Plan

- Représentation des nombres
- Circuits logiques
- Unité Arithmétique et Logique
- Notions de temps et de mémorisation
- Contrôle et jonction des composants
- Evolution des ordinateurs – Historique
- Un microprocesseur simple
- Programmation d'un microprocesseur
- Système complet
- **Les microprocesseurs actuels**
  - **Pipeline**
  - Prédiction de branchement
  - Cache
  - Exécution superscalaire
- Exploitation de la performance des microprocesseurs

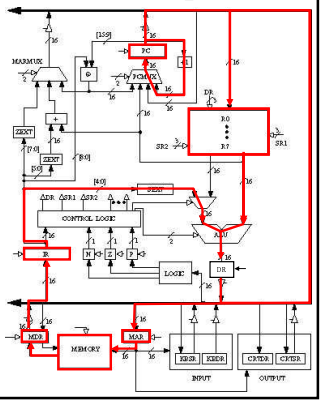
Exploitation Efficace des Composants

- Etapes d'exécution d'une instruction:
  1. Envoi adresse instruction (IA)
  2. Chargement instruction (IF)
  3. Stockage instruction (SI)
  4. Décodage; lecture des opérandes (DI)
  5. Calcul d'adresse (AC)
  6. Accès mémoire (ME)
  7. Exécution (EX)
  8. Ecriture du résultat (WB)

0001	101	100	1	00011

ADD R5, R4, #3

Un seul composant utilisé à chaque cycle

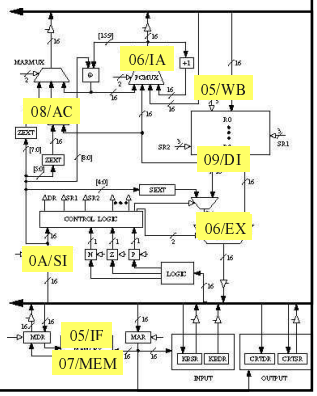


Exploitation Efficace des Composants

```
for (i=0; i < 100; i++) {
   a[i] = a[i] + 5
}
```

```
                ...
05 LOOP         LDR R1, R0, #3
06              ADD R1, R1, #5
07              STR R1, R0, #30
08              ADD R0, R0, #1
09              ADD R3, R0, R2
0A              BRn LOOP
                ...
```

- **Tous les composants sont utilisés**
- **Une instruction termine à chaque cycle**



Pipeliner

Tous les composants sont utilisés

Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12	13
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM	EX	WB						
ADD R1,R1,#5		IA	IF	SI	DI	AC	MEM	EX	WB					
STR R1,R0,#30			IA	IF	SI	DI	AC	MEM	EX	WB				
ADD R0,R0,#1				IA	IF	SI	DI	AC	MEM	EX	WB			
ADD R3,R0,R2					IA	IF	SI	DI	AC	MEM	EX	WB		
BRn LOOP						IA	IF	SI	DI	AC	MEM	EX	WB	
LDR R1,R0,#30							IA	IF	SI	DI	AC	MEM	EX	WB
ADD R1,R1,#5								IA	SI	IF	DI	AC	MEM	EX

- L'exécution du programme est jusqu'à **8 fois plus rapide**.
- Le temps d'exécution d'une instruction est inchangé.

Tendances

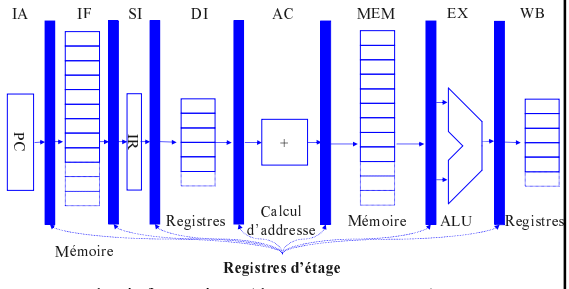
- Le pipeline du Pentium IV contient 20 étage + 8 étages de conversion `x86` → `µInstructions`.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

- La motivation initiale du pipeline était:
  - L'exploitation efficace des composants de l'architecture
  - Et par voie de conséquence: l'augmentation du débit des instructions.
- La motivation actuelle du pipeline est l'augmentation de la fréquence d'horloge:
  - On segmente les étapes/composants en sous-étapes/sous-composants.
  - La durée maximale d'une (sous-) étape est donc réduite.

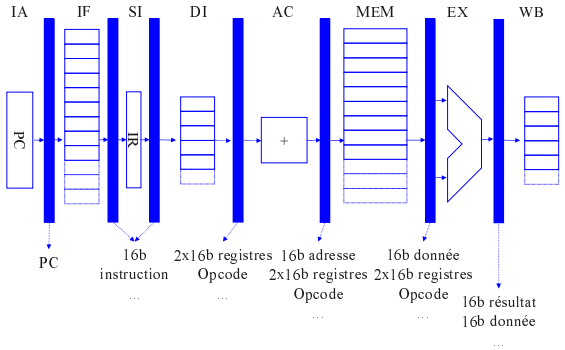
  ⇒ On peut réduire le temps de cycle donc augmenter la fréquence d'horloge.
- Cette stratégie rend difficile l'obtention de performances **soutenues** élevées.

Implémentation du Pipeline



- Toutes les informations (données et contrôle) nécessaires à l'exécution d'une instruction sont stockées et propagées dans les registres d'étage.

Implémentation du Pipeline

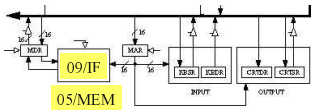


Aléas de Pipeline

- Il n'est pas toujours possible d'émettre/sortir une instruction à chaque cycle dans un pipeline.
- Lorsque l'avancement d'une instruction est bloqué, on parle d'**aléa de pipeline** (*pipeline hazard*).
- On distingue 3 types d'aléas:
  - Aléas structurels (conflit de ressource).
  - Aléas de données (dépendances de données).
  - Aléas de contrôle (branchements).
- En cas d'aléa, le circuit de contrôle introduit une **bulle** dans le pipeline; on parle de **gel de pipeline** (*stall*); l'utilisation du pipeline n'est plus optimale.
- Mesure de performance: IPC (*Instructions Per Cycle*)
  < 1 (optimal) en cas d'aléas

Aléas Structurels

- Accès au même composant au même cycle.
- Solutions:
  - Accepter un gel de pipeline à chaque conflit (SPARC).
  - Augmenter les ressources (ici, deux ports d'accès à la mémoire).



Inst.	0	1	2	3	4	5
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM
ADD R1,R1,#5		IA	IF	SI	DI	AC
STR R1,R0,#30			IA	IF	SI	DI
ADD R0,R0,#1				IA	IF	SI
ADD R3,R0,R2					IA	IF

Conflit de ressource

Aléas Structurels

Pas d'accès à la mémoire ⇒ pas d'aléa structurel

Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM	EX	WB					
ADD R1,R1,#5		IA	IF	SI	DI	AC	MEM	EX	WB				
STR R1,R0,#30			IA	IF	SI	DI	AC	MEM	EX	WB			
ADD R0,R0,#1				IA	IF	SI	DI	AC	MEM	EX	WB		
ADD R3,R0,R2					IA	●	IF	SI	DI	AC	MEM	EX	WB
BRn LOOP						●	IA	●	IF	SI	DI	AC	MEM
LDR R1,R0,#30						●		●	IA	IF	SI	DI	AC
ADD R1,R1,#5						●		●		IA	IF	SI	DI

Aléas de Données

- Dépendance de données entre deux instructions.

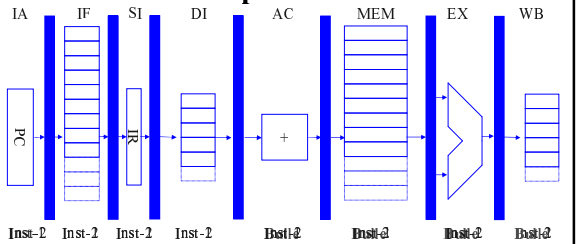
```
LDR R1 ← R0,#30
ADD R1 ← R1,#5
```

- Une instruction ne peut effectuer son étape DI (chargement des opérandes) que lorsque ses opérandes sont disponibles.

R_1 contient la valeur attendue par LDR

Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12
LDR R1,R0,#30	IA	IF	SI	DI	AC	MEM	EX	WB					
ADD R1,R1,#5		IA	IF	SI	●	●	●	●	DI	AC	MEM	EX	WB

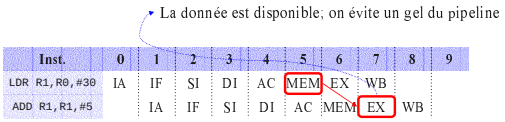
Gestion du Pipeline en Cas d'Aléa



Inst.	0	1	2	3	4	5	6	7	8	9	10	
Inst 1	IA	IF	SI	DI	AC	MEM	EX	WB				8 cycles
Inst 2		IA	SI	IF	DI	AC	MEM	EX	WB			
Inst 1	IA	IF	SI	DI	AC	MEM	EX	WB				10 cycles
Inst 2		IA	IF	SI	DI	●	●	AC	MEM	EX	WB	

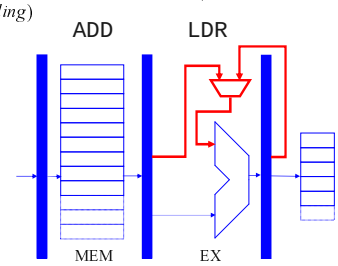
Le *Forwarding*

- En cas de dépendance, la donnée attendue est souvent disponible avant d'être écrite dans le registre
  ⇒ Passer immédiatement la donnée au composant qui l'attend sans attendre son écriture dans le registre.
  = ***Forwarding***
- Le pipeline est bloqué seulement lorsque la donnée est indispensable.



Implémentation du *forwarding*

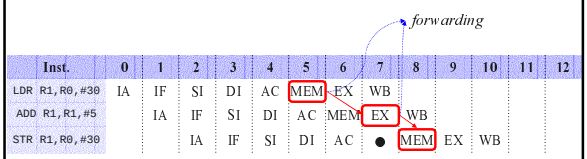
- Le *forwarding* nécessite:
  - de rajouter des chemins de données
  - d'augmenter la taille de multiplexeurs ou d'en ajouter
  - de modifier le circuit de contrôle (détection/activation du *forwarding*)



Le *Forwarding*

- Le *forwarding* ne permet pas d'éviter tous les gels de pipeline:

```
ADD R1 ← R1,#5
STR R1 → R0,#30
```



Instructions Multicycles

- Exemple: instructions flottantes.
  - FPADD: 2 cycles d'exécution
  - FPMUL: 5 cycles d'exécution

Conflit de ressource si FPADD et FPMUL dans le même composant

Inst.	0	1	2	3	4	5	6	7					
FPMUL F2,F1,F0	IA	IF	SI	DI	AC	MEM	EX	EX	EX	EX	EX	WB	
FPADD F4,F2,F3		IA	IF	SI	DI	AC	MEM	●	●	●	●	EX	EX
FPADD F2,F1,F3			IA	IF	SI	DI	AC	MEM	EX	EX	WB		

Ecriture dans le désordre

- Nouvelles dépendances de données:
  - Ecriture des registres dans le désordre
- Nouveaux conflits de ressources (ports du banc de registres)

Inst.	0	1	2	3	4	5	6	7					
FPMUL F2,F1,F0	IA	IF	SI	DI	AC	MEM	EX	EX	EX	EX	EX	WB	
FPADD F4,F2,F3		IA	IF	SI	DI	AC	MEM	●	●	●	●	EX	EX
FPADD F2,F1,F3			IA	IF	IF	DI	AC	MEM	EX	EX	●	●	WB

Pipeline et Exceptions

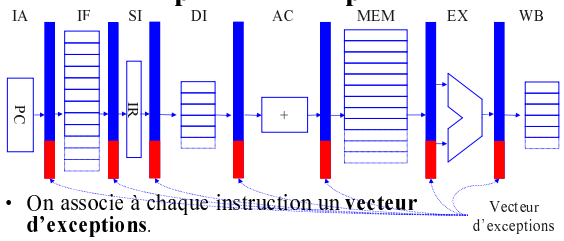
- Le pipeline rend la gestion des exceptions plus difficile. Exemple:
  - `LDR` fait une faute de page en `MEM`
  - `ADD` fait une faute de page en `IF`

Exceptions

Inst.	0	1	2	3	4	5	6	7	8
LDR	IA	IF	SI	DI	AC	MEM	EX	WB	
ADD		IA	SI	IF	DI	AC	MEM	EX	WB

- Les exceptions doivent être traitées dans l'ordre d'exécution des instructions.
- L'exception de `ADD` intervient avant celle de `LDR`; dans une machine non pipelinée, ce cas ne peut se produire.

Pipeline et Exceptions



- On associe à chaque instruction un **vecteur d'exceptions**.

Vecteur d'exceptions

- Lorsqu'une exception se produit, l'instruction ne peut plus modifier l'état de la machine (registre/mémoire).
- En WB, le circuit de contrôle examine le vecteur d'exceptions de chaque instruction et détermine si une exception doit être traitée

  ⇒ Les exceptions sont traitées dans l'ordre d'exécution des instructions.

Exceptions et Instructions Multicycles

- Exception précise sur l'instruction *i*:
  - les instructions < *i* terminent normalement,
  - les instructions > *i* peuvent être interrompues, puis réexécutées (à partir du premier étage du pipeline) après gestion de l'exception.
- Exemple: l'instruction FPADD termine et modifie l'état de la machine avant la détection de l'exception de l'instruction FPMUL.
- Le programme ne peut être redémarré normalement à l'instruction FPMUL.

Exception

Inst.	0	1	2	3	4	5	6	7					
FPMUL F2, F1, F0	IA	IF	SI	DI	AC	MEM	EX	EX	EX	EX	EX	WB	
FPADD F5, F3, F4		IA	IF	SI	DI	AC	MEM	EX	EX	WB			

- Une solution: retarder les modifications de l'état de la machine (registre/mémoire) jusqu'à ce que toutes les instructions précédentes aient terminé leur exécution.

Aléas de Contrôle

- En cas de branchement, on doit d'abord connaître la destination et le résultat (condition) du branchement, avant de charger l'instruction suivante.

```
LOOP    LDR R1 ← R0,#30
        ...
        BRn  LOOP
```

La condition (bits *n.p.z*) est lue à la fin de cette étape

L'adresse de destination du branchement est disponible à la fin de cette étape

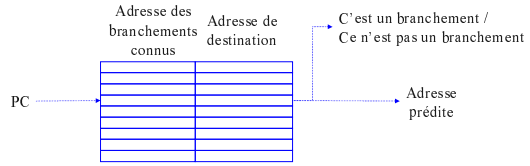
Inst.	0	1	2	3	4	5	6	7	8	9	10	11	12
BRn LOOP	IA	IF	SI	DI	AC	MEM	EX	WB					
LDR R1, R0, #30		●	●	●	●	IA	IF	DI	AC	MEM	EX	WB	

Plan

- Représentation des nombres
- Circuits logiques
- Unité Arithmétique et Logique
- Notions de temps et de mémorisation
- Contrôle et jonction des composants
- Evolution des ordinateurs – Historique
- Un microprocesseur simple
- Programmation d'un microprocesseur
- Système complet
- **Les microprocesseurs actuels**
  - Pipeline
  - **Prédiction de branchement**
  - Cache
  - Exécution superscalaire
- Exploitation de la performance des microprocesseurs

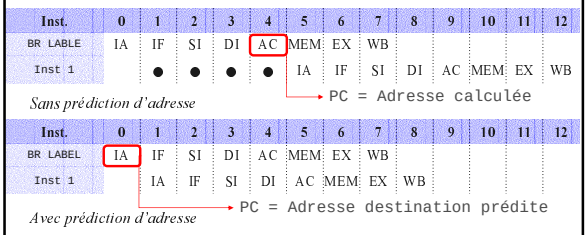
Prédiction de Branchement

- En général, l'adresse de destination d'un branchement est constante (sauf pour RET et branchements indirects).
- **Prédire** l'adresse de destination:
  - Les adresses de destination sont stockées dans une table à chaque exécution d'un branchement
  - La table est indexée par le PC de l'instruction (du branchement)
  - Lorsque le PC est envoyé à la mémoire, il est envoyé également à la table qui indique: (1) s'il s'agit d'un branchement, (2) l'adresse de destination si c'est le cas.



Prédiction d'Adresse

- Si le PC correspond à un branchement, on met à jour le PC: `PC = Adresse de destination`.
- Exemple: branchement inconditionnel

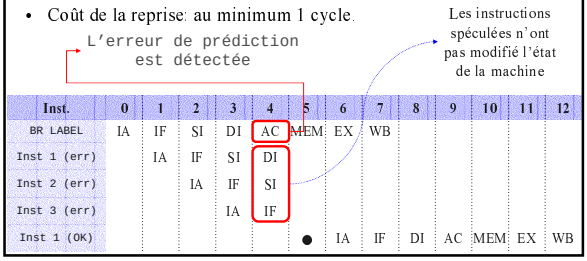


Sans prédiction d'adresse

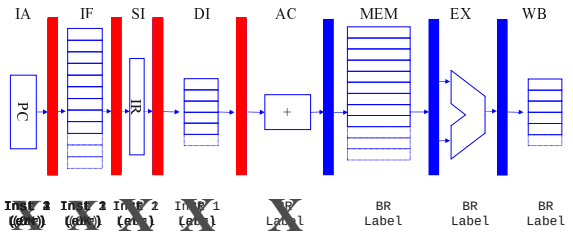
Avec prédiction d'adresse

Erreur de Prédiction

- Détecter l'erreur (on effectue toujours le calcul de l'adresse de destination).
- Eliminer du pipeline les instructions chargées spéculativement.
- Les instructions spéculées ne modifient l'état de la machine qu'après vérification de la prédiction.
- Coût de la reprise: au minimum 1 cycle.



Reprise en Cas d'Erreur



Prédiction de Condition

- Cas des branchements conditionnnels.
- Il faut prédire la valeur de la condition.
- La valeur de la condition varie souvent d'une exécution du branchement à l'autre ⇒ **la prédiction est difficile**.
- Exemple: branchement pris

Inst.	0	1	2	3	4	5	6	7	8	9
BR LABLE	IA	IF	SI	DI	AC	MEM	EX	WB		
Inst 1		●	●	●	IA	IF	DI	AC	MEM	EX	WB

PC = Adresse destination

Sans prédiction de condition, avec prédiction d'adresse prédite

Inst.	0	1	2	3	4	5	6	7	8	9
BR LABEL	IA	IF	SI	DI	AC	MEM	EX	WB		
Inst 1		IA	IF	SI	DI	AC	MEM	EX	WB	

PC = Adresse destination prédite

Avec prédiction de condition et d'adresse

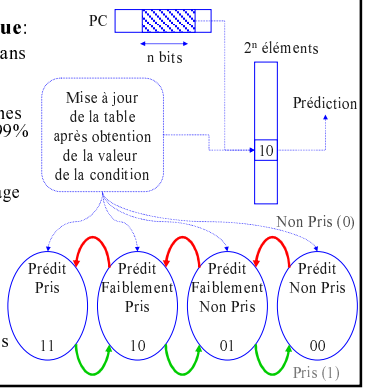
Stratégies de Prédiction

- Prédiction **statique**:
  - Toujours pris
    - Adéquat pour les boucles
  - Analyse à la compilation
    - EPIC/IA-64; limitations de l'analyse statique
  - Taux de bonnes prédictions: ≈ de 70% jusqu'à 90%

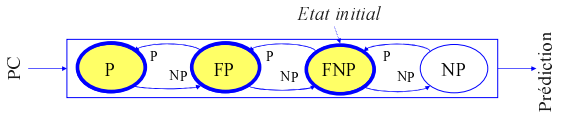
```
                ...
05 LOOP         LDR R1, R0, #3
06              ADD R1, R1, #5
07              STR R1, R0, #30
08              ADD R0, R0, #1
09              ADD R3, R0, R2
0A              BRn LOOP
                ...
```

Stratégies de Prédiction

- Prédiction **dynamique**:
  - Largement utilisée dans les processeurs
  - Mécanismes les plus récents: taux de bonnes prédictions jusqu'à 99% sur certaines applications
  - Principe: apprentissage des comportements individuels des branchements
- Un premier mécanisme: l'historique local
  - Un automate à 4 états par branchement.



Exemple



i=0 (pris)	Non Pris
i=1 (pris)	Pris
i=2 (pris)	Pris
i=3 (pris)	Pris
...	...
i=99 (non pris)	Pris

```
...
05 LOOP   LDR R1, R0, #3
06        ADD R1, R1, #5
07        STR R1, R0, #30
08        ADD R0, R0, #1
09        ADD R3, R0, R2
0A        BRn LOOP
...
```

Exécution de 100 itérations:

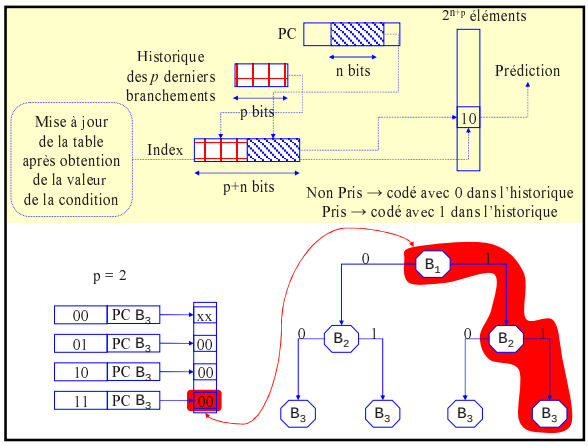
- itération i=0: BRn pris
- itération i=1: BRn pris
- ...
- itération i=99: BRn non pris

Amélioration de la Prédiction Dynamique

- Une augmentation faible du taux de bonnes prédictions a un impact important sur la performance globale du processeur.

```
if (a == 1) a = 0;      /* Branchement B1 */
...
if (b == 0) b = 1;      /* Branchement B2 */
...
if (a == b) ...;        /* Branchement B3 */
```

- Pour affiner la précision de la prédiction, on utilise le comportement des branchements antérieurs: historique global.



Impact des Aléas de Contrôle sur la Performance d'un Processeur

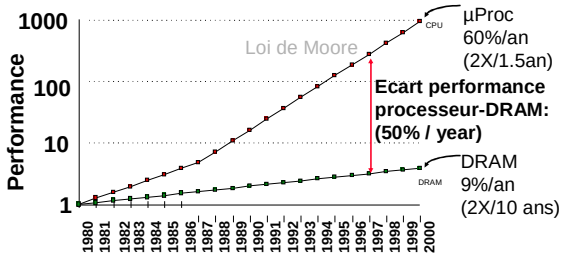
- En moyenne, 1 instruction sur 5 est un branchement.
- Sur les pipelines actuels, le coût d'un aléa de contrôle est de 5 à 10 cycles.

- 8 cycles entre le fetch et la résolution du branchement
- 4 cycles pour réinitialiser le pipeline
  ⇒ 12 cycles de pénalité.
- En moyenne 1 branchement toutes les 5 instructions.
- 1 instruction / cycle (1000 instructions) :
  - 50% mauvaises prédictions:
    1000 * (0,8*1 + 0,2*(0,5*1 + 0,5*12)) = 2100 cycles
  - 20% mauvaises prédictions:
    1000 * (0,8*1 + 0,2*(0,8*1 + 0,2*12)) = 1440 cycles
  - 5% mauvaises prédictions:
    1000 * (0,8*1 + 0,2*(0,95*1 + 0,05*12)) = 1110 cycles
- 5 instructions / cycle (1000 instructions) :
  - 50% mauvaises prédictions:
    200 * (0,5*1 + 0,5*12) = 1300 cycles
  - 20% mauvaises prédictions:
    200 * (0,8*1 + 0,2*12) = 640 cycles
  - 5% mauvaises prédictions:
    200 * (0,95*1 + 0,05*12) = 310 cycles

Plan

- Représentation des nombres
- Circuits logiques
- Unité Arithmétique et Logique
- Notions de temps et de mémorisation
- Contrôle et jonction des composants
- Evolution des ordinateurs – Historique
- Un microprocesseur simple
- Programmation d'un microprocesseur
- Système complet
- **Les microprocesseurs actuels**
  - Pipeline
  - Prédiction de branchement
  - **Cache**
  - Exécution superscalaire
- Exploitation de la performance des microprocesseurs

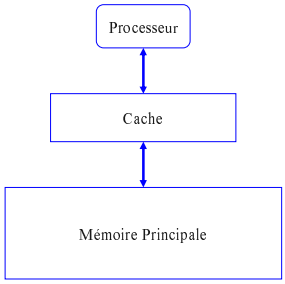
Latence d'Accès à la Mémoire



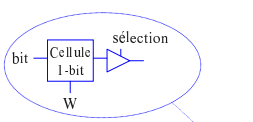
- Temps de cycle du processeur << temps d'accès à la mémoire.

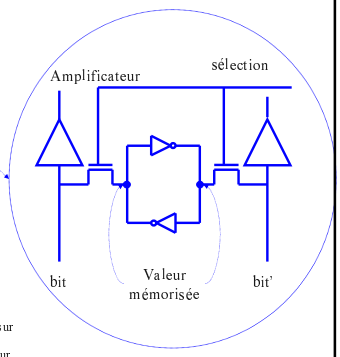
Mémoire Cache

- Mémoire rapide (SRAM) mais petite (coût):
  - ≈ entre 1 et 3 cycles (accès éventuellement pipeliné)
- Le processeur envoie ses requêtes mémoire au cache
  - Donnée dans le cache: succès (*hit*)
  - Donnée hors du cache: défaut (*miss*)
- Performance:
  - Taux de défauts de cache
  - Temps moyen d'accès à la mémoire



Cellule 1-bit SRAM





- SRAM=*Static Random Access Memory*.
- Ecriture:
  - bit=*valeur*; bit'=*valeur*'
  - sélection=1
- Lecture:
  - sélection=0
  - bit=V_{DD}, bit'=V_{DD}
  - sélection=1
  - valeur:
    - 1/V_{DD} → décroissance de V sur bit'.
    - 0/V_{SS} → décroissance de V sur bit.

Cache & Propriétés de Localité

- La plupart des programmes possèdent de fortes propriété de localité.
- Localité **temporelle**: une adresse *A* référencée à l'instant *t* a une forte probabilité d'être référencée à nouveau dans un court intervalle de temps.
- Localité **spatiale**: si une adresse *A* est référencée à l'instant *t*, il y a une forte probabilité qu'une adresse voisine de *A* soit référencée dans un court intervalle de temps.
- Le cache exploite ces deux propriétés de localité.

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    y[i] = y[i] + a[i][j] * x[j]
  }
}
```

- `y[i]`: propriétés de localités temporelle et spatiale.
- `a[i][j]`: propriétés de localité spatiale.
- `x[j]`: propriétés de localité temporelle et spatiale.

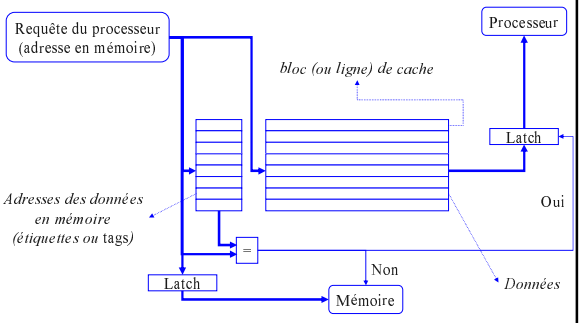
Localité des Données et des Instructions

- Les instructions, comme les données, possèdent de fortes propriétés de localité.
- La localité temporelle est simplement exploitée en conservant une donnée dans le cache.
- La localité spatiale est exploitée en chargeant les données par **blocs** et non individuellement.

```
                ...
05 LOOP         LDR R1, R0, #3
06              ADD R1, R1, #5
07              STR R1, R0, #30
08              ADD R0, R0, #1
09              ADD R3, R0, R2
0A              BRn LOOP
                ...
```

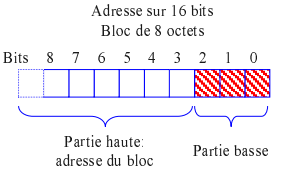
Boucle
⇒ réutilisation des instructions
⇒ localité temporelle
Instructions consécutives en mémoire
⇒ localité spatiale

Structure d'un Cache



Ligne de Cache

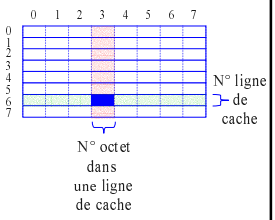
- Charger les données par blocs (localité spatiale).
- Décrire un bloc de données avec une seule adresse (contraintes du bus mémoire, calcul d'adresse).
  - La partie haute de l'adresse des données d'un même bloc est identique
  - Seule la partie basse de l'adresse varie.
- Rappel: une adresse = 1 octet.

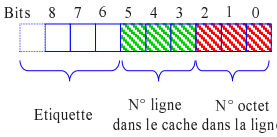


Exemple:
0...010100000
0...010100111
0...010101000
Même ligne de cache
Lignes distinctes,
adresses consécutives

Placement des Données

- Le placement des données est géré par le matériel
  - ⇒ Le programmeur n'a pas à se soucier du placement des données
  - ⇒ Le fonctionnement du cache est transparent pour le programmeur.
- L'emplacement d'une donnée est déterminé selon une fonction simple de l'adresse.
  - N° ligne dans le cache
  - N° d'octet dans la ligne

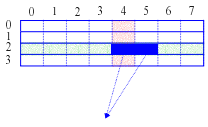




- Cache de C_S octets
- Ligne de L_S octets
- N° octet: $\log_2(L_S)$ bits de poids faible de l'adresse.
- N° ligne: $\log_2(C_S/L_S)$ bits de poids faible suivants de l'adresse.

Lecture d'une Donnée

- Exemple:
  - C_S = 32 octets
  - L_S = 8 octets
- Adresse demandée (16 bits):
  - 0110010001010**100**
  - N° ligne: **10**
  - N° octet dans la ligne: **100**
- Une requête peut avoir une taille variable:
  - octet, demi-mot, mot...
  - requête = adresse + nombre d'octets
  - adresse = adresse du premier octet
  - exemple: 2 octets (mot de 16 bits)

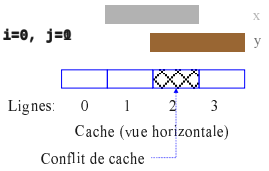


Associativité

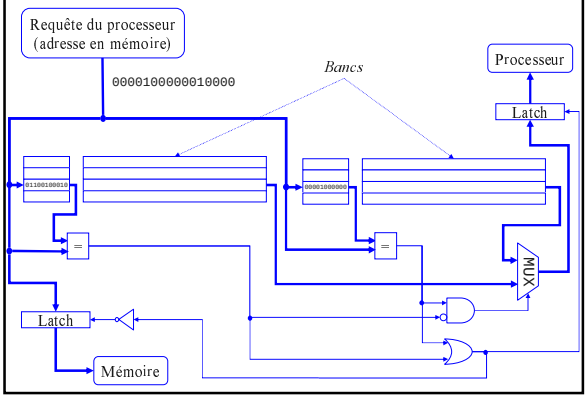
- Taille mémoire physique >> taille cache
- La fonction de placement peut engendrer des conflits entre les données.
- On peut réduire les conflits en augmentant l'**associativité** des caches.

```
for (i=0; i<N; i++) {
  for (j=0; j<2; j++) {
    x[j] = y[j]
  }
}
```

```
@x = 0110010001010000
@y = 0000100000010000
```

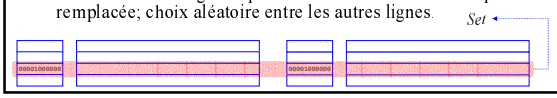


Structure d'un Cache Associatif

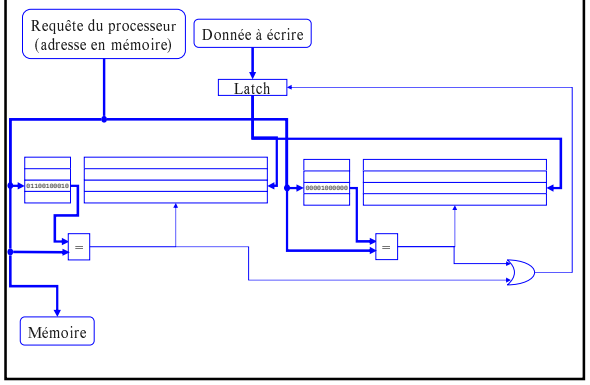


Fonctionnement Cache Associatif

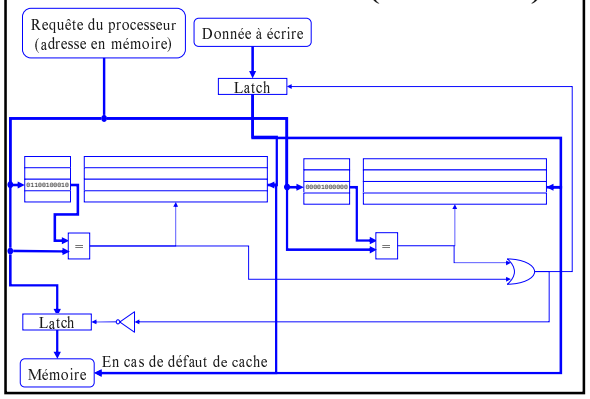
- Degré d'associativité *n*.
- Une donnée peut être stockée dans *n* emplacements différents.
- En cas de défaut de cache, il faut choisir le bloc dans lequel on va charger la nouvelle donnée.
- Le choix est effectué entre les blocs du cache qui peuvent accueillir la donnée (un *set*):
  - LRU (*Least Recently Used*) : on choisit le bloc le plus anciennement accédé.
  - FIFO (*First In First Out*)
  - *Random*
  - Pseudo-LRU: la ligne la plus récemment accédée n'est pas remplacée; choix aléatoire entre les autres lignes.



Ecriture d'une Donnée (*Write-Through*)

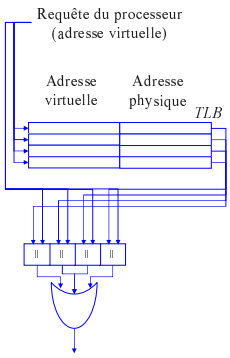


Ecriture d'une Donnée (*Write-Back*)

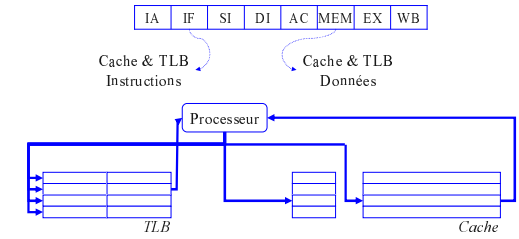


Mémoire Virtuelle / Mémoire Physique: le TLB

- Le processeur utilise des adresses *virtuelles*.
- Les données ont une adresse en mémoire *physique*.
  - ⇒ Il faut faire la traduction adresses virtuelles / adresses physiques
  - ⇒ **TLB** (*Translation Lookaside Buffer*)
- Le TLB est un *cache* de traductions d'adresses.
- Une entrée du TLB correspond à une *page*.
- Souvent le TLB est complètement associatif (n = nombre de lignes).



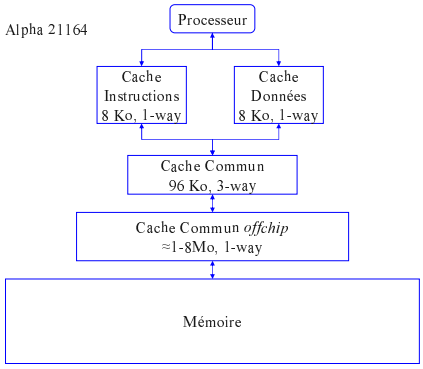
Synthèse



- L'adresse stockée dans le cache est en général l'adresse *physique*.
- Mais le processeur accède au cache avec une adresse *virtuelle*.
- Le processeur accède donc au TLB pour obtenir la traduction et effectue la comparaison ensuite.
- Souvent, on conçoit les caches pour que l'on puisse effectuer *en parallèle* l'accès au cache et au TLB.

Plusieurs Niveaux de Cache

Exemple: Alpha 21164



Impact des Défauts de Cache sur la Performance d'un Processeur

- En moyenne, 1 instruction sur 3 est une instruction mémoire.
- Il y a aussi des défauts de cache instruction.
- Les hiérarchies de cache réduisent la latence moyenne d'accès à la mémoire.

- Processeur à 1GHz, 100ns pour accéder à la mémoire
- 1000 instructions :
  - 50% défauts de cache:
    1000 * (0,67*1 + 0,33*(0,5*1 + 0,5*100)) = 17335 cycles
  - 5% défauts de cache:
    1000 * (0,67*1 + 0,33*(0,95*1 + 0,05*100)) = 2633 cycles
  - 0% défauts de cache:
    1000 cycles

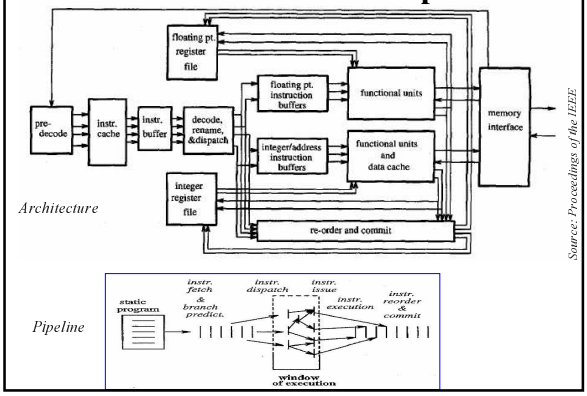
Plan

- Représentation des nombres
- Circuits logiques
- Unité Arithmétique et Logique
- Notions de temps et de mémorisation
- Contrôle et jonction des composants
- Evolution des ordinateurs – Historique
- Un microprocesseur simple
- Programmation d'un microprocesseur
- Système complet
- **Les microprocesseurs actuels**
  - Pipeline
  - Prédiction de branchement
  - Cache
  - **Exécution superscalaire**
- Exploitation de la performance des microprocesseurs

Processeur Superscalaire

- Pipeline: au plus une instruction terminée par cycle.
- Superscalaire de degré *n*: jusqu'à *n* instructions terminée par cycle (actuellement $n \approx 4$).
- Pour réaliser un processeur superscalaire, il faut:
  - Assurer un flux d'instructions suffisant.
  - Déterminer quelles instructions peuvent s'exécuter en parallèle.
  - Passer les données entre les instructions (le résultat d'une instruction *i* est l'opérande d'une instruction *j*).
  - Disposer de plusieurs unités de calcul en parallèle.
- Contrainte: maintenir des interruptions précises.
- L'architecture des différents processeurs haute-performance est très similaire.

Structure d'un Processeur Superscalaire



Source: Proceedings of the IEEE

Parallélisme entre Instructions (*ILP*)

(parallélisme à grain fin)

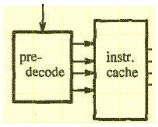
```
for (i=0; i<last; i++) {
  if (a[i] > a[i+1]) {
    temp = a[i];
    a[i] = a[i+1];
    a[i+1] = temp;
    change++;
  }
}
```

```
L2:
   move    r3,r7       #r3->a[i]
   lw      r8,(r3)     #load a[i]
   add     r3,r3,4     #r3->a[i+1]
   lw      r9,(r3)     #load a[i+1]
   ble     r8,r9,L3    #branch a[i]>a[i+1]

   move    r3,r7       #r3->a[i]
   sw      r9,(r3)     #store a[i]
   add     r3,r3,4     #r3->a[i+1]
   sw      r8,(r3)     #store a[i+1]
   add     r5,r5,1     #change++
L3:
   add     r6,r6,1     #i++
   add     r7,r7,4     #r4->a[i]
   blt     r6,r4,L2    #branch i<last
```

Chargement d'Instructions

- Il faut pouvoir déterminer rapidement si une instruction est un branchement pour éviter d'ajouter des bulles dans le pipeline:
  - Lors du chargement d'une instruction dans le cache instruction, passage par un circuit de **prédécodage**.
  - Insertion de bits de prédécodage (branchement oui/non, conditionnel oui/non…) dans le cache instructions.

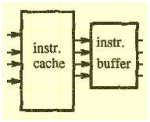


IA LC-2:

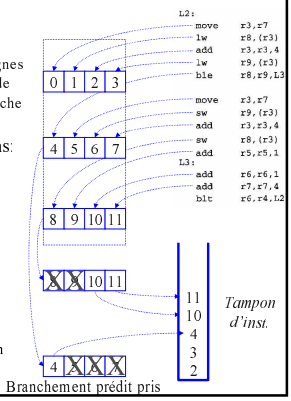
BR	$I_{15}I_{14}I_{13}I_{12}$=0000
JSR	$I_{15}I_{14}I_{13}I_{12}$=0100
JSRR	$I_{15}I_{14}I_{13}I_{12}$=1100

Bit de prédécodage *isBranch*:
$I_{14} + I_{15}'$

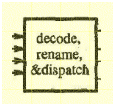
Chargement d'Instructions



- Interruptions du flux d'instructions:
  - branchements
  - défauts de cache instruction
- Eviter l'interruption du flux d'instructions:
  - Nombre d'instructions chargées à chaque cycle ≥ 4: une ou plusieurs lignes du cache instruction (cache multi-port)
  - Tampon (*buffer*) pour conserver les instructions chargées par anticipation et éviter les bulles de pipeline



Décodage, Dépendances et Mise en Attente




```
L2:
    move    r3,r7
    lw      r8,(r3)
    add     r3,r3,4
    lw      r9,(r3)
    ble     r8,r9,L3
```

RAW, WAW, WAR

- Les instructions sont sorties du tampon d'instructions.
- On détermine les dépendances de données entre instructions.
- On élimine les «fausses» dépendances liées aux alias de registres.

- RAW (*Read After Write*): vraie dépendance.
- WAW (*Write After Write*): risque d'écriture dans le désordre (fausse dépendance).
- WAR (*Write After Read*): risque d'écriture avant lecture, i.e., avant qu'une donnée ait été utilisée (fausse dépendance).

Alias de Registres: Renommage

- Le maintien de la compatibilité binaire empêche l'augmentation du nombre de registres.
- Nombre de zones de stockage *physiques* > nombre de registres *logiques* (ceux définis par le jeu d'instructions).
- Zones de stockage physiques = banc de registres et **tampon de réordonnancement** (*ReOrder Buffer* ou *ROB*).
- A chaque instruction est associée une entrée dans le ROB; la *valeur* produite par une instruction est stockée dans le ROB; le registre logique de destination est remplacé par le nom d'une entrée dans le ROB: on *renomme* le registre.
- Une table indique où se trouve la valeur courante d'un registre logique (dans le banc de registres ou dans le ROB); lors du chargement d'une instruction, ses opérandes peuvent être dans le ROB ou le banc de registres.

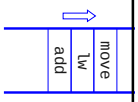
⇒ Elimination des alias de registres.

⇒ Détermination des vraies dépendances.

```
L2:
    move    r3, r7
    lw      r8, (r3)
    add     r3, r3, 4
```

Registre logique	Zone de stockage physique
r_3	ROB_8
...	...
r_8	ROB_7

Registre physique	Valeur
r_3	produite par add
...	...
r_8	produite par lw



Mise en Attente (*Dispatch*)

- Après résolution des dépendances, envoi des instructions pour exécution (*Dispatch*).
- Stations de réservation: pour chaque unité fonctionnelle (ou groupe d'unités), une file d'attente des instructions.
- Une instruction est exécutée si:
  - toutes ses opérandes sont disponibles
  - une unité fonctionnelle est disponible
- Algorithme de Tomasulo

```
add    r3, r3, 4
```

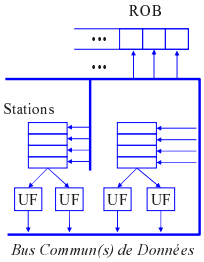
Opération	Source 1	Donnée 1	Valide 1	Source 2	Donnée 2	Valide 2	Résultat

Stations de réservation pour les unités entières (ALU)

ALUs

Envoi aux Unités Fonctionnelles (*Issue*)

- Lorsqu'une instruction peut s'exécuter, elle est envoyée à l'unité fonctionnelle.
- Après exécution, le résultat est propagée par un (des) bus:
  - à la zone de stockage de destination (entrée du ROB)
  - à toutes les stations de réservation
- Les instructions qui attendaient ce résultat peuvent être exécutées immédiatement.
- ⇒ Modèle interne plus proche du *dataflow* que de *von Neumann*: les instructions s'exécutent au fur et à mesure que leurs données sont disponibles, et non selon l'ordre du programme.



Accès Mémoire

- Calcul d'adresse dans ALU ou unités fonctionnelles dédiées.
- Requête cache+TLB.
- Contraintes:
  - Plusieurs requêtes mémoire par cycle: caches multiports.
  - Garantir une exécution correcte: dépassements `load/store`.
  - `stores` dans l'ordre; exécution spéculative des `loads`

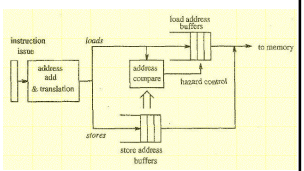
```
sw    r1, 12(r2)    ; 12+r2=100
...
lw    r4, 20(r5)    ; 20+5=100
```

Cycle	Action
10	`sw` est dispatché; attend `r2`
11	`lw` est dispatché; opérandes prêtes
12	Calcul d'adresse pour `lw`
13	Accès mémoire pour `lw` `r4` ne contient pas la bonne valeur
14	`r2` est prêt; calcul d'adresse pour `sw`
15	Accès mémoire pour `sw`

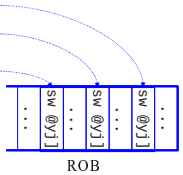
Accès Mémoire

- Lors d'un accès mémoire en lecture, on vérifie qu'il n'existe pas une écriture à la même adresse en cours.
- Si l'adresse de certaines écritures n'est pas connue: attente ou spéculation.
- On peut effectuer la même vérification en écriture (plus efficace mais pas indispensable).

```
for (i=0; i < n; i++) {
  for (j=0; j < n; j++) {
    y[j] += a[i][j]*x[j];
  }
}
```




```
...
sw @y[j]
...
sw @y[j]
...
sw @y[j]
...
```



Accès Mémoire et Exécution dans le Désordre

- L'exécution des instructions dans le désordre permet de masquer partiellement la latence en cas de défaut de cache.

```
lw    r1, 12(r2)
add   r2, r1, 3
...
lw    r3, 20(r5)
...
add   r4, r1, r3
```

Cycle	Action
10	`lw r1` : *miss*
11	`add r2, r1, r3` : bloqué
12	...
13	`lw r3` : *hit*
14	...
15	`lw r1` : terminé
16	`add r2, r1, r3` : terminé
17	`add r4, r1, r3` : terminé
	aucun délai dû au miss

Latence mémoire: 5 cycles

Finalisation (*Commit*)

- Une instruction effectue l'étape de finalisation lorsqu'elle est en tête du ROB
  - ⇒ Les instructions effectuent l'étape de finalisation dans l'ordre du programme.
- L'état *logique* de l'architecture n'est modifié que lors de l'étape de finalisation (*commit*)
  - ⇒ il est possible d'avoir des interruptions précises malgré l'exécution dans le désordre (le ROB contient le vecteur d'exception de chaque instruction).
- Etat logique: registres et mémoire.
- Lorsqu'une instruction sort du ROB:
  - le résultat est écrit dans le registre correspondant (toutes instructions sauf écriture en mémoire)
  - OU la donnée est envoyée à la mémoire (écriture en mémoire)
- Processeur superscalaire de degré *n*: *n* instructions peuvent effectuer simultanément l'étape de finalisation.
- Si l'instruction en tête du ROB n'a pas terminé son exécution: le processeur est bloqué.