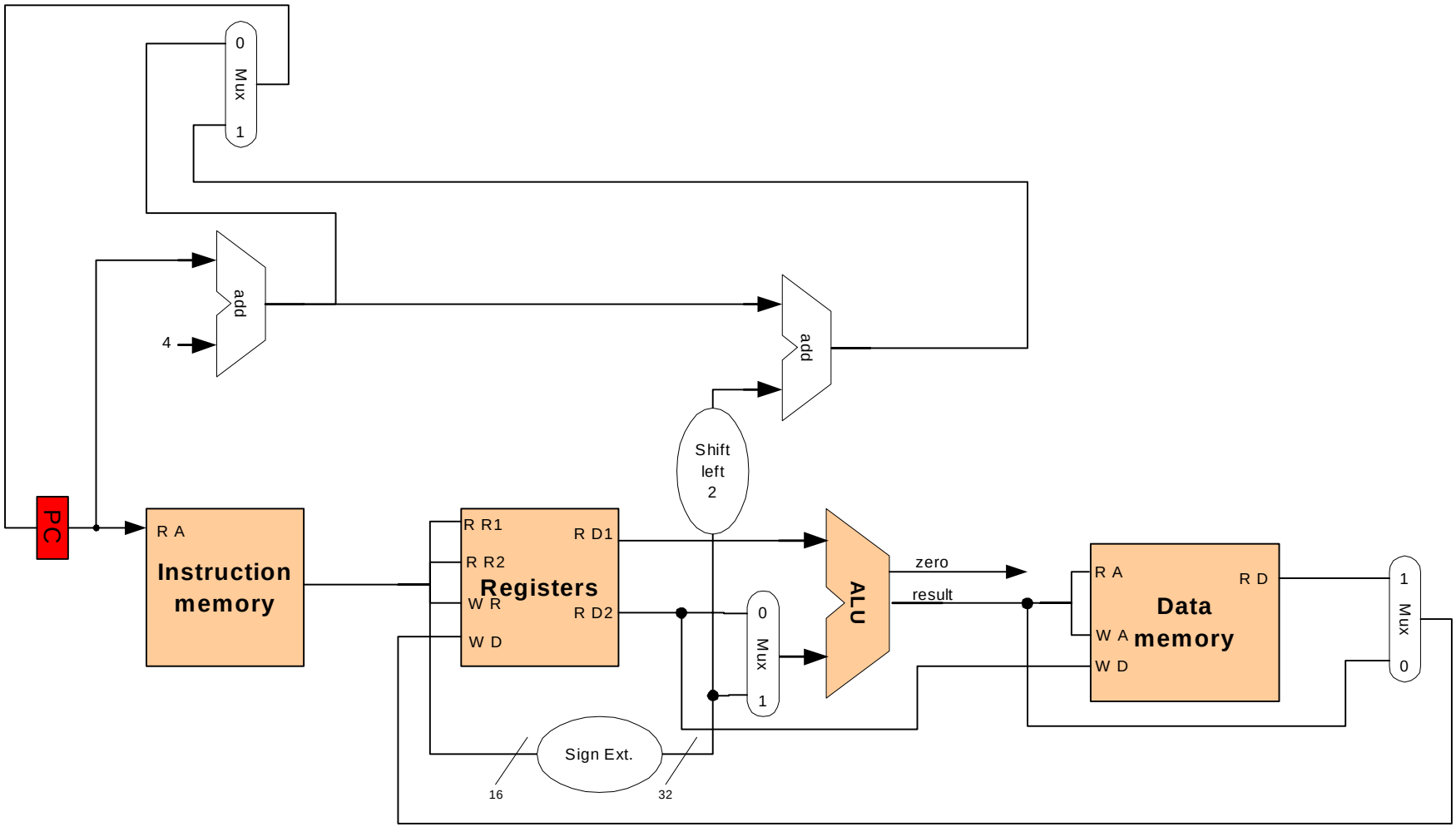
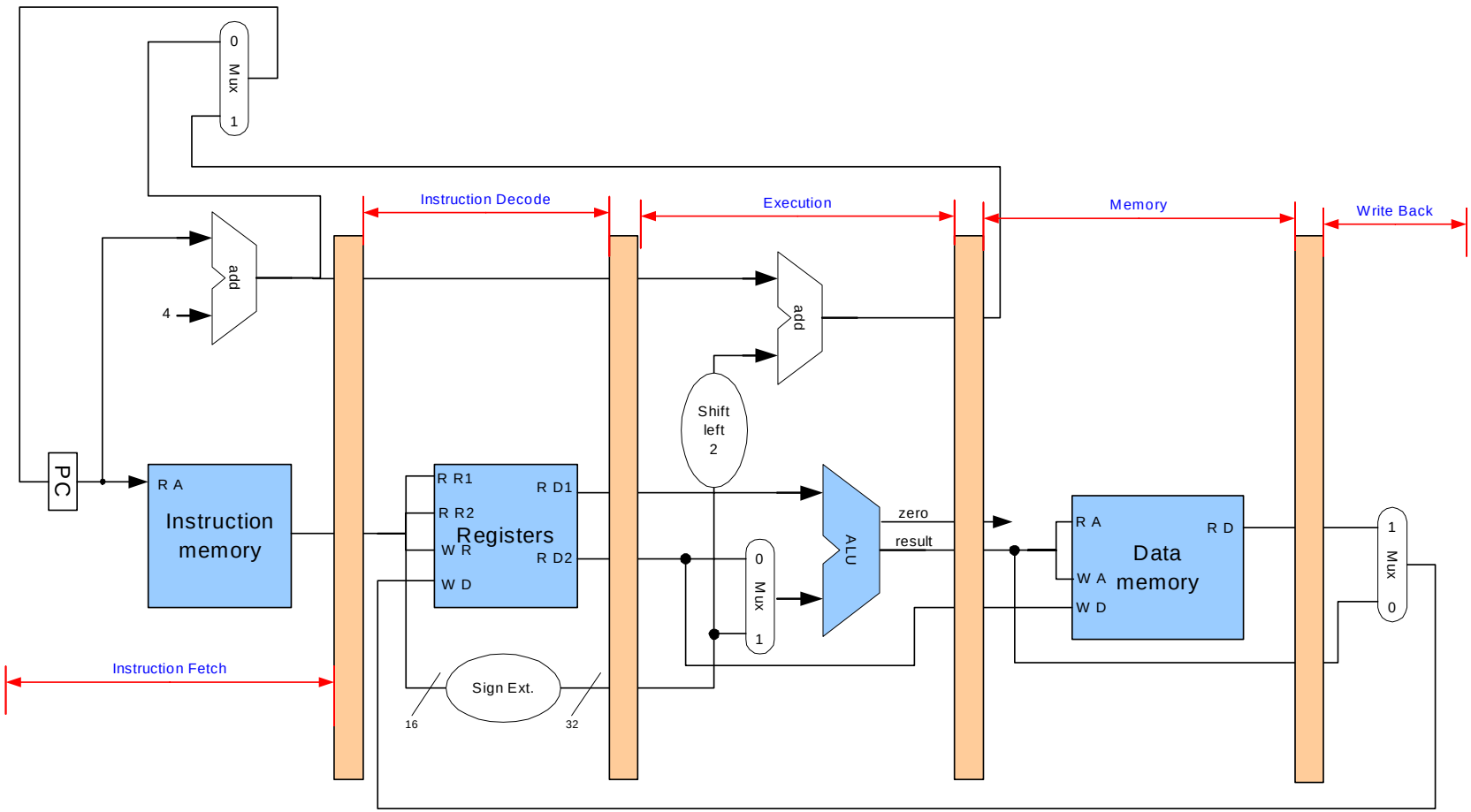
Introduction au Pipeline

Architecture du DLX

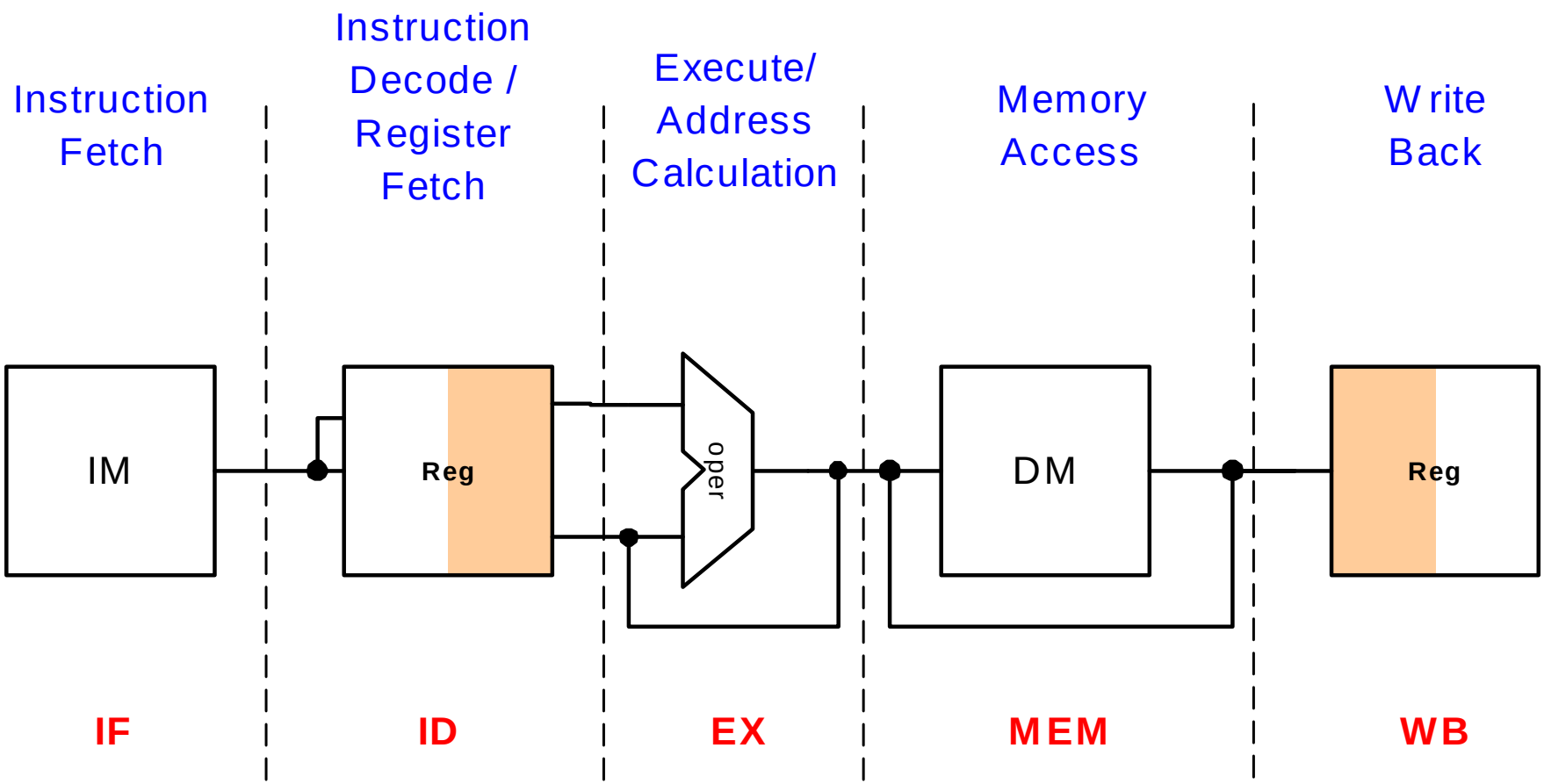


Chaque instruction peut être exécutée en 4 ou 5 cycles

Pipeline de base du DLX

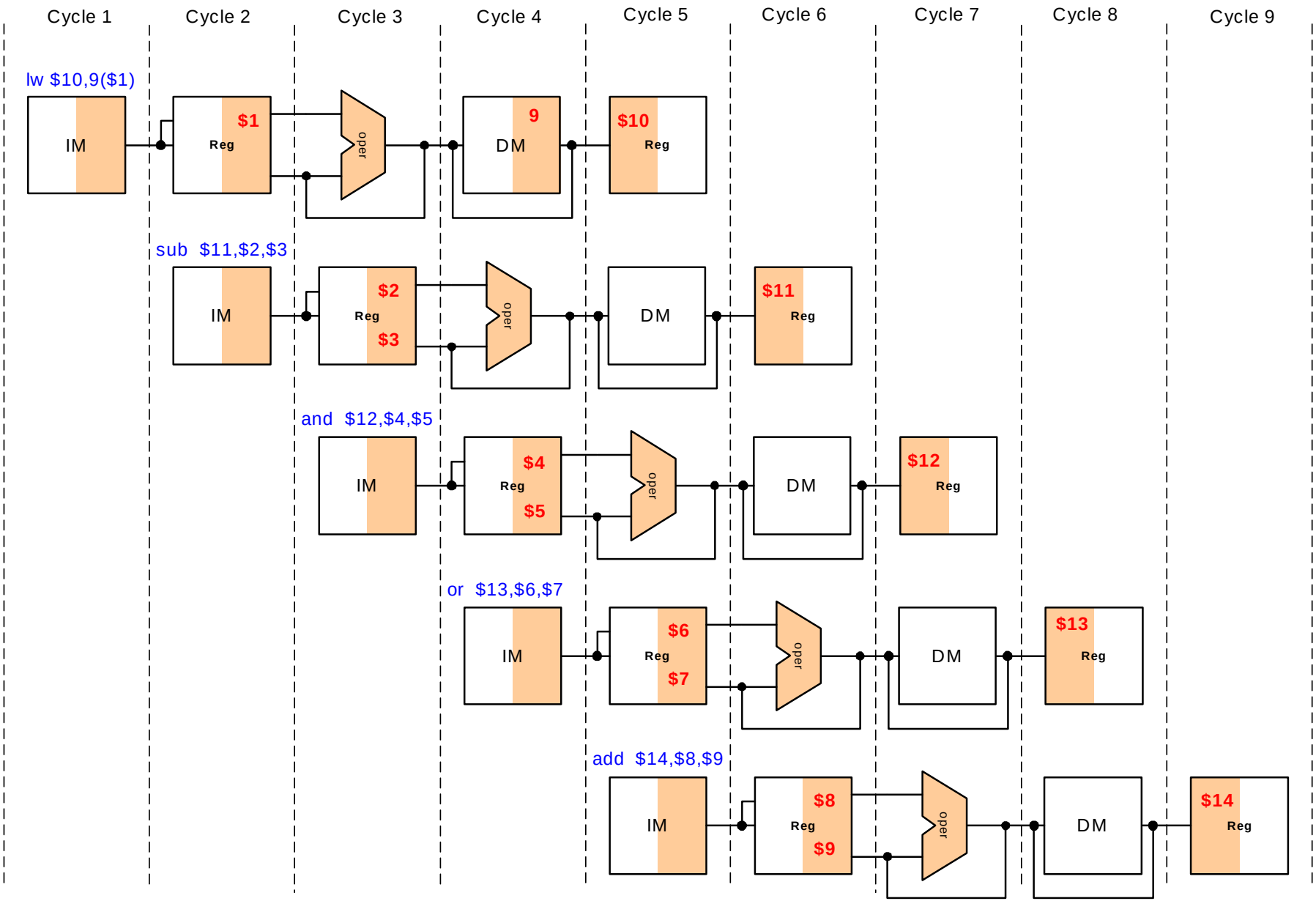


Vision simplifiée



Exemple d'exécution en pipeline

- lw $10, 9($1)
- sub $11, $2, $3
- and $12, $4, $5
- or $13, $6, $7
- add $14, $8, $9


Cycle 1
Cycle 2
Cycle 3
Cycle 4
Cycle 5
Cycle 6
Cycle 7
Cycle 8
Cycle 9
lw $10,9($1)
IM
Reg
$1
oper
DM
9
$10
Reg
sub $11,$2,$3
IM
Reg
$2
$3
oper
DM
$11
Reg
and $12,$4,$5
IM
Reg
$4
$5
oper
DM
$12
Reg
or $13,$6,$7
IM
Reg
$6
$7
oper
DM
$13
Reg
add $14,$8,$9
IM
Reg
$8
$9
oper
DM
$14
Reg

Facteur d'accélération

- C_m: cycle mineur
- C_M: cycle majeur
- E: nombre d'étages
- $C_M = C_m * E$
- A: facteur d'accélération

$$A = \frac{\text{nombre de cycles sans pipeline}}{\text{nombre de cycles avec pipeline}}$$

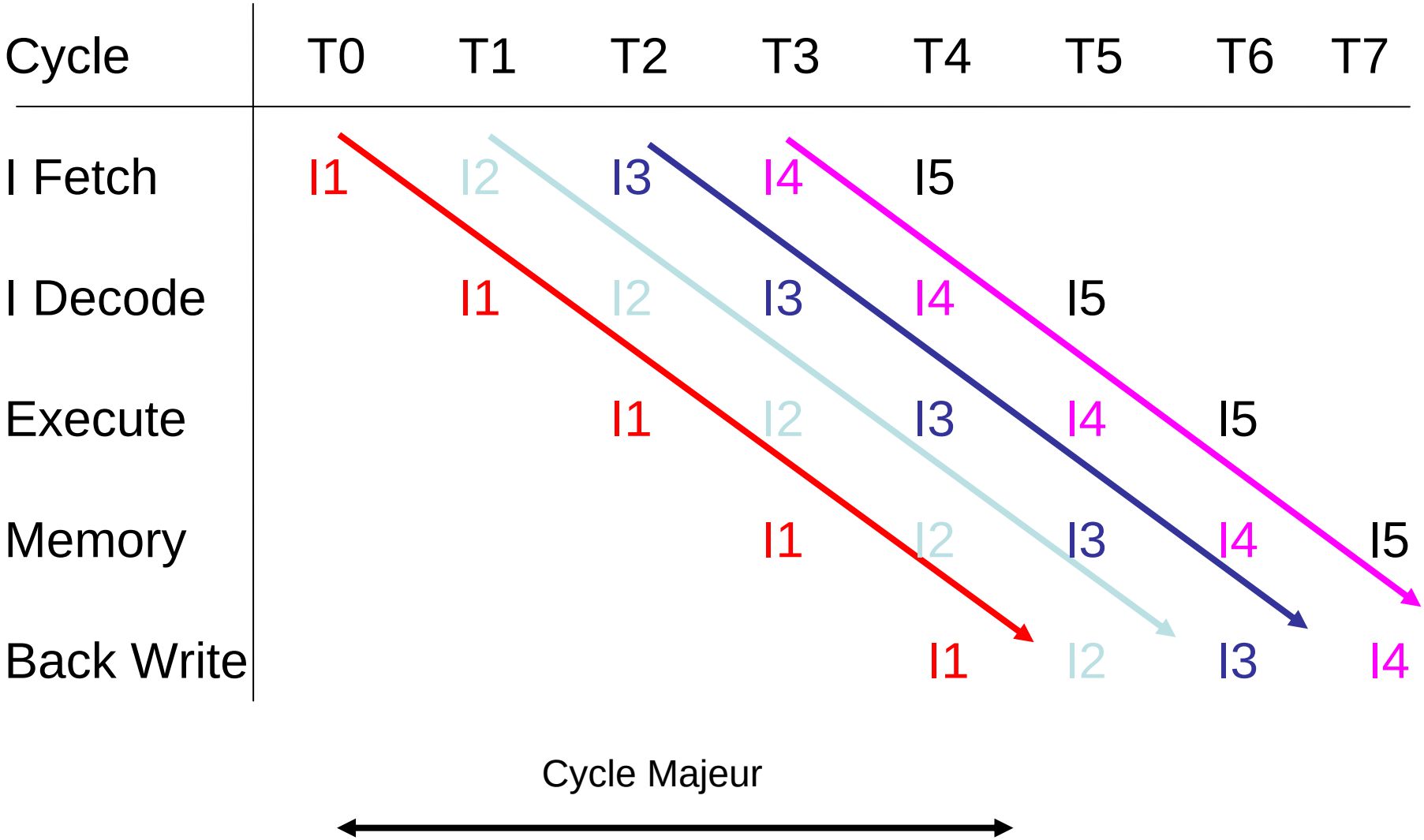
Facteur d'accélération (2)

- Pour N instructions
- Sans pipeline: $N * C_M$
- Avec pipeline en régime permanent:
- $C_M + (N-1) * C_m$
- D'où

$$A = \frac{N \times C_M}{C_M + (N-1) \times C_m} = \frac{E \times N \times C_M}{E \times C_M + (N-1) \times C_M}$$

$$A \to E \quad \text{lorsque} \quad N \gg E$$

Principe


Cycle
T0
T1
T2
T3
T4
T5
T6
T7
I Fetch
I Decode
Execute
Memory
Back Write
I1
I2
I3
I4
I5
Cycle Majeur

Chevauchements

Cycle	T0	T1	T2	T3	T4	T5	T6	T7
I1	F1	D1	E1	M1	W1			
I2		F2	D2	E2	M2	W2		
I3			F3	D3	E3	M3	W3	
I4				F4	D4	E4	M4	W4
I5					F5	D5	E5	M5

Première conclusion

- Au prix de l'ajout de matériel (registres tampon) on peut augmenter les performances de façon notable
- Forme de parallélisme « spatial »

- Semble très bien
- Sauf que …

Exemple: oddSum

N ← 5; odd ← -1; oddSum ← 0;

```
tant que  N > 0  faire
        odd ← odd + 2
        oddSum ← oddSum + odd
        N ← N - 1
fait
```

```
1)   bcl:    addi   r9,r9,2 ; odd += 2
2)           add    r10,r10,r9     ; oddSum += odd
3)           addi   r8,r8,-1       ; counter --
4)           bnez   r8,bcl
```

Exécution oddSum

IF1	ID1 Sel •R9	EX1 X ← Reg[R9]+2	Mem1 -	BW1 Reg[R9] ← X	
	IF2	ID2 Sel •R9 •R10			EX2 Y ← Reg[R10] + Reg[R9]

pénalité

Les aléas

- Une situation dans laquelle une instruction, si elle était exécuté normalement, causerait une erreur, s'appelle un aléa.
- Aléas structurels (dus au un conflit de ressources)
- Aléas de données (dépendances)
- Aléas de contrôle (branchements et autres instructions modifiant PC)

Correction des aléas

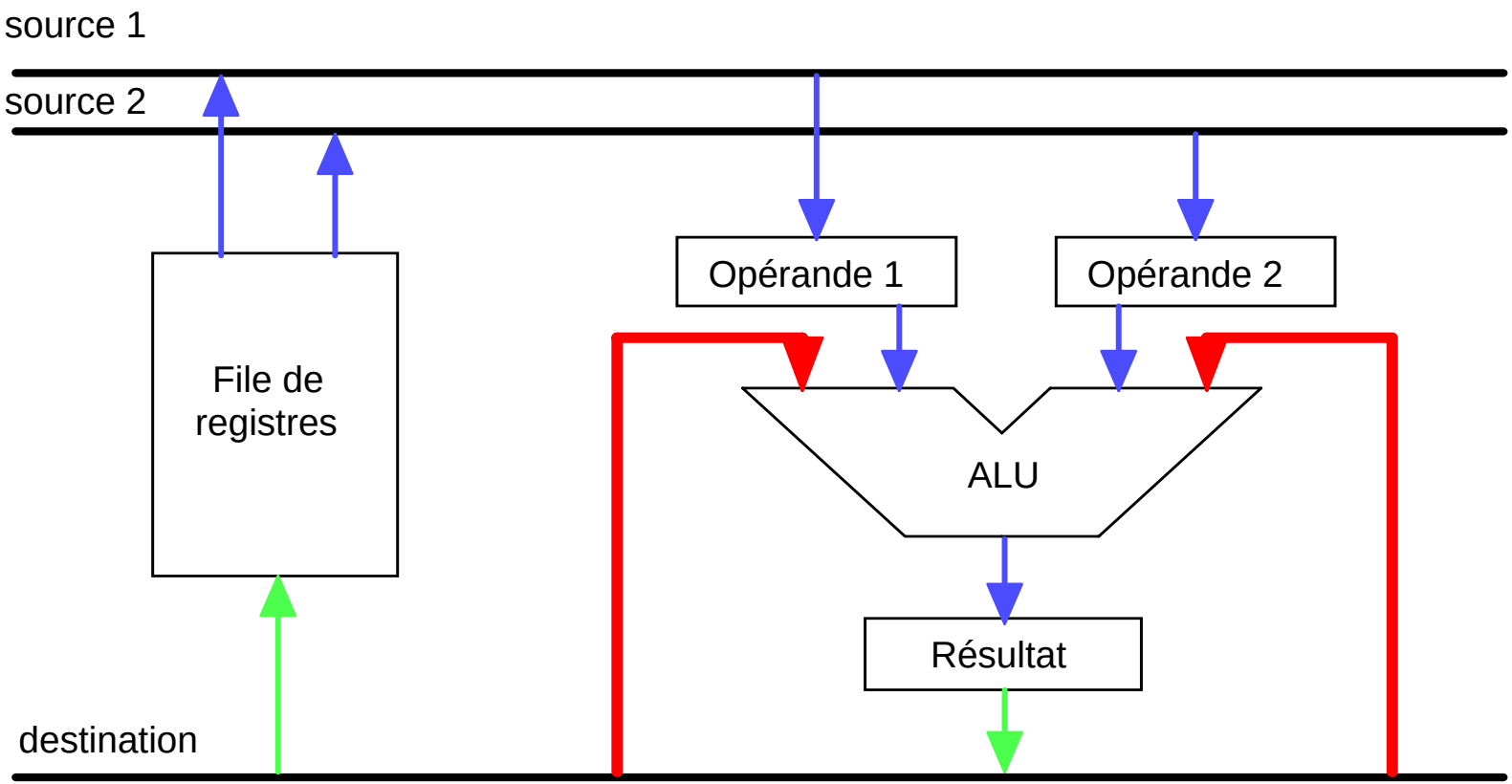
- On retarde des phases pour certaines instructions : suspension du pipeline (stall) $\Rightarrow$ perte de performance
- Dans certains cas le matériel peut corriger sans perte de temps : envoi (forward) $\Rightarrow$ hardware plus complexe et donc plus cher.
- On peut optimiser le code en déplaçant des instructions $\Rightarrow$ exercice de compilation (très) difficile

Exécution oddSum (2)

IF1	ID1 Sel •R9	EX1 X ← Reg[R9]+2	Mem1 -	BW1 Reg[R9] ← X	
	IF2	ID2 Sel •R9 •R10	EX2 Y ← Reg[R10] + X	Mem2	BW2 Reg[R10] ← Y

envoi

Modification du matériel



Aléas structurel

- Le recouvrement de l'exécution des instructions nécessite le fonctionnement pipeline des unités fonctionnelles et la duplication des ressources.
- Certaines combinaisons d'instructions peuvent entrer en conflit.
- L'exemple suivant traite du cas où il n'y a qu'un pipeline mémoire pour les données et les instructions.

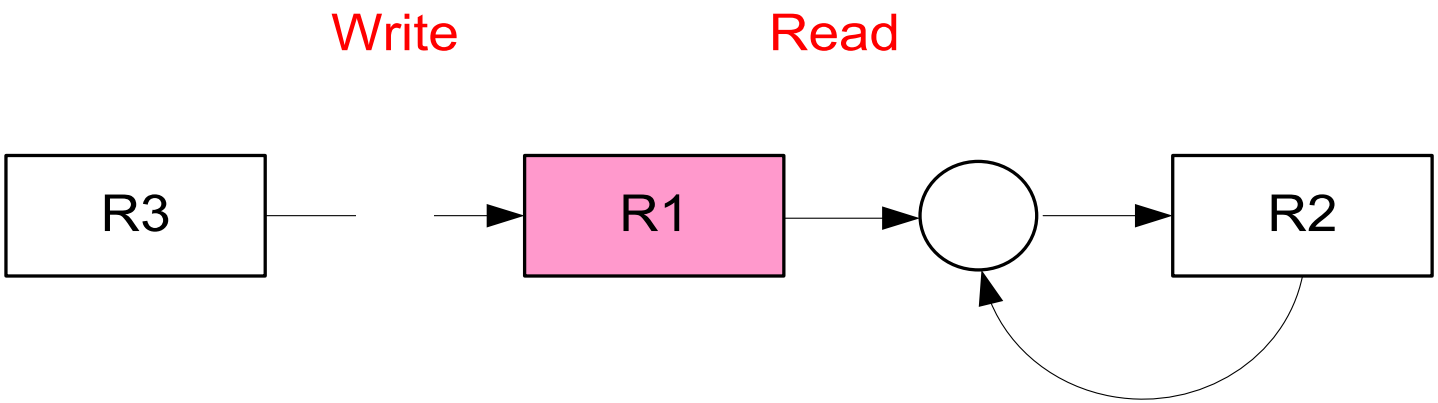
Aléas structurel : exemple

L'instruction 0 effectue un chargement

IF0	ID0	EX0	MEM0	WB0	
	IF1	ID1	EX1	MEM1	WB1
		IF2	ID2	EX2	MEM2
					
				IF3	ID3

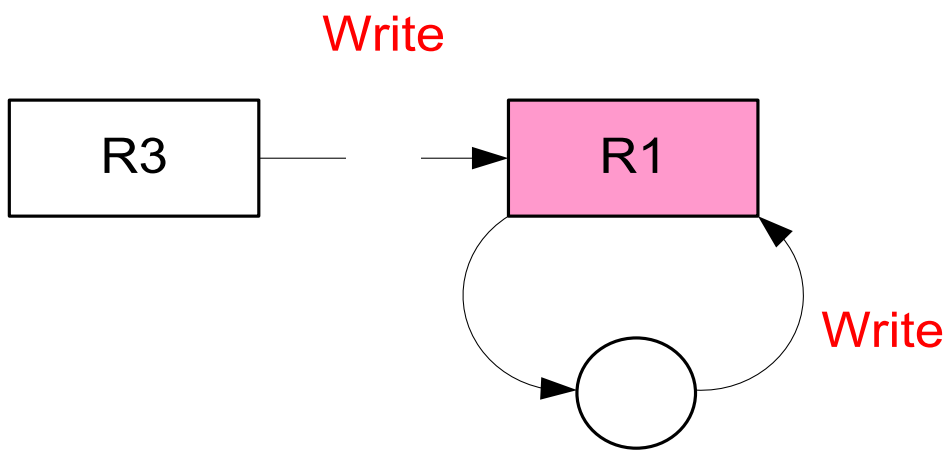
Aléas de données (1)

- Read-After-Write (RAW)
- (i1) add R1,R3,R0
- (i2) add R2,R2,R1



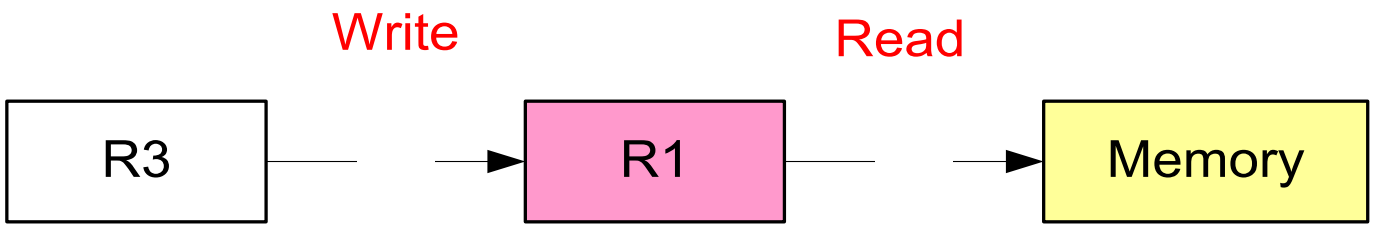
Aléas de données (2)

- Write-After-Write (WAW)
- (i1) add R1,R3,R0
- (i2) addi R1,R1,1
- Correction: retard ou envoi



Aléas de données (3)

- Write-After-Read (WAR)
- (i1) sw R1,0(R2)
- (i2) add R1,R3,R0
- Correction: retard



Aléas de contrôle

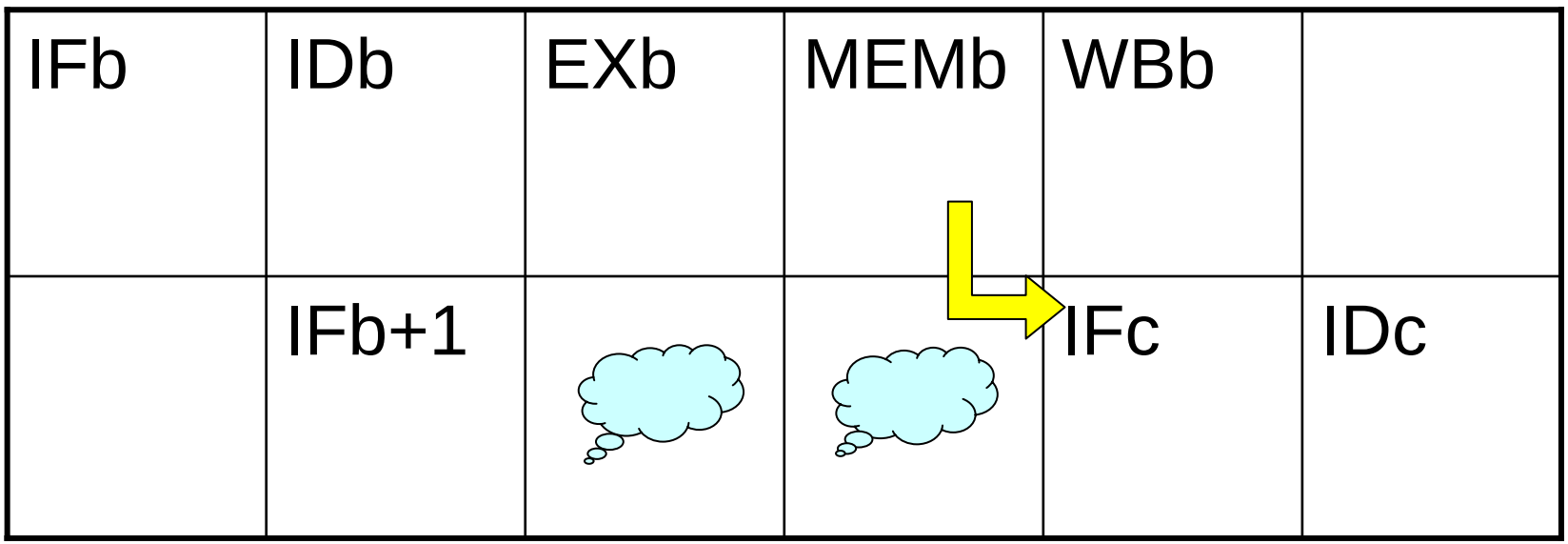
- Le problème apparaît quand le compteur programme peut être modifié pour aller à une adresse arbitraire.
- Le cas le plus courant est celui du branchement
- La solution la plus simple (et la plus pénalisante) consiste à suspendre le pipeline aussitôt qu'on détecte le branchement.

Branchement (1)

L'instruction b est une instruction de branchement.

L'instruction suivante sera l'instruction

- (b+1) si le branchement n'est pas pris,
- (c) si le branchement est pris.



Branchements (2)

- Modification du matériel pour anticiper le résultat de la décision : par ex. test à zéro et calcul adresse de branchement dans l'étage ID.
- Prédiction de branchement : en DLX on parie sur branchement non pris. Si le branchement est pris il faut vidanger (flush) le pipeline.
- Branchement différé : l'instruction suivant le branchement est toujours exécutée. Permuter des instructions ou mettre un « NOP » (travail de compilation).

Branchements : gcd

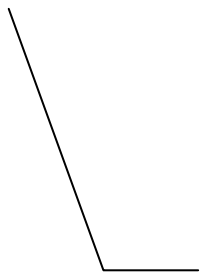
```
bcl:    beq   $t0,$t1,done
        blt   $t0,$t1,else
        sub   $t0,$t0,$t1
        j     bcl
else:   sub   $t1,$t1,$t0
        j     bcl
done:
```

```
bcl:    beq   $t0,$t1,done
        blt   $t0,$t1,else
        nop
        sub   $t0,$t0,$t1
        j     bcl
        nop
else:   sub   $t1,$t1,$t0
        j     bcl
        nop
done:
```

Branchements retardés

```
Loop:
        sll     $t0,$t0,1
        addi    $t1,$t1,-1
        beqz    $t1,Loop
Next:
        add     $t0,$t0,$t2
```

```
Loop:
        addi    $t1,$t1,-1
        beqz    $t1,Loop
        sll     $t0,$t0,1
Next:
        add     $t0,$t0,$t2
```



FAUX: Exécuté à chaque test !

Interruptions et exceptions

- En exécution séquentielle on sait exactement ce qu'on doit sauvegarder.
- Avec le pipeline:
  - Quelles instructions terminer?
  - Quelles instructions sauvegarder?
- Interruptions précises:
  - Toutes celles qui précèdent l'instruction sauvegardée ont été exécutées
  - Toutes celles qui suivent n'ont pas modifié l'état du processeur
  - Atomicité: l'instruction interrompue soit complètement exécutée, soit pas commencée.