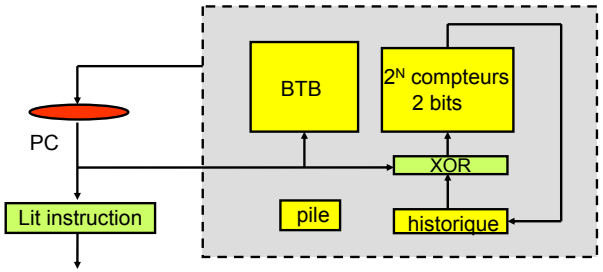
- Historique global = mémorisation des N dernières décisions de branchement conditionnel dans un registre à décalage (non pris → 0, pris → 1)
- Stocké dans la Pattern History Table
- Index de la PHT = adresse du saut xor historique global
  - Permet de capturer un comportement régulier (sauter une fois sur deux)



Branch Target Buffer
+ Pattern History Table

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

- Prévisible par cette technique à partir de
  - 11 bits d'historique
  - Nécessite 2048 compteurs 2 bits
- De nos jours
  - Utilisation d'un prédicteur dédié à la reconnaissance des boucles



Point sur l'optimisation des branchements

- Technologie basée sur l'apprentissage par la mémorisation
  - Améliore les performances des programmes au comportement régulier
    - Les boucles *for* sont optimisées
  - La taille des tables limite la capacité d'apprentissage
- Il faut donc limiter les sauts conditionnels dans les boucles les plus chaudes
  - Éviter les appels récursifs profonds
  - Ne pas trop jouer avec les pointeurs de fonction
- Démonstrations
  - Calculer le nombre d'occurrences de {0,1,2} dans un tableau ne contenant que des {0,1,2}
  - Capacité d'apprentissage d'un nehalem

Mémoire Cache

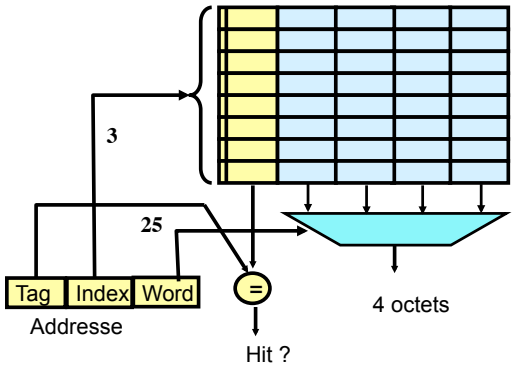
- Observations
  - Plus une mémoire est petite plus son temps de réponse peut être faible
  - Localité spatio-temporelle des programmes
    - Règle 90/10 « *90% de l'exécution se déroule dans 10% du code* »
- Contient une copie d'une faible partie de la mémoire
  - Cache hit / Cache miss
  - Nécessite 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

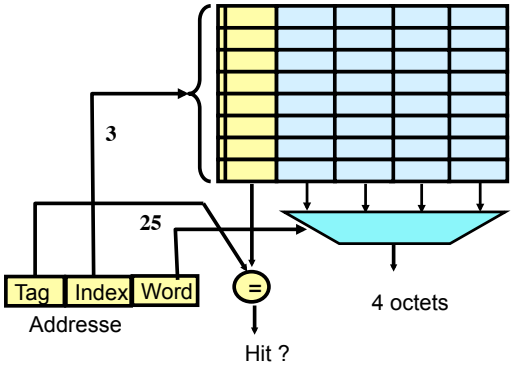
Cache direct

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



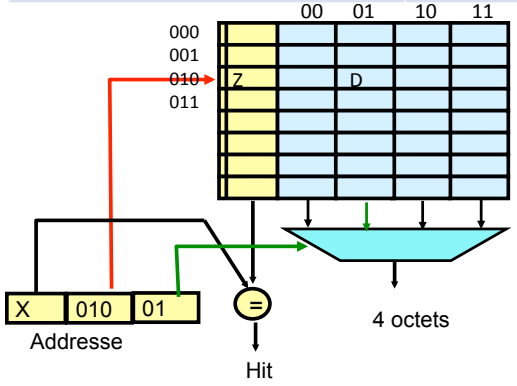
Cache direct

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

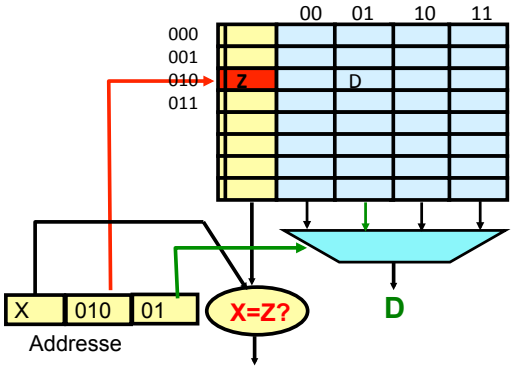


Cache direct

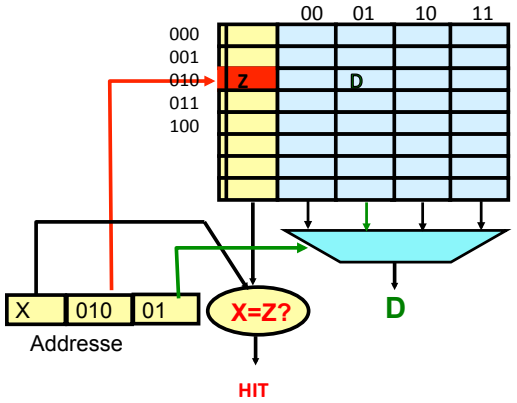
Tag = étiquette a mémoriser / comparer 25 bits	Index dans le cache 3 bits	Choix du Mot 2 bits
$X=x_{24}...x_1$	010	01



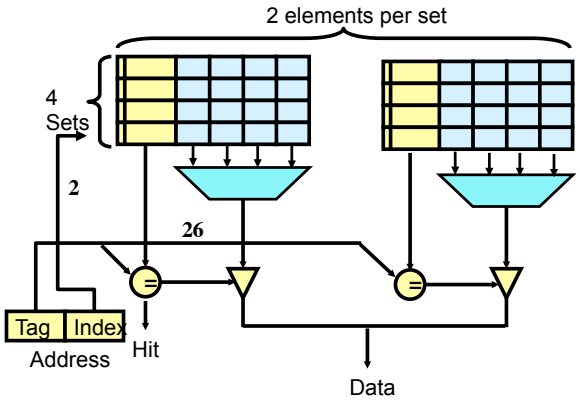
Cache direct



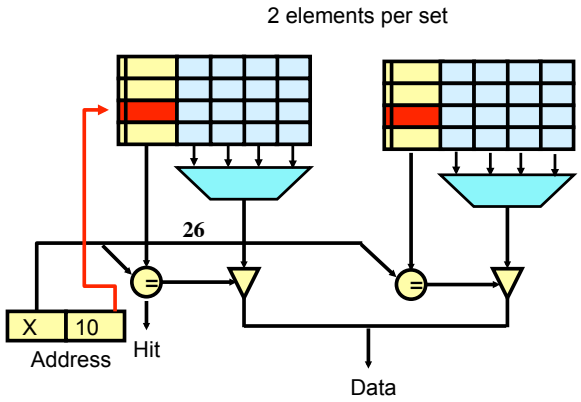
Cache direct



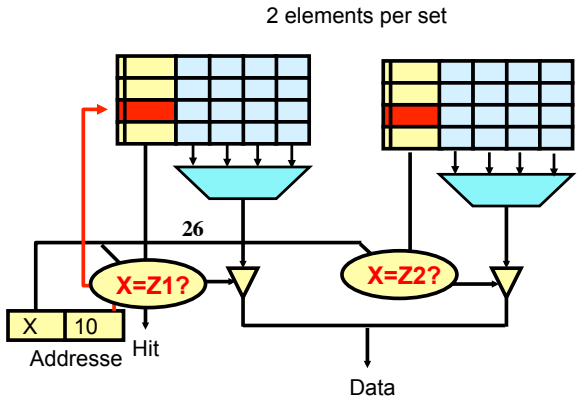
Cache 2 associatif



Cache 2 associatif

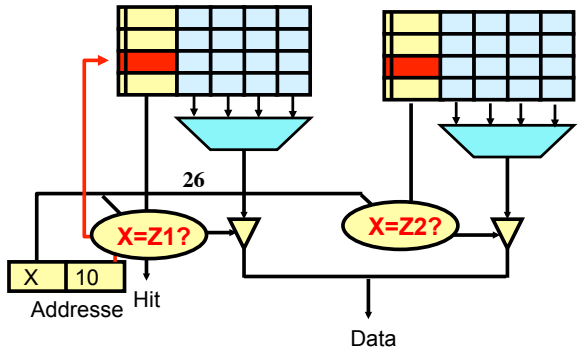


Cache 2 associatif

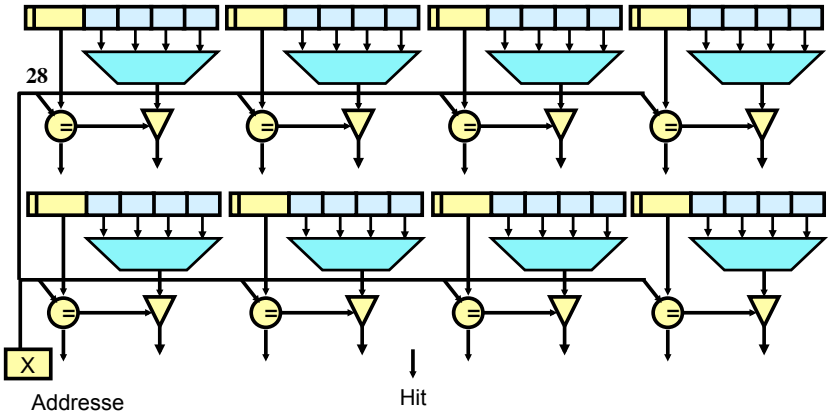


Cache 2 associatif

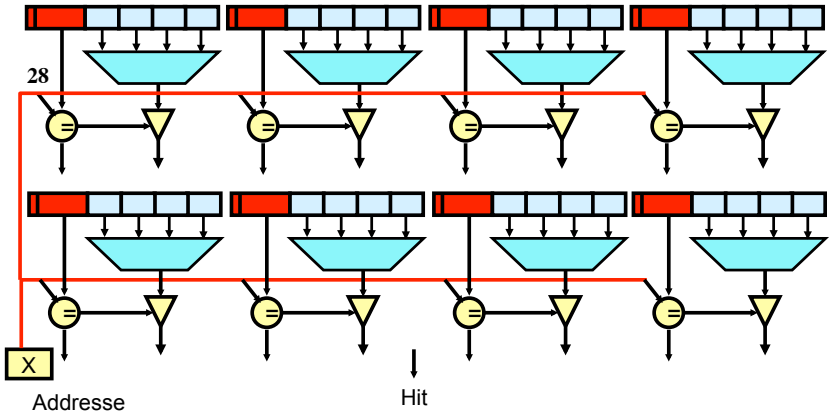
- Algo de type LRU pour l'éviction



Cache associatif



Cache associatif

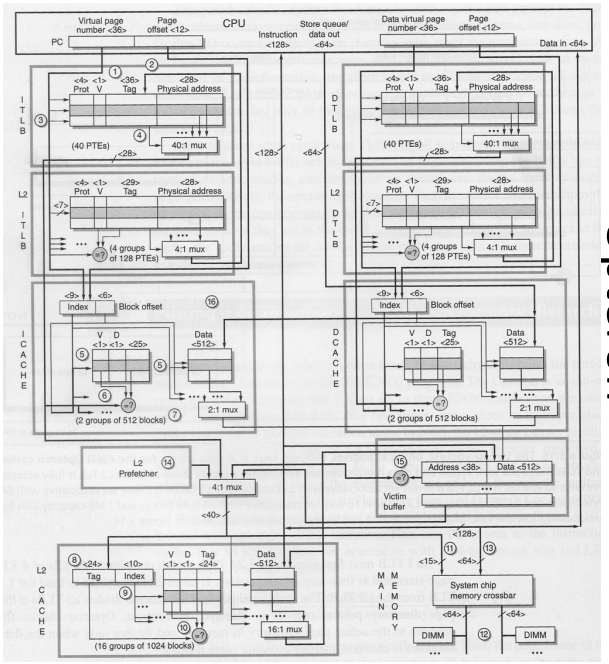


Cache

- 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

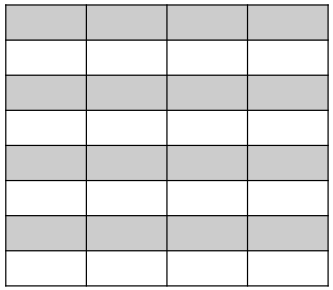


Cache
Optimisation des programmes

- 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

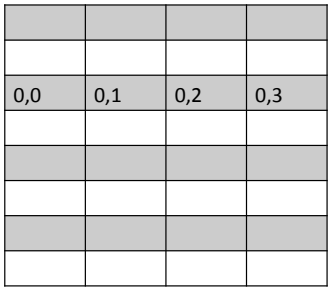
J = 0

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];
```



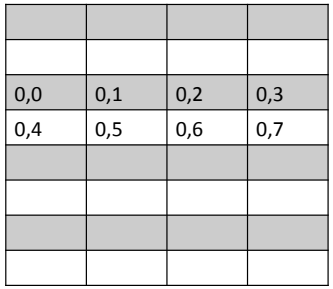
J = 0

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];
```



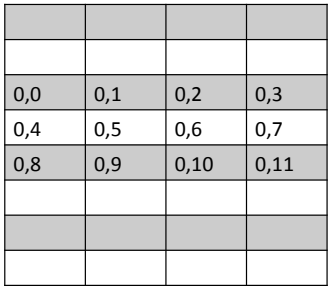
J = 4

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];
```



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];
```

Assez régulier pour que le cache anticipe 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];
```

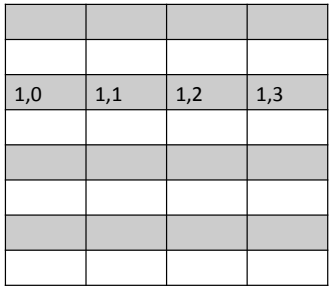
0,0	0,1	0,2	0,3

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];
```

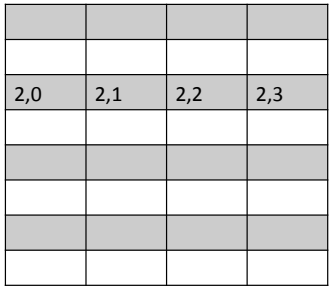


Cache
Optimisation des programmes

- 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];
```

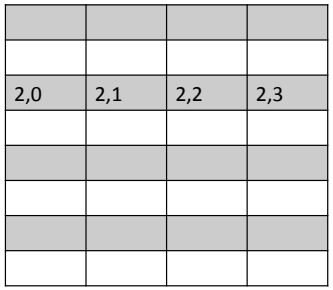


Cache
Optimisation des programmes

- 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

Optimisation
Cache

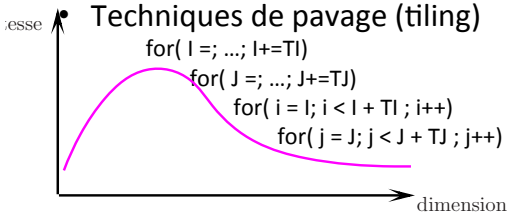
- Minimiser le nombre de défaut de cache
  - Réutiliser les données chargées dans le cache
    - Adapter les structures de données
    - Aligner les données
- Limiter le nombre de pages fréquemment accédées
  - Éviter les défauts de TLB
  - Huge page de Linux

- 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
    - Aligner les données
- Limiter le nombre de pages fréquemment accédées
  - Éviter les défauts de TLB
  - Huge page de Linux
- Techniques de pavage (tiling)



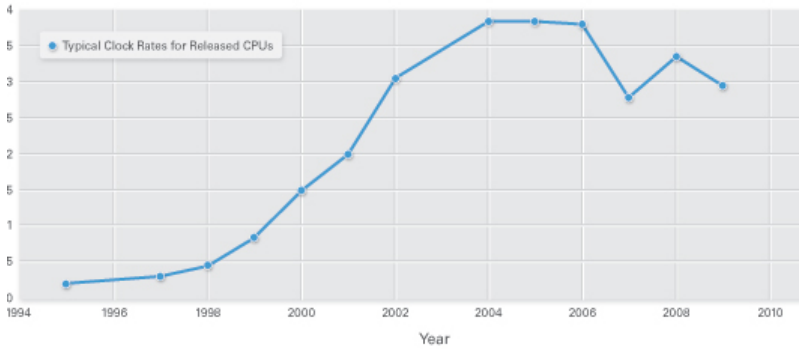
Optimisation cache

- 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 (Divide & 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*

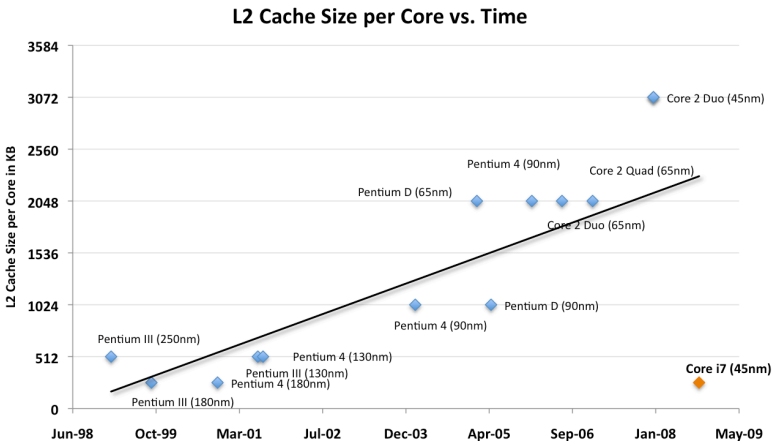
Historique Pentium

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

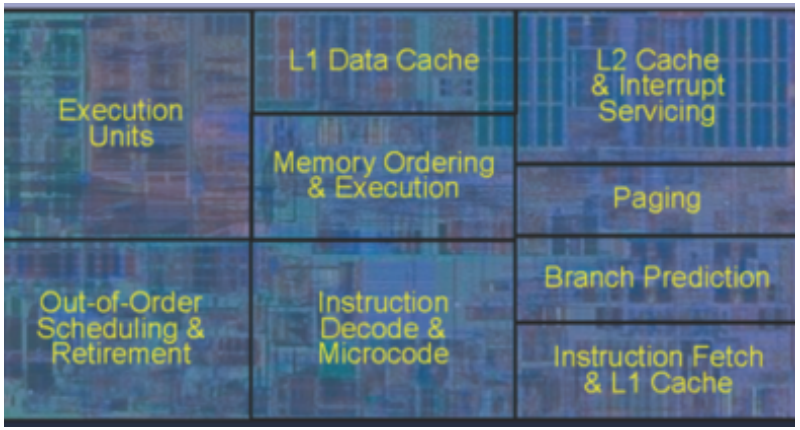
1989	Pipeline	486
1993	MMX, Superscalaire	Pentium, Pentium MMX
1995	OoO, renomage, évaluation spéculative	Pentium-pro
2000	Hyper-threading	Pentium 4
2005	Multicore	Pentium D



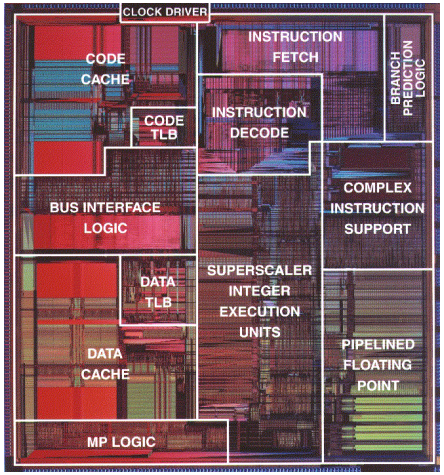
Historique

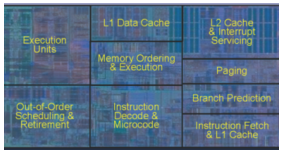


1 cœur du Nehalem

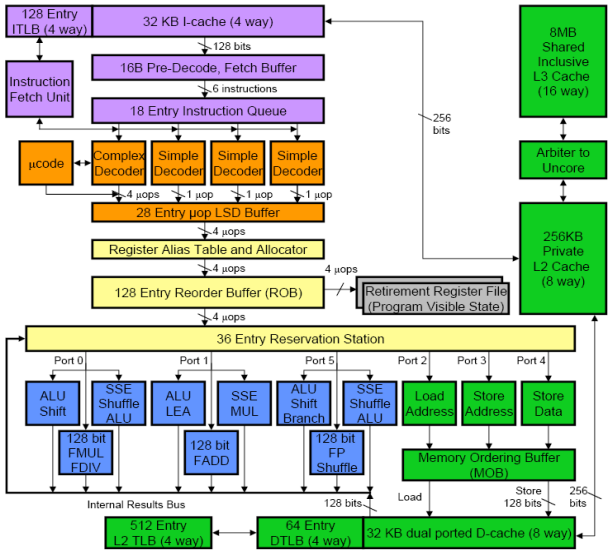


Pentium vs Nehalem



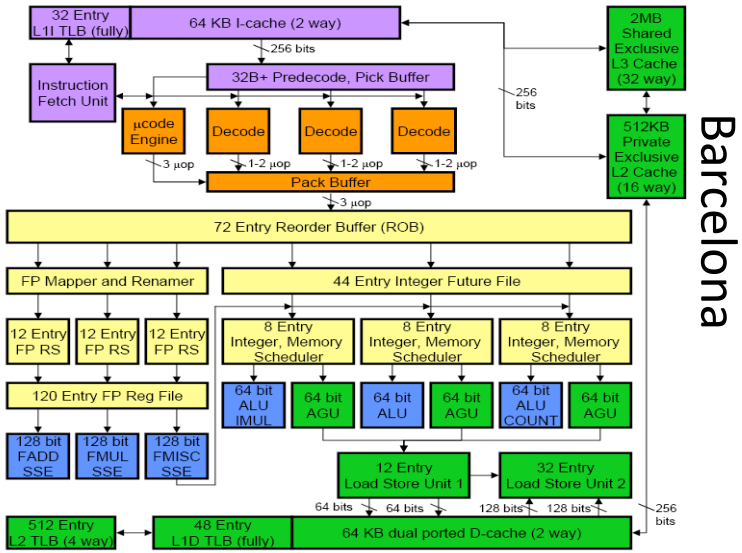


~1/10 du pentium



Nehalem

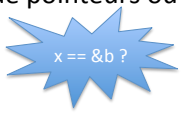
128 µinst en même temps



Barcelona

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 (restrict int *a,
                    restrict int *b,
                    restrict int *c);
```

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

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