Introduction à l'architecture des microprocesseurs

But du cours

- Avoir une idée du fonctionnement des microprocesseurs modernes
- Comprendre les idées importantes
  - Pipeline, caches, prédicteurs de branchements, etc…
- Comprendre d'où vient la « performance »

3

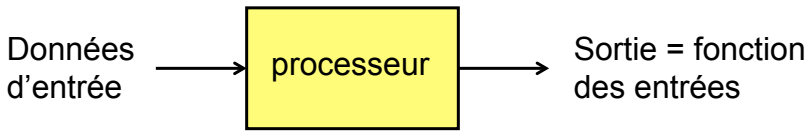
« Microprocesseur » ?

- Processeur = *processor* = calculateur (électronique)

- Micro = petite taille
  - Exemple: Intel Core 2 Duo ➔ 144 mm^2

4

Processeur



- En général, « processeur » = processeur <u>programmable</u>
  - La fonction réalisée par le processeur peut être redéfinie

- Le terme « calcul » doit être compris dans un sens large
  - Un processeur (numérique) prend en entrée des nombres entiers et rend en sortie des nombres entiers
  - Tout ce qui peut être mis en correspondance avec les nombres entiers peut être traité par un processeur
    - Grandeurs physiques discrétisées (longueur, temps,…) , textes, sons, images, programmes, etc…

5

(Pré)histoire

- Calcul arithmétique utilisé depuis des millénaires
  - Mésopotamie, Égypte ancienne, etc…
  - Administration, commerce, …
- On a trouvé des « recettes » (algorithmes)
  - Addition de 2 nombres, multiplication, division, etc…
  - Algorithme d'Euclide pour le calcul de PGCD
  - Crible d'Eratosthene pour identifier les nombres premiers
  - Etc…
- On a inventé des outils pour aider au calcul
  - Abaque (boulier)
- La *pascaline* (Blaise Pascal, 1642) ➔ première vraie machine à calculer mécanique (additions,soustractions)

6

Processeurs électroniques

- 20ème siècle
- Processeurs analogiques
  - Exemple: certaines équations différentielles peuvent être résolues en définissant un circuit électrique dont l'une des grandeurs (tension, courant) obéit à la même équation différentielle
  - Pas flexible, pas pratique
    - On ne peut pas tout résoudre par cette méthode
    - Chaque nouveau problème nécessite de construire un nouveau circuit
  - Précision limitée
    - Exemple: erreurs de mesure
- Processeurs numériques programmables
  - Traite des nombres codés en binaire
  - La fonction réalisée est décrite dans un programme, on peut changer le programme sans changer la machine

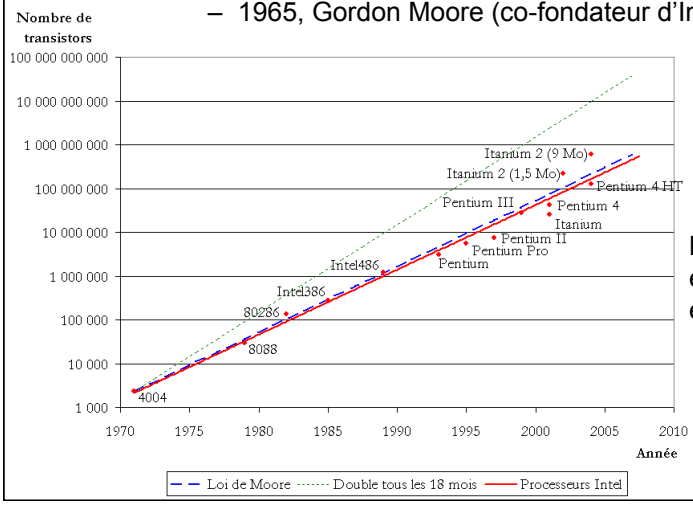
Quelques jalons

- 1946: ENIAC
  - 30 tonnes !
- 1947: invention du transistor
- 1965: IBM S/360
  - Utilise déjà la plupart des techniques utilisées dans les machines actuelles
- 1971: 1er microprocesseur
  - Intel 4004
- 1978: microprocesseur Intel 8086
  - Jeu d'instruction x86

« Loi » de Moore

Le nombre de transistors sur une puce de silicium double environ tous les 2 ans

- 1965, Gordon Moore (co-fondateur d'Intel)



L'aire d'un transistor est divisée par 2 environ tous les 2 ans

Quelques rappels

Représentation binaire

$$A = '\underbrace{a_n a_{n-1} \cdots a_2 a_1 a_0}_{a_k \in \{0,1\}}' = \sum_{k=0}^{n} a_k 2^k$$

Exemples:

$6 = '0110'$	$'110'$
$3 = '0011'$	$\times '11'$
$6+3 = '1001'$	$'110'$
	$+'1100'$
	$18 = '10010'$

Entiers signés, exemple sur 4 bits

$6 = '0110'$	$-1 = '1111'$
$'1001'$	$'0000'$
$+1$	$+1$
$-6 = '1010'$	$1 = '0001'$

bit a_3 est le bit de signe sur 4 bits

Nombres en virgule flottante

IEEE double précision

1 bit	11 bits	52 bits
Signe	**E**xposant	**M**antisse

$$X = (-1)^S \times 1.M \times 2^{E-1023}$$

Exemple: $S = 0$

E ='10000000001' = 1025

$1.M$ ='1.101'

X ='110.1' = 6.5

- Addition
  - Mettre au même exposant, additionner les mantisses, normaliser
- Multiplication
  - Additionner les exposants, multiplier les mantisses, normaliser

Attention: « flottants » ≠ réels

- Les nombres en virgule flottante (nombres « flottants ») ne sont pas les nombres réels des mathématiques

x+y arrondi au nombre flottant le plus proche, ici y

Nombres réels	Nombres flottants
$x+(y+z)=(x+y)+z$	$x+(y+z)\approx(x+y)+z$
$x\times(y\times z)=(x\times y)\times z$	$x\times(y\times z)\approx(x\times y)\times z$
$x\times(y+z)=x\times y+x\times z$	$x\times(y+z)\approx x\times y+x\times z$

Flottants: exemple

```
double x = 0;
for (i=0; i<n; i++)
     x = x + 1./n ;
If (x==1) printf(''exact'');
   else printf(''approximatif'');
```

n=65536 ➔ ''exact''

n=6 ➔ ''approximatif''

$\frac{1}{65536} = 2^{-16}$ ➔ Représentation exacte en format IEEE

$\frac{1}{6}$ ➔ Arrondi au flottant le plus proche

Processeur numérique

- On veut faire travailler les électrons pour nous
- Numérique = bits
  - Tension haute ~ V_{dd} ➔ bit à 1
  - Tension basse ~ 0 ➔ bit à 0
- On assemble des circuits qui prennent des bits en entrée et donnent des bits en sortie
- Quand on veut communiquer avec le monde extérieur, on utilise des convertisseurs analogique ⇔ numérique
  - Connaître la vitesse de la souris
  - Lire le disque dur
  - Afficher un pixel sur l'écran
  - Entendre un MP3 sur le haut-parleur
  - Etc…

15

Exemple: exp(x)

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

$y \leftarrow 1$
$z \leftarrow 1$
Pour n de 1 à 20
  $z \leftarrow z \times \frac{x}{n}$
  $y \leftarrow y + z$

Résultat dans y

- Pour résoudre cet exemple, on doit
  - disposer de « cases »mémoires pour x,y,z et n
  - pouvoir écrire ou lire une valeur dans ces cases
  - pouvoir effectuer des opérations
  - pouvoir tester la valeur de n pour arrêter lorsque n=20

16

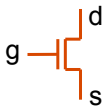
Technologie CMOS

- Il faut un réservoir d'électrons ➔ cristal de silicium « dopé »
  - silicium = semi-conducteur
  - On sait contrôler les électrons
- Élément de base: le transistor
  - On l'utilise à la manière d'un interrupteur
  - Technologie CMOS (Complementary Metal-Oxide Semi-conductor) ➔ 2 types de transistors
    - Transistor de type N
    - Transistor de type P

Transistors MOSFET

$V_{dd} \approx 1\ \mathrm{V}$

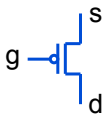
Type N



$V_{gs} = 0$ ⟷ d, s (open)

$V_{gs} = V_{dd}$ ⟷ d, s (closed)

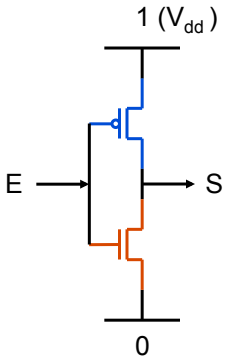
Type P



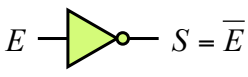
$V_{gs} = 0$ ⟷ s, d (open)

$V_{gs} = -V_{dd}$ ⟷ s, d (closed)

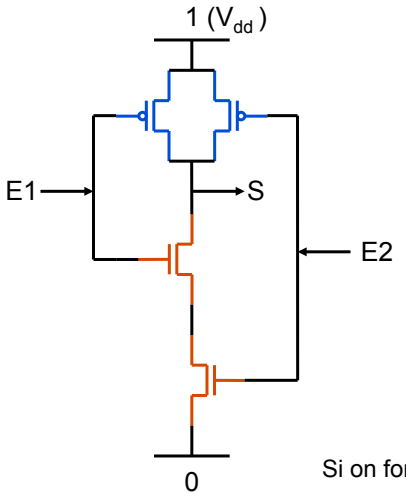
Porte NOT



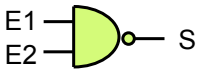
E	S
0	1
1	0



Porte NAND



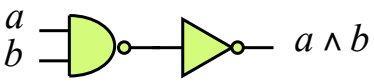
E1	E2	S
0	0	1
0	1	1
1	0	1
1	1	0



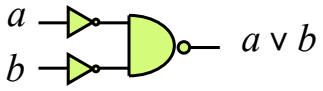
Si on force une des entrées à 1, on obtient un NOT

Portes AND,OR,XOR,…

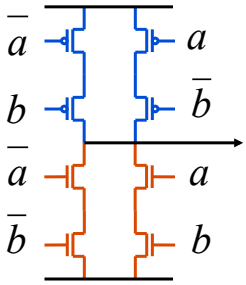
AND



OR



XOR

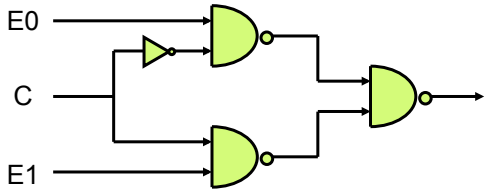


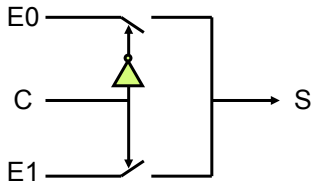
$a \oplus b = (a \wedge \overline{b}) \vee (\overline{a} \wedge b)$

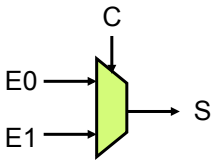
(vaut 0 si $a = b$)

21

Multiplexeur (MUX)



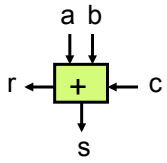




22

Additionner 2 nombres entiers

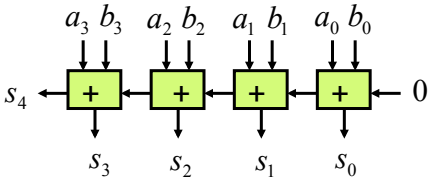
Additionneur à 1 bit



$$s = a \oplus b \oplus c$$

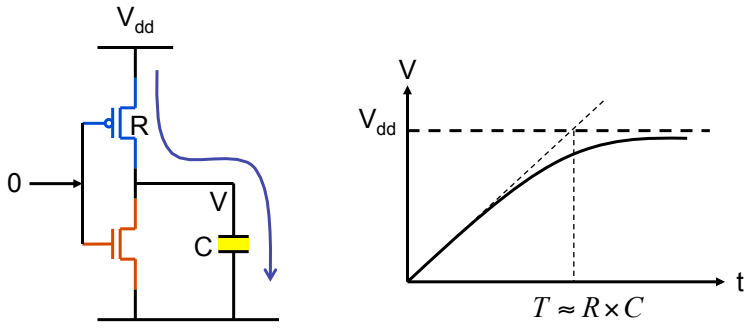
$$r = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

Additionneur à 4 bits



Temps de réponse

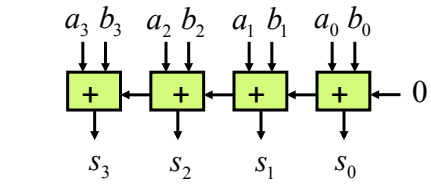
Ça dépend de ce qu'on met en sortie de la porte …



Si on connecte la porte à un grand nombre d'autres portes, la capacité C est grande, et le temps de réponse élevé

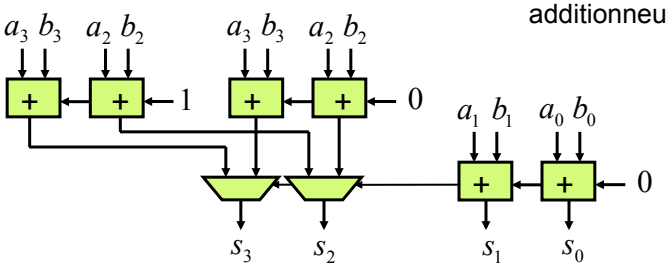
La miniaturisation des circuits diminue C ➔ circuits plus rapide

On peut parfois aller plus vite en utilisant plus de transistors …



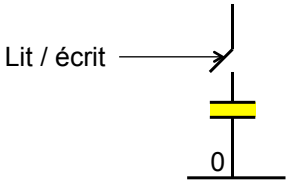
Temps de réponse pour S_3 = 4 additionneurs 1 bit

Temps pour S_3 = 2 additionneurs 1 bit et 1 MUX



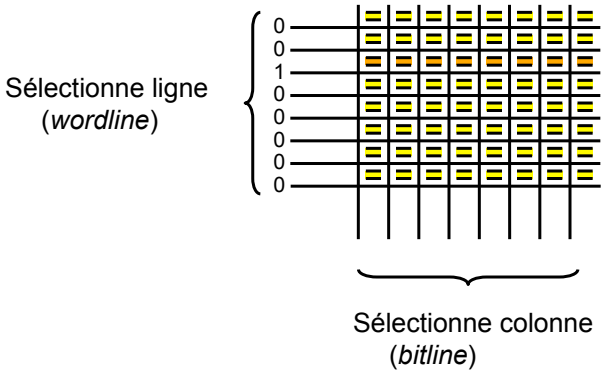
Mémoire dynamique (DRAM)

Une capacité chargée, lorsqu'elle est isolée, conserve sa charge ➔ mémorise 1 bit



- Les courants de fuite déchargent la capacité progressivement ➔ il faut « rafraîchir » le bit périodiquement
- La lecture détruit le bit ➔ Il faut réécrire le bit après chaque lecture

Mémoire à grand nombre de bits

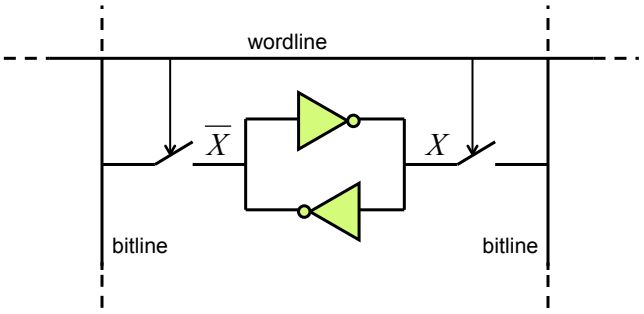


- Chaque case a un numéro distinct = adresse
- L'adresse détermine la ligne et la colonne sélectionnées

27

Mémoire statique (SRAM)

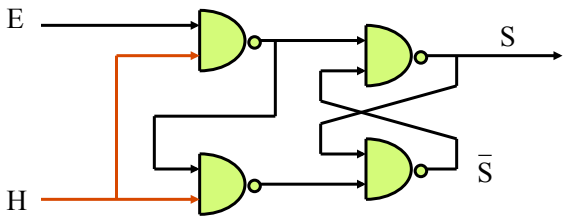
Portes NOT « tête-bêche » mémorisent 1 bit



- Écriture: activer la wordline, mettre la valeur du bit sur une bitline, son complément sur l'autre bitline, désactiver la wordline
- Lecture: précharger les 2 bitlines à V_{dd} , activer la wordline, attendre pendant un temps suffisant pour que l'amplificateur différentiel détecte une différence entre les 2 bitlines, désactiver la wordline

28

Verrou (« latch »)

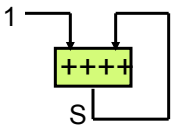


H = 1 ➡ S = E Le verrou est « passant »

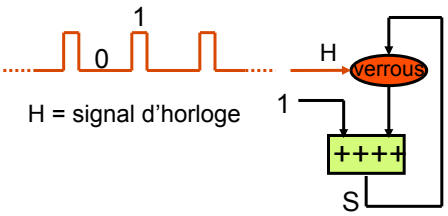
H = 0 ➡ S figé à la valeur qu'il avait juste avant que H passe de 1 à 0
La sortie est maintenant isolée de l'entrée

Comment faire un compteur ?

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



Problème: les bits de poids faible arrivent avant ceux de poids fort

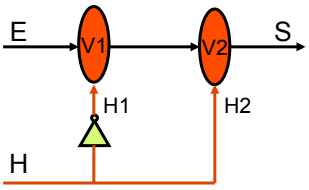


H = signal d'horloge

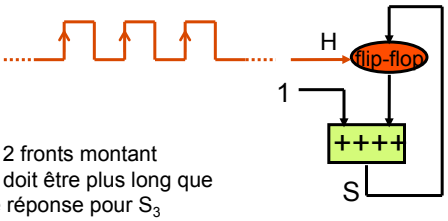
H=0: plus long que le temps de réponse pour S_3

H=1: plus court que le temps de réponse pour S_0

« Flip-flop » = 2 verrous



S prend la valeur de E lorsque H passe de 0 à 1 (= front montant)

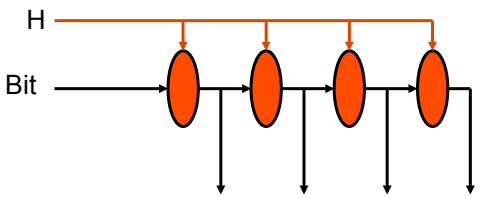


Le temps entre 2 fronts montant consécutifs doit être plus long que le temps de réponse pour S_3

31

Registre à décalage

Exemple: registre à décalage 4 bits



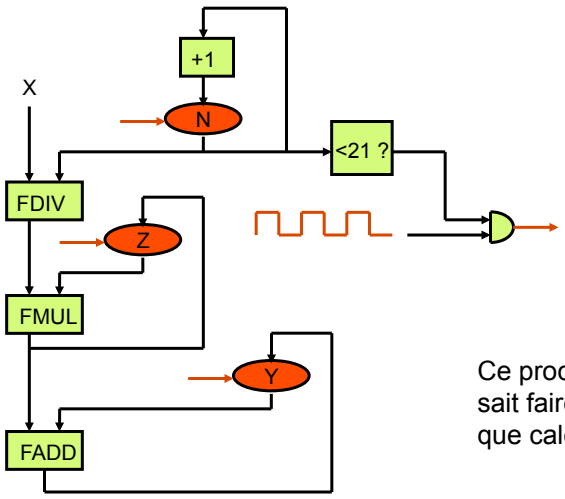
Pour insérer un nouveau bit, envoyer un front montant sur H

Le registre mémorise les 4 derniers bits insérés

32

Exp(x)

On pourrait faire un processeur spécialisé…



Ce processeur ne sait faire rien d'autre que calculer exp(x)

33

Processeur programmable

- Processeur programmable peut effectuer toutes sortes de tâches
- On crée un jeux d'instructions = « langage » codé en binaire
- On écrit un programme (= suite d'instructions décrivant la tâche à effectuer) qu'on stocke en mémoire
- On fabrique un processeur permettant d'exécuter les programmes écrits dans ce jeux d'instructions
- On exécute le programme sur le processeur
- Si on veut effectuer une autre tâche, on écrit un autre programme qu'on exécute sur le même processeur

34

Architecture / microarchitecture

- L'architecture est la machine abstraite telle qu'elle est perçue par le programmeur
  - En gros, architecture = jeux d'instruction
- La microarchitecture est la machine physique, c'est une mise en œuvre particulière d'une architecture donnée
  - Il peut exister plusieurs microarchitectures possibles pour une même architecture
  - Le programmeur ne voit pas la différence, sauf éventuellement dans la vitesse d'exécution des programmes
- Exemple:
  - Intel x86 = jeu d'instructions = architecture
  - Intel « Core » = microarchitecture
  - Le terme « architecture » est parfois utilisé pour architecture+microarchitecture

Évolution, compatibilité binaire

- Les jeux d'instructions évoluent peu dans le temps, ou lentement
  - Mais ils évoluent quand même
  - Intel x86 en 2009 différent d'Intel x86 en 1978
- La microarchitecture évolue plus rapidement
- Compatibilité binaire
  - Les programmes (= « codes ») écrits dans d'anciennes versions du jeu d'instructions sont valides dans la nouvelle version
    - Les modifications du jeu d'instructions se font par ajouts, pas par retraits
  - Cela signifie qu'un programme qui s'exécutait sur une ancienne microarchitecture s'exécute sur la nouvelle
- Ne pas assurer la compatibilité binaire serait commercialement suicidaire pour un fabricant de processeurs

Architecture / Jeux d'instructions

En anglais: *Instruction-set Architecture* (ISA)

Registres architecturaux

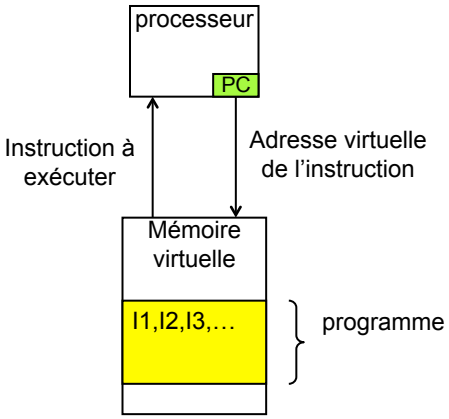
- Registres = petite mémoire
- Registres architecturaux ➔ définis par le jeu d'instructions
  - Registres vus par le programmeur
- En général, petit nombre de registres
  - Exemple: 32 registres « entiers », 32 registres « flottants » (virgule flottante), + registres spéciaux (compteur de programme, pointeur de pile, etc…)
- Mise en œuvre microarchitecturale
  - En générale, petite mémoire SRAM à l'intérieur du microprocesseur
  - Registres spéciaux peuvent être faits avec des flip-flops

Mémoire virtuelle

- C'est la mémoire vue par le programmeur
- « architecture 64 bits » = adresses virtuelles sur 64 bits
  - La taille mémoire vue par le programmeur vaut 2^{64} octets
  - On verra plus loin dans le cours comment on transforme les adresses virtuelles en adresses physiques
  - En général, dans une architecture 64 bits, un registre « entier » contient 64 bits de donnée

Compteur de programme

- Registre spécial appelé compteur de programme (*program counter*, PC)
- Initialiser PC=@I1
- Lit et exécute instruction I1 rangée à l'adresse PC
- PC← PC+taille[I1]
- Lit et exécute instruction I2 rangée à l'adresse PC
- PC← PC+taille[I2]
- Lit et exécute I3
- Etc…



Jeux d'instructions RISC / CISC

- RISC (Reduced Instruction-Set Computer)
  - Exemple: MIPS, Alpha, PowerPC, Sparc
  - Taille d'instruction fixe
    - Exemple: 4 octets par instruction (PC ← PC+4)
  - Opérations simples : lit 2 registres maxi, écrit 1 registre maxi
  - Modes d'adressage mémoire simples
- CISC (Complex Instruction-Set Computer)
  - Exemple: IBM 360, VAX, Intel X86
  - Taille d'instruction variable
  - Opérations complexes autorisées
    - Exemple: instructions x86 de manipulation de chaînes
  - Modes d'adressage mémoire complexes autorisés

Exemples d'instructions

- r1 ← CONST
  - Écrit CONST dans registre r1
  - CONST = constante codée dans l'instruction
- r3 ← r1 OP r2
  - Lit registres r1 et r2, exécute opération OP, écrit résultat dans r3
  - Opérations sur les entiers: add,sub,mul,div,and,or,xor,shift, etc…
  - Opérations sur les flottants: fadd,fsub,fmul,fdiv,sqrt, …
- r2 ← LOAD r1
  - Utilise la valeur contenue dans registre r1 comme adresse mémoire
  - Lit la valeur stockée en mémoire à cette adresse
  - Copie la valeur dans registre r2
- STORE r1,r2
  - Valeur dans r1 copiée dans la case mémoire dont l'adresse est dans r2

Instructions de contrôle

- Savoir faire des opérations n'est pas suffisant

Exp(x)

$$
\begin{array}{l}
z \leftarrow z \times \frac{x}{1} \\
y \leftarrow y + z \\
z \leftarrow z \times \frac{x}{2} \\
y \leftarrow y + z \\
\vdots \\
z \leftarrow z \times \frac{x}{20} \\
y \leftarrow y + z
\end{array}
$$

Et si on veut itérer un milliard fois ?
➔ le programme prend une place énorme en mémoire

- On veut pouvoir manipuler le compteur de programme pour faire des tests, des boucles, des procédures…

Exemples d'instructions de contrôle

- BNZ r1,DEP
  - Branchement conditionnel
  - Saute à l'adresse PC+DEP si la valeur dans r1 est non nulle
  - DEP est une constante codée dans l'instruction
    - peut être négative (saut vers l'arrière)
- JUMP r1
  - Saute à l'adresse contenue dans r1
- r2 ← CALL r1
  - Saute à l'adresse contenue dans r1 et écrit PC+4 dans r2
  - Utilisé pour connaître le point d'appel lorsqu'on fait un retour de procédure
  - on peut aussi utiliser un registre spécialisé implicite au lieu de r2

Modes d'adressage

- r2 ← LOAD r1+DEP
  - Adresse = r1+DEP
  - DEP = constante codée dans l'instruction
  - Appelé adressage basé
  - Pratique pour accéder aux champs d'une structure
- r3 ← LOAD r1+r2
  - Adresse = r1+r2
  - Appelé adressage indexé
  - Pratique pour accéder à un tableau
- r2 ← LOAD (r1)
  - 2 lectures mémoire (CISC)
  - Lit la valeur 64 bits stockée en mémoire à l'adresse contenue dans r1
  - Utilise cette valeur comme adresse finale
  - Appelé adressage indirect
  - Pratique quand on programme avec des pointeurs
  - RISC: on doit utiliser 2 LOAD

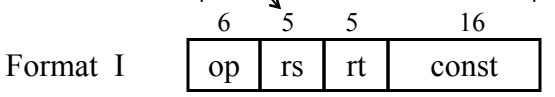
Exp(x)

x dans r33
y dans r34
z dans r35
n dans r1

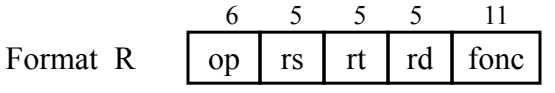
```
          r33 ← x
          r34 ← 1
          r35 ← 1
           r1 ← 1
Boucle:  r36 ← r33  FDIV  r1
         r34 ← r34  FMUL  r36
         r35 ← r35  FADD  r34
          r1 ← r1   ADD   1
          r2 ← r1   SUB   21
         BNZ r2,−5
```


Codage RISC: exemple

- instructions sur 32 bits
- 32 registres entiers
- 32 registres flottants

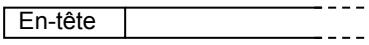


- load / store (basé)
- rt = rs op const
- branchement
- saut registre



- rd = rs op rt
- load / store (indexé)

Codage CISC



- Instruction CISC de longueur variable
- Le décodage de l'instruction se fait séquentiellement, par morceaux
  - Décoder premier morceaux (= en-tête)
  - Selon la valeur du premier morceaux, décoder deuxième morceau
  - Etc…

CISC plus compact que RISC

Une instruction complexe équivaut à plusieurs instructions simples

Exemple: BEQ r1,r2,DEP — Saute à PC+DEP si r1=r2

Équivalent RISC:

r3 ← r1 SUB r2 — Met r1-r2 dans r3

BZ r3,DEP — Saute à PC+DEP si r3=0

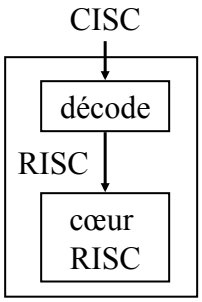
- « r3 » et « SUB » sont de l'information non essentielle
- On aurait pu mettre « r4 » et « XOR » à la place

RISC ou CISC ?

- CISC
  - Code plus compact, prend moins de place en mémoire
  - Jeu d'instructions plus facile à faire évoluer
    - Exemple x86: 16→32→64 bits, MMX→SSE→SSE2→SSE3→etc…
- RISC
  - Microarchitecture plus simple

CISC et micro-opérations RISC

Les microarchitectures x86 décomposent les instructions x86 en micro-opérations RISC



- Certaines opérations CISC complexes sont émulées avec un nombre élevé (variable) de micro-opérations
  - Fonctions transcendantes
    - cos, sin, exp, log, etc…
  - Fonctions de manipulation de chaînes
  - Etc…

Exécuter une instruction RISC: étapes

- PC stocké dans un flip-flop
- Lire l'instruction stockée en mémoire à l'adresse PC
- Décoder l'instruction
  - Extraire du code de l'instruction les différentes informations (type d'opération , registres accédés)
- Lire les registres
- Exécuter l'opération
- Écrire le résultat dans les registres
- Envoyer un « coup » d'horloge pour passer au PC suivant

La technique du pipeline

53

Exécuter une instruction RISC: étapes

- PC stocké dans un flip-flop
- Lire l'instruction stockée en mémoire à l'adresse PC
- Décoder l'instruction
  - Extraire du code de l'instruction les différentes informations (type d'opération , registres accédés)
- Lire les registres
- Exécuter l'opération
- Écrire le résultat dans les registres
- Envoyer un « coup » d'horloge pour passer au PC suivant

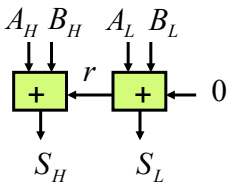
54

On veut une exécution performante

- Technique du pipeline: pour accélérer l'exécution, on n'attend pas qu'une instruction soit terminée pour commencer à exécuter l'instruction suivante
  - travail à la chaîne
  - à un instant donné, plusieurs instructions sont en cours d'exécution
- Permet de cadencer l'exécution avec une fréquence d'horloge plus élevée

La technique du pipeline: principe

Exemple: additionneur 64-bit constitué de 2 additionneurs 32-bit
➔ on veut calculer A+B, puis A'+B' puis A''+B'', etc…

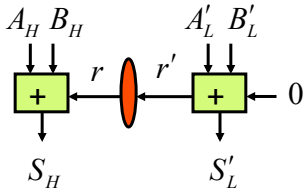


T = temps de réponse d'un additionneur 32 bits

Durée d'une addition 64 bits = 2T

Pour effectuer 10 additions 64 bits avec le même additionneur ➔ 20 T

Additionneur 32 bits inutilisé 50% du temps



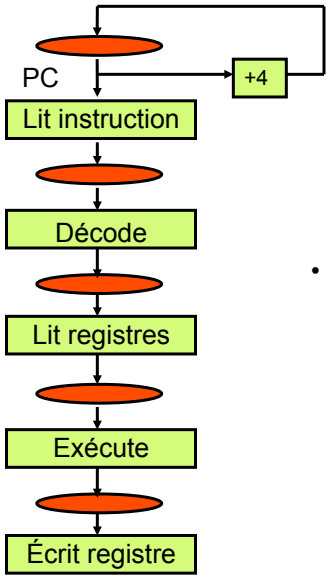
Pendant qu'on calcule les bits de poids forts de A+B, on calcule les bits de poids faible de A'+B', ainsi de suite…

Temps pour effectuer les 10 additions ➔ 11 T

Débit / latence

- La latence d'une tâche est le temps entre le début et la fin de la tâche
  - Exemples: Latence d'une instruction, latence d'une addition, latence d'un accès mémoire, etc…
- Le débit est le nombre de tâches exécutées pendant un temps fixe
  - Exemples: débit de 1 milliard d'instructions par seconde, débit de 1 accès mémoire par cycle d'horloge, etc…
- La technique du pipeline permet d'augmenter le débit
  - Elle augmente un peu la latence ➔ temps de « traversée » des flip-flops

Le pipeline d'instructions



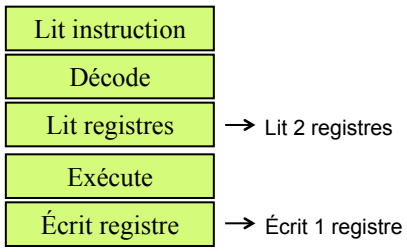
- À chaque cycle (=« coup ») d'horloge, une nouvelle instruction peut rentrer dans le pipeline

Une instruction par cycle (en théorie)

	cycle N	cycle N+1	cycle N+2 ….
Lit instruction	I5	I6	I7
Décode	I4	I5	I6
Lit registres	I3	I4	I5
Exécute	I2	I3	I4
Écrit registre	I1	I2	I3

Mais en pratique, il y a des complications …

Registres: besoins de plusieurs ports

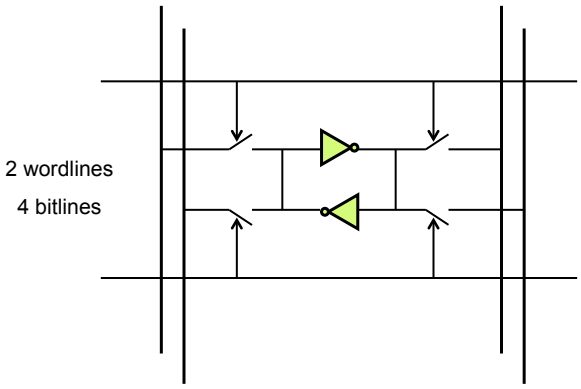


2 lectures et 1 écriture en même temps dans la mémoire de registre (« banc » de registres)

➔ On a besoin d'une SRAM avec 2 ports de lecture et un port d'écriture

SRAM multi-port

Exemple: SRAM double port



Attention, on ne peut pas écrire simultanément dans la même case (pas de problème si un seul port d'écriture)

La surface de la SRAM augmente approximativement comme le carré du nombre de ports

Types d'adressage / nombre de ports

- Adressage basé
  - r2 ← LOAD r1+DEP ➔ 1 port lecture, 1 port écriture
  - STORE r2,r1+DEP ➔ 2 ports lecture
- Adressage indexé (Sparc, PowerPC)
  - r3 ← LOAD r1+r2 ➔ 2 ports lecture, 1 port écriture
  - STORE r3,r1+r2 ➔ 3 ports lecture

Aléa structurel

- = conflit de ressource
- Registres ➔ pas d'aléa structurel si nombre de ports suffisant
- Mémoire: lecture instruction potentiellement en conflit avec LOAD/STORE
  - Solution 1: mémoire multi port
  - Solution 2: donner priorité aux LOAD/STORE et bloquer le pipeline pendant la durée d'un accès ➔ baisse de performance
  - En fait, aucune de ces 2 solutions n'est satisfaisante, on verra plus loin dans le cours comment ce problème est résolu en réalité

Aléas de contrôle: le problème des sauts

	Cycle N	N+1	N+2	N+3	N+4 ….
Lit instruction	I4	I35	I36	I37	I38
Décode	I3	bulle	I35	I36	I37
Lit registres	I2	bulle	bulle	I35	I36
Exécute	I1: jump @I35	bulle	bulle	bulle	I35
Écrit registre		I1	bulle	bulle	bulle

Cycle N: le saut est exécuté ➔ I2,I3,I4 sont « effacées » du pipeline et remplacées par des « bulles » ➔ 3 cycles perdus

Problème: si 1 saut toutes les 10 instructions ➔ 10+3 cycles pour 10 instructions ➔ débit = 10 / 13 = 0.77 instructions par cycle

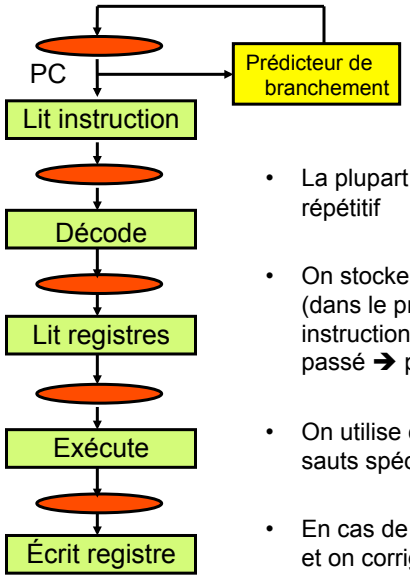
Aléas de contrôle: solutions

- Branchement différé (MIPS, Sparc)
  - le saut ne se fait pas tout de suite, mais avec un <u>délai</u>
  - exemple: le programme exécute I1, puis I2, puis I5
    - C'est comme si I2 était placée avant I1 dans le programme
  - permet d'éviter la bulle quand I2 est une instruction utile (si on ne trouve pas d'instruction utile, I2 = NOP = bulle)
  - Plus vraiment d'actualité

```
I1: jump @I5
I2:
I3:
I4:
I5:
```

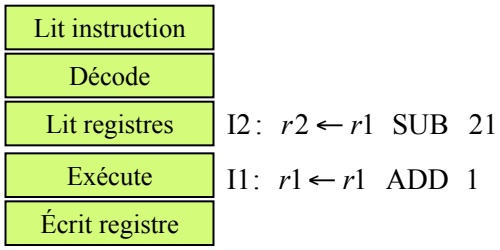
- <u>Prédiction de branchement</u>
  - solution utilisée dans les processeurs modernes
  - cf. plus loin dans le cours

Prédicteur de branchements



- La plupart des programmes ont un comportement répétitif
- On stocke dans une mémoire microarchitecturale (dans le processeur) des informations sur les instructions de contrôle et leur comportement passé ➔ prédicteur de branchements
- On utilise ces informations pour effectuer les sauts spéculativement
- En cas de mauvaise prédiction, on vide le pipeline et on corrige le PC ➔ aléa de contrôle

Dépendances entre instructions



- L'instruction I2 utilise la valeur produite par l'instruction I1
- Problème: quand I2 lit le registre r1, I1 n'a pas encore écrit dedans !

3 types de dépendance

– dépendance *Read After Write* (RAW)
  • le résultat de I1 est utilisé par I2
  • aussi appelée dépendance vraie

```
I1: r1 ← r1 ADD 1
I2: r2 ← LOAD r1+DEP
```

– dépendance *Write After Read* (WAR)
  • I2 écrit dans un registre lu par I1
  • aussi appelée anti-dépendance

```
I1: r2 ← LOAD r1+DEP
I2: r1 ← r1 ADD 1
```

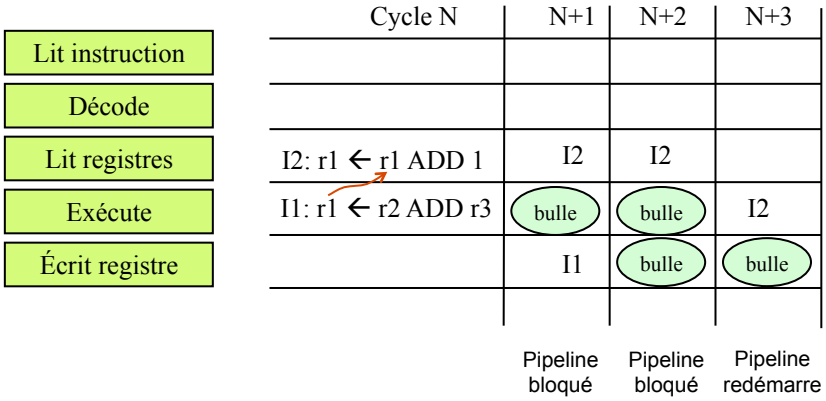
– dépendance *Write After Write* (WAW)
  • I2 écrit dans le même registre que I1
  • aussi appelé dépendance de sortie

```
I1: r2 ← LOAD r1+DEP
    BZ r5, -10
I2: r2 ← LOAD r3+DEP
```


Aléas de dépendance

- Pour respecter les dépendances entre instructions, on introduit des bulles dans le pipeline
- Aléas de dépendance
  – RAW
  – WAW: pas de problème si les instructions passent par l'étage d'écriture dans l'ordre du programme
  – WAR : pas de problème si on lit les registres dans l'ordre du programme et toujours au même étage du pipeline
- Les aléas de dépendance, comme les aléas structurels et les aléas de contrôle, diminuent la performance

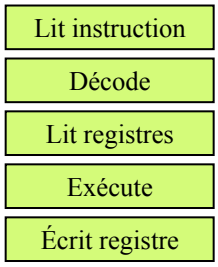
Aléa *read after write*



On attend 2 cycles avant de lancer I2 ➔ 2 cycles perdus

Exemple: exp(x)

```
                r33 ← x
                r34 ← 1
                r35 ← 1
                 r1 ← 1
Boucle:  r36 ← r33  FDIV  r1
         r34 ← r34  FMUL  r36
         r35 ← r35  FADD  r34
          r1 ← r1   ADD   1
          r2 ← r1   SUB   21
         BNZ  r2,–5
```

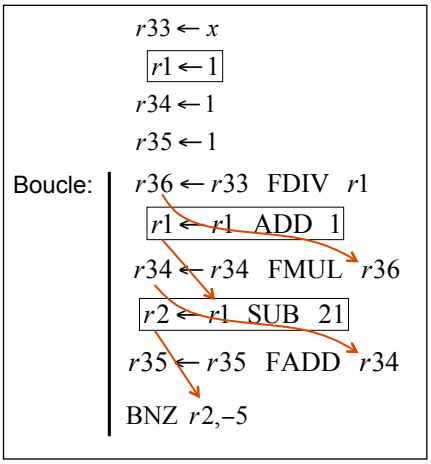


Supposons un prédicteur de branchements parfait.

Les dépendances RAW séparées par plus de 2 instructions ne coûtent rien ici

Temps d'exécution = 5 bulles (1ère instruction traverse pipeline) + 4 instructions + 2 bulles (RAW r1) + 20 x [6 instructions + 4 x 2 bulles (RAW r36,r34,r1,r2)] = 11 + 20 x 14 = 291 cycles ➔ débit réel = 124 / 291 = 0.43 instructions par cycle

Changer l'ordre des instructions



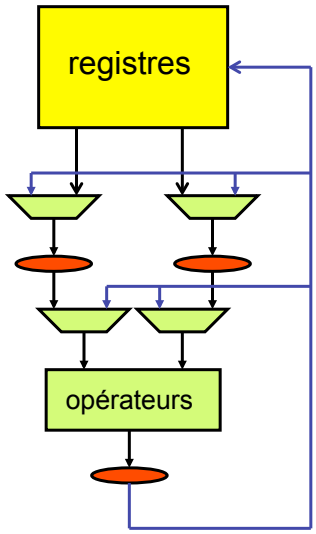
En changeant l'ordre des instructions, on peut enlever des bulles

Ici, il n'y a plus que 2 bulles par itération au lieu de 8

Temps d'exécution = 5 bulles + 4 instructions + 20 x [6 instructions + 2 bulles] = 169 cycles

Accélération = 291 / 169 = 1.72
➔ 72% de gain en performance

Mécanisme de *bypass*



Permet d'utiliser le résultat d'une instruction sans attendre qu'il soit écrit dans les registres

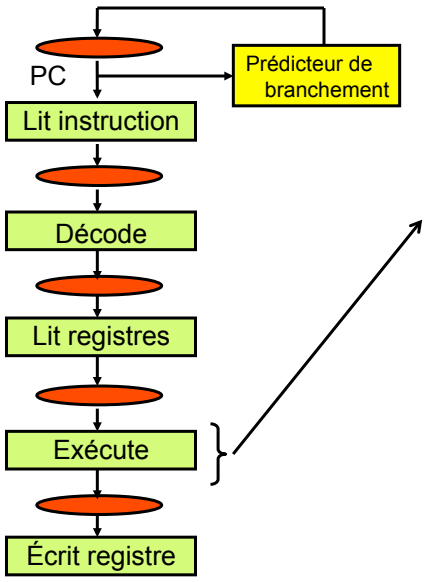
➔ Deux instructions avec une dépendance RAW peuvent s'exécuter dans des cycles consécutifs

Les MUX sont commandés par des comparaisons sur les numéros de registre

Instructions de durées différentes (1)

- Certaines instructions sont plus longues à exécuter que d'autres
  - load / store
    - Temps d'accès à la mémoire ?
  - division et multiplication entière
    - Plus complexe qu'une addition
  - opérations virgule flottante
    - Normalisation des exposants
  - …

Instructions de durées différentes (2)



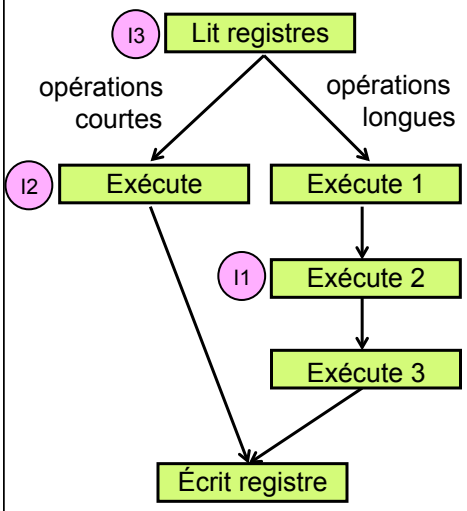
C'est l'opérateur le plus lent qui dicte la fréquence d'horloge ?

☹

Solution: pipeliner les opérateurs

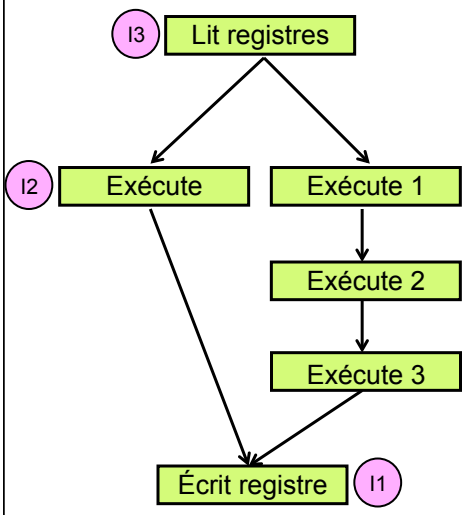
Mais attention …

Pipeliner les opérateurs: exemple



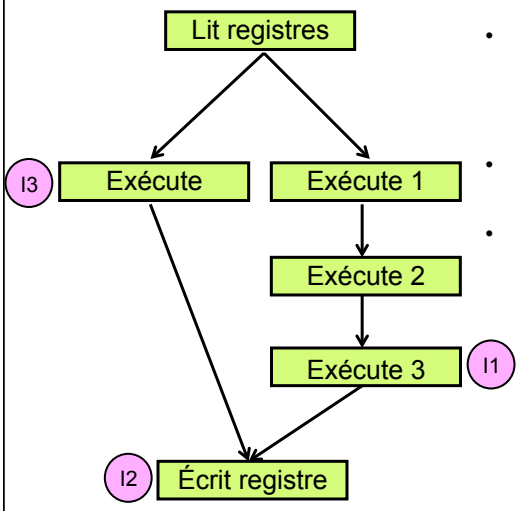
- Instruction longue I1 suivie de 2 instructions courtes I2 et I3
  - Cycle N: écriture I2
  - Cycle N+1: écriture I1 et I3
- I1 et I3 écrivent en même temps !
  - Aléa structurel si 1 seul port d'écriture
- I2 écrit avant I1
  - (Et si une interruption se produit ?)
- Si I1 et I2 écrivent le même registre ➔ aléa WAW !

Solution 1: attendre



- Attendre 2 cycles avant d'exécuter I2
- ➔ les instructions se terminent dans l'ordre du programme (ordre « séquentiel »)
  - Pas de conflit d'écriture
  - Pas d'aléa WAW
- ➔ On perd 2 cycles à chaque fois qu'une instruction courte suit une instruction longue

Solution 2: port d'écriture en plus



- Si I1 et I2 n'écrivent pas dans le même registre, autoriser I2 à écrire avant I1
  - Sinon, attendre
- I3 et I1 écrivent en même temps
- Les interruptions sont « imprécises »

Interruptions / « exceptions »

- Parfois, on veut interrompre l'exécution et pouvoir la relancer ultérieurement à partir du point d'interruption
  - Exécution en temps partagé
    - Exemple: toutes les 10 millisecondes, le timer envoie un signal d'interruption pour redonner le processeur au système d'exploitation, qui peut décider de donner momentanément le processeur à une autre tâche
    - L'état architectural (contexte) de la tâche interrompue (dont les valeurs des registres architecturaux) est sauvegardé en mémoire ou sur disque
    - Ce contexte sera utilisé pour relancer la tâche interrompue
  - Émulation en logiciel de certaines instructions
    - Par exemple pour la compatibilité binaire
  - Debugging
  - Gestion de la mémoire
    - Quand la mémoire physique est trop petite par rapport aux besoins en mémoire virtuelle, on « simule » une plus grande mémoire en utilisant le disque
    - Voir plus loin dans le cours …

Interruptions précises

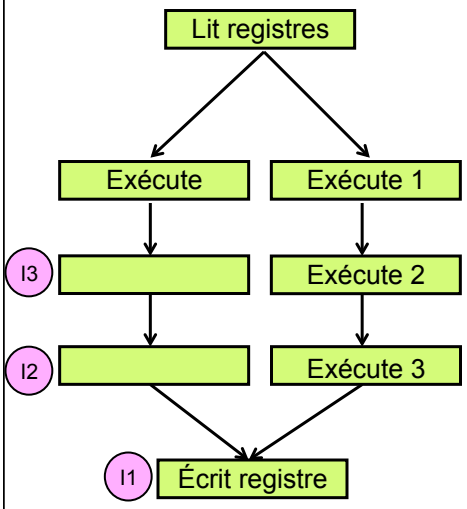
- Une interruption est dite « précise » si l'état des registres et de la mémoire au moment de l'interruption est un état architectural ➔ état résultant d'une exécution strictement séquentielle

I1: r1 ← COS r2
I2: r3 ← r3 ADD 1

→ Exemple: instruction peut être émulée en logiciel

- Si, au moment où on sauvegarde l'état des registres, I2 a déjà écrit son résultat, l'interruption est imprécise
  - Lorsqu'on reprendra l'exécution, la valeur dans r3 ne sera pas la bonne
  - Solution: mémoriser dans le contexte le fait que I2 est déjà exécutée, et ne pas re-exécuter I2 à la reprise
- Les interruptions précises sont préférables

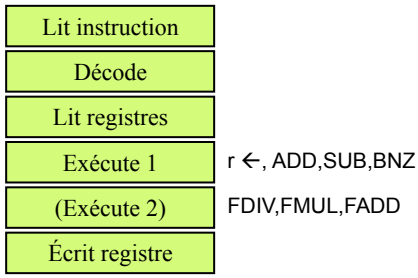
Solution 3: étages vides



- Rajouter des étages « vides » aux opérations courtes pour retarder l'écriture
  - Étage vide = flip-flop
  - Transmet l'instruction et son résultat
- ➔ Les écritures se font dans l'ordre
  - Interruptions précises
- Mécanisme de bypass plus complexe

Exemple: exp(x)

```
        r33 ← x
        r34 ← 1
        r35 ← 1
        r1 ← 1
Boucle: r36 ← r33  FDIV  r1
        r34 ← r34  FMUL  r36
        r35 ← r35  FADD  r34
        r1 ← r1  ADD  1
        r2 ← r1  SUB  21
        BNZ  r2,–5
```



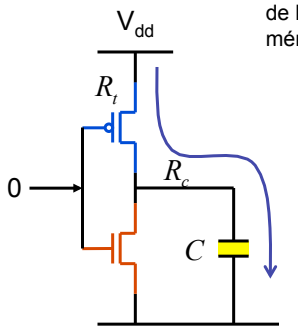
- Prédicteur de branchements parfait
- Mécanisme de bypass
- On attend 1 cycle (bulle) lorsqu'on a besoin du résultat d'une instruction longue

Temps d'exécution = 6 bulles (1ère instruction traverse pipeline) + 4 instructions + 20 x [6 instructions + 2 bulles] = 10 + 20 x 8 = 170 cycles ➔ débit réel = 124 / 170 = 0.73 instructions par cycle

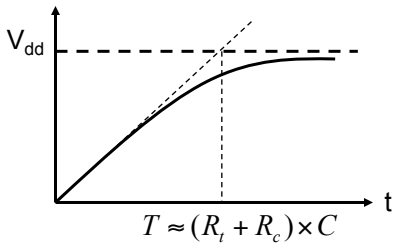
Accès mémoire

- En 2008:
  - Latence d'un additionneur entier 64 bits ➔ ~ 0.5 nanosecondes
    - = 1 cycle d'horloge
    - Dicte la fréquence d'horloge $F = \frac{1}{0.5 \times 10^{-9}} = 2 \times 10^{9}$ herz
  - Latence d'accès d'une mémoire DRAM de 1 Go ➔ ~ 100 ns = 200 cycles !
  - (Latence exact dépend de plusieurs facteurs, dont la taille mémoire)
- Problème !
  - Si on doit bloquer l'exécution pendant 200 cycles à chaque accès mémoire, la performance est très faible

D'où vient la latence ?



Pour les connexions longues de plusieurs centimètres, vitesse de la lumière à un impact. Sur une puce (processeur ou mémoire), effets capacitifs essentiellement.



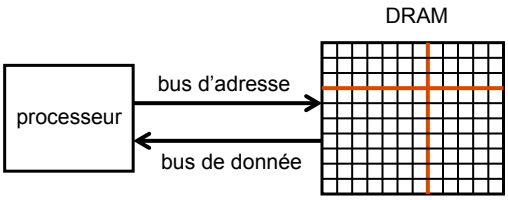
La capacité électrique C et la résistance R_c d'une connexion sont proportionnelles à sa longueur ➔ R_cC augmente comme le carré de la longueur

➔ solution: insérer des portes NOT à intervalles réguliers

➔ délai proportionnel à la longueur

$$T = n \times \left(R_t + \frac{R_c}{n} \right) \times \frac{C}{n} \approx R_t C$$

Latence d'une lecture mémoire



1. L'adresse sort de la puce processeur par le brochage et est envoyée sur le bus d'adresse
2. On active la wordline et on sélectionne les bitlines correspondant à la donnée demandée
3. La donnée est envoyée au processeur par le bus de donnée

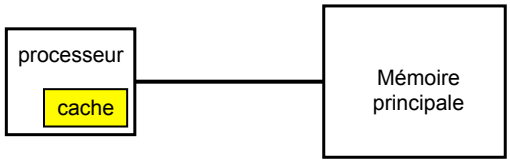
- Une grande mémoire a des wordlines et bit lines longues ➔ grand temps d'accès
- S'il y a plusieurs bancs DRAM, il faut des bus plus longs et il faut sélectionner le banc correspondant à la donnée demandée ➔ latence supplémentaire

Réduire la latence mémoire ?

- Prendre une DRAM de plus petite capacité (en octets) ?
  - Pas vraiment ce qu'on souhaite
  - Il faudrait réduire la taille mémoire considérablement pour diminuer la latence DRAM ne manière significative
    - Latence DRAM augmente avec taille, mais pas linéairement
  - La DRAM n'est pas la seule à contribuer à la latence mémoire
- Mettre la mémoire directement sur la puce ?
  - La Loi de Moore semble le permettre, mais les besoins en mémoire augmentent parallèlement à la loi de Moore
  - ➔ en 2008, la mémoire principale est encore trop grande pour tenir sur la puce processeur
    - Circuits 3D ? Recherches prometteuses mais technologie pas encore mature

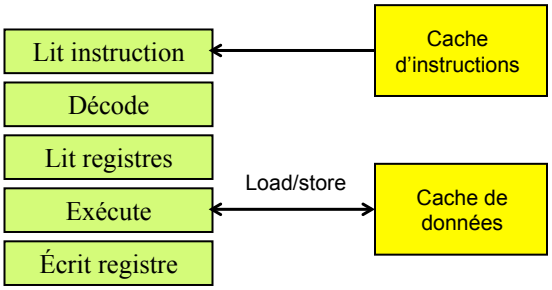
Le principe du cache

- Principe de localité temporelle
  - La plupart des programmes ont tendance à réaccéder des instructions et des données qu'ils ont accédées récemment
  - Exemple: une boucle
- Cache = petite mémoire située sur la puce processeur
  - Exemple: 64 ko
- On met dans le cache les données et instructions accédées récemment
  - Si la donnée/instruction demandée est absente du cache (défaut de cache, ou *miss*), elle est lue en mémoire principale et stockée automatiquement dans le cache
  - Technique microarchitectural transparente pour le programmeur et le compilateur



Cache d'instructions, cache de données

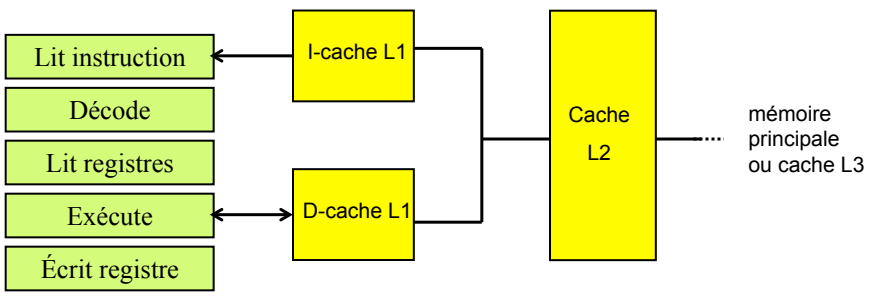
- On peut mettre les instructions et les données dans des caches séparés
- Avantage: pas besoin d'avoir 2 ports sur le cache, pas de conflit d'accès



En cas de miss, bloquer le signal d'horloge jusqu'à la résolution du miss

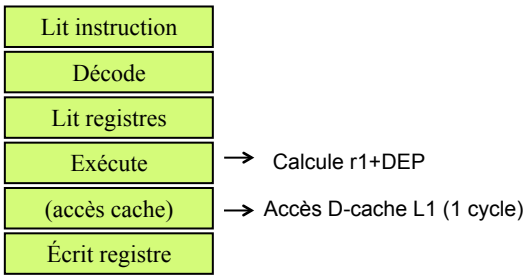
Plusieurs niveaux de cache

- La latence d'un cache augmente avec sa taille
- Plusieurs niveaux de cache (« hiérarchie mémoire »)
  - Niveau 1 (L1): caches petits et rapide d'accès
    - Exemple: 32 Ko, latence 2 cycles
  - Niveau 2 (L2): cache plus gros et plus lent
    - Exemple: 1 Mo, latence 10 cycles
  - Etc…

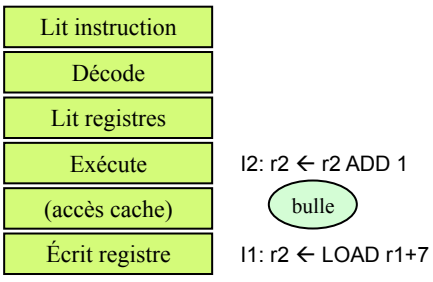


Exécution des load/store

- r2 ← LOAD r1+DEP
- STORE r2,r1+DEP



Pénalité *load-use*



Sur cet exemple, I2 ne peut s'exécuter que lorsque I1 a fini l'accès cache

➔ 1 bulle (pénalité load-use = 1 cycle)

Si le temps d'accès cache était de 2 cycles, il faudrait insérer 2 bulles (pénalité load-use = 2 cycles), etc…

Exemple (1)

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

- Prédicteur de branchements parfait
- Programme dans le cache d'instructions
- Mécanisme de bypass
- Pénalité load-use = 2 cycles
- Données déjà dans le cache L1

```
r1 ← 100
r2 ← a
r3 ← 0
r4 ← LOAD r2
r2 ← r2 ADD 4
r1 ← r1 SUB 1
r3 ← r3 ADD r4
BNZ r1,-4
```

2 instructions indépendantes du load suffisent pour masquer la latence load-use, pas de pénalité

➔ Temps d'exécution: 7 + 3 + 100 x 5 = 510 cycles

Exemple (2)

```
int a[100];
x = 0;
for (i=0; i<100; i+=2)
    x = x + a[i] + a[i+1];
```

- Prédicteur de branchements parfait
- Programme dans le cache d'instructions
- Mécanisme de bypass
- Pénalité load-use = 3 cycles
- Données déjà dans le cache L1

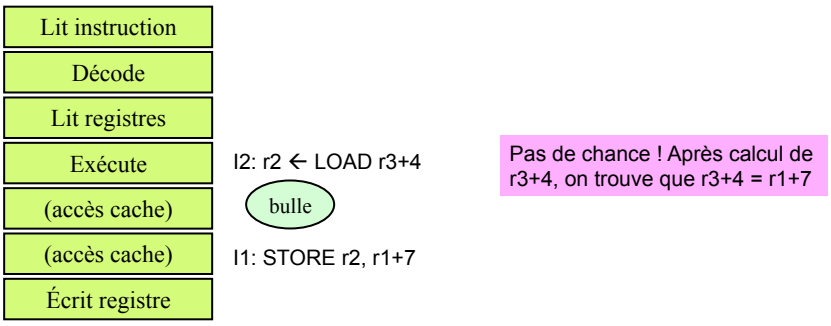
```
r1 ← 100
r2 ← a
r3 ← 0
r4 ← LOAD r2
r5 ← LOAD r2+4
r2 ← r2 ADD 8
r1 ← r1 SUB 2
r3 ← r3 ADD r4
r3 ← r3 ADD r5
BNZ r1,-6
```

Temps d'exécution = 8 + 3 + 50 x 7 = 361 cycles

Les compilateurs optimisés font cela automatiquement sans l'aide du programmeur (remarque: on a utilisé la loi d'associativité pour l'addition sur les entiers. Sur des flottants, cela changerait la sémantique du programme)

Dépendances mémoire

Exemple: latence cache = 2 cycles



Pas de chance ! Après calcul de r3+4, on trouve que r3+4 = r1+7

Opérateurs « flottants »

- L'exécution des opérations en virgule flottante prend plusieurs cycles
- Exemple: Intel Core
  - FADD: 6 cycles, complètement pipeliné (1 FADD/cycle possible)
  - FMUL: 8 cycles, complètement pipeliné
  - FDIV: 40 cycles, non pipeliné (1 FDIV tous les 40 cycles)
- Quand on écrit des programmes performants, on essaie d'éviter les divisions si possible

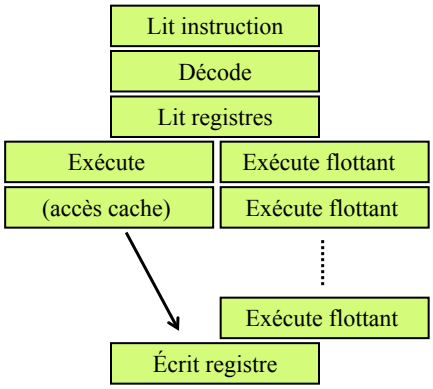
$$z = \frac{1 + \frac{1}{x}}{y} \quad \Rightarrow \quad z = \frac{x+1}{x \times y}$$

1 addition, 2 divisions

On a remplacé une division par une multiplication

Problème

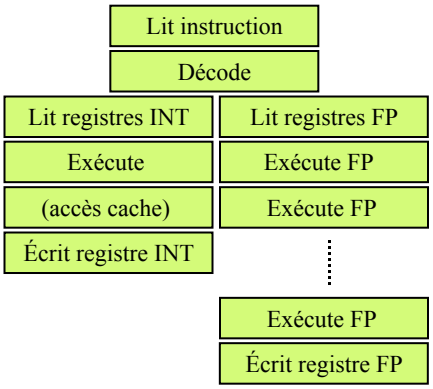
- Si on veut pipeliner l'addition et la multiplication flottante tout en maintenant l'ordre séquentiel des écritures registre, il faut rajouter beaucoup d'étages vides pour les opérations courtes, ce qui rend le mécanisme de bypass complexe
- Revenons un peu en arrière …



- Interruptions imprécises
- Besoin de 2 ports d'écriture
- Aléa WAW peut se produire si LOAD dans registre FP ➔ attente

Pipeline entier, pipeline flottant

- Deux SRAMs au lieu d'une: une SRAM pour les registres entiers (INT=*integer*), une pour les registres flottants (FP=*floating-point*)



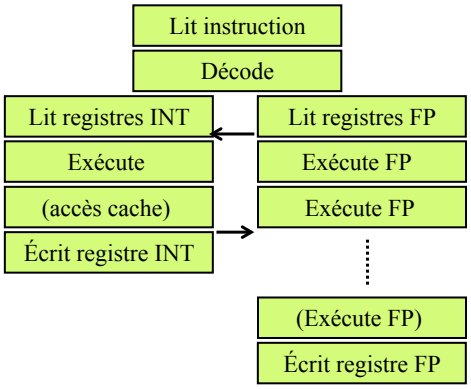
- SRAMs avec 1 seul port d'écriture
- Rappel: complexité d'une SRAM multi-port augmente comme le carré du nombre de ports
  - Exemple: 32 INT, 32 FP

$$\frac{2\times 32\times (2+1)^2}{64\times (2+2)^2} = \frac{9}{16}$$

- ➔ moins complexe qu'une SRAM à 2 ports d'écriture

Load/store flottant

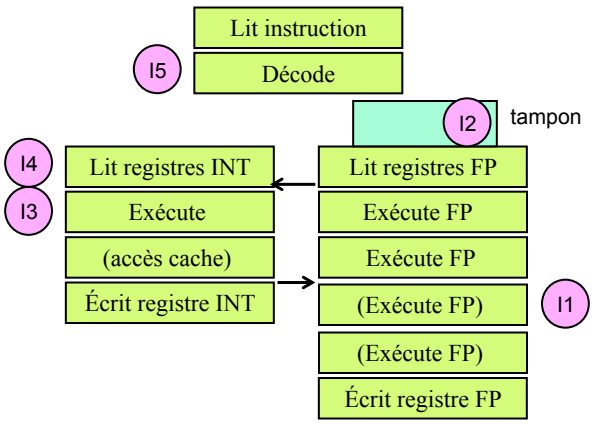
- Problème:
  - Load/store flottant accède à la fois aux registres INT (calcul d'adresse) et aux registres FP (valeur lue ou écrite)



- Envoyer copies de l'instruction dans les 2 pipelines simultanément
- f1 ← LOAD i2+10
  - Quand l'accès cache est terminé, envoyer valeur au pipeline FP
  - En général, augmente la pénalité load-use FP de 1 cycle (voire plus)
- STORE f1,i2+10
  - Quand lecture FP terminée, envoyer valeur FP au pipeline INT
- Pas d'aléa WAW, mais interruptions imprécises

Exécution dans le désordre

- Exemple: I2 dépend de I1, mais I3,I4 ne dépendent ni de I1 ni de I2
- I2 doit attendre que I1 se termine (RAW), mais pourquoi I3,I4 devraient elles attendre ?
- On pourrait mettre I2 dans une mémoire tampon et continuer le lancement d'instructions



Load/store flottant doit rester bloqué au décode tant que tampon FP pas vide

Le calcul entier prend de l'avance sur le calcul flottant tant que le tampon n'est pas plein

```
double y[100] ;
double x = 0 ;
for (i=0; i<100; i++)
      x = x + y[i] ;
```

```
f2 ← LOAD i1
f1 ← f1 FADD f2
i1 ← i1 ADD 8
i2 ← i2 SUB 1
BNZ i2,-4
```

On suppose ici 1 étage d'accès cache et FADD pipeliné sur 4 étages

	cycle décode	cycle lecture reg	cycle résultat	
f2 ← LOAD i1	1	2	4	
f1 ← f1 FADD f2	2	4	8	
i1 ← i1 ADD 8	3	4	5	
i2 ← i2 SUB 1	4	5	6	
BNZ i2,-4	5	6		
f2 ← LOAD i1	6	7	9	
f1 ← f1 FADD f2	7	9	13	
i1 ← i1 ADD 8	8	9	10	
i2 ← i2 SUB 1	9	10	11	
BNZ i2,-4	10	11		
f2 ← LOAD i1	11	12	14	
f1 ← f1 FADD f2	12	14	18	
i1 ← i1 ADD 8	13	14	15	
i2 ← i2 SUB 1	14	15	16	
BNZ i2,-4	15	16		

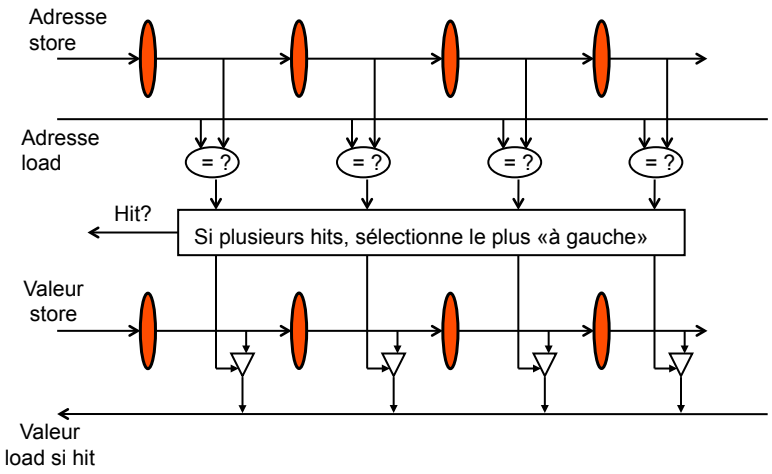
Dépendances registre

- WAW,WAR
  - pas de problème
  - load/store flottant bloqué au décode tant que tampon FP non vide
- RAW:
  - SRAM spéciale où sont stockés 1 *bit de présence* pour chaque registre
  - Quand une instruction I1 est lancée dans l'étage de lecture registre, met le bit du registre destination à 0
  - Une instruction I2 utilisant le résultat de I1 doit attendre tant que le bit de présence est nul.
  - Dès que le résultat de I1 est connu et accessible via le bypass, met le bit à 1, ce qui autorise I2 à rentrer dans l'étage de lecture registre

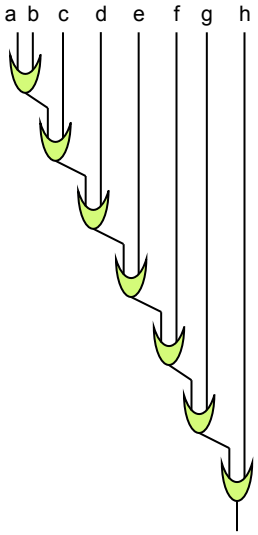
Dépendances mémoire

- File de STORE = petite mémoire « associative »
  - Mémoire associative (*content addressable memory*, CAM) ➔ mémoire contenant des comparateurs
- Mettre le STORE (adresse calculée et valeur) dans la file pendant la durée de l'écriture cache
  - Nombre d'entrées de la file augmente avec la latence du cache
- Quand écriture cache terminée, retirer le STORE de la file
- Quand adresse LOAD calculée, comparer l'adresse avec toutes les adresses dans la file
  - Comparaisons faites en parallèle
- Si il existe un STORE à la même adresse (*hit*), le LOAD prend la valeur directement dans la file au lieu d'accéder au cache (mécanisme de bypass mémoire)
- Si plusieurs STORE à la même adresse, prendre le plus récent

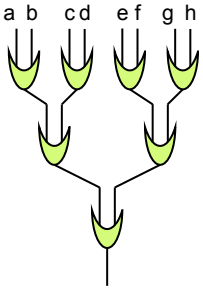
Exemple: file de STORE à 4 entrées



Petite parenthèse …



Mieux !

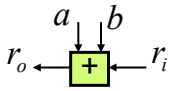


L'opération OR (comme AND et XOR) est une opération associative

$$(a \vee b) \vee c = a \vee (b \vee c)$$

…fin de la parenthèse

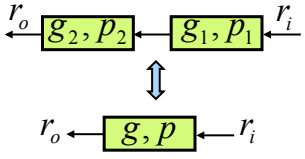
- Quand la sortie d'un circuit peut être exprimée sous la forme a*b*c*d* … etc., où «* » est une opération associative, alors il y a de bonnes chances qu'il existe une mise en œuvre matérielle efficace de ce circuit
- Exemples:
  - Détection de hit ➔ utilise des OR
  - Comparateur d'adresses ➔ utilise des XOR et des OR
  - Détection du hit le plus « à gauche » ➔ OR (« y a-t-il un hit plus à gauche ? »)
  - Retenue d'un additionneur ➔ voir ci-dessous



$$r_o = g \vee (p \wedge r_i)$$

$g = a \wedge b$ ➔ génère retenue

$p = a \oplus b$ ➔ propage retenue



$$\begin{pmatrix} g \\ p \end{pmatrix} = \begin{pmatrix} g_2 \\ p_2 \end{pmatrix} \otimes \begin{pmatrix} g_1 \\ p_1 \end{pmatrix} = \begin{pmatrix} g_2 \vee (g_1 \wedge p_2) \\ p_2 \wedge p_1 \end{pmatrix}$$

L'opération $\otimes$ est associative

Les caches

Rappel: localité temporelle

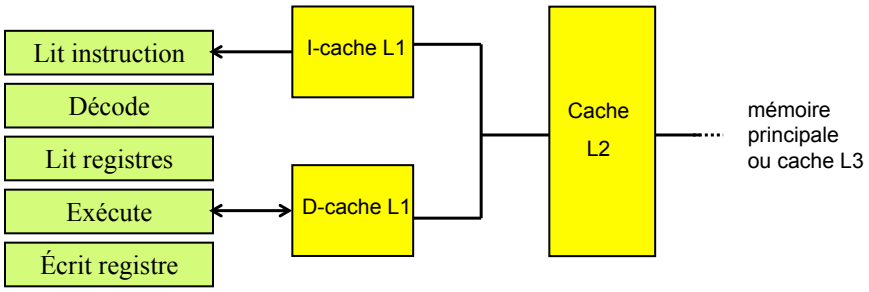
- La plupart des programmes ont tendance à accéder des instructions et des données qu'ils ont accédées récemment

Exemple: produit de matrice $c_{ij} = \sum_{k=0}^{n} a_{ik} b_{kj}$

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)  {
      c[i][j] = 0 ;
      for (k=0; k<n; k++)
         c[i][j] += a[i][k] * b[k][j] ;
   }
```

- Localité des instructions
  - Intérieur boucle *for i* → n fois
  - Intérieur boucle *for j* → n^2 fois
  - Intérieur boucle *for k* → n^3 fois
- Localité des données
  - Chaque a[i][j] et b[i][j] est utilisé n fois

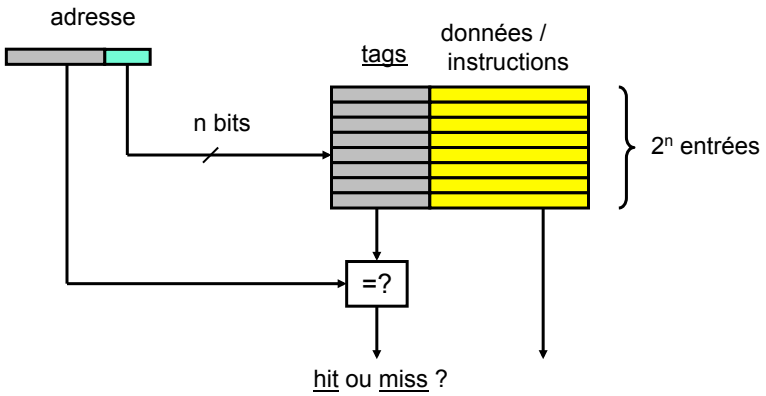
Rappel: hiérarchie mémoire



- Ordres de grandeur:
  - Fréquence processeur 2 GHz ➔ cycle d'horloge = 0.5 ns
  - Cache L1 de 64 Ko ➔ latence = 2 cycles (2 étages d'accès cache)
  - Cache L2 de 2 Mo ➔ latence = 10-20 cycles
  - Mémoire 2 Go ➔ 100-200 cycles

Cache *direct-mapped*

- C'est la structure de cache la plus simple
- Sélectionner l'entrée avec les bits de poids faible de l'adresse
- Comparer les bits de poids fort avec le tag stocké dans l'entrée



Lignes de cache

- Principe de localité spatiale
  - La plupart des programmes ont tendance à accéder dans des temps rapprochés des données ou instructions situées à des adresses proches
- Au lieu de stocker dans une entrée du cache une seule donnée ou une seule instruction, on stocke une ligne
  - Ligne = nombre fixe d'octets consécutifs en mémoire
  - Taille typique: 64 octets

Exemple:

```
int x[N];
for (i=0; i<N; i++)
    s += x[i] ;
```

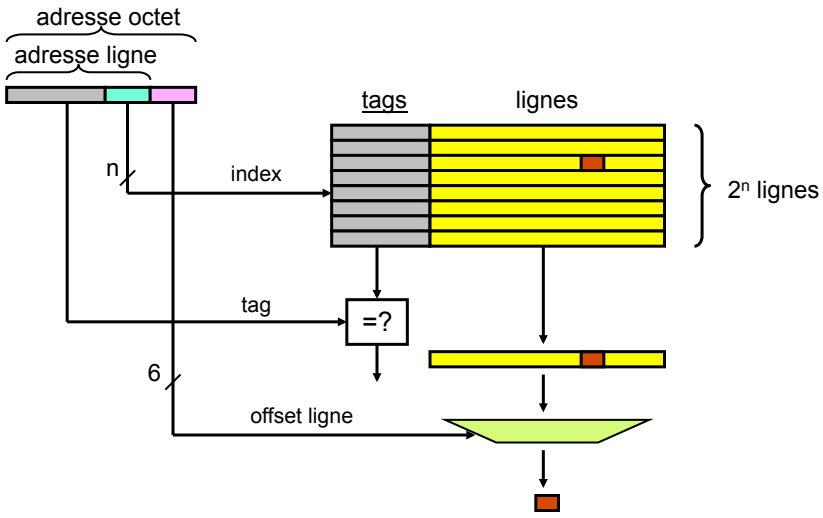
1 *int* ➔ 4 octets

1 ligne de 64 octets ➔ 16 *int*

1 miss toutes les 16 itérations

Exemple: ligne de 64 octets

Les 6 bits de poids faible de l'adresse sélectionnent la donnée (ou l'instruction) à l'intérieur de la ligne (utilise MUX)

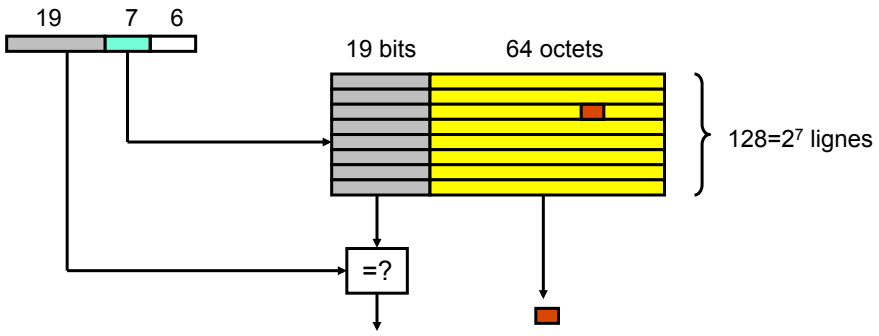


« taille » d'un cache

- Taille d'un cache ?
  - Surface de silicium (mm^2) ?
  - Capacité SRAM (octets) ?
  - Nombre d'instructions/données ?

- Exercice
  - Soit une architecture 32 bits
  - Soit un cache direct-mapped juste assez grand pour contenir un tableau de 1024 flottants double précision. Les lignes font 64 octets.
  - Quelle est la capacité (octets) de la SRAM où sont stockées les lignes ?
  - Quelle est la capacité (octets) de la SRAM où sont stockés les tags ?

- Réponse
  - Architecture 32 bits ➔ adresses 32 bits
  - 1 flottant DP ➔ 8 octets
  - 1 ligne 64 octets ➔ 8 flottants DP
  - 1024 flottants DP ➔ 128 lignes



  - ➔ 128 x 64 = 8 Ko de SRAM pour les lignes
  - ➔ 128 x 19 / 8 = 304 octets de SRAM pour les tags

On dit que le cache fait 8 Ko

Cache *write-through*

(cache à écriture transmise)

- Plusieurs types de caches selon la manière dont sont traitées les écritures
- Cache *write-through*
  - On envoie systématiquement l'ordre d'écriture (adresse et valeur) au niveau suivant de la hiérarchie mémoire (cache L2, cache L3, ou mémoire)
  - Si l'adresse d'écriture est présente dans le cache (hit en écriture), on met à jour la donnée dans le cache
  - Sur un miss en lecture, on va chercher la ligne manquante dans le niveau suivant (L2,L3,mémoire) et on écrit la ligne et son tag dans le cache
    - L'ancienne ligne à cette entrée peut être écrasée car il existe une copie à jour dans le niveau suivant
  - Sur un miss en écriture, 2 méthodes possibles
    - Allocation sur écriture ➔ on va chercher la ligne manquante, on la met dans le cache, et on fait comme pour un hit en écriture
    - Pas d'allocation sur écriture ➔ le cache est inchangé, on se contente de transmettre l' écriture au niveau suivant

Cache *write-back*

(cache à écriture différée)

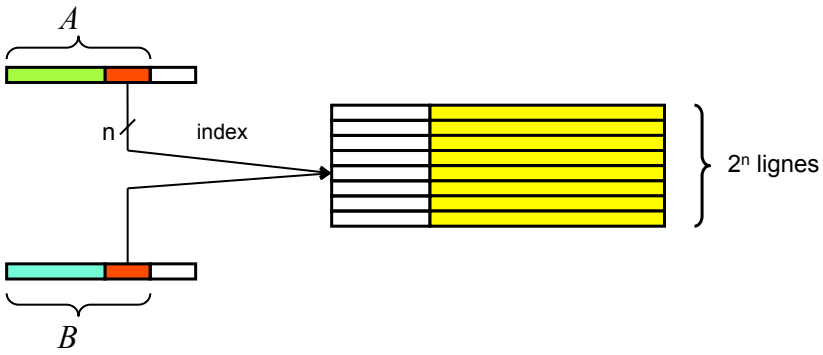
- On rajoute un bit *dirty* dans chaque entrée du cache
- Lorsqu'il vaut 1, ce bit signifie que la ligne a été modifiée
- Sur un hit en écriture, on met à jour la donnée dans le cache et on positionne le bit *dirty* de la ligne à 1
- Sur un miss en lecture, on va chercher la ligne manquante dans le niveau suivant et pendant ce temps on regarde si le bit dirty de la ligne qu'on va évincer vaut 1
  - Si le bit dirty est à 0, cela signifie que la ligne n'a pas été modifiée et donc que la ligne est à jour dans le niveau suivant ➔ on peut écraser la ligne
  - Si le bit dirty est à 1, il faut sauvegardée la ligne modifiée dans le niveau suivant avant de pouvoir la remplacer par la ligne manquante
- Sur un miss en écriture, on va chercher la ligne manquante, on la met dans le cache, puis on fait comme pour un hit en écriture

Classification des miss

- Pour avoir une bonne performance, on cherche à minimiser le nombre de miss
- 3 types de miss
  - Miss de démarrage ➔ première fois qu'on accède à une donnée ou à une instruction
    - On peut diminuer le nombre de miss de démarrage en prenant des lignes de cache plus longues
  - *Miss de capacité* ➔ le cache n'est pas assez grand pour contenir le programme et/ou les données de ce programme
    - On peut diminuer le nombre de miss de capacité en augmentant la taille du cache
  - Miss de conflit ➔ le cache est assez grand, mais certaines lignes « veulent » aller dans la même entrée du cache

Miss de conflit

2 lignes sont en conflit si elles ont le même index cache



Sur un cache direct-mapped, les lignes en conflit sont les lignes dont les adresses sont distantes d'un nombre entier de fois le nombre de lignes de cache

$$A \equiv B \pmod{2^n}$$

Exemple

- Soit un cache direct-mapped de 64 Ko dont les lignes font 64 octets
- Question:
  - les adresses d'octet 87436 et 218500 sont elles en conflit dans le cache ?
- Réponse:
  - Le cache contient 1024 lignes
  - D'abord on calcule les adresses de ligne
    - 87436 = 1366 x 64 + 12
    - 218500 = 3414 x 64 + 4
  - Puis on calcule la différence entre les adresses de ligne
    - 3414 - 1366 = 2048
  - C'est un multiple de 1024 ➔ ces 2 octets sont en conflit dans le cache

Exemple

- Cache direct-mapped de 32 Ko, lignes de 64 octets
  - ➔ 512 lignes
- On suppose des matrices de 20 x 20 flottants double précision chacune
  - ➔ 20 x 20 x 8 = 3200 octets par matrice, soit 50 ou 51 lignes de cache
  - A priori, il y a assez de place dans le cache pour les 3 matrices
- On suppose que les adresses des matrices sont
  - A = 70432 = 1100 x 64 + 32
  - B = 76448 = 1194 x 64 + 32
  - C = 106240 = 1660 x 64

```
for (i=0; i<20; i++)
   for (j=0; j<20; j++)  {
      C[i][j] = 0 ;
      for (k=0; k<20; k++)
         C[i][j] += A[i][k] * B[k][j] ;
   }
```

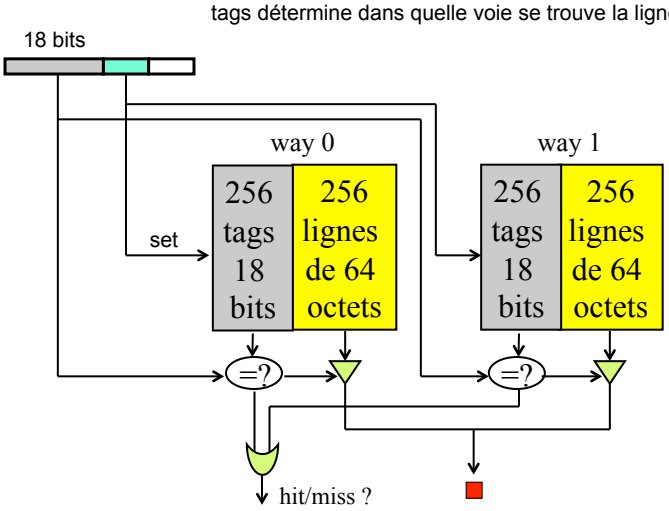
- Entrées du cache utilisées
  - 1100 ≡ 76 (mod 512) ➔ matrice A utilise les entrées 76 à 126 (51 entrées)
  - 1194 ≡ 170 (mod 512) ➔ matrice B utilise les entrées 170 à 220 (51 entrées)
  - 1660 ≡ 124 (mod 512) ➔ matrice C utilise les entrées 124 à 173 (50 entrées)
  - ➔ la matrice C est partiellement en conflit avec les matrices A et B

Comment éliminer les conflit ?

- Au niveau du programmeur
  - Propriété du cache direct-mapped: si les données (ou instructions) d'un programme sont située à l'intérieur d'une zone mémoire dont la taille n'excède pas la taille du cache, il n'y a pas de conflit dans le cache direct-mapped
    - (en fait, pas toujours vrai lorsque index physique, cf. plus loin)
  - ➔ essayer de placer les données qui sont accédées ensemble à des adresses mémoire proches
- Au niveau du matériel ➔ cache « set-associatif »
  - En anglais, *set-associative* ➔ associatif par ensembles
  - Au lieu d'avoir 1 seule entrée possible dans le cache, chaque ligne en a plusieurs
  - Cache N-way set-associatif (cache à N voies) ➔ N entrées possibles
  - Cache full-associatif ➔ N = nombre d'entrées du cache (toutes les entrées sont possibles)

Exemple: cache 2-way set-associatif

- cache 2-way SA de 32 Ko, lignes de 64 octets, adresses 32 bits
- les 2 parties du cache sont accédés en parallèle, la comparaison des tags détermine dans quelle voie se trouve la ligne demandée



Exemple

- Cache 2-way SA de 32 Ko, lignes de 64 octets
  - ➔ 512 lignes total, 256 lignes par way (= 256 sets)
- A,B,C matrices 20 x 20 x 8 octets
- Adresses matrices
  - A = 70432 = 1100 x 64 + 32
  - B = 76448 = 1194 x 64 + 32
  - C = 106240 = 1660 x 64

```
for (i=0; i<20; i++)
   for (j=0; j<20; j++)  {
      C[i][j] = 0 ;
      for (k=0; k<20; k++)
         C[i][j] += A[i][k] * B[k][j] ;
   }
```

- Entrées du cache utilisées
  - 1100 ≡ 76 (mod 256) ➔ matrice A utilise les sets 76 à 126
  - 1194 ≡ 170 (mod 256) ➔ matrice B utilise les sets 170 à 220
  - 1660 ≡ 124 (mod 256) ➔ matrice C utilise les sets 124 à 173
  - ➔ il n'y a pas de conflit car chaque set contient 2 entrées

Politique de remplacement

- Comme il y a plusieurs entrées possibles dans un cache set-associative, il faut en choisir une ➔ politique de remplacement

- Exemples de politiques de remplacement
  - RANDOM ➔ évince une ligne au hasard dans le set
  - LRU (*Least-Recently Used*) ➔ évince la ligne qui n'a pas été utilisée depuis le temps le plus long

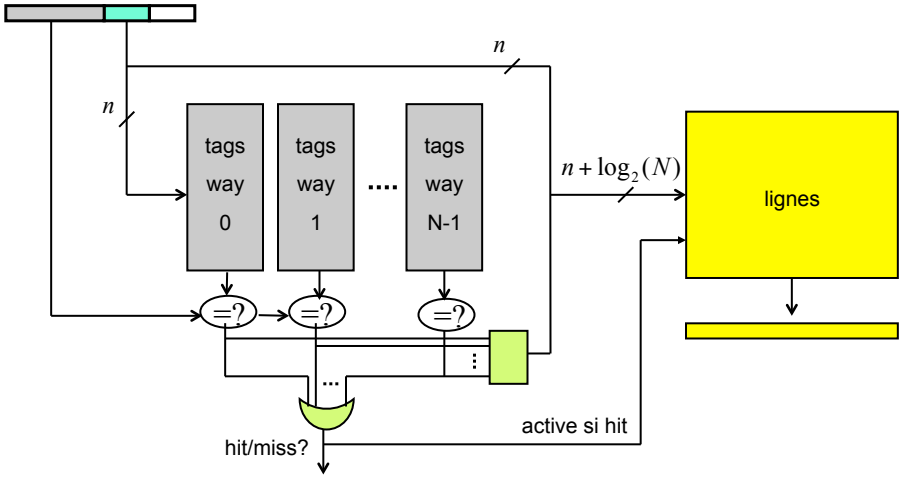
Politique LRU

- Politique assez efficace la plupart du temps
  - Si une ligne n'a pas été utilisée depuis longtemps, il est probable qu'elle ne sera pas réutilisée avant longtemps
    - Peut-être même qu'elle est « morte »
- Politique LRU facile à mettre en œuvre sur un cache 2-way SA
  - 1 bit MRU par set (MRU = *most recently used*)
  - Lorsqu'une ligne est accédée, mettre le bit à 0 si la ligne est dans le way 0, à 1 si la ligne est dans le way 1
  - Quand on doit choisir une ligne à évincer d'un set, on choisit le way 0 si le bit MRU vaut 1 et le way 1 si le bit MRU vaut 0
- Quand l'associativité est élevée, la mise en œuvre de LRU est complexe
  - ➔ il existe des approximations de LRU qui sont plus simples à implémenter

Quelle degré d'associativité ?

- La complexité matérielle d'un cache N-way set-associatif augmente avec N
- La latence du cache augmente avec N
- Le nombre de conflits de cache décroît (en général) quand N augmente
- Exemple
  - Cache L1 ➔ 32 Ko, 4-way set-associatif
    - On veut que la latence ne dépasse pas quelques cycles donc, en général, faible degré d'associativité
  - Cache L2 ➔ 2 Mo, 8-way set-associatif
- En général, les caches de niveau 2 et 3 ont une associativité élevée (8 à 32 ways)
  - Un miss qui nécessite d'accéder à la mémoire principale peut « coûter » plusieurs centaines de cycles de latence

- Pour les caches L2 et L3 d'associativité élevée, on préfère en général accéder le cache en 2 temps: d'abord les tags, puis les lignes
  - Cela augmente la latence mais simplifie le matériel (et diminue la consommation électrique)
- Exemple: N way, 2^n entrées par way



LRU: exemple

Cache 4-way set-associatif LRU

Supposons que les lignes A,B,C,D,E aient le même index cache (= même set)

Ligne accédée	Set avant	Hit / miss ?	Ligne évincée	Set après
A	DBCA	hit		DBCA
B	DBCA	hit		DBCA
C	DBCA	hit		DBCA
D	DBCA	hit		DBCA
E	DBCA	miss	A	DBCE
A	DBCE	miss	B	DACE
E	DACE	hit		DACE
B	DACE	miss	C	DABE

LRU et distance de réutilisation

- Soit une liste ordonnée comportant toutes les lignes qui ont le même index cache
- Quand on accède une ligne, on l'enlève de la liste et on la remet en tête de liste
  - Les lignes proches de la tête de liste sont celles qui ont été accédées récemment
- Distance de réutilisation d'une ligne = position de la ligne dans la liste
  - Tête de liste ➔ distance = 1
  - distance de réutilisation = nombre de lignes distinctes ayant été accédées depuis la dernière utilisation de la ligne courante
- Quand la distance est plus grande que l'associativité du set, c'est un miss, sinon c'est un hit

Lignes accédées	A	B	C	D	E	A	E	B	A	D
Distance de réutilisation						5	2	5	3	4
Set 4 way LRU						M	H	M	H	H

- Attention, cas pathologique: accès circulaire sur un nombre de lignes supérieur à l'associativité ➔ 100% de miss
  - …ABCDE ABCDE ABCDE ABCDE ABCDE… ➔ 100 % de miss sur 1 set 4 way

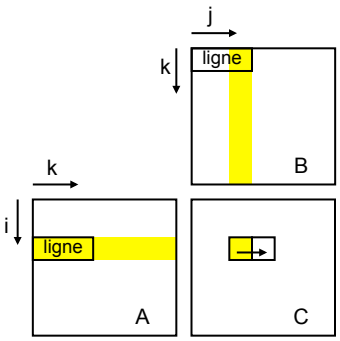
Mise en œuvre de LRU (N-way)

- LRU exact: plusieurs méthodes
  - Implémenter la liste
    - ligne LRU est celle en dernière position
    - Nécessite $\log_2(N)$ bits pour chaque entrée du cache
  - Matrice de bits
    - Bit M(i,j) vaut 1 si entrée i du set utilisée plus récemment qu'entrée j
    - Accès entrée i ➔ met tous les M(i,j) à 1 et tous les M(j,i) à 0
    - LRU = entrée dont la ligne est remplie de 1 et la colonne remplie de 0
    - Nécessite N x (N-1) / 2 bits par set
- LRU approximatif
  - NMRU (Not the Most Recently Used)
    - Évince une ligne au hasard sauf celle qui a été accédée le plus récemment
    - $\text{Log}_2(N)$ bits par set
  - LRU en arbre
    - 1 bit MRU par paire d'entrées dans le set, 1 bit MRU pour chaque paire de paires, 1 bit MRU pour chaque paire de paires de paires, etc…
    - N-1 bits par set

Analyser la localité: exemple

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

Remarque: en C, A[i][j] et A[i][j+1] sont adjacents en mémoire

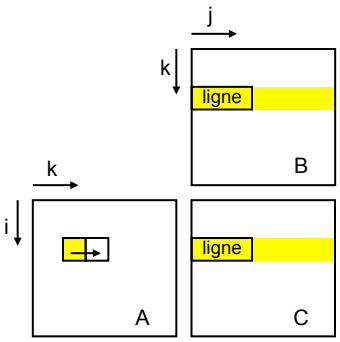


- Matrice C ➔ très bonne localité
  - En fait, on peut utiliser un registre pour c[i][j]
- Matrice A ➔ bonne localité
  - ligne réutilisée dans plusieurs itérations consécutives de k
- Matrice B ➔ mauvaise localité
  - N lignes différentes de B sont accédées pour chaque itération j

On peut parfois améliorer la localité

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

On a permuté les boucles *for k* et *for j*



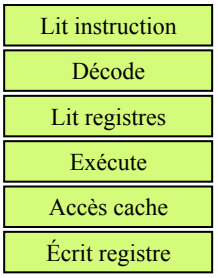
- Matrice A ➔ très bonne localité
  - On peut utiliser un registre pour A[i][k]
- Matrice B et C ➔ bonne localité
  - Lignes réutilisées dans plusieurs itérations consécutives de j
- ➔ On a amélioré la localité !
- Expérience sur un Pentium 4 avec N=1000
  - Premier programme ➔ 11.7 secondes
  - Nouveau programme ➔ 3.1 secondes

Bande passante

- Bande passante = nombre moyen d'octets par unité de temps que peut délivrer une mémoire, un cache, ou un bus
- Important pour la performance. Exemple:
  - Bus mémoire de 64 bits de largeur, cycle bus = 8 cycles processeur
  - ➔ bande passante = 1 octet par cycle processeur
  - Ligne de cache 64 octets ➔ 64 cycles processeur
  - Exemple: latence mémoire de 100 cycles ➔100 cycles pour le premier octet + 3 cycles pour obtenir une donnée complète de 4 octets, + 60 cycles pour le reste de la ligne
  - Théoriquement, le pipeline d'instructions pourrait redémarrer sans attendre que le reste de la ligne soit stocké dans le cache ➔ miss non bloquant
  - Mais si un 2ème miss de cache se produit moins de 60 cycles après le redémarrage du pipeline, il faudra attendre de toute manière
- La bande passante doit être suffisante pour supporter des rafales de miss de cache

Bande passante: exemple

```
Boucle:  STORE f1,i2
         i2 ← i2 ADD 8
         i3 ← i3 SUB 1
         BNZ i3, Boucle
```



- Hypothèses:
  - Cache write-back
  - On suppose que les données ne sont pas dans le cache
  - On suppose miss bloquant ➔le pipeline est bloqué tant que la ligne entière n'est pas dans le cache
  - Sur un miss, on suppose qu'il faut 100 cycles pour que le premier octet arrive
- Quelle est la performance en fonction de la bande passante X (octets/cycle) ?
  - Ligne de 64 octets contient 8 flottants DP ➔ 1 miss toutes les 8 itérations
  - on perd 99+64/X cycles toutes les 8 itérations (32 inst)

Cycles par instruction $\quad CPI = \frac{32 + 99 + \frac{64}{X}}{32} \approx 4 + \frac{2}{X}$

- X=1 octet/cycle ➔ CPI = 6
- X=2 octets/cycle ➔ CPI = 5
- X=4 octets/cycle ➔ CPI = 4.5
- X=64 octets/cycle ➔ CPI ~ 4

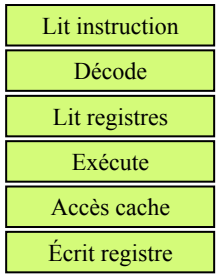
↓ La performance augmente

Miss non bloquant

- On aimerait pouvoir redémarrer le pipeline sans attendre que la ligne soit complètement écrite dans le cache.
- Commencer par charger la donnée demandée par le LOAD/STORE
  - Puis charger le reste de la ligne au rythme autorisé par la bande passante
  - Exemple: LOAD demande octets 12 à 15 dans une ligne de 64 octets. On commence par charger les octets 12 à 15, puis on charge les octets 16 à 63, puis les octets 0 à 11
- On a un registre de miss dans lequel on écrit l'adresse de la ligne en cours de chargement, ainsi que des bits de présence associés à chaque octet de la ligne
  - Ligne 64 octets ➔ 64 bits de présence
  - On pourrait aussi associer 1 bit à un groupes de plusieurs octets
- Au début du miss, mettre tous les bits à 0. Quand octet arrive, met bit correspondant à 1
- Si un LOAD/STORE essaie d'accéder à la ligne en cours de chargement (*hit on miss*), regarder les bits de présence pour savoir si le chargement de la donnée est terminé
- Quand tous les bits de présence sont à 1, la ligne est valide et on peut libérer le registre de miss

Miss non bloquant: exemple

```
Boucle:  STORE f1,i2
         i2 ← i2 ADD 8
         i3 ← i3 SUB 1
         BNZ i3, Boucle
```



- Hypothèses:
  - Cache write-back
  - On suppose que les données ne sont pas dans le cache
  - Miss non bloquant
  - Sur un miss, on suppose qu'il faut 100 cycles pour que le premier octet arrive, les octets suivants arrivent par adresse croissante d'abord
- Bande passante X (octets/cycle)
- Cycle T: miss premier octet de la ligne ➔ pipeline bloqué
- Cycle T+99+8/X ➔ pipeline redémarre
- Cycle T+99+8/X+4 ➔ STORE hit ou hit on miss
- Cycle T+99+16/X ➔ flottant suivant chargé
- Si X ≤ 2 octets/cycle, 16/X ≥ (8/X+4)

$$CPI = \frac{32 + (99 + \frac{8}{X}) + 7 \times (\frac{8}{X} - 4)}{32} \approx 3.2 + \frac{2}{X}$$

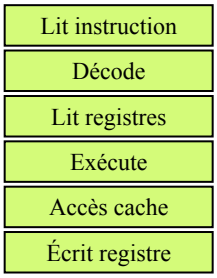
- X=1 octet/cycle ➔ CPI = 5.2
- X=2 octets/cycle ➔ CPI = 4.2

Quelle taille de ligne ?

- Petites lignes
  - on ne met sur le bus et dans le cache que des données utiles la plupart du temps
  - la place occupée par les tags augmente
    - Pour une taille de cache donnée, diviser par 2 la taille de ligne multiplie par 2 le nombre de lignes dans le cache et donc le nombre de tags
- Grandes lignes
  - Si bonne localité spatiale, effet de préchargement bénéfique
  - Si localité spatiale insuffisante, espace du cache mal utilisé (« trous » dans les lignes, augmentation des conflits) et bande passante gaspillée
- On choisit une taille de ligne ni trop grande ni trop petite
  - 64 octets est un choix fréquent
  - Caches L1 ➔ parfois petites lignes (exemple 32 octets)
  - Caches L2 et L3 ➔ parfois grandes lignes (exemple 256 octets)

Taille de ligne: exemple 1

```
Boucle:  STORE f1,i2
         i2 ← i2 ADD 8
         i3 ← i3 SUB 1
         BNZ i3, Boucle
```



- Hypothèses:
  - Cache write back
  - Données pas dans le cache
  - Miss bloquant
  - Latence miss: 100 cycles pour premier octet
  - Bande passante 1 octet/cycle
- Quel est le débit d'instruction en fonction de la taille de ligne L ?
  - miss toutes les L/8 itération ➔ pipeline bloqué pendant 99 + L cycles

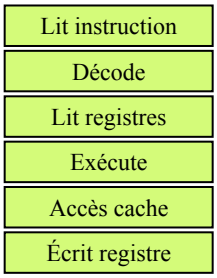
$$CPI = \frac{\frac{L}{8} \times 4 + 99 + L}{\frac{L}{8} \times 4} = 3 + \frac{198}{L}$$

- L = 128 ➔ CPI = 4.5
- L = 64 ➔ CPI = 6
- L = 32 ➔ CPI = 9

↓ La performance diminue

Taille de ligne: exemple 2

```
        i0← 8000
Boucle: STORE f1,i2
        i2 ← i2 ADD i0
        i3 ← i3 SUB 1
        BNZ i3, Boucle
```



- Hypothèses:
  - Cache write back
  - Données pas dans le cache
  - Miss bloquant
  - Latence miss: 100 cycles pour premier octet
  - Bande passante 1 octet/cycle
- Quel est la performance en fonction de la taille de ligne L ?
  - miss à chaque itération ➔ pipeline bloqué pendant 99 + L cycles

$$CPI = \frac{4 + 99 + L}{4} \approx 26 + L$$

- L = 64 ➔ CPI = 90
- L= 32 ➔ CPI = 58
- L=16 ➔ CPI = 42

La performance augmente

Cache write-back et bande passante

- Cache write-back permet de réduire le débit d'écriture vers le cache L2 quand on écrit plusieurs fois de suite dans la même donnée
  - Exemple: C[i][j] += A[i][k] * B[k][j]
- Les écritures se font à l'éviction d'une ligne ➔ on utilise pleinement la largeur des bus
  - Exemple: quand une donnée de 64 bits passe sur un bus 128 bits, la moitié du bus est inutilisée
- ➔ cache write-back intéressant quand la bande passante est limitée
- Les caches L2 et L3 sont généralement write-back

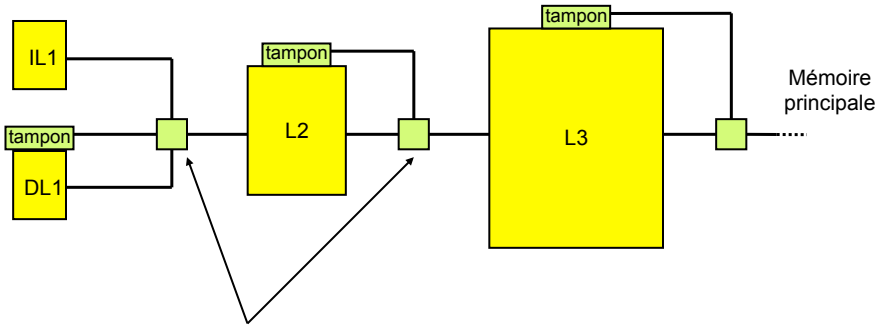
Latence des écritures ?

- La latence d'une écriture, est-ce important ?
- Du point de vue de la performance, il y a une différence fondamentale entre les LOAD et les STORE: on a en général besoin que le LOAD soit terminé pour pouvoir continuer à travailler ➔ on veut que la latence du LOAD soit la plus faible possible
- Pour un STORE, on n'est pas obligé de faire l'écriture immédiatement
- Sur un hit en écriture dans un cache L1 write-back, le pipeline ne se bloque pas
  - Et pourtant, l'écriture n'est pas complètement terminée puisque lorsque la ligne de cache sera évincée du cache L1, on consommera à ce moment là de la bande passante à cause de cette écriture
- Question: doit-on bloquer le pipeline lorsqu'on écrit dans un cache L1 write-through ?
  - ➔ ce n'est pas souhaitable

Tampon d'écritures

- Lorsqu'on écrit dans un cache L1 write-through, qu'il y ait un hit ou un miss en écriture dans le L1, le pipeline d'instructions ne se bloque pas ➔ l'écriture vers le cache L2 est faite par un mécanisme indépendant du pipeline d'instructions
- Les caches write-through sont généralement associés à un tampon d'écritures = mini-cache write-back entre les caches L1 et L2
  - Contient seulement quelques entrées (exemple: 4 STORE) ➔ suffisamment petit pour être full-associatif
  - on peut grouper les STORE à des adresses adjacentes pour exploiter la largeur du bus
- Un STORE écrit simultanément dans le cache L1 (s'il y a un hit) et dans le tampon d'écritures
  - S'il n'y a pas de hit dans le tampon d'écritures, on crée une entrée pour cette écriture
  - Lorsque le tampon est plein, on fait des écritures dans le cache L2 pour faire de la place dans le tampon
- Lorsqu'un LOAD accède le cache L1 et qu'il y a un miss, on doit regarder dans le tampon d'écritures ➔ en cas de hit dans le tampon, la ligne manquante doit être mise à jour avant son chargement dans le L1
- Les caches write-back sont en général également associés à un tampon d'écritures
  - Une ligne « dirty » évincée du cache est mise dans le tampon
  - La ligne est écrite dans le niveau de cache suivant lorsque le tampon est plein

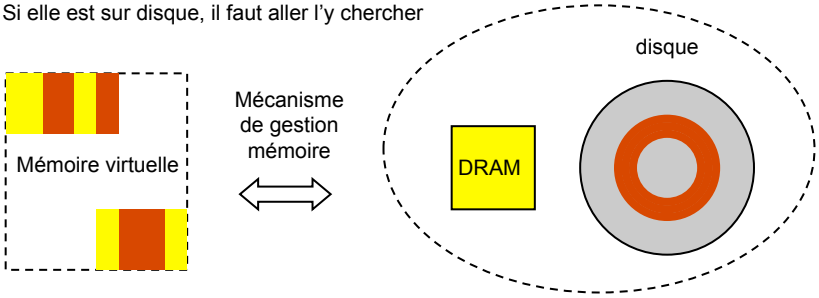
Hiérarchie mémoire



- Arbitrage bande passante:
  - lectures données ont priorité sur lectures instructions
  - Lectures on priorité sur écritures sauf si tampon d'écritures plein
  - Lorsque pas de miss lecture en cours, utiliser bande passante disponible pour vider les tampons d'écritures

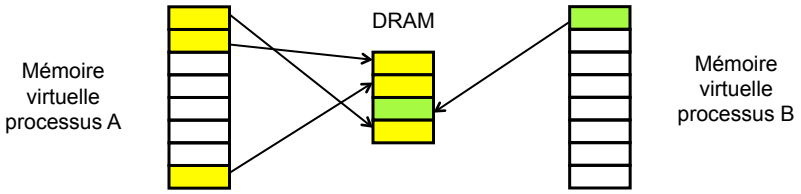
Mémoire virtuelle ⇔ mémoire physique

- Les adresses manipulées par le programmeur sont des adresses virtuelles
  - Exemple: adresses virtuelles sur 64 bits
- La mémoire physique (DRAM) est beaucoup plus petite que la mémoire virtuelle
- Une partie du contenu de la mémoire virtuelle est stockée sur disque
- Il faut un mécanisme permettant, à partir d'une adresse virtuelle, de savoir si l'adresse est « mappée » en DRAM ou bien sur disque
  - Si elle est « mappée » en DRAM, if faut un mécanisme de transformation de l'adresse virtuelle en adresse physique
  - Si elle est sur disque, il faut aller l'y chercher



Pagination

- L'espace mémoire est divisé en pages
  - Exemple: page de 4 Ko



- Il faut traduire le numéro de page virtuelle en numéro de page physique ➔ table des pages stockée en mémoire
- Si la page demandée n'est pas en mémoire physique, il faut récupérer la page sur disque et mettre à jour la table des pages ➔ « défaut » de page
- Pénalité défaut de page très grande !
  - Latence d'un accès disque (aléatoire) se mesure en millisecondes (dépend de la vitesse de rotation du disque. Exemple: 5 ms)
  - Processeur 2 Ghz ➔ cycle d'horloge 0.5 ns ➔ latence disque = plusieurs millions de cycles !

Table des pages

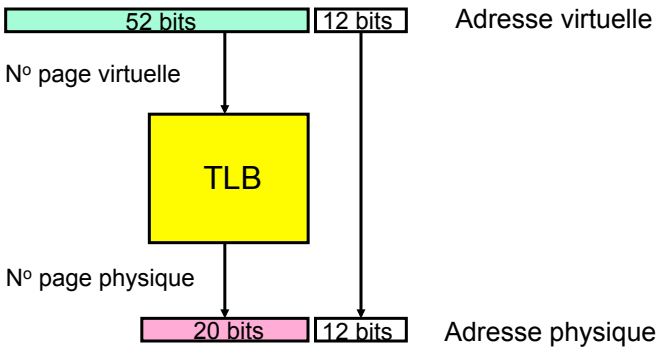
- La table des pages est stockée en mémoire et gérée par le système d'exploitation (OS)
  - DRAM de 2 Go, page de 4 Ko ➔ 512k pages
- Plusieurs organisations possibles pour la table des pages
- Table inverse: une entrée par page physique ➔ table de hachage
- Une entrée par page virtuelle utilisée
  - Table potentiellement très grande (2^{52} pages virtuelles de 4k) mais en pratique une grande partie de la mémoire virtuelle n'est pas utilisée
- ➔ plusieurs niveaux de tables
  - Partie A de l'adresse virtuelle désigne entrée dans table premier niveau, laquelle donne pointeur vers table second niveau
  - Partie B de l'adresse virtuelle désigne entrée dans table second niveau, laquelle donne pointeur vers table troisième niveau
  - Partie C de l'adresse virtuelle désigne entrée dans table troisième niveau, laquelle donne numéro de page physique
  - (ou 4ème niveau, etc…)
- La table des pages peut être elle-même soumise à la pagination

TLB

- Comme la table des pages se trouve en mémoire, chaque accès mémoire du programme nécessite plusieurs accès mémoire ☹
  - C'est-à-dire plusieurs accès cache si la partie de la table des pages dont on a besoin se trouve dans le cache
- TLB (Translation Lookaside Buffer) ➔ petite SRAM sur la puce contenant les correspondances page virtuelle/page physique pour les pages accédées récemment
  - = cache de traduction d'adresse
  - Tag = numéro page virtuelle, entrée donne numéro page physique (+ infos supplémentaires)
- Miss TLB
  - Un miss TLB déclenche une exception (d'où l'intérêt des interruptions précises)
  - L'OS (ou le microcode) accède la table des pages ➔ plusieurs cycles de pénalité
  - L'OS (ou le microcode) met la traduction manquante dans le TLB et relance le programme interrompu
- Exemple: Pentium 4, pages 4 Ko
  - TLB d'instructions 128 pages, 4-way set-associatif
  - TLB de données 64 pages, full-associatif
- Il peut y avoir plusieurs niveaux de TLB
  - Petit TLB L1, « gros » TLB L2

TLB: exemple

- Pages de 4 Ko ➔ 12 bits d'offset page
- Adresses virtuelles sur 64 bits
- Adresses physique sur 32 bits



Mémoire virtuelle et caches

- Accède-t-on aux caches avec l'adresse virtuelle ou l'adresse physique ?
- Cache à adresses virtuelles
  - Avantage: on n'accède au TLB que sur un miss
  - Inconvénients:
    - Problème des « synonymes » : pages logiques distinctes « mappées » sur la même page physique (exemple: process fork) ➔ si j'écris dans une ligne de cache, je dois mettre à jour ou invalider les lignes synonymes
    - Gros tags
- Cache à adresses physiques
  - Avantage: pas de problème de synonymes
  - Inconvénient: il faut accéder au TLB avant d'accéder au cache
- Cache L1 instructions ➔ adresse virtuelle ou index virtuel/tag physique
- Cache L1 données ➔ index virtuel / tag physique
- Caches L2,L3 ➔ index et tag physiques

Index virtuel / tag physique

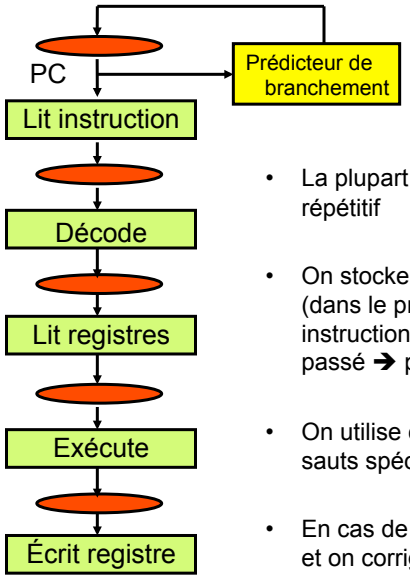
- Principe:
  - Les tags stockés dans le cache correspondent aux adresse physiques
  - La structure du cache est telle que les bits utilisés pour débuter l'accès au cache ne sont pas changés par la traduction d'adresse
  - On commence donc à accéder (« indexer ») le cache avec les bits d'adresse virtuelle
  - En parallèle, on lit le TLB et on obtient l'adresse physique
  - Une fois qu'on a le tag et l'adresse physique, on peut faire la comparaison et savoir si on a un hit ou un miss
- C'est strictement équivalent à un cache à adresse physique, mais avec la rapidité d'accès d'un cache à adresse virtuelle ➔ très intéressant pour cache L1
- Attention: ce n'est possible que si certaines conditions sur la taille de page, la taille du cache et son associativité sont vérifiées
  - Voir exemple transparent suivant

Exemple

- Soit un cache de 64 Ko 2-way set-associatif avec des lignes de 64 octets
- Les pages mémoire font 4 Ko
- Question 1: ce cache peut-il être indexé virtuellement et taggé physiquement ?
  - D'abord, on identifie les bits d'adresse utilisés pour indexer le cache
    - Le cache comporte 64 Ko / 64 = 1024 lignes. Comme le cache est 2-way SA, il y a 1024/2 = 512 sets (= 512 lignes par way). Il faut donc $\log_2(512) = 9$ bits pour indexer le cache
    - Comme les bits 0 à 5 de l'adresse correspondent à l'offset à l'intérieur de la ligne, les 9 bits utilisés pour l'indexage sont les 9 bits suivants, c'est-à-dire les bit 6 à 14
  - On regarde si ces bits d'index sont altérés par la traduction d'adresse
    - Page de 4 Ko ➔ les 12 bits de poids faible sont inchangés par la traduction d'adresse
    - ➔ les bits 12 à 14 de l'adresse virtuelle sont a priori différents de ceux de l'adresse physique
  - ➔ Le cache ne peut pas être indexé virtuellement / taggé physiquement
- Question 2: comment modifier le cache pour que ce soit possible ?
  - Solution 1: augmenter l'associativité ➔ 16-way (64 sets, 6 bits d'index) ou plus
  - Solution 2: diminuer la taille du cache ➔ 8 Ko (64 sets, 6 bits d'index) ou moins
  - Solution 3: combiner solutions 1 et 2
- Question 3: peut-on résoudre le problème sans changer le cache ?
  - ➔ oui, en augmentant la taille des pages (ici, pages de 32 Ko ou plus)

La prédiction de branchement

Prédicteur de branchement: rappel



- La plupart des programmes ont un comportement répétitif
- On stocke dans une mémoire microarchitecturale (dans le processeur) des informations sur les instructions de contrôle et leur comportement passé ➔ prédicteur de branchements
- On utilise ces informations pour effectuer les sauts spéculativement
- En cas de mauvaise prédiction, on vide le pipeline et on corrige le PC

Classification des branchements

- Conditionnel / inconditionnel
  - Branchement conditionnel
    - Le saut se fait seulement si la condition est vraie (on dit que le branchement est « pris »)
    - Si la condition est fausse, le branchement est « non pris » (pc ← pc+4)
    - L'évaluation de la condition est généralement faite à l'étage d'exécution
  - Branchement inconditionnel (ou saut)
    - On saute systématiquement (pc ← adresse cible du saut)
- Immédiat / indirect
  - Branchement immédiat
    - Adresse cible connue à la compilation, peut être calculée au décodage
    - Exemple: branchement relatif ➔ adresse cible = PC + CONST
  - Branchement indirect
    - Adresse cible lue dans un registre ➔ connue seulement à l'étage d'exécution

Utilisation des branchements en C

- Branchement conditionnel immédiat
  - Boucles *for* et *while*
  - *If else*
  - *Switch case*
- Branchement inconditionnel (saut) immédiat
  - Boucles
  - *if else* (sortie)
  - Break, continue
  - Appel de fonction
- Branchement inconditionnel (saut) indirect
  - Retour de fonction
  - Pointeur de fonction
  - *Switch case*
  - Saut longue distance
- Branchement conditionnel indirect
  - N'existe pas dans tous les jeux d'instructions

Exemple

Calcul de valeur min sur des valeurs aléatoires x[i]

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

Branchement conditionnel de boucle

Branchement conditionnel de test

- Branchement de boucle: non pris à la dernière itération seulement
- Branchement de test: une chance sur *N* pour que la *N*ème valeur soit le min des *N* premières valeurs
  - 1ère itération: une chance sur 2 d'être non pris
  - 2ème itération: une chance sur 3
  - 3eme itération: une chance sur 4
  - …
  - 999ème itération: une chance sur 1000

$$\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\cdots+\frac{1}{1000}\approx 6.5$$

Autres exemples

Branchement toujours pris (normalement !)

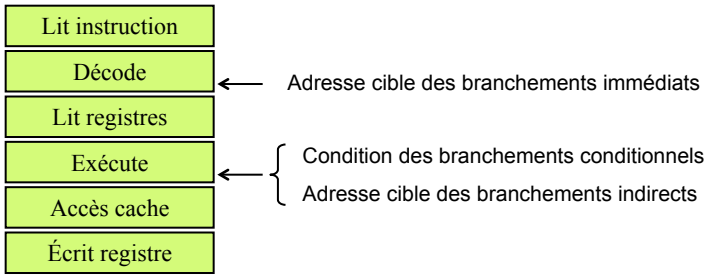
```
adresse = (int *)  malloc(sizeof(int)) ;
if (adresse == NULL)
   erreur("mémoire pleine") ;
```

Branchement toujours non pris lorsque b=256

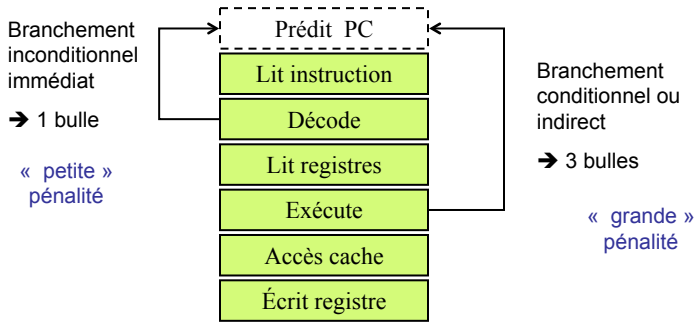
```
int  modulo(int a,int b)   {
    if ((b & (b -1))==0)
        return(a & (b -1));
    return(a % b);
}
```

```
#include "ma_librairie.h"
#define HASH_INDEX(v)  modulo(v,256)
```


Branchements et pipeline



Pénalité de mauvaise prédiction



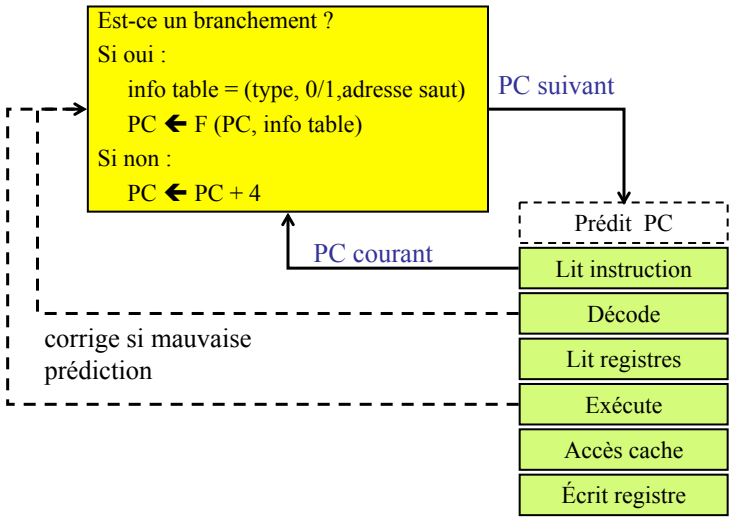
Le nombre de bulles sur une mauvaise prédiction dépend de la longueur du pipeline

Exemple: Intel Core ➔ 14 cycles de pénalité de mauvaise prédiction sur les branchements conditionnels

Prédiction de branchement: principe

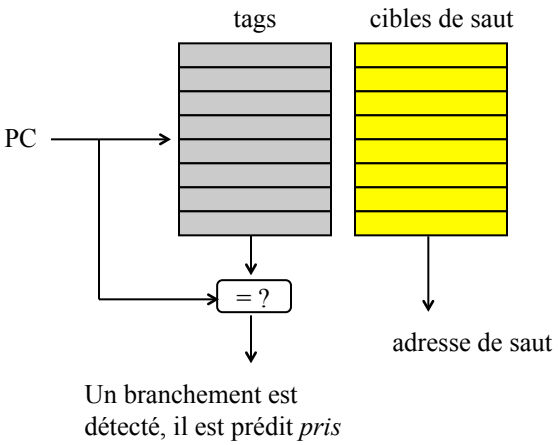
- Table de prédiction accédée en même temps que la cache d'instructions
- Table de prédiction de branchement permet de
  - Prédire qu'à l'adresse indiquée par le compteur de programme se trouve une instruction de branchement
  - Identifier le type de branchement
    - branchement conditionnel ou inconditionnel ?
    - appel de fonction ?
    - retour de fonction ?
  - Prédire l'adresse de saut
  - Pour les branchements conditionnels, prédire si le branchement est pris (1) ou non pris (0)
- En cas de mauvaise prédiction, profiter du temps de réparation du pipeline pour corriger l'information stockée dans la table de prédiction

Table de prédiction



Cache de branchements

BTB (*Branch Target Buffer)*



Utilisation du BTB

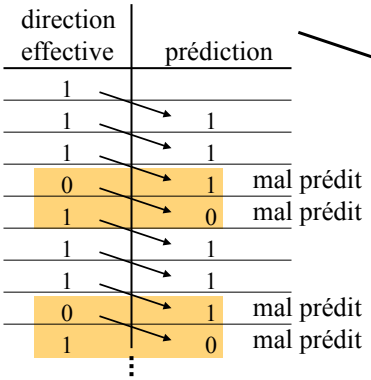
- On stocke dans le BTB les branchements inconditionnels et les branchements conditionnels pris
- La présence d'une entrée dans le BTB entraîne un saut à l'adresse indiquée par le BTB
- En l'absence d'une entrée BTB, on suppose que l'instruction n'est pas un branchement ou est un branchement conditionnel non pris
- En cas de mauvaise prédiction, corriger le BTB
  - si on a manqué un saut, rajouter l'entrée manquante
  - si on a prédit un branchement pris alors qu'en réalité il est non pris, enlever l'entrée du BTB
  - si on a sauté à une mauvaise adresse, corriger l'adresse de saut dans le BTB

BTB: remarques

- Parfois appelé autrement
  - BTAC (*branch target address cache*), *next-fetch predictor*, etc…
- Taille: ça dépend …
  - Petit sur certains processeurs (exemple: 8 entrées)
  - Gros sur d'autres (exemple: 2k entrées)
- Associativité: ça dépend
  - de 1 (direct-mapped) à 4
- Prédiction « comme la dernière fois » (= on prédit le même comportement que la dernière fois)
  - fonctionne bien pour les branchements inconditionnels immédiats
    - le comportement est toujours le même
  - pas très performant pour les branchements conditionnels et les retours de fonction

Exemple

Prédiction « comme la dernière fois »




```
for (i=0;i<1000;i++) {
    for (j=0;j<4;j++) {
        : corps de la boucle
    }
}
```

Branchement mal prédit à la première et à la dernière itération de la boucle j

Petit exercice

- On considère un branchement conditionnel ayant une probabilité *p* d'être pris.
- Quelle serait la probabilité de mauvaise prédiction si la prédiction était toujours « non pris » ?

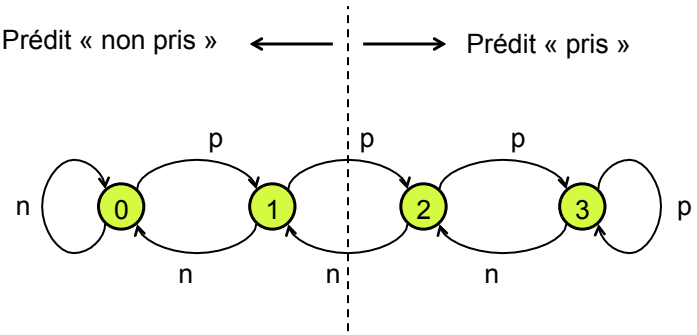
$$P_{mp} = p$$

- En prédisant « comme la dernière fois », quelle est la probabilité de mauvaise prédiction ?

$$P_{mp} = p(1-p) + (1-p)p = 2p(1-p)$$

Exemple: p = 0.1 ➔ P_{mp} = 0.18

Compteur 2 bits



- Incrémente lorsque le branchement est pris
- Décrémente lorsque le branchement est non pris
- Prédit « pris » si valeur ≥ 2

Prédiction avec compteur 2 bits

compteur avant	prédiction	direction effective	compteur après
2	1	1	3
3	1	1	3
3	1	1	3
3	1	0	2
2	1	1	3
3	1	1	3
3	1	1	3
3	1	0	2
2	1	1	3

```
for (i=0;i<1000;i++) {
   for (j=0;j<4;j++) {
      : corps de la boucle
   }
}
```

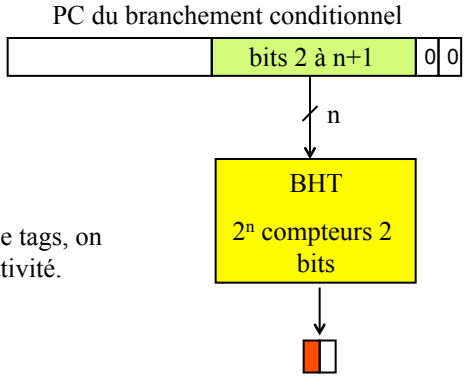
Ici, on fait 2 fois moins de mauvaises prédictions qu'en prédisant « comme la dernière fois »

Compteur 2 bits : mise en œuvre

- Première possibilité : rajouter un compteur 2 bits dans chaque entrée du BTB
  - on bénéficie de l'associativité du BTB
  - si un branchement est prédit pris mais qu'en réalité il est non pris, on décrémente le compteur et on laisse l'entrée dans le BTB
- Deuxième possibilité : stocker les compteurs 2 bits dans une table spécifique, la BHT (*branch history table*)
  - pas de tags, juste 2 bits dans chaque entrée
  - la BHT peut comporter plus d'entrées que le BTB
  - Exemple: IBM POWER6 ➔ BHT de 16k compteurs 2 bits
  - quand le compteur passe de l'état 2 à l'état 1, l'entrée dans le BTB n'est plus vraiment nécessaire
    - BTB set associatif: quand un branchement (pris) a besoin d'une entrée BTB et que les entrées du set sont toutes déjà occupées, on prend de préférence une entrée dont le compteur 2 bits vaut 0 ou 1

BHT

Aussi appelé prédicteur bimodal

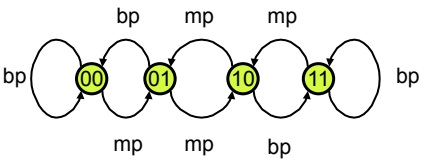


Rem: comme il n'y a pas de tags, on ne peut pas avoir d'associativité.

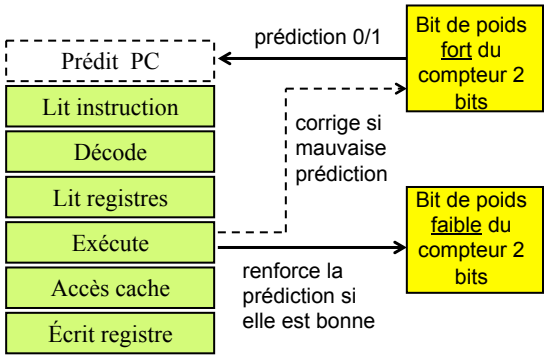
Interférences entre branchements. Problème analogue aux conflits dans un cache direct-mapped

Le bit de poids fort du compteur donne la prédiction

BHT ➔ 2 SRAM



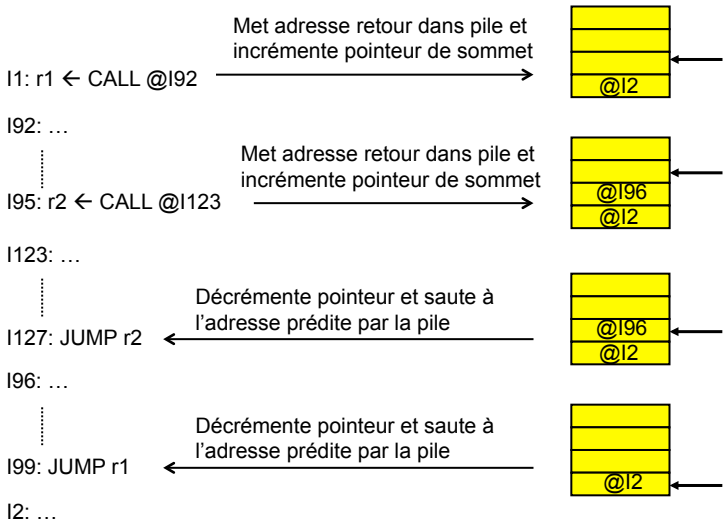
- Le bit de poids fort du compteur ne change que sur une mauvaise prédiction (mp)
- Sur une bonne prédiction (bp), la prédiction est renforcée (état 0 ou 3)



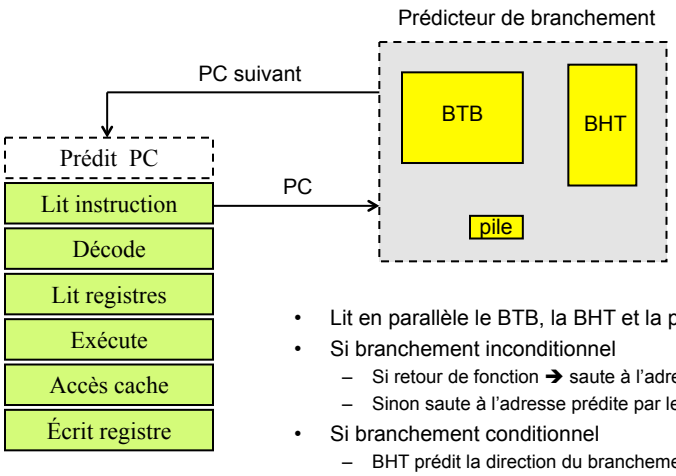
Prédire les retours de fonction

- Utiliser une pile d'adresses de retour
  - pour chaque CALL exécuté, on empile l'adresse de retour
  - pour prédire un retour, au lieu d'utiliser l'adresse fournie par le BTB, dépiler l'adresse au sommet de la pile et sauter à cette adresse
- Peut être utilisé aussi pour prédire les retours d'interruptions et d'exceptions
- Si pile suffisamment profonde, 100 % de bonne prédiction
  - Taille typique: 8 à 32 entrées
  - ne pas abuser de la récursivité
- Seule difficulté éventuelle ➔ identifier les appels et les retours
  - utiliser le code-op (ex. appel = saut avec lien)
  - le compilateur peut aider le processeur
    - exemple: *hint* (Alpha), jr r31 (MIPS) ...

Pile d'adresses de retour: exemple



Résumé: BTB / BHT / pile



- Lit en parallèle le BTB, la BHT et la pile de retour
- Si branchement inconditionnel
  - Si retour de fonction ➔ saute à l'adresse prédite par la pile
  - Sinon saute à l'adresse prédite par le BTB
- Si branchement conditionnel
  - BHT prédit la direction du branchement
  - Si prédit « pris », saute à l'adresse prédite par le BTB
  - Sinon PC ← PC+4

D'où viennent les mauvaises prédictions ?

- Branchements inconditionnels immédiats (« petite » pénalité)
  - BTB: miss de démarrage, miss de capacité, miss de conflit
- Retours de fonctions
  - Pile pas assez profonde (exemple: appels récursifs)
  - Retour de fonction pas « orthodoxe » (ça peut arriver …)
- Branchements conditionnels et branchements inconditionnels indirects (sauf retours de fonctions)
  - Certains branchements sont intrinsèquement difficiles à prédire ➔ comportement plus ou moins aléatoire
  - Limitations de l'algorithme de prédiction
  - Interférences entre branchements dans la BHT
  - Miss du BTB pour les indirects

Petit exercice

- Soit un processeur dont la pénalité de mauvaise prédiction vaut 14 cycles
- Soit un programme dont 1 instruction sur 10 exécutées est un branchement conditionnel
- Combien les mauvaises prédictions coûtent-elle en cycles par instruction (CPI) si le prédicteur de branchement prédit correctement 90% des branchements conditionnels ?
  - 1 branchement conditionnel sur 10 est mal prédit ➔ il y a une mauvaise prédiction toutes les 100 instructions ➔ on perd 14 cycles toutes les 100 instructions
  - Les mauvaises prédictions coûtent 14/100 = 0.14 en CPI
- Et si on prédit correctement 95% des branchements conditionnels ?
  - On perd 14 cycles toutes les 200 instructions
  - Les mauvaises prédictions coûtent 0.07 en CPI
- Passer de 90% de bonne prédiction à 95% de bonne prédiction veut dire qu'on divise par 2 le nombre de mauvaises prédictions

Prédiction à 2 niveaux d'historique

- Inventé au début des années 90 pour les branchements conditionnels
- Utilisé dans les processeurs actuels
- Permet d'atteindre des taux de bonne prédiction souvent supérieurs à 95%, parfois proches de 100%
- Principe: exploiter les corrélations entre branchements et les motifs répétitifs

Corrélation: exemple 1

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

for (i=0; i<10; i++) ---------> p^9n

for (j=0; j<10; j++) ——————> p^9n

pppppppppnpppppppppnppp…

Le même motif se répète

Après avoir observé plusieurs fois le même motif, on peut en déduire qu'il faut prédire le branchement de la boucle j « non pris » après qu'il a été pris 9 fois consécutivement

Corrélation: exemple 2

```
B1: If (cond1 && cond2) {…}
B2: If (cond1) {…}
```

Si le 1er branchement est non pris (cond1 et cond2 vrais), on est sûr que le 2ème branchement sera non pris (cond1 vrai)

cond1	cond2	cond1 && cond2
F	F	F
F	V	F
V	F	F
V	V	V

- Supposons les 4 cas équiprobables (aléatoires)
- La probabilité de bien prédire B2 en se basant uniquement sur son comportement passé est de 50%

- Supposons maintenant qu'on connaisse le résultat de B1 au moment de prédire B2
- Si B1 est non pris (probabilité 1/4), prédire B2 non pris ➔ 100% de bonnes prédictions
- Si B1 est pris (probabilité 3/4), prédire B2 pris ➔ 2/3 de bonnes prédictions

- ➔ La probabilité de bien prédire B2 vaut (1/4)x(1) + (3/4)x(2/3) = 75%

Petit exercice

cond1 et cond2 aléatoires (50% vrai/faux)

Quelle version est la plus intéressante du point de vue de la prédiction de branchement ?

```
B1: If (cond1 && cond2) {…}
B2: If (cond1) {…}
```

Prédit B1 « pris » ➔ 75 % bien prédit

Prédit B2 « non pris » si B1 est non pris, sinon prédit « pris » ➔ 75 % bien prédit

La connaissance de B1 permet d'améliorer la prédiction de B2

```
B2: if (cond1) {…}
B1: If (cond1 && cond2) {…}
```

Prédit B2 « pris » ➔ 50 % bien prédit

Prédit B1 « pris » ➔ 75 % bien prédit

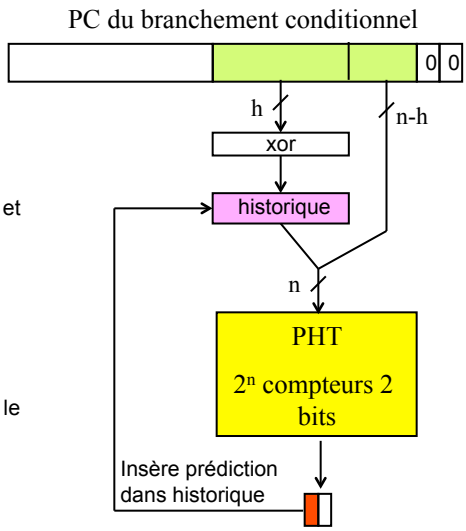
La connaissance de B2 ne permet pas d'améliorer la prédiction de B1

Prédicteur à historique global

- Forme la plus répandue de prédicteur à 2 niveaux d'historique
- Premier niveau d'historique = historique global
  - Historique global = registre à décalage de *h* bits qui contient les directions prises par les *h* derniers branchements rencontrés
- Deuxième niveau d'historique
  - chaque branchement utilise plusieurs compteurs 2 bits, stockés dans une PHT (*pattern history table*)
  - utiliser le PC et la valeur d'historique global pour sélectionner le compteur dans la PHT
- Remarque: une PHT, c'est comme une BHT sauf que la BHT n'est accédée qu'avec le PC alors que la PHT est aussi accédée avec l'historique global

Prédicteur *gshare*

- gshare ➔ prédicteur à historique global parmi les plus simples
- Index PHT = XOR entre les bits d'adresse et les *h* bits d'historique global
- Chaque prédiction est insérée dans l'historique global pour prédire les branchements suivants
- En cas de mauvaise prédiction, on corrige le compteur 2 bits dans la PHT et on répare l'historique global

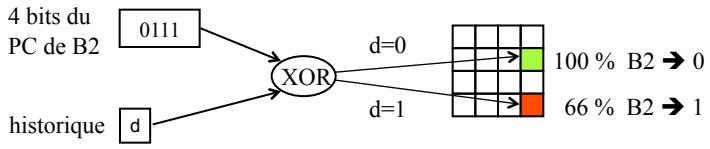


Gshare: exemple

B1: If (cond1 && cond2) {…}

B2: If (cond1) {…}

- PHT de 16 entrées
- Historique global de 1 bit
- Direction de B1 = d



Le nombre d'entrées utilisées par chaque branchement augmente généralement avec la longueur d'historique global

Problème des interférences

- La taille de la PHT est limitée
- Il peut arriver que deux branchements différents utilisent une même entrée de la PHT: c'est une interférence
- Problème analogue aux miss dans les caches
- Attention au « paradoxe de l'anniversaire »
  - 23 personnes dans une même pièce ➔ 50% de chance qu'il y ait au moins 2 personnes fêtant leur anniversaire le même jour

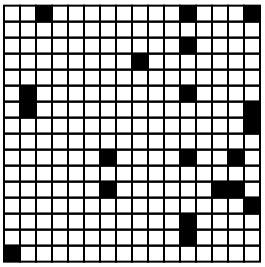
Petite expérience

PHT de 256 entrées

Tirer 20 entrées au hasard

Répéter l'expérience plusieurs fois :

Conflit en moyenne une fois sur 2



Et pourtant, il y a largement la place !

Longueur d'historique global ?

- Le nombre d'interférences est élevé quand …
  - …la taille de la PHT est petite
  - …le programme est gros et comporte beaucoup de branchements conditionnels
  - …l'historique global est long
- Pour un programme donné, il existe généralement une longueur d'historique global optimale
  - Gros programmes ➔ historique pas trop long sinon interférences
    - Sur programmes très gros, simple BHT parfois meilleure que gshare
  - Certains programmes ont des corrélations entre branchement qui ne peuvent être exploitées qu'avec un historique global long

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

p^9n

p^9n

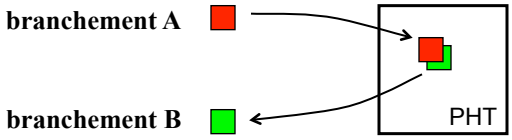
pppppppppnpppppppppppnppp…

Prédit non-pris lorsque historique = pppppppppp

➔ Besoin de 10 bits d'historique pour prédire boucle j correctement

➔ PHT (au moins) de 1024 compteurs 2 bits

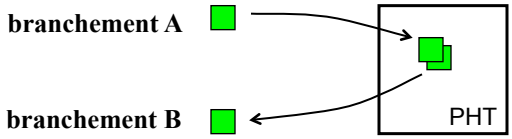
Interférences destructives / neutres



Branchements de biais opposés

interférence destructive

Génère des mauvaises prédictions

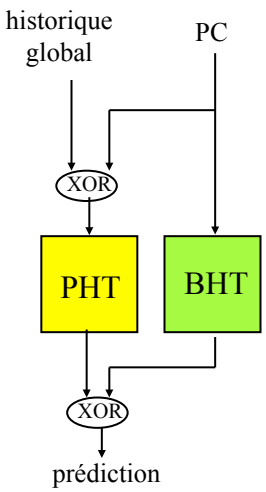


Branchements de même biais

interférence neutre

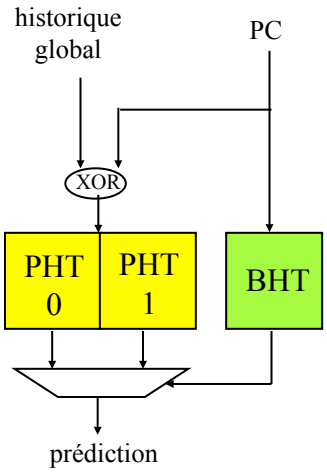
« indolore »

Prédicteur *agree*



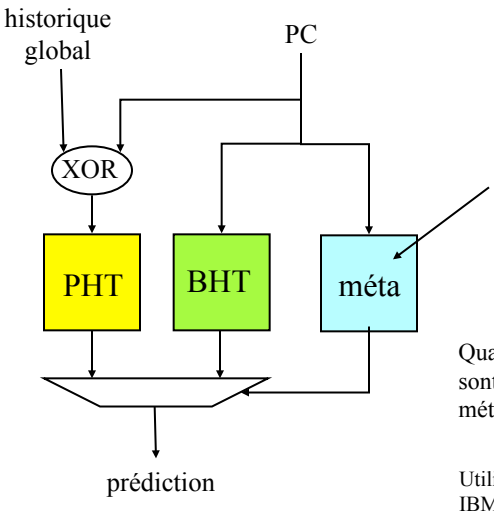
- Une BHT + une PHT
- Si le XOR de la direction du branchement avec la prédiction de la BHT vaut 1, incrémenter le compteur dans la PHT, sinon le décrémenter
- Diminue les interférences destructives dans la PHT
- Exemple
  - 2 branchements A et B
  - A toujours non pris (0), B toujours pris (1)
  - Dans les 2 cas, le XOR de la direction avec la prédiction de la BHT vaut 0
  - Si A et B utilisent un même compteur de la PHT, c'est une interférence neutre
- Au lieu d'utiliser une BHT, on peut aussi rajouter un bit de biais dans chaque entrée BTB
  - Allouer une entrée BTB pour chaque branchement, y compris conditionnels non pris
  - Biais = direction prise la première fois
- Prédicteur agree était utilisé dans les processeurs HP/PA
  - Bit de biais était un *hint* dans le jeu d'instructions

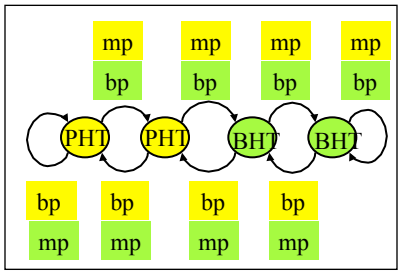
Prédicteur *bimode*



- Une BHT plus une PHT
- La PHT est divisée en 2 parties
- La prédiction de la BHT sélectionne quelle partie de la PHT donne la prédiction
- Permet de réduire les interférences destructives dans la PHT
  - Les prédictions stockées dans la PHT 0 sont majoritairement des prédictions « non pris », celles dans la PHT 1 des prédictions majoritairement « pris »
- A peu prés équivalent à prédicteur agree
- Utilisé dans les processeurs PA6T

Prédicteur hybride





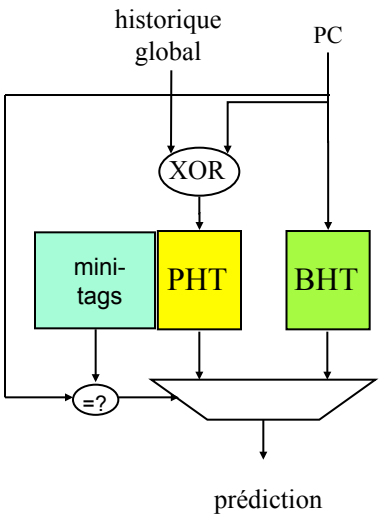
Quand les prédictions de la PHT et de la BHT sont identiques (bonnes ou mauvaises), la méta-prédiction ne change pas

Utilisé dans les processeurs Alpha 21264, IBM POWER, AMD, …

Prédicteur hybride: avantages

- Quand le programme est gros, la BHT subit moins d'interférences que la PHT
  - Si un branchement est mal prédit par la PHT à cause d'interférences, le méta-prédicteur désignera la BHT comme prédicteur le plus fiable pour ce branchement
- Les interférences dans le méta-prédicteur sont majoritairement neutres
  - Effet ressemble à prédicteur agree
- La BHT se réchauffe plus vite que la PHT au démarrage du programme ou lorsque 2 programmes s'exécutent en temps partagé
- On peut avoir un historique global plus long qu'avec gshare, agree ou bimode
- Variantes:
  - On peut indexer le méta-prédicteur avec des bits d'historique global
  - Si la fonction d'index du méta-prédicteur est différente de celle de la PHT et de la BHT, on peut ne pas corriger un prédicteur (BHT ou PHT) si la prédiction a été fournie par l'autre prédicteur et qu'elle était correcte

Prédicteur YAGS



- Chaque entrée de la PHT contient un compteur 2 bits et un mini-tag
- Exemple: mini-tag = 6 bits d'adresse
- Si hit, la PHT fournit la prédiction
- Si miss, la BHT fournit la prédiction
- Si miss et que la prédiction de la BHT est correcte, on n'alloue pas d'entrée dans la PHT pour ce branchement
  - Permet de réserver l'espace de la PHT pour les branchements qui en ont vraiment besoin
- Prédicteur de type YAGS utilisé dans les processeurs Intel Core

Prédicteur de branchement indirect

- Pour les branchements indirects autres que retours de fonctions
- Prédicteur à historique global ➔ exploite les corrélations
- Comme YAGS, sauf que …
  - Au lieu d'avoir des compteurs 2 bits dans la PHT, on a des adresses cibles
  - Sur un miss de la PHT, c'est le BTB qui fournit la prédiction
- Comme YAGS, on n'alloue une entrée dans la PHT que lorsque le branchement indirect est mal prédit par le BTB
- Utilisé dans les processeurs PA6T et Intel Core

Prédicteur de boucle

- Prédicteur à historique global n'arrive à prédire correctement la sortie d'une boucle que lorsque celle-ci fait un petit nombre d'itérations
- Les processeurs Intel Core comporte un prédicteur spécialisé pour les boucles
- Principe: détecter les motifs du type 1111….11110
  - Avec un compteur 8 bits, on peut prédire correctement la sortie des boucles faisant moins de 256 itérations

En résumé …

- Plus le pipeline d'instructions est long, plus le processeur aura intérêt à avoir un prédicteur de branchement sophistiqué
  - Actuellement, les prédicteurs les plus sophistiqués se trouvent dans les processeurs Intel ➔ BTB + pile + YAGS + boucles + indirects

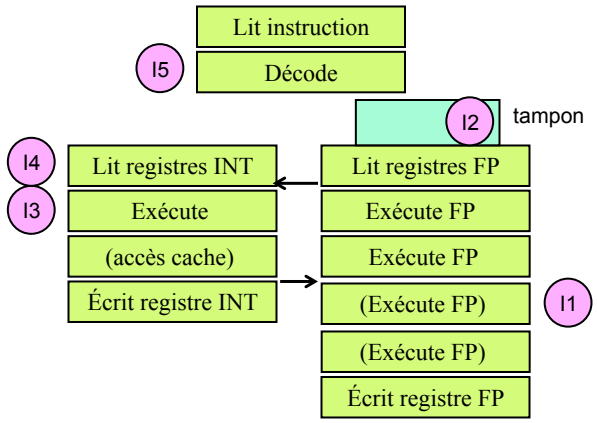
- Quand on programme avec un soucis de performance, on a rarement à se préoccuper des branchements
  - La plupart des branchements sont bien prédits
  - Il peut arriver que les mauvaises prédictions de branchement dégradent la performance de manière importante, mais il est de toute manière difficile de savoir ce qui se passe exactement dans le prédicteur

Processeurs superscalaires

« superscalaire »

- Processeur superscalaire = processeur pouvant décoder plusieurs instructions du même *thread* par cycle (*thread* ~ programme)
  - Cela implique que l'IPC (instructions/cycle) peut être supérieur à 1
- Les processeurs généralistes actuels sont tous superscalaires

Sur l'exemple ci-contre, l'étage de décodage traite au plus une instruction par cycle ➔ ce n'est pas un pipeline superscalaire



Parallélisme d'instructions

- Le parallélisme d'instructions est une notion qualitative caractérisant la proportion d'instructions d'un programme à pouvoir être exécutées en parallèle
  - Remarque: la technique du pipeline est une forme d'exploitation du parallélisme d'instruction
- Si un programme comporte un nombre relativement important de suites d'instructions consécutives indépendantes, alors son parallélisme d'instructions est élevé
- Si des instructions exécutées consécutivement sont dépendantes et forment une seule longue chaîne de dépendance, le parallélisme d'instructions est faible
- Un processeur superscalaire n'est intéressant pour la performance que si les programmes comportent du parallélisme d'instructions

Différents types de superscalaires

- In-order
  - Les instructions sont lancées à l'exécution dans l'ordre séquentiel, c'est-à-dire l'ordre du programme
- Out-of-order
  - Les instructions peuvent être lancées dans un ordre différent de l'ordre séquentiel
- Degré 2,3,4,…
  - Un processeur superscalaire de degré 2 peut décoder 2 instructions par cycle
  - Un processeur superscalaire de degré 3 peut décoder 3 instructions par cycle
  - Etc…
- La complexité matérielle augmente avec le degré superscalaire
  - Un superscalaire de degré 4 est plus complexe qu'un superscalaire de degré 2, qui plus complexe qu'un processeur non-superscalaire (degré 1)
  - Le degré superscalaire des processeurs existants ne dépasse pas 6
  - Limiteurs principaux: nombre de ports sur les registres et complexité du mécanisme de bypass

Le plus simple: in-order de degré 2

- Doubler la largeur du pipeline
  - Lire 2 instructions par cycle (PC ← PC+8)
  - Décoder 2 instructions par cycle (il faut 2 décodeurs)
  - Lecture registres: 4 ports de lecture
  - Exécution: dupliquer les opérateurs
  - Cache de données: 2 ports de lecture/écriture
    - Pareil pour le TLB de données
  - Écriture registres: 2 ports d'écriture
- En pratique, on ne double que les ressources qui sont utilisées fréquemment
  - Exemple: 1 seul diviseur au lieu de 2
  - Cache de données: 1 seul port de lecture/écriture

Exemple

	Cycle N	Cycle N+1	Cycle N+2	
Lit instruction	I3 / I4	I3 / I4	I5 / I6	
Décode	I1 / I2	I2	I3 / I4	
Lit registres		I1	I2	
Exécute			I1	
(accès cache)				
Écrit registre				

Si au décodage on trouve que l'instruction I2 dépend de l'instruction I1, on retarde le lancement de I2 et on « gèle » le pipeline en amont du décodage

Exercice

- Processeur superscalaire in-order de degré 2
- On suppose qu'il y a une probabilité p = ½ qu'une instruction dépende de l'instruction précédente
- Lorsqu'on détecte ce cas au décodage, on retarde le lancement de la deuxième instruction de 1 cycle, et on gèle le pipeline en amont du décodage
- Question: quel est le débit maximum du pipeline en instructions par cycle (IPC) ?
- Réponse:
  - En moyenne, 1+p cycles par groupe de 2 instructions
  - ➔ IPC= 2 / (1+p) = 1.33
- Le débit réel d'un processeur superscalaire est généralement inférieur à sont degré superscalaire
  - Miss de cache, load-use, branchement mal prédit, dépendances RAW, etc…

Décaler les groupements ?

	Cycle N	Cycle N+1	Cycle N+2	
Lit instruction	I3 / I4	I5 / I4	I7 / I6	
Décode	I1 / I2	I3 / I2	I5 / I4	
Lit registres		I1	I3 / I2	
Exécute			I1	
(accès cache)				
Écrit registre				

- Si I2 dépend de I1 et si I3 ne dépend pas de I2, lancer I2 et I3 ensemble
- Exemple: probabilité p= ½ qu'une instruction dépende de l'instruction précédente
- ➔ la probabilité de lancer 2 instructions simultanément vaut 1-p
- ➔ IPC = p + 2(1-p) = 1.5

Tampon de chargement: exemple

Lit instruction	I3	I4		I5	I6		I7	I8				I9	I10		
Décode	I1	I2		I3	I2		I5	I4		I5	I6	I7	I8		
Lit registres				I1			I3	I2			I4	I5	I6		
Exécute							I1			I3	I2			I4	
(accès cache)										I1		I3	I2		
Écrit registre												I1			

Remarques

- Rôle du compilateur
  - Éviter de mettre 2 instructions dépendantes l'une à la suite de l'autre
  - Éviter de mettre l'une à la suite de l'autre 2 instructions utilisant une ressource non dupliquée
  - Exemple: si un seul port sur le cache de données, intercaler si possible une instruction entre des LOAD/STORE consécutifs
- Si on agrandit le tampon de chargement, il est possible de charger plus de 2 instructions par cycle ➔ permet d'absorber les bulles dues aux miss du BTB et, dans une certaine mesure, les miss du cache d'instructions
- Attention aux dépendances WAW
  - Cas de 2 instructions exécutées en parallèle et écrivant dans le même registre

Cache de données double ports

- Souhaitable pour degré superscalaire supérieur à 2
- On n'implémente que très rarement de véritables caches double ports
  - Étant donnée la taille d'un cache comparée à celle d'un banc de registres, l'impact d'un véritable 2ème port est trop coûteux en complexité matérielle, surface de silicium, énergie consommée, latence d'accès, …
- « faux » double port ➔ plusieurs méthodes
- Méthode possible: dupliquer le cache
  - Les 2 copies contiennent exactement les même données
  - Permet 2 LOAD à des adresses arbitraires en parallèle, mais un seul STORE (on écrit dans les 2 copies simultanément)
- Autre méthode: diviser la SRAM du cache en plusieurs bancs
  - Exemple 2 bancs: banc pour les adresses paires et banc pour les adresses impaires ➔ on peut accéder simultanément à une adresse paire et à une adresse impaire
  - Nécessite un mécanisme d'arbitrage en cas de conflit de banc
  - Conflits ne peuvent être détecté qu'après le calcul d'adresse ➔ complique la mise en oeuvre

Lecture des instructions

- Contrairement au cache de données, pas besoin de plusieurs ports sur le cache d'instructions
- On lit à chaque cycle un groupe d'instructions adjacentes en mémoire
  - Taille du groupe peut être plus grand que degré superscalaire
  - Exemple: groupe = demi-ligne de cache
- Tampon de chargement peut contenir un ou plusieurs groupes
  - Si pas de place dans le tampon pour mettre le groupe, différer de 1 ou plusieurs cycles (pendant ce temps le pipeline d'instructions continue de travailler)
- Quand il y a la place dans le tampon, lire le groupe correspondant au PC courant et prédire le PC suivant
  - Le prédicteur de branchement peut être indexé avec l'adresse du point d'entrée dans le groupe
  - Si le groupe ne contient pas de branchement, ou des branchements toujours non pris, c'est un miss BTB: PC ← PC + taille groupe
  - En cas de hit BTB, le BTB et les différents prédicteurs permettent d'identifier les instructions valides dans le groupe et de prédire le PC suivant

Préchargement (*prefetch*)

- Quand la localité spatiale est bonne, les miss de cache sont plus ou moins prévisibles
  - Exemple: miss à la ligne X, puis à la ligne X+1, puis X+2, etc…
- Mécanisme matériel de préchargement (*prefetch*) des lignes de cache
  - Met dans le cache les lignes dont le programme a besoin avant qu'il ne les demande
  - But: éviter les miss de cache
- Implémenté sur la plupart des processeurs généralistes actuels (in-order et out-of-order)
- Cache d'instructions (IL1) ➔ laisser le prédicteur de branchement prendre de l'avance sur le pipeline d'instructions
  - Mettre les PC générés par le prédicteur dans une file
  - Plus efficace si les accès L2 sont pipelinés (les miss IL1 sont pipelinés)
- Ne précharger que si la ligne n'est pas déjà dans le cache
  - Le mécanisme de préchargement doit avoir accès aux tags du cache

Préchargement (suite)

- Basé sur l'observation des requêtes cache récentes
  - cache à index physique ➔ on stoppe généralement le prefetch dès qu'on arrive à une frontière de page physique
- Exemple: prefetch dans cache L2
  - Maintenir une petite table de prefetch indexée avec le numéro de page
  - Chaque entrée de la table contient les adresses des 2 dernières lignes X et Y de la page demandées au cache L2 (demandes initiées par des LOAD)
  - Si cache L1 demande une ligne Z telle que (Z-Y)=(Y-X), alors le mécanisme précharge dans L2 la ligne Z+N*(Y-X)
  - Plus la latence d'un miss L2 est grande, plus N doit être grand afin que la ligne arrive dans le cache à temps
- Les requêtes de prefetch consomment de la bande passante
  - Les « vraies » requêtes ont priorité sur les requêtes de prefetch

Superscalaire in-order: exemple

- Superscalaire in-order degré 2, toutes ressources dupliquées
- Mécanisme de chargement parfait (prédicteur de branchement parfait, tampon de chargement toujours plein)
- 100 % hit dans cache de données

```
double y[1000] ;
double x = 0 ;
for (i=0; i<1000; i++)
      x = x + y[i] ;
```

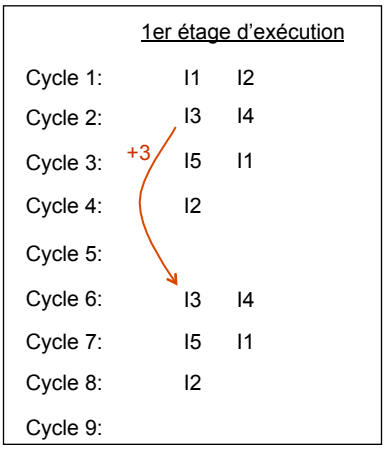
I1: f2 ← LOAD i1 +1 cycle
I2: f1 ← f1 FADD f2 +3 cycles
I3: i1 ← i1 ADD 8
I4: i2 ← i2 SUB 1
I5: BNZ i2,-4

	1er étage d'exécution	
Cycle 1:	I1	
Cycle 2:		
Cycle 3:	I2	I3
Cycle 4:	I4	
Cycle 5:	I5	I1
Cycle 6:		
Cycle 7:	I2	I3
Cycle 8:	I4	
Cycle 9:	I5	I1

➔ 4 cycles par élément

Changer l'ordre des instruction ?

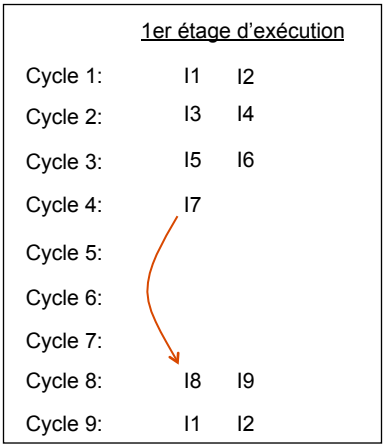
I1: i2 ← i2 SUB 1
I2: i1 ← i1 ADD 8
I3: f1 ← f1 FADD f2
I4: f2 ← LOAD i1
I5: BNZ i2,-4



➔ Parfois ça marche, mais pas ici: toujours 4 cycles par élément
➔ c'est le FADD qui limite la performance

Dérouler la boucle pour trouver plus de parallélisme ?

I1: f2 ← LOAD i1
I2: f3 ← LOAD i1+8
I3: f4 ← LOAD i1+16
I4: i2 ← i2 SUB 3
I5: i1 ← i1 ADD 24
I6: f5 ← f1 FADD f2
I7: f6 ← f3 FADD f4
I8: f1 ← f5 FADD f6
I9: BNZ i2,-8



➔ 8 cycles pour 3 éléments, soit 2.7 cycles par élément

Sur cet exemple, si on veut exploiter les ressources au maximum, il faut augmenter le parallélisme d'instructions

I1: f2 ← LOAD i1

I2: f3 ← LOAD i1+8

I3: f4 ← LOAD i1+16

I4: i2 ← i2 SUB 3

I5: i1 ← i1 ADD 24

I6: f1 ← f1 FADD f2

I7: f5 ← f5 FADD f3

I8: f6 ← f6 FADD f4

I9: BNZ i2,-8

f5 ← f5 FADD f6

f1 ← f1 FADD f5

	1er étage d'exécution	
Cycle 1:	I1	I2
Cycle 2:	I3	I4
Cycle 3:	I5	I6
Cycle 4:	I7	I8
Cycle 5:	I9	I1
Cycle 6:	I2	I3
Cycle 7:	I4	I5
Cycle 8:	I6	I7
Cycle 9:	I8	I9

➔ 9 cycles pour 6 éléments, soit 1.5 cycles par élément

In-order de degré supérieur à 2

- Exemples:
  - Alpha 21164 (1995) ➔ in-order degré 4
  - IBM POWER6 (2007) ➔ in-order degré 5
  - Intel Itanium ➔ in-order degré 6
- Degré 4 ➔ registres à 8 ports de lecture et 4 ports d'écriture
- Cache de données ➔ 2 ports de lecture souhaitables
- En pratique, seules les ALUs sont complètement répliquées
  - ALU = unité arithmétique et logique (ADD,SUB,OR,XOR,AND, …)
- Nécessite un compilateur très performant capable d' « exposer » le parallélisme d'instructions

Exposer le parallélisme d'instructions

- Le compilateur peut améliorer la performance en augmentant le parallélisme d'instructions exploitable par le processeur
- Mais il y a des contraintes
  - Respecter la sémantique du programme étant données les options spécifiées à la compilation
    - Le compilateur n'a pas nécessairement le droit de supposer l'associativité des opérations en virgule flottante, sauf si ce droit est accordé explicitement par une option de compilation
    - Attention aux instructions susceptibles de déclencher une exception
  - Ne pas augmenter le nombre d'instructions exécutées
    - Une « optimisation » qui augmente de manière significative le nombre d'instructions exécutées est une optimisation douteuse
  - Attention à la localité des accès mémoire
  - Le nombre de registres disponibles est limité
  - Les branchements limitent les possibilités de « bouger » les instructions
  - Certaines dépendances mémoire sont impossibles à détecter à la compilation

Bloc de base

- Bloc de base = instructions consécutives en mémoire avec un seul point d'entrée et un seul point de sortie
- Chaque instruction du programme appartient à un bloc de base et un seul
- Un bloc de base ne peut contenir qu'un seul branchement, à la fin du bloc
- La destination d'un saut ne peut se faire qu'au début d'un bloc de base
  - ➔ Certains blocs de base ne se terminent pas sur un branchement mais parce que l'instruction suivante en mémoire est la destination potentielle d'un saut
- Exemple

```
do {
   if (x[n]==0)  {
      a = a + 1;
   }
   n = n-1;
} while (n != 0);
```

```
boucle:  r2 ← r5 SHL 2
         r1 ← r0 ADD r2
         r2 ← LOAD r1
         BNZ r2, suite
                                  Bloc de base 1
         r3 ← r3 ADD 1
                                  Bloc de base 2
suite:   r5 ← r5 SUB 1
         BNZ r5, boucle
                                  Bloc de base 3
```

Scheduling d'instructions

- Le compilateur peut changer l'ordre des instructions au sein d'un bloc de base pour essayer d'augmenter le parallélisme d'instruction
- Contraintes
  - Les branchements ne bougent pas et restent à la fin de leurs blocs
  - Il faut respecter les dépendances RAW
  - Le nombre de registres disponibles est limité
  - Il faut respecter l'ordre des LOAD et STORE
    - Sauf si on est capable de déterminer à la compilation que les adresses sont différentes. Exemple: STORE R1,R2+1 et R1 ← LOAD R2+3
- Trouver un schedule optimal connaissant les dépendances entre instructions et les caractéristiques de la microarchitecture (latences des opérations, nombre de ressources, …) est NP-difficile dans le cas général ➔ on utilise des heuristiques

« Déroulage » de boucle

Loop unrolling

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

⇒

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

- Permet de former des blocs de base plus grands et ainsi d'exposer (en général) plus de parallélisme d'instructions
- Le facteur de déroulage optimal dépend des latences des instructions
  - Des instructions de grande latence peuvent nécessiter un grand facteur de déroulage pour obtenir de bonnes performances

« Pipeline » logiciel

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

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

- On essaye d'augmenter le parallélisme dans le bloc de base en « pipelinant » les itérations

Inlining de fonction

```
int add (int x, int y)  {
    return x + y ;
}

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

```
int add (int x, int y)  {
    return x + y ;
}

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

- Permet d'éliminer le coût de l'appel de fonction
- Permet de former des blocs de base plus grands

Sortir les branchements des boucles

(si possible)

```
for (i=0; i<100; i++)  {
    if (x==0) {
        a[i] = b[i] + c[i];
    } else {
        a[i] = b[i];
    }
}
```

```
if (x==0) {
    for (i=0; i<100; i++)
        a[i] = b[i] + c[i];
} else {
    for (i=0; i<100; i++)
        a[i] = b[i];
}
```

Loop unswitching

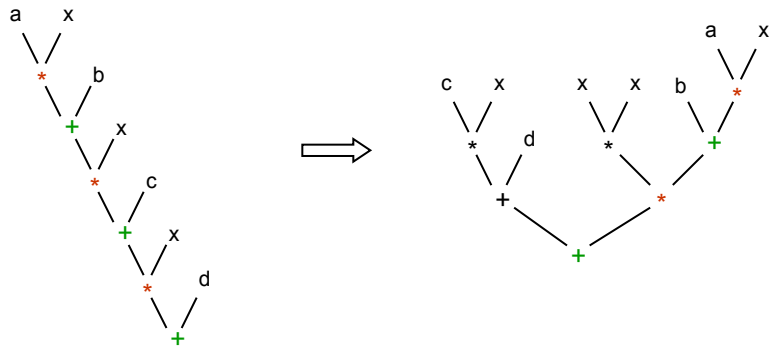
Attention à la taille du programme

- Les transformations augmentant la taille des blocs de base (déroulage, inlining, …) augmentent généralement la taille du programme
- Pas un problème tant que le programme tient dans le cache d'instructions
- Peut devenir un problème si la taille du programme dépasse la taille des caches
  - gcc –O2 n'applique ni inlining ni déroulage de boucle
- Il vaut mieux n'appliquer ces transformations qu'aux parties du programme qui contribuent de manière importante au temps d'exécution total
  - Exemple: boucles les plus internes

Exploiter les propriétés des opérations

(c'est généralement le programmeur qui doit le faire explicitement)

$((ax+b)x+c)x+d$ $\qquad$ $(ax+b)(x^2)+(cx+d)$



Maintenant il y a du parallélisme, le compilateur n'a plus qu'à faire un « bon » schedule

Le compilateur a-t-il le droit de faire cette transformation automatiquement ?

```
typedef double mat [1000][1000];

void mulmat(mat a, mat b, mat c, int n)
{
    int i,j,k;
    for (i=0; i<n; i++)
       for (j=0; j<n; j++)   {
          c[i][j] = 0;
          for (k=0; k<n; k++)
             c[i][j] += a[i][k] * b[k][j];
       }
 }
```

```
typedef double mat [1000][1000];

void mulmat(mat a, mat b, mat c, int n)
{
    int i,j,k;
    for (i=0; i<n; i++)   {
       for (j=0; j<n; j++)  c[i][j] = 0
       for (k=0; k<n; k++)
          for (j=0; j<n; j++)
             c[i][j] += a[i][k] * b[k][j];
    }
}
```

➔ Le comportement de mulmat(a,b,a,100) est-il le même dans les 2 cas ?

Le compilateur a-t-il le droit de faire cette transformation automatiquement ?

```
void truc(int n)  {
    int i
    for (i=0; i<n; i++)  {
        if ((100 % n)==0)  {
            a[i] = 0 ;
        } else {
            a[i] = 1 ;
        }
    }
}
```

```
void truc(int n)  {
    int i
    if ((100 % n)==0)  {
        for (i=0; i<n; i++)
            a[i] = 0 ;
    } else {
        for (i=0; i<n; i++)
            a[i] = 1;
    }
}
```

➔ que se passe-t-il si n=0 ?

Exécution conditionnelle

```
if (x >= 0 ) {
   a++;
}
```

Certains jeux d'instructions offrent le MOV conditionnel

Le jeu d'instructions Intel IA-64 généralise le principe en offrant la prédication (= exécution conditionnelle)

avec un branchement

```
BLZ r1,4
r2 ← LOAD r0
r2 ← r2 ADD 1
STORE r2, r0
```

MOV conditionnel

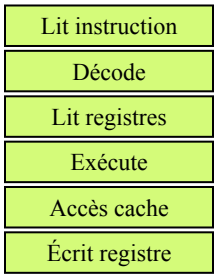
```
r4 ← SGE r1
r2 ← LOAD r0
r3 ← r2 ADD 1
If r4: r2 ← MOV r3
STORE r2, r0
```

prédication

```
r4 ← SGE r1
If r4: r2 ← LOAD r0
If r4: r2 ← r2 ADD 1
If r4: STORE r2, r0
```

Si le prédicat est faux, l'instruction se comporte comme un NOP

If r4: r2 ← MOV r3



Lit r3 et r4

Si r4 = 0, transforme l'instruction en NOP

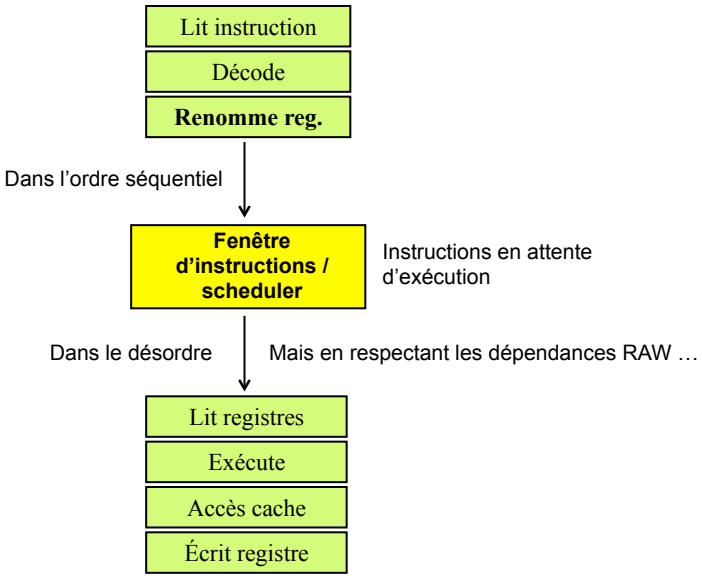
Soit NOP, soit écrit r3 dans r2

- L'exécution conditionnelle transforme les dépendances de contrôle en dépendances de donnée
- Intéressant sur processeur superscalaire in-order ➔ permet de créer des blocs de base plus grands
- Attention: cela augmente le nombre d'instructions décodées

Superscalaire out-of-order (OoO)

- Les instructions sont toujours lues et décodées dans l'ordre séquentiel, mais elles peuvent être lancées à l'exécution dans un ordre différent de l'ordre séquentiel
- Superscalaire OoO partiel
  - Différentes formes possibles. Exemple: pipeline INT et pipeline FP, lit et décode 2 instructions par cycle ➔ lance 2 instruction simultanément si l'une INT et l'autre FP
  - Rôle du compilateur important
- Superscalaire OoO complet ➔ capable de lancer simultanément et dans le désordre plusieurs instructions INT et/ou plusieurs instructions FP
  - Tous les processeurs Intel x86 depuis le Pentium Pro (« P6 », 1995)
  - Tous les processeurs AMD depuis le K5 (1996)
  - Processeurs PowerPC et IBM POWER (sauf le récent POWER6, qui est OoO partiel)
  - Années 90: Alpha 21264, MIPS R10000, HP-PA
  - …
- Mise en œuvre matérielle très complexe

Fenêtre d'instructions



Renommage de registre

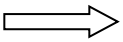
- Dans un superscalaire OoO (complet), on s'affranchit des dépendances WAR et WAW grâce au renommage de registre
- Pourquoi ?
  - Le nombre de registres offerts par le jeu d'instructions est limité
  - Quand il a besoin d'un registre pour stocker une valeur, le compilateur utilise un registre contenant une valeur « morte » → cela crée des dépendance WAR et WAW
  - Si on s'oblige à respecter les dépendances WAR et WAW, cela limite les possibilités d'exécution dans le désordre
- Le renommage de registre est un mécanisme matériel permettant d'avoir un plus grand nombre de registres microarchitecturaux (registres physiques) que de registres architecturaux
- À un instant donné, chaque instruction dans la fenêtre d'instructions écrit dans un registre physique distinct

Renommage: exemple

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

```
I1: r5 ← LOAD r2+r1
I2: r6 ← LOAD r3+r1
I3: r5 ← ADD r5, r6
I4: STORE r5, r4+r1
I5: r1 ← r1 SUB 8
I6: BGZ r1, @I1
```

La boucle est déroulée à l'intérieur de la fenêtre d'instructions



Le renommage a supprimé les dépendances WAR et WAW

```
I1: p9 ← LOAD p2+p8
I2: p10 ← LOAD p3+p8
I3: p11 ← ADD p9, p10
I4: STORE p11, p4+p8
I5: p12 ← p8 SUB 8
I6: BGZ p12, @I1
I1: p13 ← LOAD p2+p12
I2: p14 ← LOAD p3+p12
I3: p15 ← ADD p13, p14
I4: STORE p15, p4+p12
I5: p16 ← p12 SUB 8
I6: BGZ p16, @I1
```


Chemin critique

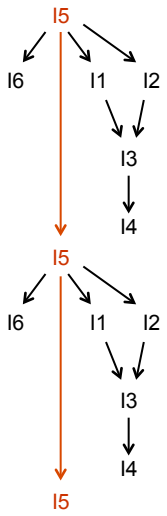
Chemin critique = chemin le plus long dans le graphe de dépendance de données

Sur l'exemple, il y a une seule instruction sur le chemin critique, l'instruction I5

Quand il y a suffisamment de ressources, c'est la durée totale du chemin critique dans la fenêtre d'instructions qui détermine la performance

Exemple: on peut, s'il y a suffisamment de ressources, exécuter en parallèle l'instruction I5 de l'itération n, I6,I1 et I2 de l'itération n-1, I3 de l'itération n-3 et I4 de l'itération n-4

➔ À condition que la fenêtre soit suffisamment grande pour contenir simultanément les instructions de 5 itérations consécutives



Taille de la fenêtre

- Plus la fenêtre d'instructions est grande, plus le processeur a des chances de trouver suffisamment de parallélisme d'instructions pour saturer ses ressources d'exécution
- Mais attention, la fenêtre d'instructions n'est pas juste un tampon
- ➔ La fenêtre d'instructions est associée à un mécanisme appelé *scheduler d'instructions*
  - Mécanisme matériel faisant le travail de schedule d'instructions que ferait le compilateur sur un superscalaire in-order
  - Mécanisme très complexe, très difficile à pipeliner, c'est un point chaud du processeur
- ➔ Comme chaque instruction de la fenêtre écrit dans une registre physique distinct, le nombre de registres physiques doit être au moins égal à la somme du nombre de registres architecturaux et du nombre d'entrées de la fenêtre d'instructions
  - La SRAM des registres physiques est plus grande qu'une SRAM de registres architecturaux ➔ difficile d'avoir beaucoup de ports si il y a un grand nombre de registres physiques
- Sur les processeurs actuels, on sait faire des fenêtres de plusieurs dizaines d'instructions

Comme un récipient percé…

- Le récipient, c'est la fenêtre d'instructions
- La hauteur d'eau *h*, c'est le nombre d'instructions en attente d'exécution
- Le débit auquel l'eau s'écoule du trou au fond du récipient, c'est la performance (IPC: nombre d'instructions exécutées par cycle)
- Le débit avec lequel on remplit le récipient, c'est le degré superscalaire (nombre d'instructions décodées par cycle)
- Plus il y a d'instructions dans la fenêtre, plus la performance est grande. Si on veut avoir le maximum de performance, il faut que la fenêtre soit pleine ➔ nécessite un bon prédicteur de branchements

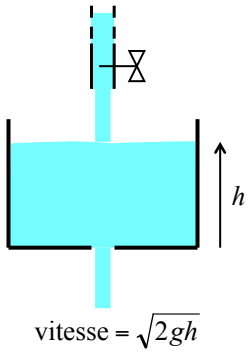


Table de renommage de registre

- Une fois les registres renommés, on peut exécuter les instructions dans le désordre
  - attention aux load/store cependant (cf. plus loin dans le cours)
- On utilise une table de renommage, consultée et mise à jour au décodage
  - maintient la correspondance entre registres architecturaux et registres physiques
  - pour les opérandes: on accède la table avec les numéros de registres architecturaux, et elle renvoie les numéros de registres physiques correspondant
  - la table est mise à jour sur chaque instruction écrivant dans un registre
    - on affecte un registre physique libre au registre architectural destination de l'instruction
  - en cas de branchement mal prédit ou d'exception, on répare la table de renommage
- Exemple de table de renommage
  - 1 entrée par registre architectural, chaque entrée contient un numéro de registre physique

Mise en œuvre OoO

- Mécanisme microarchitecturaux pour l'exécution OoO
  - *reorder buffer* (ROB)
  - table de renommage
  - points de reprise (checkpoint)
  - scoreboard
  - Scheduler d'instructions
- Points à préciser
  - comment réparer l'état du processeur en cas de branchement mal prédit ou d'exception ?
  - quand un registre physique devient-il libre ?
  - Comment savoir quand une instruction est prête à être lancée ?
  - comment traiter le cas des load/store ?

235

Reorder buffer (ROB)

- Reorder buffer = tampon matérialisant la fenêtre d'instructions
  - Mémoire *first-in first-out* (file circulaire avec pointeur de tête et pointeur de queue)
- Le ROB maintient l'ordre séquentiel des instructions ➔ offre un support pour les exceptions/interruptions précises
  - Vue de l'extérieur, l'exécution parait séquentielle
- Les instructions sont insérées dans le ROB au décodage, dans l'ordre séquentiel, et sont retirées du ROB après l'exécution, dans l'ordre séquentiel
  - à la fin du pipeline, il y a un (ou plusieurs) étage de retirement pour traiter ce qui doit être fait dans l'ordre et/ou non spéculativement
    - détection des exceptions
    - écritures mémoire
    - accès mémoire non-cachables (entrées/sorties)
  - pour pouvoir être retirée, une instruction doit attendre que toutes les instructions plus anciennes qu'elle soient exécutées et retirées
- Exemple Intel Core: ROB de 96 entrées (96 micro-ops)

236

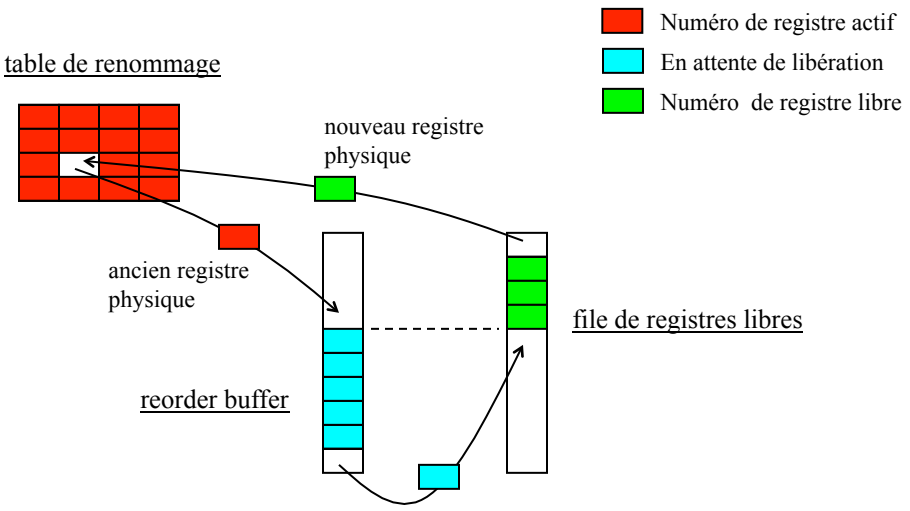
Renommage: mise en oeuvre

- Les processeurs OoO n'utilisent pas tous la même méthode de renommage de registres
- Certains processeurs utilisent le renommage « permanent »
  - Exemple: années 90 ➔ MIPS R1000, Alpha 21264
- Certains processeurs utilisent le renommage « temporaire »
  - Exemple: Intel Pentium Pro (1995), et aussi très probablement tous les processeurs x86 récents (Intel et AMD)

Renommage « permanent »

- 1 banc de registres physiques (en fait, 2 bancs: 1 entier + 1 flottant)
  - on affecte un registre physique libre à chaque nouvelle instruction écrivant dans un registre
- Taille du banc de registres = nombre de registres architecturaux + nombres d'entrées du ROB
  - ex. MIPS R10000: ROB 32 entrées, 32 registres architecturaux entiers, 64 registres physiques entiers (pareil pour les flottants)
- Un registre physique P associé au registre architectural R est libéré lorsqu'une nouvelle instruction I écrit dans R et que cette instruction est retirée du ROB
  - lorsque I est renommée, le registre P est mis en attente de libération dans le ROB
  - lorsque I est retirée, on est sûr qu'on n'aura plus besoin de la valeur, et on peut en toute sécurité libérer P, qui pourra être réutilisé par une autre instruction

Cycle des registres physiques



Renommage « temporaire »

- 1 banc de registres architecturaux + 1 banc de registres temporaires
  - le nombre de registres temporaires est égal au nombre d'entrées du ROB
  - En pratique, registres temporaires et ROB peuvent être fusionnés: 1 registre temporaire dans chaque entrée du ROB
- Tant que l'instruction est dans le ROB, registre physique = registre temporaire
  - à l'exécution, l'écriture registre se fait dans le registre temporaire
- Une fois l'instruction retirée du ROB, registre physique = registre architectural
  - la valeur stockée dans le registre temporaire est recopiée dans le banc de registres architecturaux, et le registre temporaire est libéré en même temps que l'entrée ROB correspondante
  - une écriture associative dans le ROB est nécessaire
    - si la lecture des registres est faite avant l'entrée dans le ROB, il faut, à l'exécution, propager la valeur aux autres instructions dans le ROB
    - si la lecture des registres est faite au lancement de l'exécution, il faut, au retirement, indiquer aux instructions dans le ROB que la valeur se trouve désormais dans le banc de registres architecturaux

Exécution spéculative

- L'exécution dans le désordre conduit à exécuter certaines instructions spéculativement
- Une mauvaise spéculation se produit lorsqu'une instruction Y est exécutée avant une instruction X qui précède Y dans l'ordre séquentiel et que l'instruction X est un branchement potentiellement mal prédit ou est une instruction susceptible de déclencher une exception (par exemple un LOAD/STORE)
- Sur une mauvaise spéculation, il faut annuler les instructions qui se sont exécutées spéculativement et annuler les modifications faites par ces instructions

Réparation sur exception

- Les exceptions sont déclenchées au retirement, quand l'instruction fautive est la plus vieille instruction dans le ROB
- La table de renommage doit retrouver l'état qu'elle avait avant que l'instruction fautive soit décodée
- Renommage « permanent »: plusieurs solutions
  - dépiler les registres physique du ROB et les remettre dans la table (MIPS R10000) ➔ prend plusieurs cycles
  - maintenir une copie non-spéculative de la table au retirement
  - points de reprise (checkpoint) sur chaque instruction (voir plus loin)
- Renommage « temporaire »
  - l'état non spéculatif est contenu dans le banc de registres architecturaux
- Effacer les instructions spéculatives (en particulier, vider le ROB), libérer les registres physiques spéculativement alloués, puis passer la main au système

Réparation sur branchement mal prédit

- Remettre la table de renommage dans l'état où elle était avant qu'on décode le branchement et effacer les instructions qui ont été chargées dans le processeur après le branchement mal prédit
- Solution 1: réparer la mauvaise prédiction au retirement du branchement, comme pour les exceptions
  - entre l'exécution d'un branchement et son retirement, il peut s'écouler plusieurs cycles, ce qui rallonge la pénalité de mauvaise prédiction
- Solution 2: réparer la mauvaise prédiction dès que le branchement est exécuté
  - intéressant à condition d'être capable de réparer la mauvaise prédiction rapidement
  - solution: établir des points de reprise (checkpoint)

243

Points de reprise (*checkpoint*)

- Checkpoint = « photographie » de l'état du processeur avant de décoder un nouveau branchement
  - en particulier, état de la table de renommage
- Mémoire de checkpoint = mémoire microarchitecturale où on sauvegarde les checkpoints
  - Pas nécessairement une mémoire centralisée, les informations constituant le checkpoint peuvent être physiquement stockées à des endroits divers
- Si le branchement est mal prédit, le point de reprise associé est utilisé pour restaurer l'état du processeur
- Le nombre de branchements dans le ROB est limité par le nombre de points de reprise disponibles
  - ex. MIPS R10000: ROB 32 entrées, 4 points de reprise
  - quand il n'y a plus de points de reprise disponibles pour un nouveau branchement, on suspend le décodage et on attend qu'un branchement soit retiré et libère un point de reprise

244

Scoreboard

- Pour savoir si les opérandes d'une instruction sont disponibles, on consulte le scoreboard
- Scoreboard = table de bits de présence (cf. transparent N° 100)
- Au lieu d'avoir un bit de présence par registre architectural, on a un bit de présence par registre physique
- Bit à 1 indique que la valeur du registre physique est disponible
- Quand une instruction est insérée dans le ROB, le bit correspondant au registre physique destination de l'instruction est mis à 0
- Quand l'instruction est exécutée, le bit est mis à 1

Scheduler d'instructions

- Scheduler = tampon + circuits pour sélectionner les instructions à exécuter
- En même temps que l'instruction est insérée dans le ROB, une version exécutable de l'instruction est insérée dans le scheduler
- L'instruction attend dans le scheduler jusqu'à ce que ses opérandes et les ressources d'exécution nécessaires soient disponibles
  - le scheduler « réveille » les instructions dont les opérandes sont prêts et décide quelles instructions doivent être lancées sur quelles unités
  - arbitrage entre instructions en cas de conflit de ressource
- Une fois exécutée, l'instruction est en général retirée du scheduler d'instructions sans attendre son retirement du ROB
  - Dans ce cas, le tampon du scheduler comporte moins d'entrées que le ROB
- Sur certains processeurs, le ROB et le scheduler d'instructions sont confondus

Différents types de scheduler

- Scheduler unifié
  - Un seul scheduler dans le processeur
  - Exemple Intel Core : un seul scheduler de 32 entrées (32 micro-ops)
  - Exemple PA6T: un seul scheduler de 64 entrées (confondu avec le ROB)
- Scheduler par type d'instructions
  - Scheduler entier, scheduler flottant, scheduler mémoire
- Scheduler par unité d'exécution
  - aussi appelé *station de réservation*
  - L'arbitrage des ressources est simplifié
- Exemple: AMD Opteron
  - 1 scheduler flottant de 36 entrées (36 micro-ops)
  - 1 scheduler entier de 24 entrées divisé en 3 stations de 8 entrées chacune

File de load/store

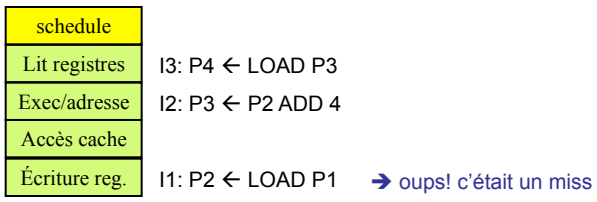
- Les LOAD/STORE sont insérés simultanément dans le ROB et dans une file de load/store dans l'ordre séquentiel
  - File de load/store aussi appelée *Address Reorder Buffer,* ou encore *Memory Ordering Buffer* (MOB)
- Les LOAD/STORE sont retirés de la file dans l'ordre séquentiel
- Rôle de la file
  - tamponner les STORE : un store n'a le droit d'écrire dans les caches ou en mémoire qu'après qu'il a été retiré du ROB ➔ les écritures mémoire sont non spéculatives
  - bypass mémoire: si un load essaie de lire une donnée alors que le store qui écrit cette donnée se trouve encore dans le tampon, possibilité d'obtenir la valeur directement sans attendre qu'elle soit écrite dans le cache
  - La file de load/store assure le respect des dépendances mémoire
- Exemples:
  - AMD Opteron ➔ file de load/store de 44 entrées
  - PA6T ➔ file de load/store de 32 entrées

Dépendances mémoire

- Plusieurs solutions possibles pour forcer le respect des dépendances mémoire
- Solution « simple » : ne lancer un LOAD que lorsque les adresses de tous les STORE précédant le LOAD (dans l'ordre séquentiel) sont connues
  - ➔ On fait parfois attendre certains LOAD plus que nécessaire
  - On peut améliorer la performance si on autorise un STORE à s'exécuter partiellement afin que l'adresse soit calculée même si la valeur à écrire n'est pas encore disponible
- Solution « performante »: LOAD spéculatif avec mécanisme de réparation
  - Utilisé sur Intel Core
  - Un load peut s'exécuter spéculativement sans attendre que les adresses de tous les STORE précédant le LOAD soient calculées
  - Lorsque l'adresse d'un STORE est calculée, on vérifie que l'adresse est différente de celles des LOAD exécutés spéculativement par rapport à ce STORE
  - En cas de mauvaise spéculation, on répare l'état du processeur, comme sur une mauvaise prédiction de branchement
  - Prédicteur de dépendance ➔ un LOAD qui a été exécuté trop tôt sera exécuté non spéculativement la prochaine fois, afin d'éviter la pénalité de mauvaise spéculation

Relance d'instructions (*replay*)

- Pour de bonnes performances, le scheduler d'instructions est capable de lancer des instructions sans attendre que les instructions dont elles dépendent soient terminées (on exploite le mécanisme de bypass)



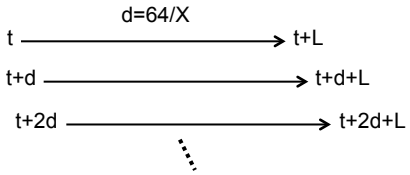
- Si miss de cache: les instructions I1,I2,I3 devront être relancées (*replay)*
  - Elles sont marquées « non-exécutées » dans le scheduler
  - En fait, il n'est pas si facile d'identifier rapidement les instructions dépendant de I1 de celles n'en dépendant pas ➔ il est plus simple de relancer toutes les instructions lancées après I1

File de requêtes de miss

- File de miss = liste d'adresses de ligne sur lesquelles on a fait un miss
  - Généralisation de l'idée présentée dans le transparent N° 133
- Quand un LOAD/STORE fait un miss dans DL1, on met l'adresse de la ligne manquante dans la file de miss
  - Sauf si une entrée dans la file a déjà été allouée pour cette adresse par un LOAD/ STORE antérieur
- La file de miss a son propre scheduler qui organise les accès au cache L2
  - Par exemple, premier arrivé premier servi
- Sur les processeurs modernes, les miss sont pipelinés
  - ➔ les latences de miss se recouvrent partiellement
  - Exemple: sur le PA6T, on peut avoir jusqu'à 16 miss en cours simultanément
- Quand une requête de miss est satisfaite (la ligne manquante a été chargée complètement ou partiellement dans le DL1), la file de miss doit informer le scheduler que les LOAD dépendant des données chargées peuvent être relancés

Exercice

- Lignes de cache de 64 octets
- Latence mémoire (DRAM) de L cycles
- Bande passante mémoire de X octets/cycle
- Calculer en fonction de L et X le nombre idéal N d'entrées de la file de miss
  - La valeur de N idéale est la plus petite valeur qui permet de saturer la bande passante lorsque la file de miss est pleine



- Réponse
  - Quand la bande passante est saturée, on lance une requête tous les d=64/X cycles
  - Entre le début et la fin d'un miss, on a le temps de lancer (L/d)-1 autres requêtes
  - On a donc N=L/d = (L*X)/64 requêtes simultanément en cours

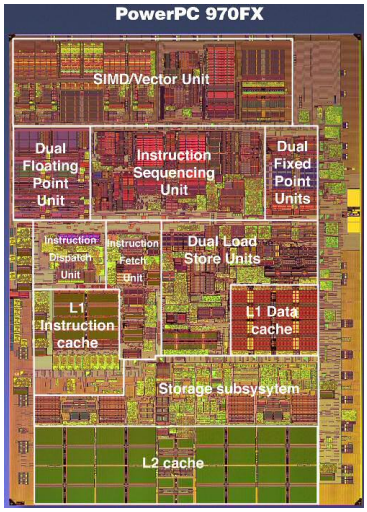
$$\text{Nombre de Requêtes} = \frac{\text{Latence} \times \text{Bande Passante}}{\text{Taille Ligne}}$$

(cf. loi de Little)

- Autre exemple:
  - Lignes 64 octets
  - Cache L2: pipeliné, latence 16 cycles, bande passante 16 octets/cycle
  - Mémoire: pipelinée, latence 128 cycles, bande passante 8 octets/cycle
  - Reorder buffer (ROB) de 64 instructions
  - On suppose qu'on fait dans le pire des cas 1 miss DL1 toutes les 8 instructions
  - On suppose que la file de miss est suffisamment grande
  - Question: sans considérer le prefetch, les bandes passantes sont elle correctement dimensionnées ?

- Réponse
  - Nombre maximum de requêtes dans le ROB → 64/8 = 8
  - Requêtes pour saturer la bande passante L2 → (16x16)/64 = 4
  - Requêtes pour saturer la bande passante mémoire → (128x8)/64 = 16
  - On peut saturer la bande passante L2
  - La bande passante mémoire est surdimensionnée

Exemple: l'IBM PowerPC 970FX



- Technologie 90 nm
- 58 millions de transistors
- Taille: 9.4 mm × 7.1 mm
- Fréquence 2 GHz
- Superscalaire degré 5, out-of-order
- 80 registres physiques entiers + 80 registres physiques flottants
- Ligne cache 128 octets
- Cache IL1: 64 Ko, direct-mapped
- Cache DL1: 32 Ko, 2-way set-associatif LRU 2 ports lecture 1 port écriture
- Cache L2: 512 Ko, 8 way set-associatif LRU
- Prédicteur de branchement hybride (3x16k entrées)
- Branchement mal prédit ➔ 12 cycles de pénalité (minimum)

Superscalaire OoO: en résumé

- Sur un processeur OoO, le scheduling d'instructions est fait par le matériel
  - Le scheduling d'instructions fait par le compilateur est en général inutile, sauf si le compilateur travaille sur une « fenêtre » d'instructions plus grande que celle du processeur (déroulage de boucle, pipeline logiciel, inlining, …)
- Ce que le programmeur peut faire (ou essayer de faire) pour améliorer la performance lorsque nécessaire
  - Augmenter le parallélisme d'instructions en diminuant la longueur du chemin critique dans le graphe de dépendance de données
  - Bien utiliser les caches et le prefetch (localité temporelle, localité spatiale)
  - Lire la documentation du constructeur qui fournit des indications aux programmeurs

La quête de la performance

- Avant de se préoccuper du processeur, il faut écrire des algorithmes efficaces dans le langage de programmation correspondant à ses besoins
- Ensuite, si on veut augmenter la performance, il faut utiliser un compilateur performant et utiliser les « bonnes » options de compilation
  - Cela dépend de la plateforme processeur concernée
- Si on veut encore plus de performance, alors on peut regarder les caractéristiques du processeur et les « lacunes » du compilateur
- La plupart des processeurs comportent des *compteurs de performance*
  - ➔ donnent de multiples informations sur l'exécution d'un programme
  - Exemple: nombre d'instructions exécutées, nombre de miss DL1, nombre de branchements mal prédits, etc…

Exemple:

- Intel Pentium 4 (3 Ghz)
  - gcc 4.1.1
  - Niveau d'optimisation –O3
  - Temps d'exécution 13.1 s
- Intel Core 2 Duo (2.33 Ghz)
  - gcc 4.0.1
  - Niveau d'optimisation –O3
  - Temps d'exécution 6.3 s

```
#include <stdio.h>

#define NN 5000000
#define NITER 200

double t[NN];

void f(double x, double *y1, double *y2)
{
  double y = ((*y1)+(*y2))*0.5;
  if ((y*y) < x) {
    *y1 = y;
  } else {
    *y2 = y;
  }
}

double rac(double x)
{
  int n;
  double y1,y2;
  y1 = 0;
  y2 = x;
  for (n=0; n<NITER; n++) {
    f(x,&y1,&y2);
  }
  return y2;
}

main()
{
  int i;
  for (i=0; i<NN; i++) t[i] = rac(i);
  printf("%f\n",t[(int)t[NN-1]]);
}
```

257

- On augmente le parallélisme d'instructions
- Pentium 4
  - Temps d'exécution 8.1 s
- Core 2 Duo
  - Temps d'exécution 3.0 s

```
#include <stdio.h>

#define NN 5000000
#define NITER 200

double t[NN];

void f(double x, double *y1, double *y2)
{
  double y = ((*y1)+(*y2))*0.5;
  if ((y*y) < x) {
    *y1 = y;
  } else {
    *y2 = y;
  }
}

void rac4(double xa, double xb, double xc, double xd,
          double *ya, double *yb, double *yc, double *yd)
{
  int n;
  double ya1,ya2,yb1,yb2,yc1,yc2,yd1,yd2;
  ya1 = yb1 = yc1 = yd1 = 0;
  ya2 = xa;
  yb2 = xb;
  yc2 = xc;
  yd2 = xd;
  for (n=0; n<NITER; n++) {
    f(xa,&ya1,&ya2);
    f(xb,&yb1,&yb2);
    f(xc,&yc1,&yc2);
    f(xd,&yd1,&yd2);
  }
  *ya = ya2;
  *yb = yb2;
  *yc = yc2;
  *yd = yd2;
}

main()
{
  int i;
  for (i=0; i<NN; i+=4)
    rac4(i,i+1,i+2,i+3,&t[i],&t[i+1],&t[i+2],&t[i+3]);
  printf("%f\n",t[(int)t[NN-1]]);
}
```

258

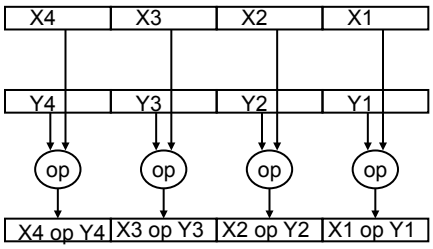
Quand les données ne tiennent pas dans les caches, il est parfois fructueux de regarder la localité spatiale et/ou la localité temporelle

- Core 2 Duo
- gcc –O3 liste.c –o liste –DPAS=1
  - Temps d'exécution ➔ 1.1 s
- gcc -O3 liste.c –o liste –DPAS=3
  - Temps d'exécution ➔ 3.2 s
- gcc -O3 liste.c –o liste –DPAS=5
  - Temps d'exécution ➔ 5.2 s
- gcc -O3 liste.c –o liste –DPAS=7
  - Temps d'exécution ➔ 7.3 s
- gcc –O3 liste.c –o liste –DPAS=15
  - Temps d'exécution ➔ 23.3 s
- gcc –O3 liste.c –o liste –DPAS=1001
  - Temps d'exécution ➔ 50 s

```
#include <stdio.h>

#define MM (1<<20)
#define NN (MM/2)

typedef struct noeud {
  int a;
  struct noeud *p;
} liste;

liste m[MM];
int k = 0;

liste * listalloc()
{
  k = (k+PAS) & (MM-1);
  return &m[k];
}

main()
{
  int i,j,x=0;
  liste *d = listalloc();
  liste *p = d;
  for (i=0; i<NN; i++) {
    p->a = i;
    p->p = NULL;
    if (i==(NN-1)) break;
    p->p = listalloc();
    p = p->p;
  }
  for (j=0; j<1000; j++) {
    p = d;
    while (p!=NULL) {
      x ^= p->a;
      p = p->p;
    }
  }
  printf("%d\n",x);
}
```

Extensions SIMD

- SIMD = single instruction multiple data
- ➔ fait plusieurs opérations simultanément avec une seule instruction
  - Exemple: registres 128 bits ➔ 4 opérations FP simple précision ou 2 opérations FP double précision
- Exemple: Intel MMX, Intel SSE, SSE2, SSE3, etc.



- Il faut que le compilateur soit capable de vectoriser le code automatiquement ou bien que le programmeur utilise explicitement les instructions SIMD

- Intel Core 2 Duo (2.33 Ghz), gcc 4.0.1
- gcc –O3 –funroll-loops ➔ temps d'exécution = 8.7 s
- gcc –O3 –funroll-loops –ftree-vectorize –msse ➔ temps d'exécution = 2.2 s
  - Le compilateur essaie de vectoriser le code automatiquement et si il réussit il utilise les instructions SIMD

```
#include <stdio.h>

#define NN 100

float a[NN];
float b[NN];

main()
{
  int i,j;
  for (i=0; i<NN; i++) {
    a[i] = 1;
    b[i] = 0;
  }
  for (i=0; i<50000000; i++) {
    for (j=0; j<NN; j++) {
      b[j] += a[j];
    }
  }
  printf("%f\n",b[((unsigned)random()) % NN]);
}
```

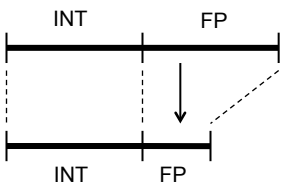
- On peut aussi vectoriser « à la main » si on veut …
- gcc –O3 –funroll-loops –msse ➔ temps d'exécution = 2.4 s

```
#include <stdio.h>
#include <xmmintrin.h>

#define NN 100

float a[NN] __attribute__ ((aligned(16)));
float b[NN] __attribute__ ((aligned(16)));

main()
{
  int i,j;
  __m128 x,y;
  for (i=0; i<NN; i++) {
    a[i] = 1;
    b[i] = 0;
  }
  for (i=0; i<50000000; i++) {
    for (j=0; j<NN; j+=4) {
      x = _mm_load_ps(&a[j]);
      y = _mm_load_ps(&b[j]);
      y = _mm_add_ps(y,x);
      _mm_store_ps(&b[j],y);
    }
  }
  printf("%f\n",b[((unsigned)random()) % NN]);
}
```


Accélération locale / globale

Exemple: supposons qu'un programme passe 50% du temps d'exécution dans du calcul en virgule flottante, et supposons qu'on multiplie par 2 les performances du calcul en virgule flottante. Quelle accélération globale peut-on espérer ?

$$T' = \frac{T}{2} + \frac{T}{2} \times \frac{1}{2} = \frac{3}{4}T \qquad \text{accélération globale} = \frac{T}{T'} = \frac{4}{3} = 1.33$$



Loi d'Amdhal

T = temps total d'exécution
F = fraction de l'exécution qui est accélérée
a = accélération locale
A = accélération globale

$$A = \frac{T}{T'} = \frac{T}{\frac{FT}{a} + (1-F)T}$$

$$\boxed{A = \frac{1}{\frac{F}{a} + (1-F)}}$$

$$\boxed{A < \frac{1}{1-F}}$$

- L'accélération globale ne peut pas dépasser 1/(1-F)

Le sens de la loi d'Amdahl

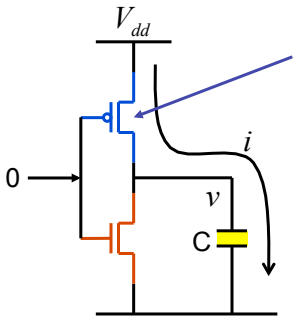
- Quand on veut améliorer les performances d'un programme, on cherche d'abord à savoir sur quelle partie du programme doit porter l'effort en priorité
  - On utilise pour cela un outil de *profilage* (exemple: *gprof*) qui donne les contributions de chaque fonction au temps d'exécution total
- Parfois, une fonction en particulier contribue de manière importante au temps d'exécution ➔ si on arrive à accélérer cette fonction, on peut obtenir une accélération globale significative
  - Mais attention, une fois qu'on a accéléré une fonction particulière, une autre fonction peut alors être le contributeur principal au temps d'exécution, et il faudra éventuellement travailler sur cette autre fonction, etc.
- Parfois, le temps d'exécution total est plus ou moins également réparti entre un grand nombre de fonctions ➔ accélérer le programme « à la main » va demander un effort important
  - Dans ce cas, à moins de trouver un meilleur compilateur, des options de compilation plus efficaces, ou de disposer d'outils faisant des optimisations automatiques de code, il est difficile d'obtenir des gains de performance importants

Consommation d'énergie

Consommation d'énergie / dissipation

- Contrainte majeure (technique et économique)
- Énergie (joules)
  - Coût (€,$) ➔ important pour les *data centers* (exemple: Google)
  - Systèmes alimentés sur batterie
    - on veut maximiser le temps de déchargement de la batterie
- Température
  - Température sur la puce: 85 à100 °C maxi pour un bon fonctionnement
  - Machine de bureau: coût de la machine, bruit du ventilateur
  - Système portable: fortes contraintes géométriques et de poids
- Puissance (watts)
  - Ampérage limité
  - *Data centers*: échauffement de la pièce par effet de masse
    - ➔ limite le nombre de processeurs par pièce

Consommation dynamique



$$\int_0^{\infty}(V_{dd}-v)i\,dt = \int_0^{V_{dd}}C(V_{dd}-v)\,dv = \boxed{\frac{1}{2}CV_{dd}^2}$$

L'énergie dissipée par transition de bit ne dépend que de la capacité et de la tension

On peut diminuer l'énergie consommée en réduisant l'échelle de gravure (C plus petit) et en réduisant la tension d'alimentation

Attention: la latence d'une porte logique est inversement proportionnelle à la différence entre V_{dd} et la tension de seuil du transistor

➔ baisser V_{dd} augmente la latence des portes

Consommation statique

- Les transistors ne sont pas des interrupteurs parfaits, il y a des courants de fuite
- ➔ Une porte consomme de l'énergie même lorsqu'elle ne travaille pas (pas de transition)
- 20 à 30 % de la consommation dans les processeurs des années 2000
  - C'est la consommation principale dans les caches de niveau 2 et 3
  - ➔ Pour les caches L2 et L3, on peut utiliser des transistors un peu plus lents mais qui ont des courants de fuite plus faibles
  - La situation s'est améliorée depuis l'introduction récente de la techno 45 nm d'Intel (high-K dielectric)
- Pour supprimer la consommation statique dans une partie de circuit, il faut la déconnecter de l'alimentation électrique
  - *Power gating*
  - Attention: les mémoires perdent leur contenu

Puissance électrique (watts)

$$P = \overbrace{a \times N \times C \times F \times V_{dd}^2}^{\text{dynamique}} + \overbrace{N \times I_{fuite} \times V_{dd}}^{\text{statique}}$$

- N = nombre de portes
  - On peut diminuer *N* en faisant un circuit plus simple
- a = fraction des portes qui changent d'état (0→1,1→0) par cycle d'horloge
  - On peut diminuer *a* en faisant du clock gating (cf. transparent suivant)
- C = capacité moyenne en sortie de porte
  - La réduction d'échelle de gravure diminue *C*
- F = fréquence d'horloge
- V_{dd} = tension d'alimentation
  - Réduire la tension diminue à la fois la consommation dynamique et la consommation statique
- I_{fuite} = courant de fuite moyen (augmente avec la température)

Clock gating

- Quand un circuit n'est pas utilisé, il consomme de l'énergie inutilement
- Exemples:
  - certains programmes n'utilisent pas les opérateurs virgule flottante
  - sur un accès mémoire long et bloquant, les opérateurs sont inutilisés durant l'accès
- *Clock gating*
  - Quand un circuit n'a pas été utilisé depuis quelques cycles, le signal d'horloge est déconnecté automatiquement de ce circuit
  - ➔ supprime les transitions non voulues, consommation dynamique minimale
  - Le circuit se « réveille » à la prochaine utilisation
    - Quasi immédiat, 1 ou 2 cycles

Tension / fréquence

- En théorie, grâce au clock gating, un processeur non utilisé pendant un certain temps ne devrait pas consommer d'énergie dynamique
- En pratique, le clock gating seul ne permet pas d'éliminer complètement la consommation dynamique d'un processeur. De plus, il y a toujours la consommation statique
- ➔ Quand le processeur n'a pas été actif pendant un certain temps, on peut obtenir une plus grande réduction de la consommation en diminuant simultanément la tension d'alimentation et la fréquence d'horloge
- ➔ permet de réduire à la fois la consommation dynamique et la consommation statique sans perdre le contenu des flip-flops et des diverses SRAMs (registres, caches, TLBs, prédicteurs de branchement, …)
- Cependant, revenir à la fréquence maximum prend un certain temps
  - Exemple: PA6T ➔ 0.5 millisecondes pour revenir à la fréquence maximum

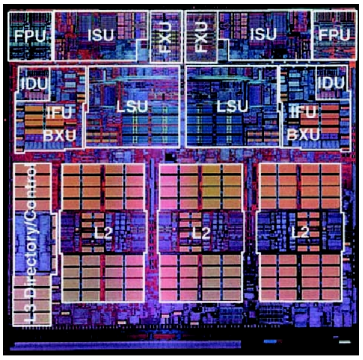
Température

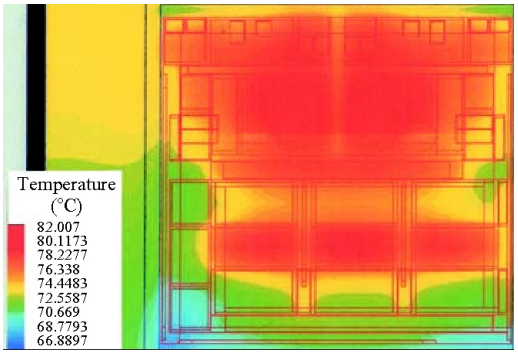
- Température nominale sur la puce = entre 85 et 100 °C
- Durée de vie du circuit dépend exponentiellement de la température
  - Un circuit en fonctionnement se dégrade avec le temps
    - Exemple: electromigration
  - Température élevée augmente la vitesse de dégradation
  - +10 °C ➔ durée de vie moyenne divisée par 2
- Le temps de réponse des circuits augmente avec la température
  - Augmentation de la résistivité des matériaux (silicium,cuivre)
- Les courants de fuite augmentent rapidement avec la température

La température sur la puce n'est pas uniforme

jusqu'à plusieurs dizaines de °C de différence d'un point à un autre

Exemple: IBM POWER4





Régulation puissance / température

- La puissance dissipée sur la puce processeur dépend de l'application qui s'exécute
  - Peut aller du simple au double, voire plus (« virus » thermique)
- La température dépend …
  - De l'application qui s'exécute
  - De la vitesse du ventilateur
  - De la température ambiante
- Capteurs de température et de puissance intégrés sur les processeurs récents. Lorsque risque de dépassement de la limite, le processeur est ralenti pour diminuer la consommation
  - Les processeurs récents comportent jusqu'à plusieurs dizaines de capteurs thermiques, placés aux endroits potentiellement chauds
- Différentes méthodes possibles:
  - Réduire la fréquence d'horloge
  - Réduire la fréquence d'horloge et la tension d'alimentation
  - *Clock gating* global periodique (on/off/on/off…)
  - Étage de lecture d'instructions du pipeline travaille un cycle sur 2

Parallélisme et consommation

1 opérateur	2 opérateurs
Fréquence F	Fréquence F/2 chacun
Tension V	Tension V/2
op	op op
Débit max = F op/s	Débit max = 2(F/2)= F op/s
Puissance dynamique ~ FV^2	Puissance dynamique ~ $2(F/2)(V/2)^2 = FV^2/4$

- Le parallélisme permet d'obtenir la même performance en consommant moins de puissance dynamique
- ➔ À condition qu'on baisse la tension et à condition qu'il y ait du parallélisme à exploiter dans le programme

Haute performance / faible consommation

- Certaines applications demandent à la fois une haute performance et une faible consommation électrique ➔ processeurs « embarqués » haute performance
  - Systèmes alimentés sur batterie
  - Contraintes géométriques et/ou économiques ne permettant pas un système de refroidissement performant
- Quand la contrainte énergétique est forte, le processeur doit être le plus simple possible
  - Processeurs embarqués ont généralement des contraintes de *time-to-market* ➔ dans le cas où les processeurs déjà existants ne conviennent pas, un processeur simple nécessite moins d'ingénieurs et prend moins de temps à développer
- SIMD très intéressant pour les applications embarquées multimédia
  - Avantage performance/consommation (moins d'instructions à exécuter pour un même nombre d'opérations)
- Différents choix possibles selon les besoins
  - Pipeline in-order, faiblement superscalaire (voire non superscalaire) avec des opérateurs SIMD
  - Processeur OoO low-power, exemple PA6T (low-power, ici, veut dire quelques watts)
  - VLIW

Processeur VLIW

- VLIW = *Very Long Instruction Word*

- Philosophie
  - On exploite le parallélisme en mettant plusieurs opérateurs
  - Contrairement au superscalaire, le matériel pour contrôler l'exécution est réduit à sa plus simple expression ➔ c'est le compilateur qui fait tout, ou presque
  - Pour un processeur embarqué, la compatibilité binaire est moins importante que pour un processeur généraliste

- Caractéristiques
  - Une instruction consiste en une nombre fixe d'opérations (à la différence du SIMD, pas des opérations identiques)
  - Les opérations d'une même instruction s'exécutent en parallèle
  - Sauts différés (pas de prédicteur de branchement)
  - Si le compilateur ne trouve pas assez d'opérations à exécuter dans un cycle, il complète avec des NOP
    - Afin que le programme ne prennent pas trop d'espace en mémoire, certains VLIW ont un format d'instruction qui permet de compresser les NOP
  - Exécution conditionnelle (prédication)
  - Le compilateur expose le parallélisme au maximum ➔ prédication, scheduling, déroulage de boucle, pipeline logiciel, etc… ➔ nécessite un nombre assez élevé de registres architecturaux

Exemple de VLIW: TriMedia TM3270

- 5 opérations par instruction
- Saut différé de 5 cycles (le compilateur doit trouver 25 opérations à mettre dans le «délai» du saut, à défaut il met des NOP)
- Format d'instructions compresse les NOP
- Adresses et données 32 bits
- 128 registres architecturaux 32 bits
  - 10 ports de lecture, 5 ports d'écriture
  - Prédicats (1 bit): 5 ports de lecture, 5 ports d'écriture
- 31 unités fonctionnelles
- Cache d'instructions 64 Ko 8-way set-associatif LRU
- Cache de données 128 Ko 4-way set-associatif LRU
- Mécanisme de prefetch dans le cache de données contrôlé par logiciel

- Technologie 90 nm
- Surface 8.1 mm2
- Tension 1.2 V
- Fréquence > 350 MHz
- Consommation ~ 1 milliwatt/MHz ➔ ~ 0.35 W

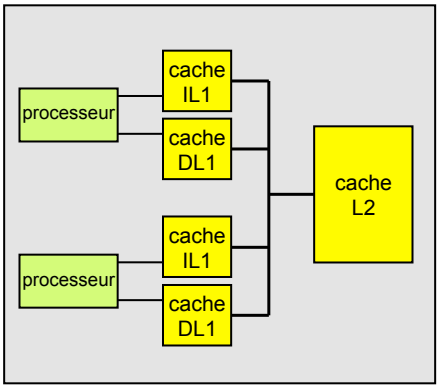
Parallélisme de thread

Superscalaire: une technique mature ?

- Années 2000
  - Alpha racheté par Intel, projet Alpha EV8 (21464) arrêté
  - Intel arrête le projet Tejas (Pentium 4 à 8 GHz): problème de consommation électrique et de dissipation de chaleur
  - Intel abandonne progressivement la microarchitecture Pentium 4 pour revenir à la microarchitecture précédente (P6), améliorée et renommée en Pentium M puis Intel Core
  - Les principaux constructeurs se tournent vers le multi-coeur ➔ le marketing ne se fait plus sur les Gigaherz mais sur le nombre de cœurs d'exécution
- Multi-cœur: les raisons
  - Problème de complexité matérielle ➔ coût et temps de développement des processeurs
  - Problème de consommation électrique ➔ dû à la complexité matérielle, mais aussi à la technologie (forte consommation statique dans les technologies récentes)

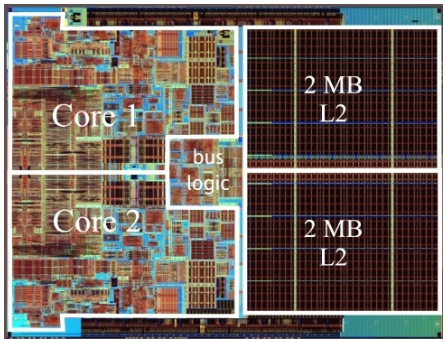
Processeurs multi-coeurs

- Multi-cœur aussi appelé *chip multiprocessor* (CMP)
- Plusieurs processeurs (cœurs) sur une même puce de silicium
- Les caches peuvent être locaux (un pour chaque cœur) ou partagés



- Exemples
  - Intel Core Duo ➔ 2 cœurs
  - AMD Phenom ➔ 4 cœurs
  - Sun Ultrasparc T2 ➔ 8 coeurs

Multi-cœur: exemple



- Exemple: Intel Core 2 Duo (65nm, 2.66 Ghz)
  - 2 coeurs
  - 291 millions de transistors (total)
  - 144 mm^2
  - 65 watts
  - Cache L2 de 4 Mo partagé par les 2 coeurs

Parallélisme de thread

- Pour exploiter la performance potentielle d'un multi-cœur, il faut exécuter plusieurs *threads* simultanément (*thread* ~ tâche)
- Exemple: 2 cœurs, 2 threads ➔ 1 thread s'exécute sur chaque cœur
- Chaque thread a son propre compteur de programme et ses propres registres
- Les threads se partagent le même espace d'adressage et peuvent communiquer en accédant à des variables partagées
- C'est au programmeur de créer du parallélisme de thread
  - Avec les langages de programmation actuels, le parallélisme de thread est beaucoup plus difficile à exposer automatiquement que le parallélisme d'instructions

- Exemple de programme multi-thread (cf. transparent 256)
- Intel Core 2 Duo ➔ 2 coeurs
- gcc –O3 -lpthread
- Temps d'exécution 3.3 s

```
#include <stdio.h>
#include <pthread.h>

#define NN 5000000
#define NITER 200

typedef struct {
  int debut,fin;
} iterations;

double t[NN];

void f(double x, double *y1, double *y2)
{
  double y = ((*y1)+(*y2))*0.5;
  if ((y*y) < x) {
    *y1 = y;
  } else {
    *y2 = y;
  }
}

double rac(double x)
{
  int n;
  double y1 = 0;
  double y2 = x;
  for (n=0; n<NITER; n++) f(x,&y1,&y2);
  return y2;
}

void * calcule(void *ptr)
{
  int i;
  iterations *p = (iterations *) ptr;
  for (i=p->debut; i<=p->fin; i++) t[i] = rac(i);
}

main()
{
  pthread_t thread1,thread2;
  iterations i1 = {0,NN/2-1};
  iterations i2 = {NN/2,NN-1};
  pthread_create(&thread1,NULL,calcule,(void *) &i1);
  pthread_create(&thread2,NULL,calcule,(void *) &i2);
  pthread_join(thread1,NULL);
  pthread_join(thread2,NULL);
  printf("%f\n",t[(int)t[NN-1]]);
}
```

- On peut aussi essayer d'utiliser un outil qui crée les threads (plus ou moins) automatiquement
  - exemple OpenMP
- gcc 4.2 sur Core 2 Duo
- gcc –O3 –funroll-loops –ftree-vectorize –fopenmp
- Temps d'exécution 3.3 s

```
#include <stdio.h>
#include <omp.h>

#define NN 5000000
#define NITER 200

double t[NN];

void f(double x, double *y1, double *y2)
{
  double y = ((*y1)+(*y2))*0.5;
  if ((y*y) < x) {
    *y1 = y;
  } else {
    *y2 = y;
  }
}

double rac(double x)
{
  int n;
  double y1,y2;
  y1 = 0;
  y2 = x;
  for (n=0; n<NITER; n++) {
    f(x,&y1,&y2);
  }
  return y2;
}

main()
{
  int i;
  omp_set_num_threads(2);
#pragma omp parallel for shared(t) private(i)
  for (i=0; i<NN; i++) t[i] = rac(i);
  printf("%f\n",t[(int)t[NN-1]]);
}
```


Processeur SMT

- Le multi-cœur n'est pas la seule manière d'exploiter le parallélisme de thread sur une même puce
- Une autre manière ➔ *Simultaneous multithreading* (SMT)
  - Dans un processeur superscalaire, il est rare qu'on arrive à exécuter à chaque cycle un nombre d'instructions égal au degré superscalaire
    - Dépendances entre instructions, miss de cache, etc…
  - Un processeur SMT est capable d'exécuter plusieurs threads simultanément
    - Plusieurs PC (cycle N lit instructions thread 1, cycle N+1 lit instructions thread 2, etc…)
    - À un instant donné, le pipeline d'instructions contient des instructions de plusieurs threads
    - Les threads se partagent les ressources (caches, opérateurs, …)
    - Chaque thread a ses propres registres
- Un processeur peut être à la fois multi-cœur et SMT (les cœurs sont SMT)
- Exemples de processeur SMT
  - Intel Pentium 4 (*hyperthreading*) ➔2 threads
  - IBM Power 5 ➔ 2 threads par cœur
  - Sun Ultrasparc T2 ➔ 8 threads par coeur

Architecture à mémoire partagée

- Les multi-cœurs généralistes actuels sont des architectures à mémoire partagée ➔ ils peuvent exécuter simultanément plusieurs threads ayant le même espace d'adressage
- L'architecture doit définir un modèle de comportement de la mémoire partagée (*consistency model*) ➔ contrat avec le programmeur

Exemples

Avant la création des threads, on initialise f1 et f2 à 0

Processeur 1	Processeur 2
f1 = 1;	f2 = 1;
while (f2==0);	while (f1==0);

Le programmeur veut exprimer le fait que le premier thread qui arrive à son point de synchronisation attend que l'autre thread arrive au sien

➔ On veut que la microarchitecture se comporte d'une manière qui soit intuitive pour le programmeur

Processeur 1	Processeur 2
A=1;	X=B
B=1;	Y=A;

- Initialement, A=B=0
- Quatre cas possibles:
  - X=0,Y=0 ➔ Y=A exécuté avant A=1
  - X=0,Y=1 ➔ X=B exécuté avant A=1 et Y=A exécuté après A=1
  - X=1,Y=1 ➔ X=B exécuté après B=1
  - X=1,Y=0 ➔ peut se produire si le processeur 2 exécute Y=A avant X=B (c'est théoriquement possible sur un processeur OoO) ou si il voit B=1 avant A=1
- Le 4ème cas n'est pas cohérent avec la vision séquentielle que le programmeur a du processeur ➔ ce cas est à interdire

Processeur 1	Processeur 2	Processeur 3
A=1;	X=A;	Y=B;
	B=1;	Z=A;

- Initialement, A=B=0
- Si X=1 et Y=1, alors on s'attend à avoir Z=1
  - le processeur 2 a vu l'instruction A=1 s'exécuter avant l'instruction B=1, donc on s'attend à ce que le processeur 3 voit aussi A=1 s'exécuter avant B=1
  - ➔ on veut que l'ordre des écritures soit le même pour tous les processeurs

Sequential consistency (SC)

- Le modèle de mémoire partagée connu sous le nom de *sequential consistency* (SC) peut être décrit de la manière suivante
- Une architecture qui respecte le modèle SC ne génère que les exécutions qu'on obtiendrait si on effectuait les accès mémoire de tous les threads les uns après les autres et d'une manière telle que chaque thread effectue ses accès dans l'ordre séquentiel (l'ordre spécifié par le programme)
  - Cela ne veut pas forcément dire que la microarchitecture effectue réellement les accès mémoire les uns après les autres et dans l'ordre séquentiel. C'est juste l'impression qu'elle doit donner au programmeur.
- Il peut exister plusieurs exécutions valides dans le modèle SC
  - Plusieurs manières d'entrelacer les threads
- En pratique, le modèle SC est difficile à mettre en œuvre sans sacrifier de la performance ➔ certains processeurs mettent en œuvre des versions «relâchées» de SC

Exemple: modèle de mémoire partagée de l'architecture Intel 64

- Du point de vue du programmeur (en assembleur),
  - Les LOAD sont exécutés dans l'ordre séquentiel
  - Les STORE sont exécutés dans l'ordre séquentiel
  - Un STORE ne peut pas s'exécuter avant un LOAD qui le précède dans l'ordre séquentiel
  - Un LOAD peut s'exécuter avant un STORE qui le précède dans l'ordre séquentiel pourvu que le LOAD et le STORE accèdent des adresses différentes
  - L'ordre dans lequel s'exécutent les STORE est le même pour tous les processeurs
- La microarchitecture doit simuler ce comportement

Mise en œuvre du modèle SC

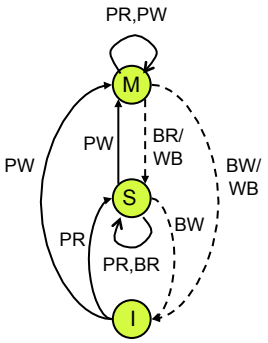
- Si un load a été exécuté spéculativement et que la donnée lue par le LOAD a été modifiée entre l'exécution du LOAD et son retirement du reorder buffer (ROB), il faut vider le ROB (comme pour une exception ou un branchement mal prédit) et réexécuter le LOAD
- Il faut un mécanisme permettant de simuler des écritures « atomiques »
  - On doit pouvoir répondre à la question « à quel cycle a été effectuée cette écriture ? » ➔ toute lecture effectuée avant ce cycle voit l'ancienne valeur, toute lecture effectuée après voit la nouvelle valeur
  - ➔ l'ordre des écritures est le même pour tous les coeurs
- ➔ mécanisme de cohérence
- Attention: les optimisations de code faites par le compilateur risquent de changer la sémantique du programme
  - Si l'exécution correcte du programme repose sur le modèle SC (ou un autre), il est nécessaire de dire explicitement au compilateur de faire certains accès comme spécifié (exemple: variables déclarées *volatiles*)

Mécanisme de cohérence

- Certains niveaux de cache sont locaux. Exemple: chaque cœur a son propre cache L1 de données. Sur certains multi-cœurs, les caches L2 sont aussi locaux.
- Si plusieurs coeurs accèdent en lecture à une même ligne de cache, une copie de la ligne est stockée sur chaque coeur
- Si un des coeurs veut modifier la ligne, il faut invalider les autres copies avant d'autoriser la ligne à être modifiée ➔ écriture « atomique » effectuée après qu'on est sûr que plus aucun coeur ne peut lire l'ancienne valeur
- Avec des caches locaux write-back, en cas de miss, il faut regarder dans les caches locaux des autres cœurs
  - Si une copy « dirty » de la ligne existe, c'est cette copie qu'il faut lire (écritures « atomiques »: une fois que la nouvelle valeur peut être lue par un processeur, c'est cette valeur que tous les processeurs doivent lire)
- Il faut aussi maintenir la cohérence des TLB

Protocole MSI

- Caches locaux write-back
- Chaque (copie de) ligne de cache a 3 états possibles
  - État I (« invalid ») ➔ la ligne est invalide
  - État M (« modified ») = « dirty » ➔ la ligne a été modifiée, les autres copies de la ligne sont dans l'état I
  - État S (« shared ») ➔ toutes les copies valides de la ligne sont non modifiées (état S)



- Intentions
  - PR (processor read): le coeur veut lire la ligne
  - PW (processor write): le coeur veut modifier la ligne
  - BR (bus read): un autre coeur veut lire la ligne
  - BW (bus write): un autre coeur veut modifier la ligne
- Actions
  - WB (write-back): la ligne est sauvegardée dans la partie partagée de la hiérarchie mémoire (exemple: cache L2)
- Pour regarder dans les autres cœurs, on fait du bus snooping
  - On exploite le fait d'avoir un bus partagé entre les coeurs
  - Les tags des caches locaux sont dupliqués pour que le snooping ne consomme pas de bande passante cache

Autres protocoles de cohérence

- Protocole MESI
  - On rajoute un état E (« exclusive ») ➔ la ligne est non modifiée et aucun autre cœur n'a de copie
  - On met la ligne dans l'état E lorsqu'on fait un miss et qu'aucun autre cœur n'a de copie
  - Quand on veut modifier la ligne et qu'elle est dans l'état E, on n'a pas besoin d'aller regarder sur les autres cœurs ➔ on génère moins de snooping
  - Utilisé dans les Intel Core
- Protocole MOESI
  - On rajoute un état O (« owned ») ➔ il peut exister plusieurs copies de la ligne: cette copie est « dirty », les autres copies sont dans l'état S
  - On passe de l'état M à l'état O quand un autre cœur veut lire la ligne ➔ pas besoin de faire un write-back dans ce cas
  - Utilisé dans les AMD Opteron
- Problème du *false sharing*: lorsque 2 variables non partagées sont dans la même ligne de cache et que 2 coeurs veulent écrire dans ces variables, la ligne fait du « ping-pong » entre les 2 cœurs ➔ problème de performance

Le modèle SC est-il suffisant ?

- Théoriquement, le modèle SC est suffisant pour la programmation multi-thread
  - Par exemple, on sait faire des sections critiques (algorithme de Dekker, algorithme de Peterson,…)
- Problème(s)
  - Modèle SC incompatible avec certaines optimisations de compilation
  - La performance est importante (sinon pourquoi faire du multi-cœur ?)
  - Les boucles d'attente consomment des ressources d'exécution et de l'énergie
  - Et si la mémoire partagée n'est pas strictement SC ?
- ➔ la plupart des architectures proposent des instructions pour la programmation multi-thread

Instructions pour le multi-thread

- Exemples
  - « Barrière » mémoire (fence)
    - on force les accès mémoire antérieurs à la barrière (dans l'ordre du programme) à se terminer avant d'exécuter la suite du programme ➔ pipeline flush
  - Swap (or *exchange*)
    - Opération atomique qui échange le contenu d'un registre et d'une adresse mémoire
  - Fetch-and-add
    - Opération atomique qui modifie le contenu d'une adresse mémoire
  - Load-locked / Store-conditional
    - r1 ← LL x; modifie r1; SC r1,x
    - Si aucun autre processeur n'écrit à x entre l'exécution du LL et du SC, alors le SC écrit la valeur de r1 à l'adresse x et écrit 1 dans r1, sinon il écrit 0 dans r1
- Boucles d'attente
  - Faire un HALT pour permettre à un autre thread d'utiliser le processeur si la boucle d'attente est trop longue
  - Instruction PAUSE (Intel) ➔ rend les boucles d'attente moins « agressives »
  - MONITOR/MWAIT (Intel) ➔ permet de réveiller un thread lorsqu'un certain évènement se produit (écriture à une adresse particulière)

Cache partagé

- Certains niveaux de cache peuvent être partagés entre plusieurs cœurs
  - Exemple: cache L2 de l'Intel Core Duo
  - Si un niveau est partagé, les niveaux suivants le sont aussi
    - L2 partagé ➔ L3 partagé
- Avantages
  - Le protocole de cohérence entre cœurs n'a pas besoin d'aller au-delà du niveau de cache partagé
  - Meilleure utilisation de l'espace de cache
    - Pas de réplication de ligne dans un cache partagé
    - Un cœur peut faire du prefetch pour un autre coeur
    - Lorsqu'un seul cœur travaille à un instant donné, il a accès à tous l'espace du cache
    - Lorsque plusieurs cœurs travaillent en même temps et que l'un des cœurs n'utilise qu'une petite partie de l'espace du cache partagé, cela laisse plus d'espace pour les autres cœurs
- Inconvénients
  - Nécessite une grande bande passante d'accès au cache
  - Cache plus complexe, latence du cache plus grande
  - La politique de remplacement LRU (ou pseudo-LRU) ne garantit pas un usage équitable de l'espace du cache lorsqu'on exécute des tâches indépendantes ➔ il faut une politique de remplacement spécifique, plus complexe que LRU

Quelques références

- Principes de base de l'architecture des ordinateurs
  - David A. Patterson & John L. Hennessy, « Computer Organization and Design: the hardware/software interfaces », 3ème édition, Morgan Kaufmann.
- Architecture avancée
  - John L. Hennessy & David A. Patterson, « Computer Architecture: a quantitative approach », 4ème édition, Morgan Kaufmann.
- Architectures multiprocesseurs
  - David E. Culler & Jaswinder Pal Singh, « Parallel Computer Architecture: a hardware/software approach », Morgan Kaufmann.