Parallélisme d'instructions

Christophe Mouilleron

ensiie
école nationale supérieure d'informatique
pour l'industrie et l'entreprise

Rappel : performances

Temps d'exécution d'un programme :

Rappel : performances

Temps d'exécution d'un programme :

$$T_{ex} = NI \times CPI \times T_c$$

où :

- NI = nombre d'instructions
- CPI = nombre moyen de cycles par instruction
- T_c = durée d'un cycle

Objectif : diminuer *CPI* / augmenter *IPC*

Plan

Execution séquentielle

Déroulement d'un cycle :

- Lecture / décodage de l'instruction — Dec
- Lecture des opérandes — Lec
- Executer l'instruction — Exec
- Écrire le résultat — Ecr

Execution séquentielle de plusieurs cycles :

Execution pipelinée

Idée :

- 1 cycle = 4 étapes
- travailler étape par étape
- commencer l'étape 1 dès que possible

Pipeline :

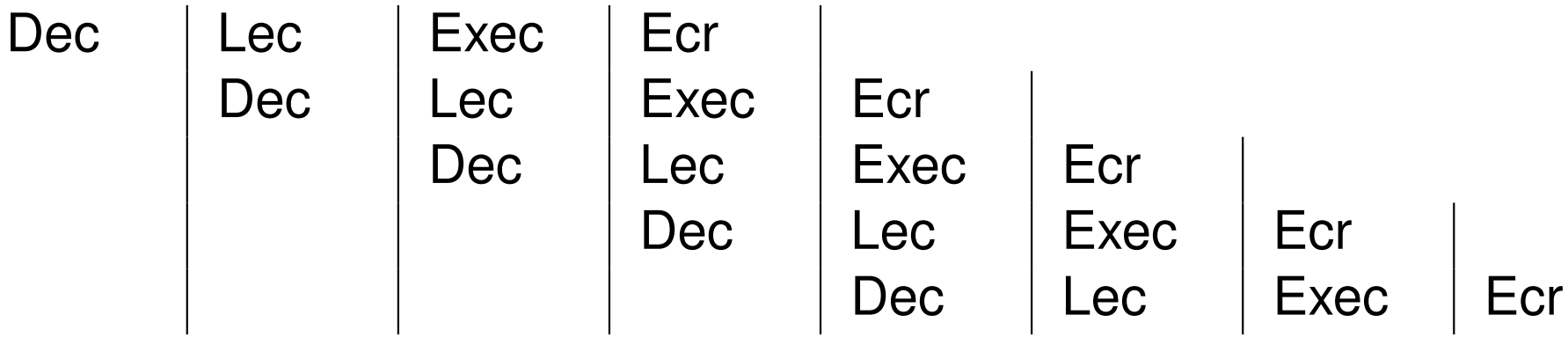
Dec	Lec	Exec	Ecr

Execution pipelinée

Idée :

- 1 cycle = 4 étapes
- travailler étape par étape
- commencer l'étape 1 dès que possible

Pipeline :



Séquentiel VS Pipeline (1)

Hypothèse H1 = chaque étape prend 0.2 ns

Hypothèse H2 = Exec en 0.2 ns, reste en 0.3 ns

Séquentiel :

Temps d'exécution en ns ?

	H1	H2
1 cycle		
5 cycles		
n cycles		

Séquentiel VS Pipeline (1)

Hypothèse H1 = chaque étape prend 0.2 ns

Hypothèse H2 = Exec en 0.2 ns, reste en 0.3 ns

Séquentiel :

Dec	Lec	Exec	Ecr	Dec	Lec	Exec	Ecr	Dec	···

Temps d'exécution en ns ?

	H1	H2
1 cycle	0.8	1.1
5 cycles		
n cycles		

Séquentiel VS Pipeline (1)

Hypothèse H1 = chaque étape prend 0.2 ns

Hypothèse H2 = Exec en 0.2 ns, reste en 0.3 ns

Séquentiel :

Dec	Lec	Exec	Ecr	Dec	Lec	Exec	Ecr	Dec	···

Temps d'exécution en ns ?

	H1	H2
1 cycle	0.8	1.1
5 cycles	4	5.5
n cycles	$0.8n$	$1.1\,n$

Séquentiel VS Pipeline (2)

Pipeline :

Dec	Lec	Exec	Ecr				
	Dec	Lec	Exec	Ecr			
		Dec	Lec	Exec	Ecr		
			Dec	Lec	Exec	Ecr	
				Dec	Lec	Exec	Ecr

Temps d'exécution en ns ?

	H1	H2
1 cycle		
5 cycles		
n cycles		

Séquentiel VS Pipeline (2)

Pipeline :

Dec	Lec	Exec	Ecr				
	Dec	Lec	Exec	Ecr			
		Dec	Lec	Exec	Ecr		
			Dec	Lec	Exec	Ecr	
				Dec	Lec	Exec	Ecr

Temps d'exécution en ns ?

	H1	H2
1 cycle	0.8	
5 cycles	1.6	
n cycles	$0.6 + 0.2n$	

Séquentiel VS Pipeline (2)

Pipeline :

Dec	Lec	Exec	Ecr				
	Dec	Lec	Exec	Ecr			
		Dec	Lec	Exec	Ecr		
			Dec	Lec	Exec	Ecr	
				Dec	Lec	Exec	Ecr

Temps d'exécution en ns ?

	H1	H2
1 cycle	0.8	1.2
5 cycles	1.6	2.4
n cycles	$0.6 + 0.2n$	$0.9 + 0.3n$

Séquentiel VS Pipeline (3)

Séquentiel :

✓ 1er cycle plus rapide, bonne latence

Pipeline :

✗ 1er cycle un peu plus lent, mauvaise latence

1 cycle = k étages

Séquentiel VS Pipeline (3)

Séquentiel :

- ✓ 1er cycle plus rapide, bonne latence
- ✗ débit limité

n cycles = $n \times$ 1 cycle

Pipeline :

- ✗ 1er cycle un peu plus lent, mauvaise latence
- ✓ meilleur débit

1 cycle = k étages

n cycles $\simeq$ n étages

Séquentiel VS Pipeline (3)

Séquentiel :

- ✓ 1er cycle plus rapide, bonne latence
- ✗ débit limité

n cycles = $n \times$ 1 cycle

Pipeline :

- ✗ 1er cycle un peu plus lent, mauvaise latence
- ✓ meilleur débit

1 cycle = k étages

n cycles $\simeq$ n étages

Bilan :

- pipeline à k étages $\Rightarrow$ $IPC \simeq k$ à plein régime

Mise en place d'un pipeline

Circuit logique $\rightarrow$ version pipelinée :

1. découper le chemin de données
   - $\rightsquigarrow$ 1 circuit $\rightarrow$ k étages
2. calculer la profondeur des étages
   - $\rightsquigarrow$ vérifier si étages de $\approx$ même profondeur
3. ajouter des bascules entre les étages
   - $\rightsquigarrow$ sauvegarde de l'état courant

Exemple : ajout d'un pipeline pour un multiplieur 4×4

$\rightsquigarrow$ cf `mul_pipeline.lgf`

Exercice

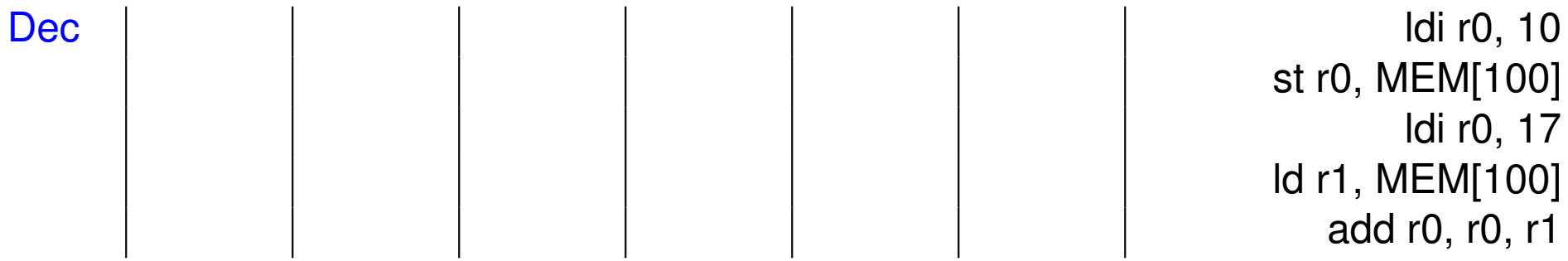
On considère le code assembleur suivant :

```
ldi r0, 10
st r0, MEM[100]
ldi r0, 17
ld  r1, MEM[100]
add r0, r0, r1
```

Décrire l'exécution de ce code :

1. dans le cas séquentiel,
2. en présence du pipeline décrit précédemment.

Exercice – Réponse si pipeline



1. décodage premier `ldi`
2.
3.
4.
5.
6.
7.
8.

Exercice – Réponse si pipeline

Dec	Lec						ldi r0, 10
	Dec						st r0, MEM[100]
							ldi r0, 17
							ld r1, MEM[100]
							add r0, r0, r1

1. décodage premier `ldi`
2. décodage `st`
3.
4.
5.
6.
7.
8.

Exercice – Réponse si pipeline

Dec	Lec
	Dec
		Exec
		Lec
		Dec
							ldi r0, 10
							st r0, MEM[100]
							ldi r0, 17
							ld r1, MEM[100]
							add r0, r0, r1

1. décodage premier `ldi`
2. décodage `st`
3. lecture de `r0` + décodage deuxième `ldi`
4.
5.
6.
7.
8.

Exercice – Réponse si pipeline

Dec	Lec	Exec	Ecr				ldi r0, 10
	Dec	Lec	Exec				st r0, MEM[100]
		Dec	Lec				ldi r0, 17
			Dec				ld r1, MEM[100]
							add r0, r0, r1

1. décodage premier `ldi`
2. décodage `st`
3. lecture de `r0` + décodage deuxième `ldi`
4. écriture de 10 dans `r0` + décodage `ld`
5.
6.
7.
8.

Exercice – Réponse si pipeline

Dec	Lec	Exec	Ecr				ldi r0, 10
	Dec	Lec	Exec	Ecr			st r0, MEM[100]
		Dec	Lec	Exec			ldi r0, 17
			Dec	Lec			ld r1, MEM[100]
				Dec			add r0, r0, r1

1. décodage premier `ldi`
2. décodage `st`
3. lecture de `r0` + décodage deuxième `ldi`
4. écriture de 10 dans `r0` + décodage `ld`
5. écriture de ? à l'adresse 100 + lecture de ? + décodage `add`
6.
7.
8.

Exercice – Réponse si pipeline

Dec	Lec	Exec	Ecr				ldi r0, 10
	Dec	Lec	Exec	Ecr			st r0, MEM[100]
		Dec	Lec	Exec	Ecr		ldi r0, 17
			Dec	Lec	Exec		ld r1, MEM[100]
				Dec	Lec		add r0, r0, r1

1. décodage premier `ldi`
2. décodage `st`
3. lecture de `r0` + décodage deuxième `ldi`
4. écriture de 10 dans `r0` + décodage `ld`
5. écriture de ? à l'adresse 100 + lecture de ? + décodage `add`
6. écriture de 17 dans `r0` + lecture de `r0` et `r1`
7.
8.

Exercice – Réponse si pipeline

Dec	Lec	Exec	Ecr				ldi r0, 10
	Dec	Lec	Exec	Ecr			st r0, MEM[100]
		Dec	Lec	Exec	Ecr		ldi r0, 17
			Dec	Lec	Exec	Ecr	ld r1, MEM[100]
				Dec	Lec	Exec	add r0, r0, r1

1. décodage premier `ldi`
2. décodage `st`
3. lecture de `r0` + décodage deuxième `ldi`
4. écriture de 10 dans `r0` + décodage `ld`
5. écriture de ? à l'adresse 100 + lecture de ? + décodage `add`
6. écriture de 17 dans `r0` + lecture de `r0` et `r1`
7. écriture de ? dans `r1` + calcul de ?
8.

Exercice – Réponse si pipeline

Dec	Lec	Exec	Ecr					ldi r0, 10
	Dec	Lec	Exec	Ecr				st r0, MEM[100]
		Dec	Lec	Exec	Ecr			ldi r0, 17
			Dec	Lec	Exec	Ecr		ld r1, MEM[100]
				Dec	Lec	Exec	Ecr	add r0, r0, r1

1. décodage premier `ldi`
2. décodage `st`
3. lecture de `r0` + décodage deuxième `ldi`
4. écriture de 10 dans `r0` + décodage `ld`
5. écriture de ? à l'adresse 100 + lecture de ? + décodage `add`
6. écriture de 17 dans `r0` + lecture de `r0` et `r1`
7. écriture de ? dans `r1` + calcul de ?
8. écriture de ? dans `r0`

résultat non défini

Problèmes liés au pipeline

Aléas de données = utilisation avant écriture d'une valeur RaW

ex. exercice, étape 6

$\rightsquigarrow$ lecture de `r0` avant écriture de 17 dans `r0`

Problèmes liés au pipeline

Aléas de structure = accès à la même ressource par 2 étages

ex. exercice, étape 5

⇝ double accès à la mémoire (Lec et Ecr) triple avec Dec

Aléas de données = utilisation avant écriture d'une valeur RaW

ex. exercice, étape 6

⇝ lecture de `r0` avant écriture de 17 dans `r0`

Problèmes liés au pipeline

Aléas de structure = accès à la même ressource par 2 étages

ex. exercice, étape 5

⇝ double accès à la mémoire (Lec et Ecr) triple avec Dec

Aléas de données = utilisation avant écriture d'une valeur RaW

ex. exercice, étape 6

⇝ lecture de `r0` avant écriture de 17 dans `r0`

Aléas de contrôle = branchement conditionnels

⇝ adresse de la prochaine instruction non connu à l'avance

Solutions (1)

Solutions (1)

1. retarder l'exécution en cas d'aléa — paresser
   - ✓ corrige tous les aléas
   - ✓ simple ajout d'instructions `nop`
   - ✗ vide le pipeline ⇒ IPC ↘

```
ldi r0, 10
st r0, MEM[100]
ldi r0, 17
ld  r1, MEM[100]
add r0, r0, r1
```

⇝

Solutions (1)

1. retarder l'exécution en cas d'aléa — paresser
   - ✓ corrige tous les aléas
   - ✓ simple ajout d'instructions `nop`
   - ✗ vide le pipeline ⇒ IPC ↘

```
ldi r0, 10
st r0, MEM[100]
ldi r0, 17
ld  r1, MEM[100]
add r0, r0, r1
```

⇝

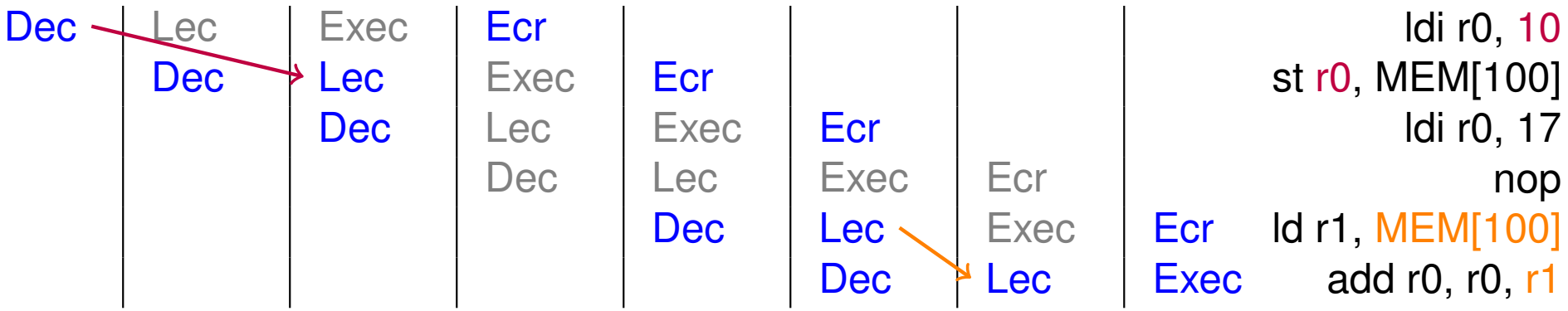
```
   ldi r0, 10
nop ; nop
   st r0, MEM[100]
   ldi r0, 17
nop
   ld  r1, MEM[100]
nop ; nop
   add r0, r0, r1
```

Solutions (2)

2. bypass = transmettre valeur calculée avant écriture — anticiper
   - ✓ règle partiellement les aléas de données
   - ✗ nouvelles sources $\Rightarrow$ multiplexeurs
   - ✗ ajout portes logiques pour contrôler le bypass

Solutions (2)

2. bypass = transmettre valeur calculée avant écriture — anticiper
   - ✓ règle partiellement les aléas de données
   - ✗ nouvelles sources ⇒ multiplexeurs
   - ✗ ajout portes logiques pour contrôler le bypass



Solutions (3)

3. changer le code — adapter
   - ✓ meilleure alloc. registre $\Rightarrow$ moins d'aléas de données
   - ✗ travail fastidieux à la main / compilateur dédié

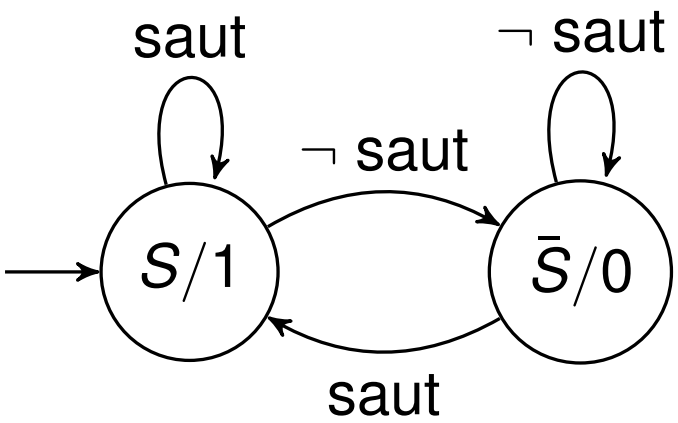
4. prédiction de saut — prédire
   - ✓ minimise l'impact des aléas de contrôle
   - ✗ dépend de la qualité de la prédiction

Prédiction de sauts (1)

Idée :

- choix entre 1 (faire le saut) et 0 (passer à l'instr. suivante)
- décision selon adresse de l'instr. de saut
- association adresse $\rightarrow$ état, selon historique

Prédicteur à 1 bit



État stocké dans une SRAM

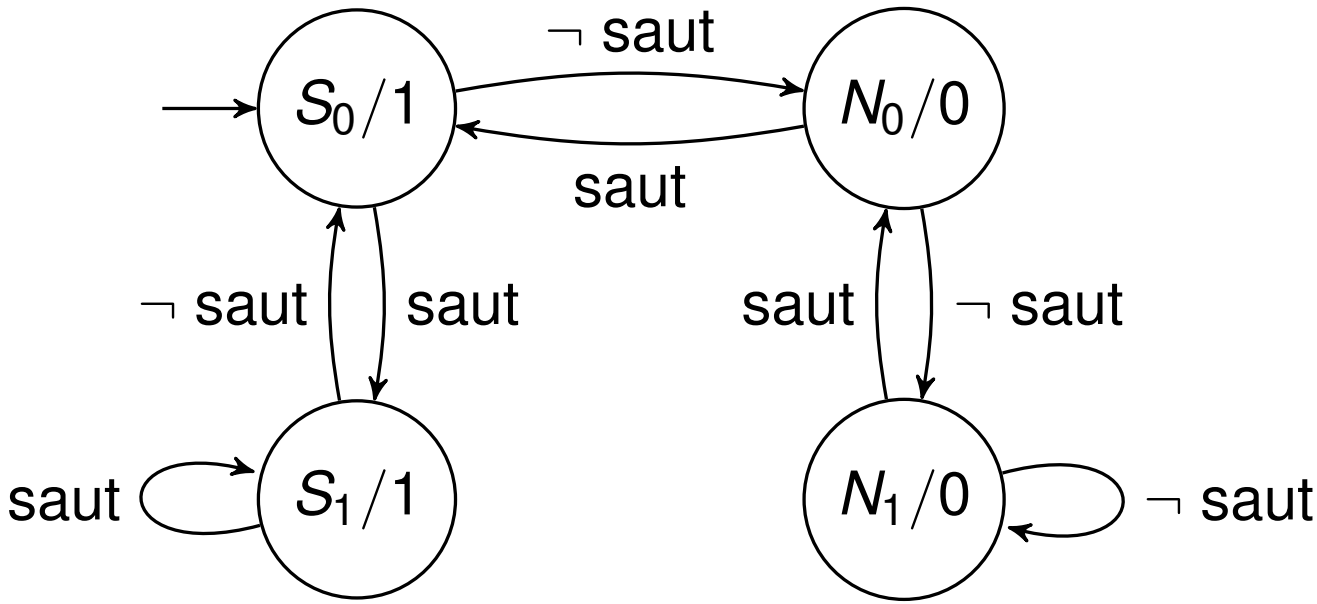
État codé sur 1 bit

Mise à jour selon issue à l'exécution :

saut ou ¬ saut

Prédiction de sauts (2)

Prédicteur à 2 bits



Davantage d'états
$\Rightarrow$ meilleure prise en compte de l'historique
$\Rightarrow$ meilleure prédiction

Ce qu'il faut retenir sur les pipelines

Pipeline dans le processeur :

- ✓ améliore le débit — mais latence + grande
- ✓ IPC = nb. étages en plein régime
- ✓ diminue la profondeur entre deux registres $\Rightarrow T_c \searrow$
- ✗ introduit des problèmes = aléas
- ✗ difficile à mettre en place — en maintenant un IPC élevé

Ce qu'il faut retenir sur les pipelines

Pipeline dans le processeur :

- ✓ améliore le débit — mais latence + grande
- ✓ IPC = nb. étages en plein régime
- ✓ diminue la profondeur entre deux registres $\Rightarrow T_c \searrow$
- ✗ introduit des problèmes = aléas
- ✗ difficile à mettre en place — en maintenant un IPC élevé

Technique applicable dans de nombreux contextes :

- pipeline 3D en traitement d'image
- restauration libre-service
- …

Plan

Limite des pipelines

Taille du pipeline ? Core i7 Nehalem = 14*

- nb. étages élevé $\Rightarrow$ *IPC* max élevé mais remplissage faible
- nb. étages faible $\Rightarrow$ gain moindre

Limite des pipelines

Taille du pipeline ? Core i7 Nehalem = 14*

- nb. étages élevé $\Rightarrow$ *IPC* max élevé mais remplissage faible
- nb. étages faible $\Rightarrow$ gain moindre

Autre solution pour augmenter IPC :

- utiliser plusieurs unités de calcul . . . ici : 2

Dec	Lec	Exec	Ecr
Dec	Lec	Exec	Ecr

Limite des pipelines

Taille du pipeline ? Core i7 Nehalem = 14*

- nb. étages élevé $\Rightarrow$ *IPC* max élevé mais remplissage faible
- nb. étages faible $\Rightarrow$ gain moindre

Autre solution pour augmenter IPC :

- utiliser plusieurs unités de calcul . . . ici : 2
- . . . et du pipeline

Dec	Lec	Exec	Ecr	
Dec	Lec	Exec	Ecr	
	Dec	Lec	Exec	Ecr
	Dec	Lec	Exec	Ecr

Parallélisme d'instructions

Architecture superscalaire :

- gestion matérielle du parallélisme
- circuit pour choisir où/quand lancer chaque instruction
- exécution potentiellement *out-of-order*

Architecture VLIW :

- gestion logicielle du parallélisme
- ordonnancement généré par le compilateur

Approche superscalaire

Dec	Lec	Exec	Ecr	
Dec	Lec	Exec	Ecr	
	Dec	Lec	Exec	Ecr
	Dec	Lec	Exec	Ecr

Idée générale :

- chargement d'un lot de *n* instructions — Core i7 Nehalem = 18*
- ordonnancement selon dépendances + contraintes des unités
- vérification de cohérence — saut ?
- passage au lot suivant

Approche VLIW

Dec	Lec	Exec	Ecr	
Dec	Lec	Exec	Ecr	
	Dec	Lec	Exec	Ecr
	Dec	Lec	Exec	Ecr

VLIW = Very Long Instruction Word

Idée générale :

- plusieurs voies de calcul potentiellement différentes
- chargement d'un VLIW à chaque cycle
- décodage de l'instruction à exécuter sur chaque voie
- pas de détection / gestion d'aléas

VLIW : exemple du ST231

ST 231 =

- processeur entier 32 bits avec 64 registres
- architecture VLIW à 4 voies
- seulement 2 multiplieurs (latence 3, débit 1) voies 2 et 4

Compilation de `r3 = r0 * r0 + r1 * r1 + r2 * r2` :

cycle	voie 1	voie 2	voie 3	voie 4
0				
1				
2				
3				
4				
5				

VLIW : exemple du ST231

ST 231 =

- processeur entier 32 bits avec 64 registres
- architecture VLIW à 4 voies
- seulement 2 multiplieurs (latence 3, débit 1) voies 2 et 4

Compilation de `r3 = r0 * r0 + r1 * r1 + r2 * r2` :

cycle	voie 1	voie 2	voie 3	voie 4
0		mul r3, r0, r0		mul r4, r1, r1
1				
2				
3				
4				
5				

VLIW : exemple du ST231

ST 231 =

- processeur entier 32 bits avec 64 registres
- architecture VLIW à 4 voies
- seulement 2 multiplieurs (latence 3, débit 1) voies 2 et 4

Compilation de `r3 = r0 * r0 + r1 * r1 + r2 * r2` :

cycle	voie 1	voie 2	voie 3	voie 4
0		mul r3, r0, r0		mul r4, r1, r1
1		mul r5, r2, r2		
2				
3				
4				
5				

VLIW : exemple du ST231

ST 231 =

- processeur entier 32 bits avec 64 registres
- architecture VLIW à 4 voies
- seulement 2 multiplieurs (latence 3, débit 1) voies 2 et 4

Compilation de `r3 = r0 * r0 + r1 * r1 + r2 * r2` :

cycle	voie 1	voie 2	voie 3	voie 4
0		mul r3, r0, r0		mul r4, r1, r1
1		mul r5, r2, r2		
2				
3	add r3, r3, r4			
4				
5				

VLIW : exemple du ST231

ST 231 =

- processeur entier 32 bits avec 64 registres
- architecture VLIW à 4 voies
- seulement 2 multiplieurs (latence 3, débit 1) voies 2 et 4

Compilation de `r3 = r0 * r0 + r1 * r1 + r2 * r2` :

cycle	voie 1	voie 2	voie 3	voie 4
0		mul r3, r0, r0		mul r4, r1, r1
1		mul r5, r2, r2		
2				
3	add r3, r3, r4			
4	add r3, r3, r5			
5				

Comparaison des deux approches

Superscalaire	VLIW
✗ coût matériel	✓ parallélisation à la compilation
✓ transparent pour l'utilisateur	✗ exige un support par le compilateur
✓ portabilité	
✗ performance réelle difficile à prédire	✓ parallélisme explicite
	✓ performances connues avant l'exécution