T1 Architecture des processeurs

Pipeline scalaire

A. Mérigot

M2 SETI 2022-23

Performances d'un processeur

Formule de Hennessy-Patterson

$$T_{ex} = N_i \times \overline{CPI} \times T_c$$

N_i *Nombre d'instructions executées* : jeu d'instruction, compilateur

$\overline{CPI}$ *cycles par instruction* : microarchitecture du processeur, pipeline

T_c *temps de cycle* : technologie, microarchitecture

Exécution pipeline des instructions

Principales étapes

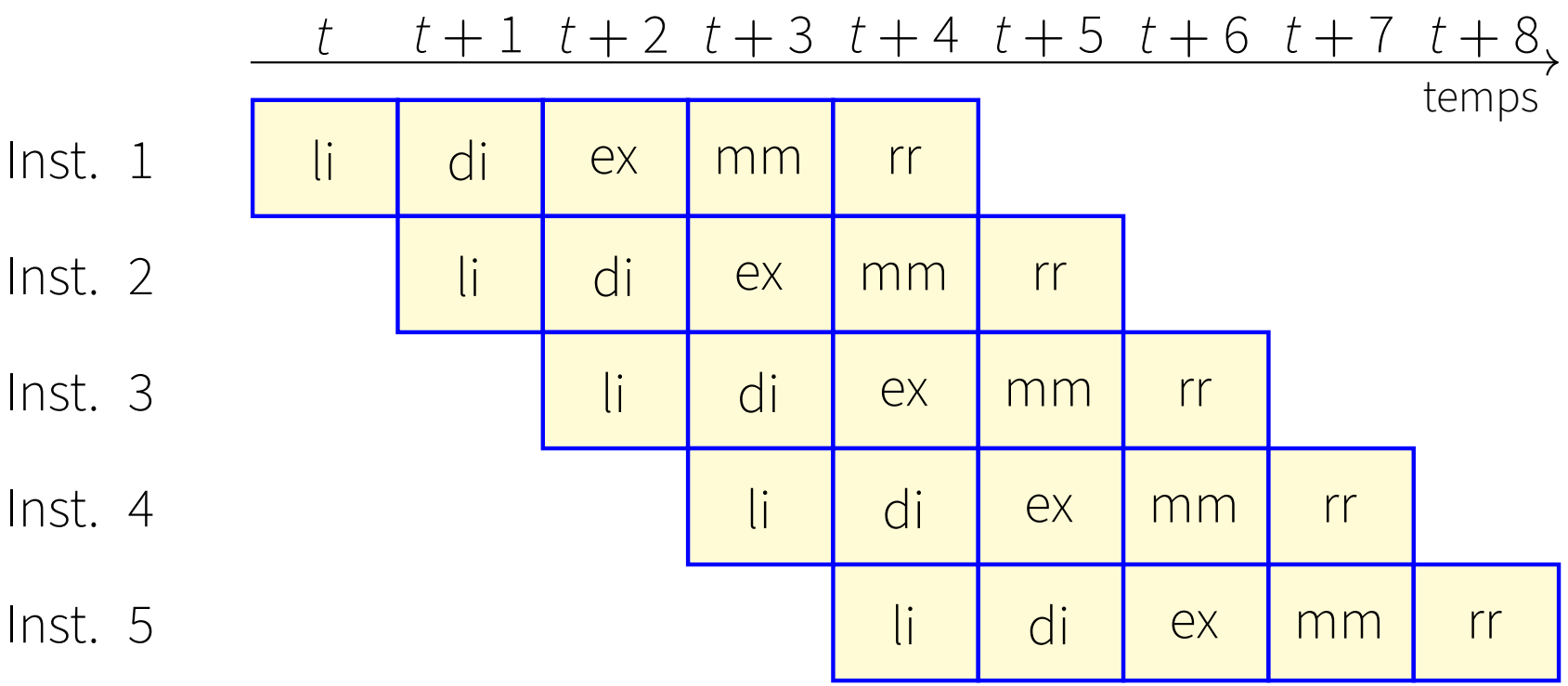
Instruction ALU	**Instruction mémoire**	**Branchement**
Lecture instruction/ Increm. `pc`	Lecture instruction/ Increm. `pc`	Lecture instruction/ Increm. `pc`
Décodage instr./ Lecture opérandes	Décodage instr./ Lecture opérandes	Décodage instr./ Calcul adr. branch.
Exécution	Calcul Adr. Mém.	Exécution
	Accès mémoire	
Rangement résultat	Rangement résultat	

Exécution pipeline des instructions (cont.)

- Instructions entières
  - LI/CP DI/LR EX ER
- Instructions flottantes
  - LI/CP DI/LR EX1 EX2 ... ER
- Instructions mémoire
  - LI/CP DI/LR CA AM ER
- Instructions branchement
  - LI/CP DI/CAB/EX

LI/CP : Lecture Instruction/
Incrémentation pc
DI/LR : Décodage Instruction/
Lecture registres
EX : Exécution (ALU)
CA : Calcul Adresse
CAB : Calcul Adresse Brcht.
AM : Accès mémoire
ER : Ecriture Registre

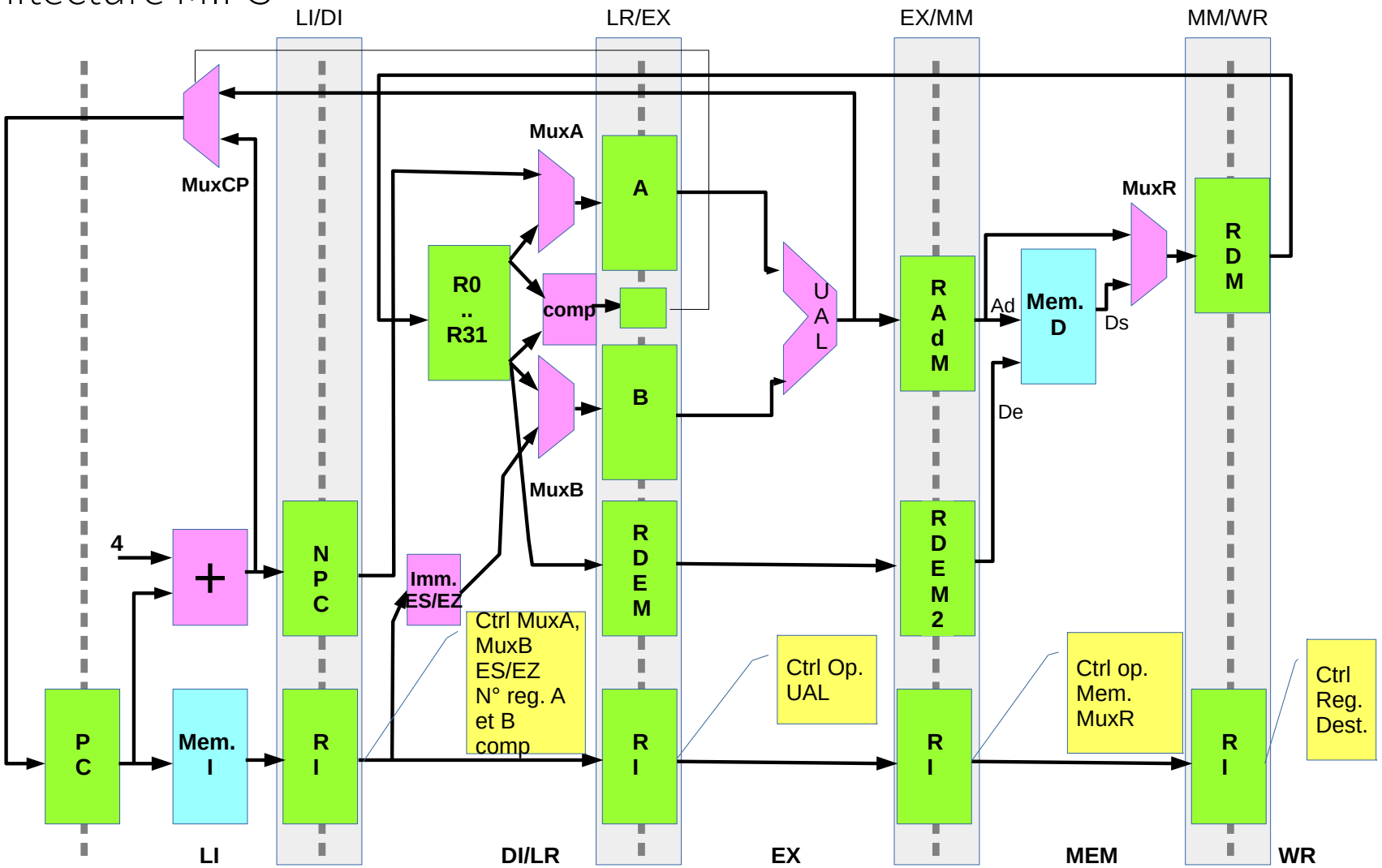
Pipeline RISC classique : exemple MIPS R2000/R3000



Latence : 5 cycles
Débit : 1 instruction/cycle

Exécution scalaire en pipeline

Architecture MIPS



Exécution scalaire en pipeline (cont.)

pipeline MIPS

li (lecture instruction)
- mem(`pc`) → `ri`
- `pc` += 4

di/lr (décodage instruction/lecture registres)
- `r0..r31` (ou `npc`) → `A` (pour UAL)
- `r0..r31` (ou imm) → `B` (idem)
- `r0..r31` → `RDEM` (pour écriture mémoire)
- calcul cond (pour branchements)

ex (exécution)
- `A` op `B` → `RAdM` (pour instr ALU et MEM)
- `REDM` → `REDM2` (pour écr. mém.)
- `A` + `B` → `pc` (pour branchement)

mm (mémoire)
- mem(`RAdM`) → `RDM` (lect. mém.)
- ou `RAdM` → `RDM` (inst UAL)
- `REDM2` → mem(`RADM`) (écrit. mém.)

rr (réécriture registres)
- `RDM` → `r0..r31` (lect mem et inst ALU)

Les aléas

A priori le pipeline permet de lancer une instruction par cycle…

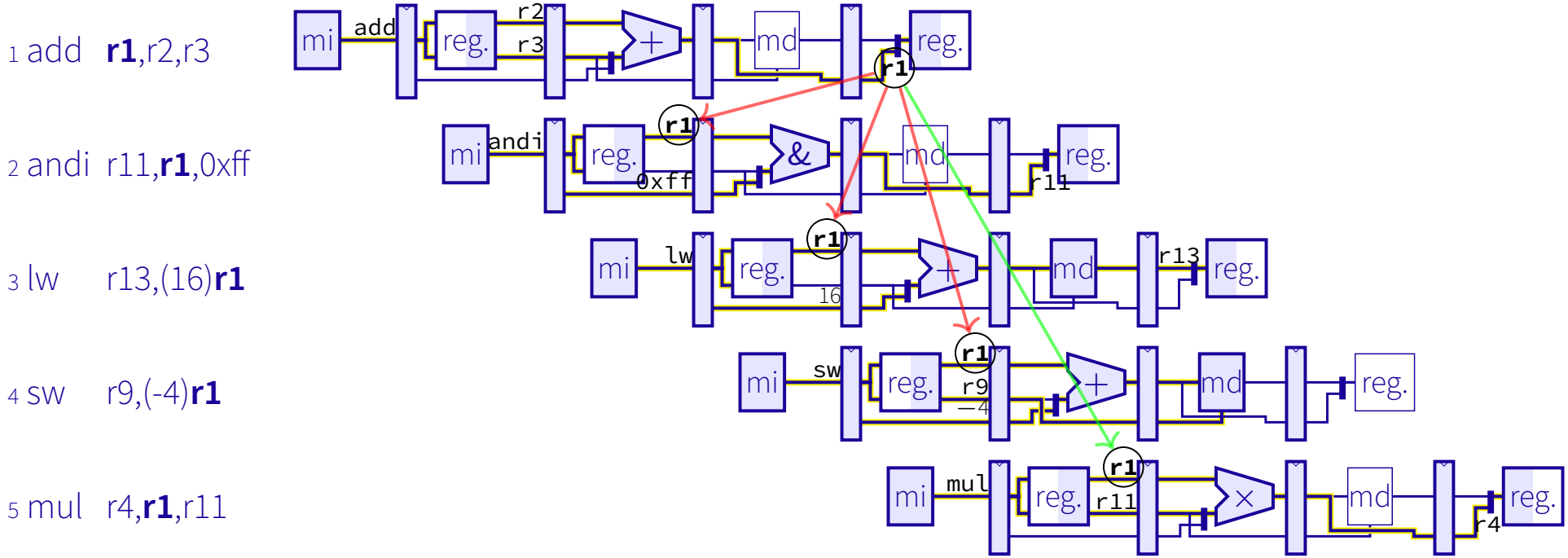
…**mais** possibilité d'*aléa*

aléa de données donnée non disponible à cause de la latence du pipeline

aléa de contrôle délai lors de l'exécution d'une instruction de branchement/saut

aléa de structure conflit pour l'accès à une ressource matérielle

Les aléas de données



Pour les instructions 2–4 :

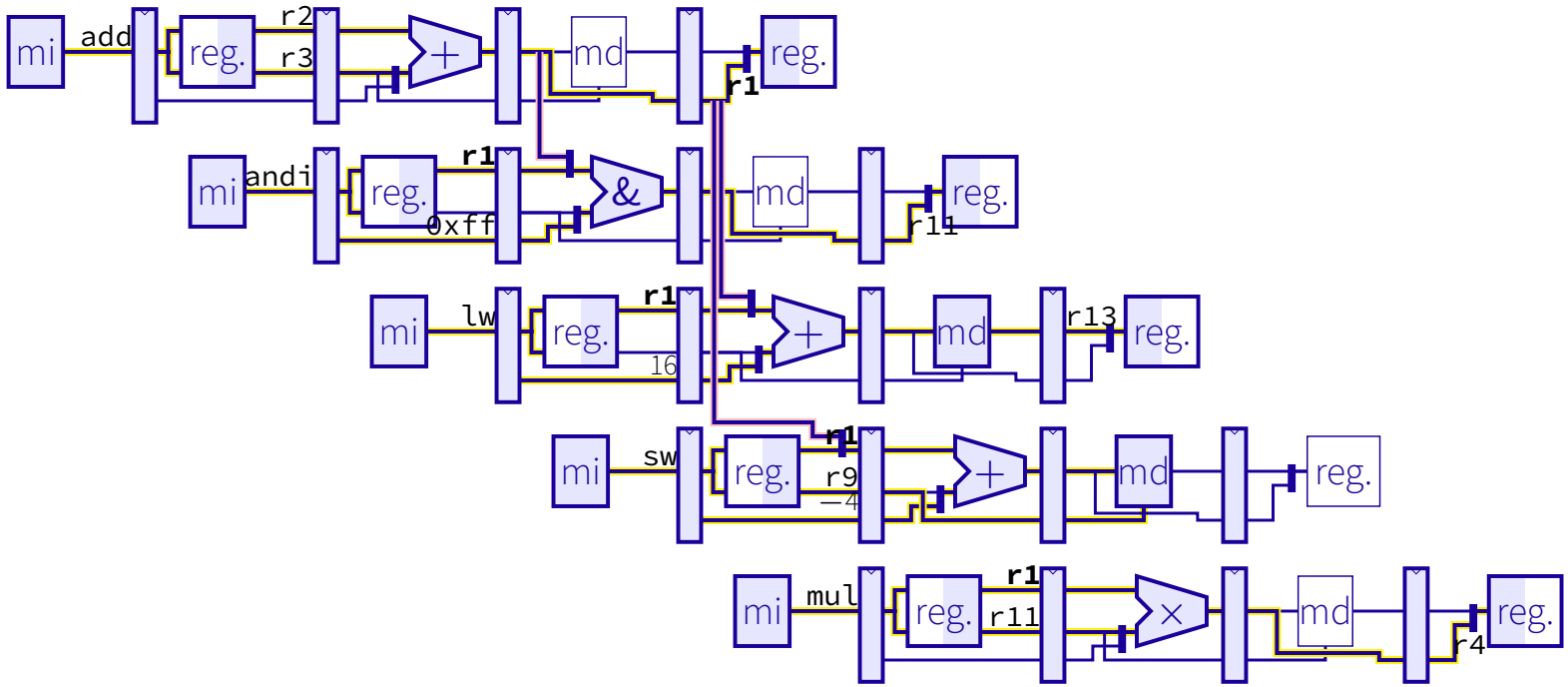
- lecture lors de la phase `li` de la valeur de `r1` stockée dans le banc de registre
- mais on lit l'*ancienne* valeur de `r1` et non la valeur calculée par l'instruction 1 (et copiée dans le banc de registres en fin de phase 5).

Les aléas de données (cont.)

envoi

Utilisation de mécanismes d'envoi (ou court-circuit) (*forwarding*).
On va chercher la donnée là où elle est présente...

1 add **r1**,r2,r3
2 andi r11,**r1**,0xff
3 lw r13,(16)**r1**
4 sw r9,(-4)**r1**
5 mul r4,**r1**,r11

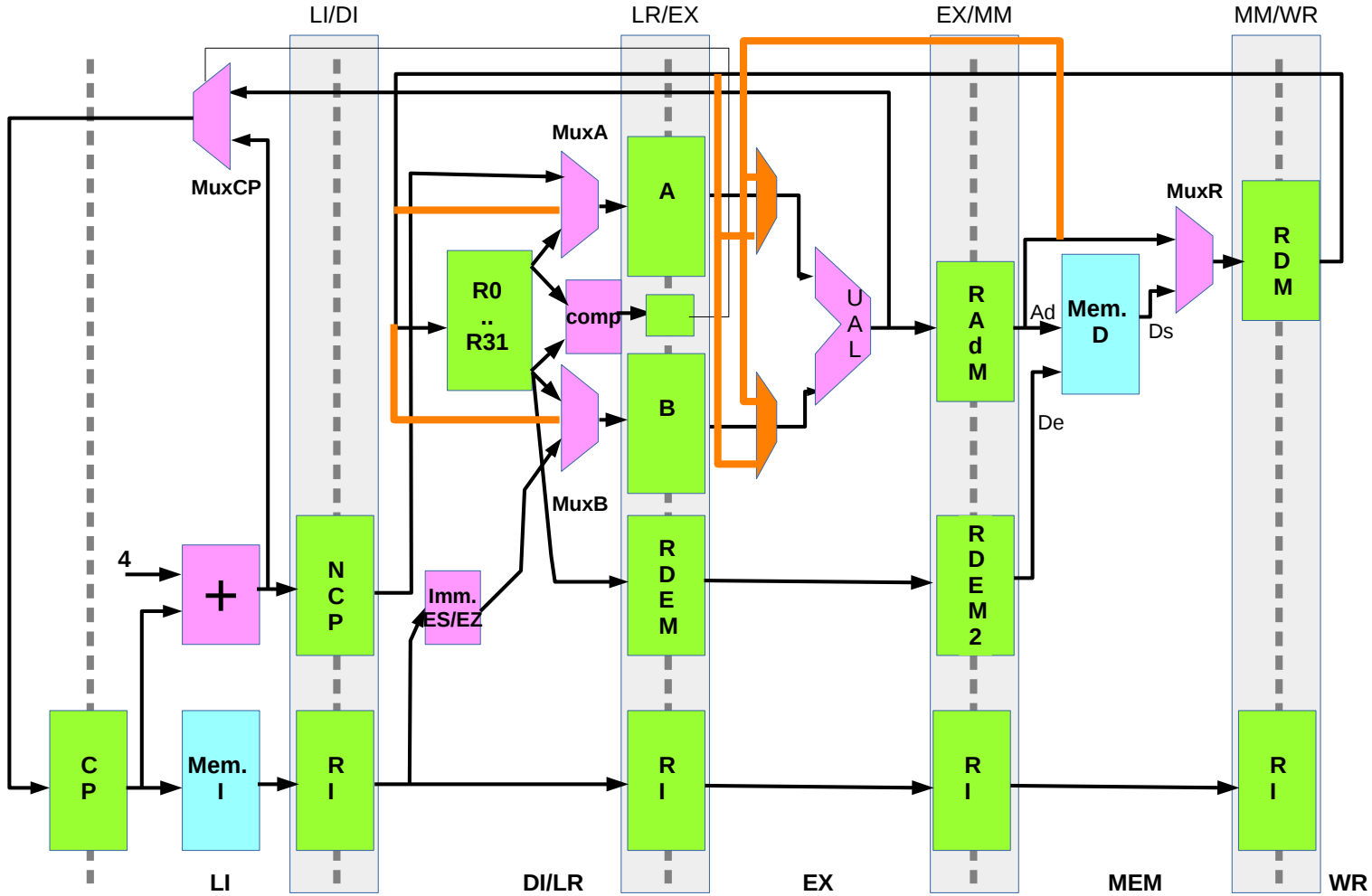


Les aléas de données (cont.)

envoi

si `LR/EX.ri.src1` == `EX/MM.ri.dest`
alors utiliser `EX/MM.RAdm` comme opérande1 de l'ALU

idem pour les autres situations



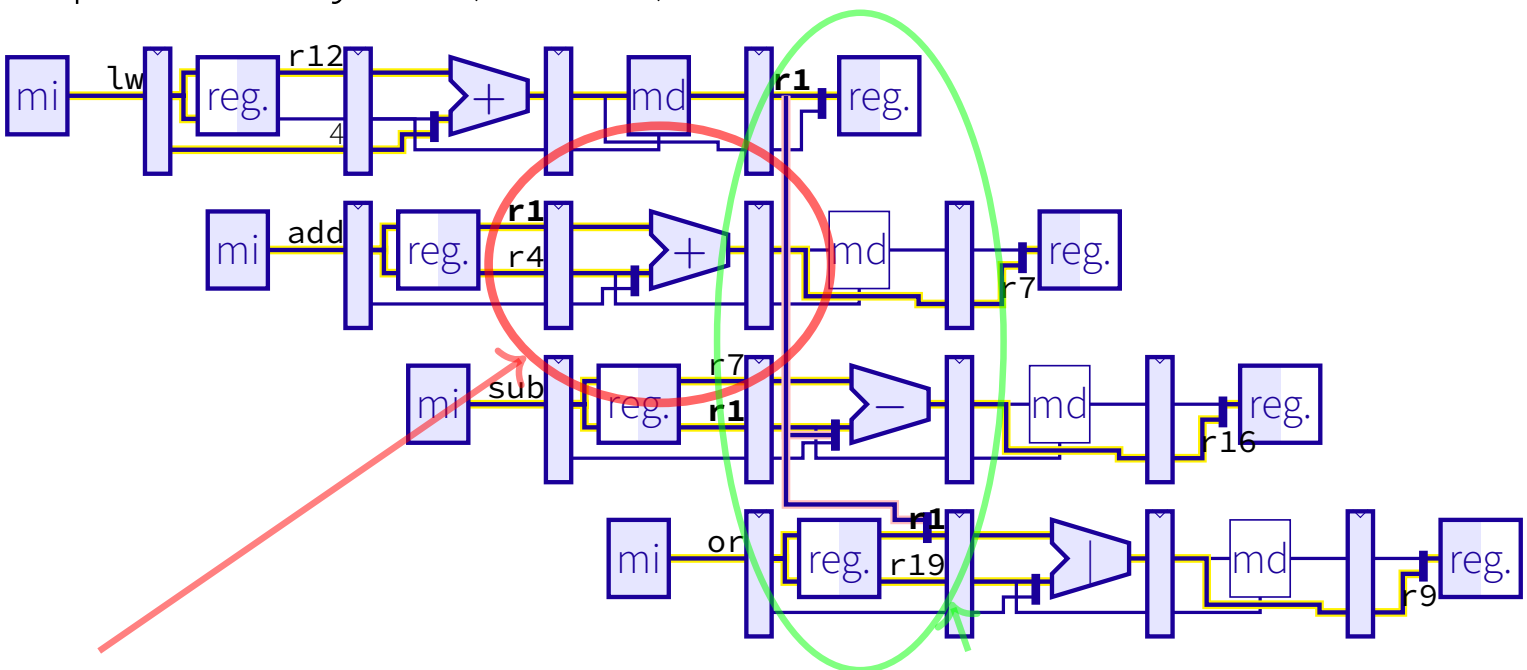
Les aléas de données (cont.)

suspensions

Mais quand la donnée n'est pas disponible dans le processeur, les aléas de données ne peuvent être résolus par des court-circuits.

- Lecture mémoire
- opérations en plusieurs cycles (flottant)

1 lw **r1**,(4)r12
2 add r7,**r1**,r4
3 sub r16,r7,**r1**
4 or r9,**r1**,r19

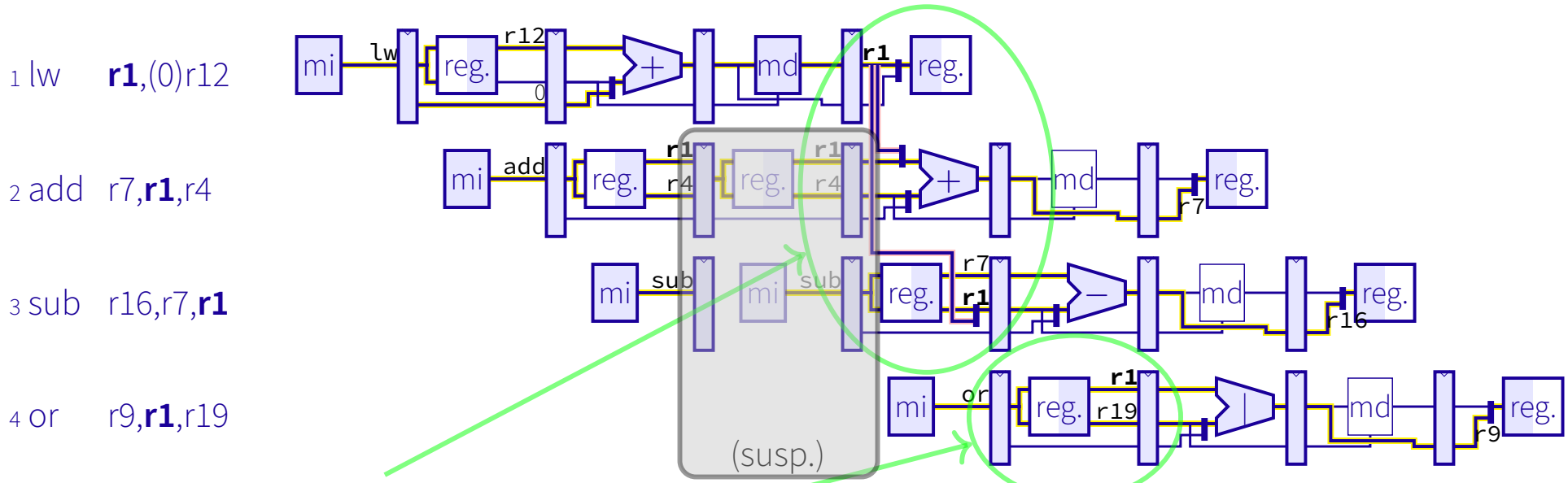


La nouvelle valeur de `r1` n'est pas dans le processeur !

Utilisation possible des mécanismes d'envoi

Les aléas de données (cont.)

On opére alors une *suspension du pipeline* (*bubble* ou *pipeline stall*)



Utilisation possible des mécanismes d'envoi

`r1` est déjà copié dans le banc de registres

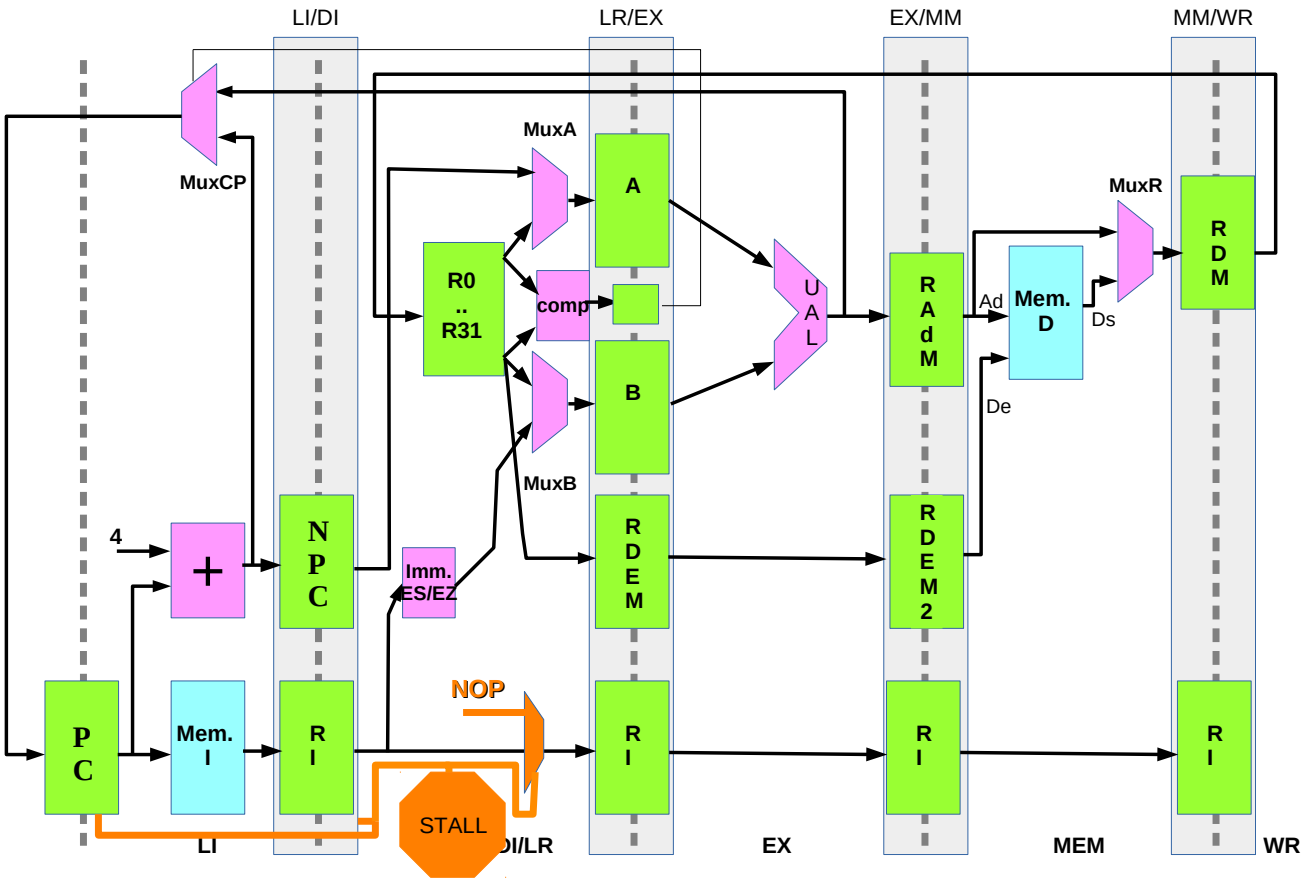
En pratique, pour une suspension :

- on fige le pipeline pour les étages `LI` et `DI` (pas d'écriture des registres pipeline)
- on injecte un **nop** dans le registre `LR/EX`

Les aléas de données (cont.)

suspensions

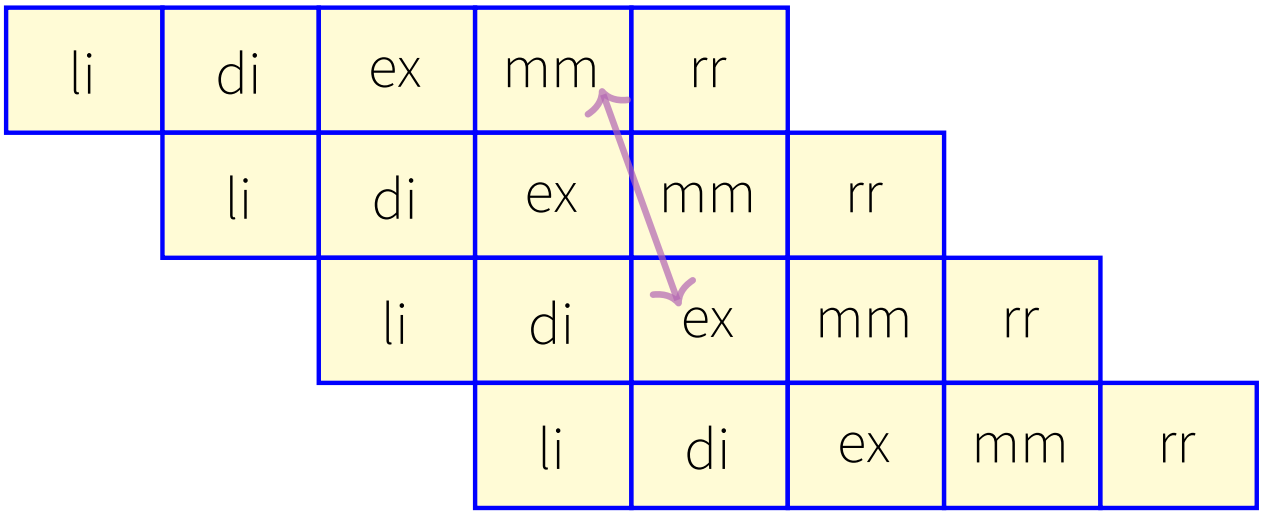
si `LR/EX.ri.codeop` == `lw`
et ((`LR/EX.ri.dest` == `LI/DI.ri.src1`)
ou (`LR/EX.ri.dest` == `LI/DI.ri.src2`))
alors suspension :
pas de chargement de `pc` ou `LI/DI`
`LR/EX.ri` ← `nop`



Les aléas de données (cont.)

latence

Après une instruction de chargement, la donnée n'est disponible qu'avec un délai de 1 cycle



Chargement mémoire : *latence* de 2 cycles (ou *délai de chargement* d'un cycle).

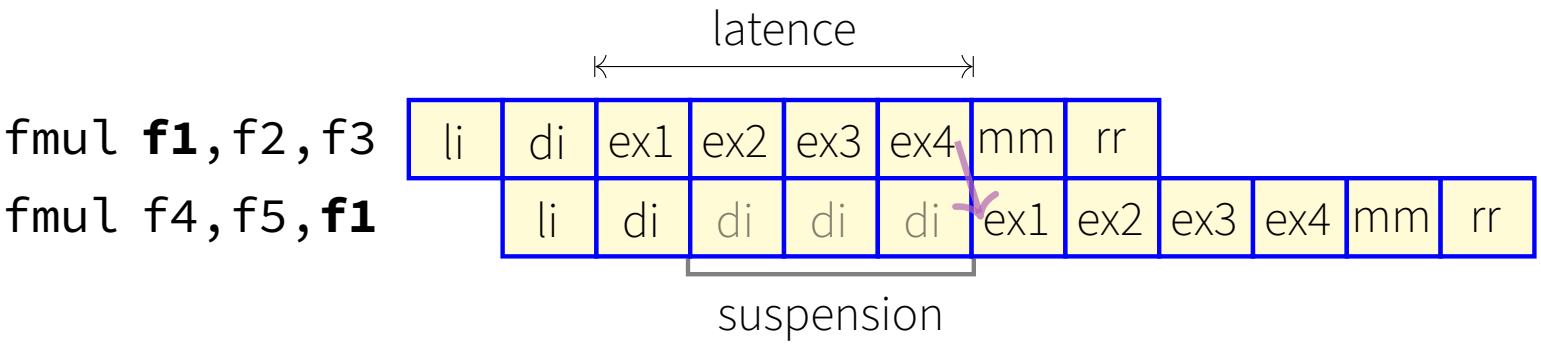
Les aléas de données (cont.)

latence

Les opérations entières complexes (multiplication/division) et les opérations flottantes ont souvent une latence de plusieurs cycles.

Les aléas de données entraînent alors un délai d'attente.

Exemple : supposons multiplication flottante `fmul` latence de 4 cycles.
Un aléa de données nécessitera une suspension (délai) de trois cycles.

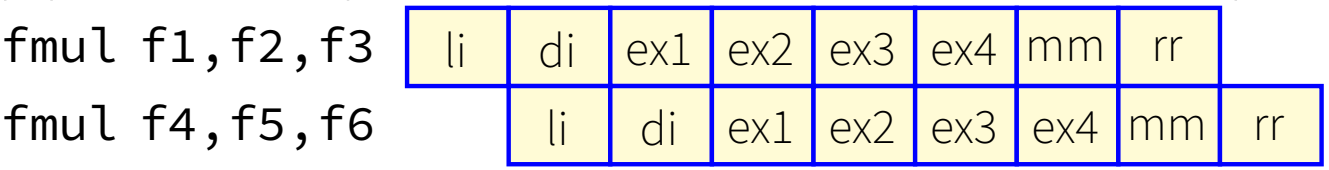


Les aléas de données (cont.)

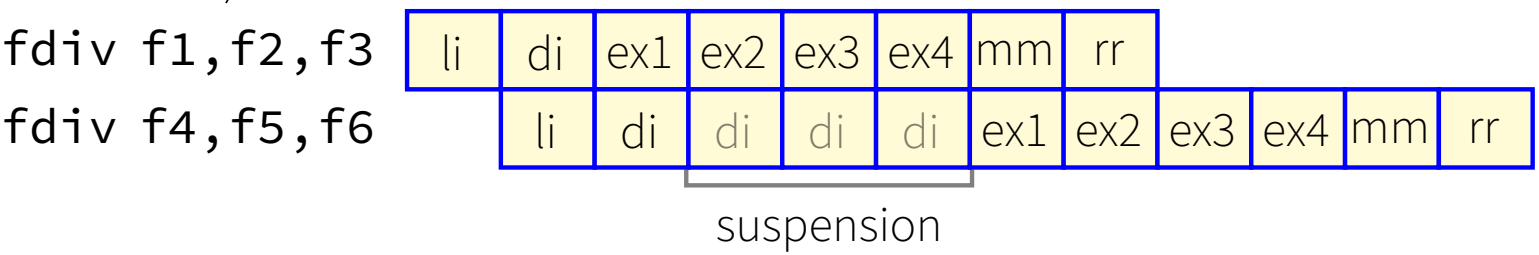
latence

De plus, ces opérations peuvent être *pipelinées* ou *non pipelinées*.

- pipelinée : on peut lancer un nouveau calcul à chaque cycle.



- non pipelinée : opérateur indisponible pour la durée du calcul (aléa *structurel*).



Les opérations peu courantes (division, $\sqrt{}$) sont non pipelinées.

	+	×	÷	$\sqrt{}$	+	×	÷	$\sqrt{}$
	(latence)				(débit démarrage)			
P4-x87	5	7	38	38	1	2	38	38
P4-SSE2	4	6	35		2	2	35	

Rappels : représentation des nombres flottants

N flottant représenté par triplet (s, m, e) $N = (-1)^s \times 1.m \times 2^e$
IEEE 754 simple précision et double précision

SP 32bits :	signe (1)	exposant (8)	mantisse (23)
DP 64bits :	signe (1)	exposant (11)	mantisse (52)

Addition flottante $A(s_a, e_a, m_a) + B(s_b, e_b, m_b)$:

1. Si $e_a < e_b$, permuter A et B
2. Décaler $1.m_b$ de $e_a - e_b$ positions vers la droite $\rightarrow m'_b$.
3. Calculer $S = 1.m_a \pm m'_b$
4. Renormaliser S si $S \geq 2$ ou $S < 1$
   Pour cela décaler la mantisse et incr./décr. l'exposant
5. Arrondir le résultat er repasser en codage IEEE 754

Les aléas de données (cont.)

Multiplication flottante $A(s_a, e_a, m_a) \times B(s_b, e_b, m_b)$:

1. Calculer $m = 1.m_a \times 1.m_b$
   $e = e_a + e_b$
   $s = s_a \oplus s_b$
2. si $2.0 \leq m < 4.0$, $m >>= 1$ et $e++$
3. Arrondir le résultat
4. Repasser en codage IEEE 754

Les opérations flottantes sont longues et complexes.

Latence de plusieurs cycles.

Addition/soustraction et multiplication sont pipelinées
Division non pipelinée

Les dépendances

Dépendance = contrainte de séquentialité entre instructions qui lisent ou écrivent une donnée dans un même registre.

On distingue les cas suivants :

dépendance de données ou **RAW (*read-after-write*)**
: l'instruction i écrit le registre r et l'instruction $i + n$ lit r

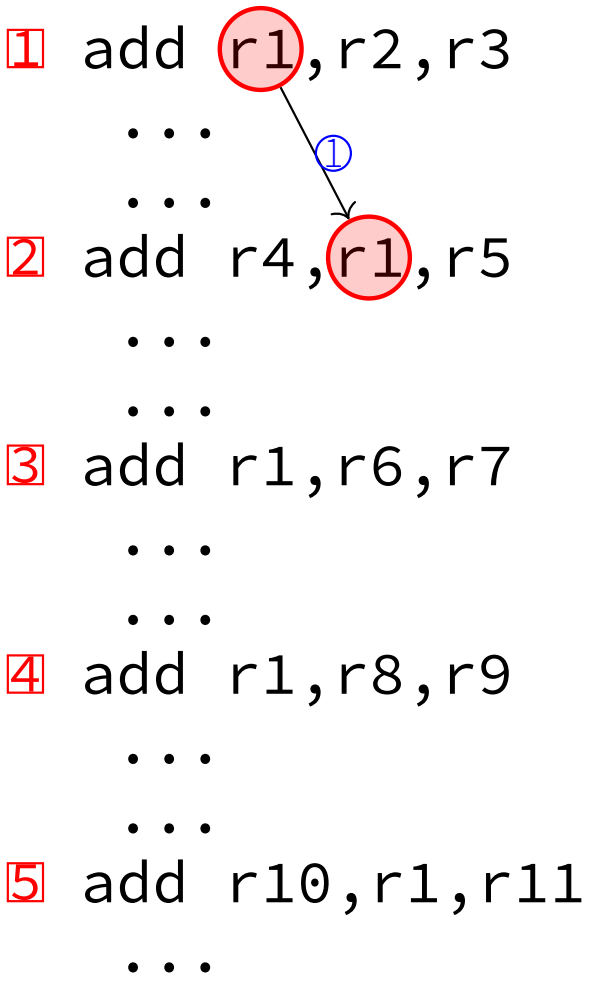
anti-dépendance ou **dépendance de nom** ou **WAR (*write-after-read*)**
: l'instruction i lit la donnée dans r et l'instruction $i + n$ génère une nouvelle valeur dans r

dépendance de sortie ou **WAW (*write-after-write*)**
: les instructions i et $i + n$ génèrent une information dans le même registre r. L'ordre des écritures doit être respecté.

Les dépendances (cont.)

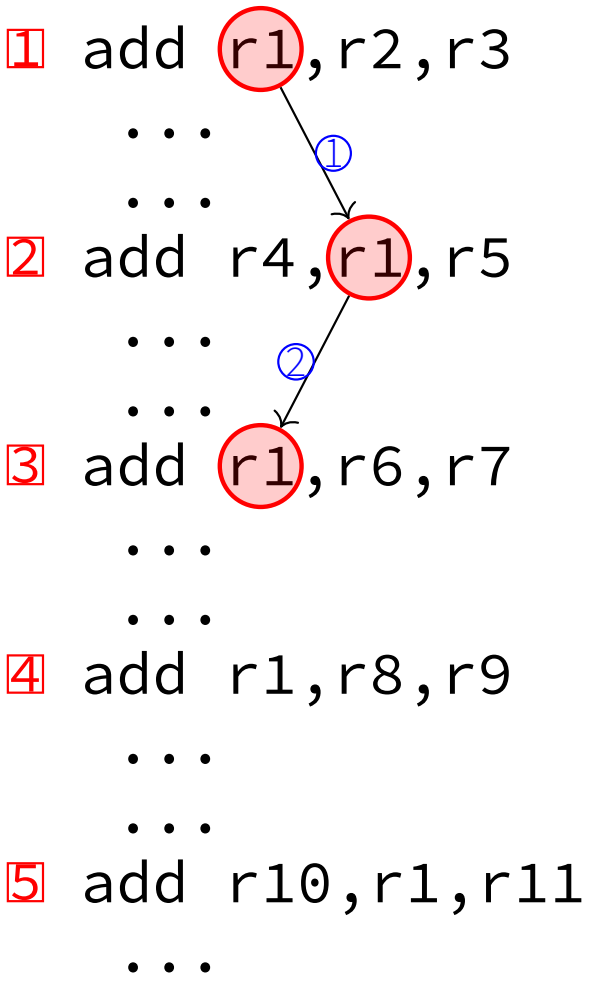
```
1 add r1,r2,r3
    ...
    ...
2 add r4,r1,r5
    ...
    ...
3 add r1,r6,r7
    ...
    ...
4 add r1,r8,r9
    ...
    ...
5 add r10,r1,r11
    ...
```

Les dépendances (cont.)



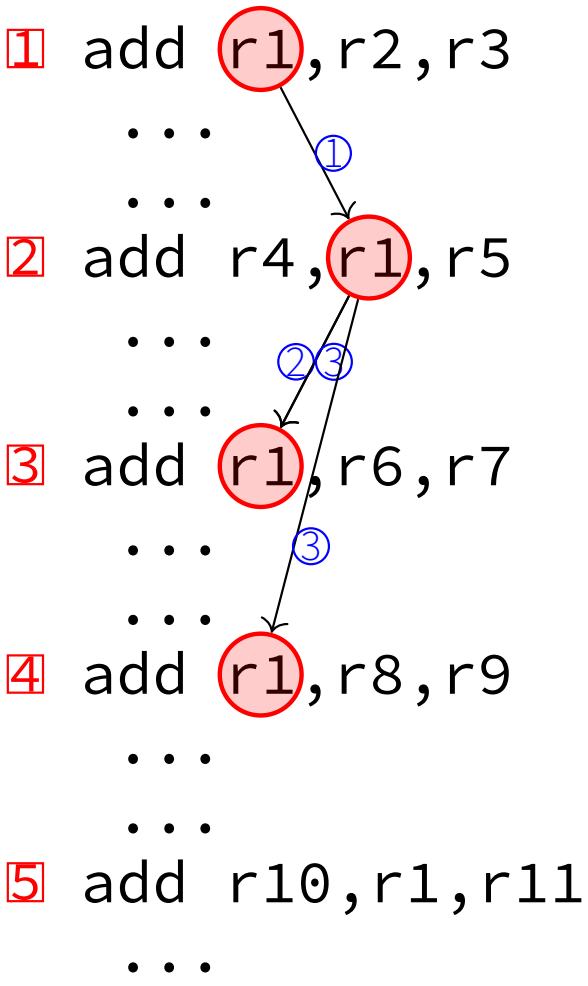
① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée **dépendance de données** (**RAW**)

Les dépendances (cont.)



① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée
dépendance de données (**RAW**)
② Le registre `r1` est écrit par l'instruction 3 et l'instruction 2 *doit* lire son contenu *avant*.
Par contre, il suffit d'utiliser un autre registre que `r1` pour que le problème disparaisse.
antidépendance ou **dépendance de nom** (**WAR**)

Les dépendances (cont.)

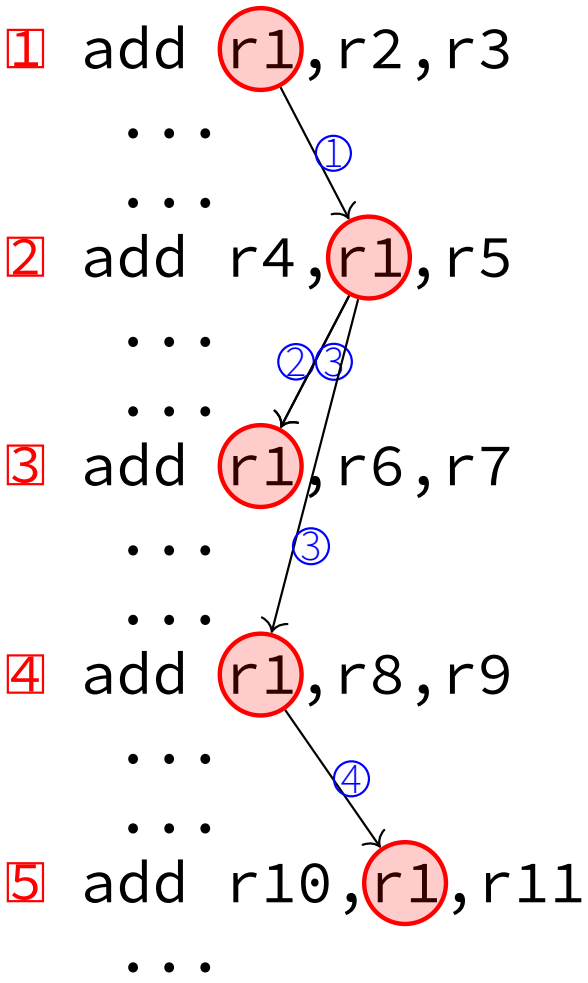


① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée
dépendance de données (**RAW**)

② Le registre `r1` est écrit par l'instruction 3 et l'instruction 2 *doit* lire son contenu *avant*.
Par contre, il suffit d'utiliser un autre registre que `r1` pour que le problème disparaisse.
antidépendance ou **dépendance de nom** (**WAR**)

③ Les instructions 3 et 4 écrivent successivement le registre `r1`. L'ordre des écritures doit être respecté.
dépendance de sortie (**WAW**)

Les dépendances (cont.)

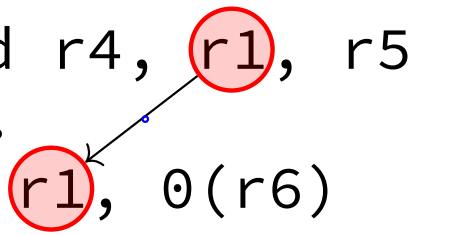


① L'information produite par l'instruction 1 dans `r1` est utilisée par l'instruction 2. C'est une vraie dépendance qui doit être respectée
dépendance de données (**RAW**)
② Le registre `r1` est écrit par l'instruction 3 et l'instruction 2 *doit* lire son contenu *avant*.
Par contre, il suffit d'utiliser un autre registre que `r1` pour que le problème disparaisse.
antidépendance ou **dépendance de nom** (**WAR**)
③ Les instructions 3 et 4 écrivent successivement le registre `r1`. L'ordre des écritures doit être respecté.
dépendance de sortie (**WAW**)
④ A nouveau une **dépendance de données** entre 4 et 5.

Renommage de registres

Soit le programme ci-dessous

```
add r1, r2, r3
...
add r4, r1, r5
...
lw r1, 0(r6)
...
add r7, r1, r8
```



Dépendance de nom du fait de l'utilisation du même registre `r1`

Il suffit de *renommer* le (deuxième) registre `r1` en `r9`

```
add r1, r2, r3
...
add r4, r1, r5
...
lw r9, 0(r6)
...
add r7, r9, r8
```

La dépendance a disparu sans que cela change le programme.

Le renommage de registres permet de supprimer les dépendances WAR et WAW.

Exceptions et interruptions

L'exécution d'un programme peut être interrompue par

- un événement extérieur (*interruption*)
- un problème interne au processeur (*exception*) (erreur arithmétique ($\div 0$), pb d'accès mémoire (*bus error*), etc)

Le processeur exécute alors un routine associée à l'exception.

Pour une exception *propre* :

- *toutes* les instructions avant celle qui a provoqué l'exception doivent se terminer
- *aucune* instruction suivant celle qui a provoqué l'exception ne doit se terminer

Exceptions et interruptions (cont.)

Pour le traitement des exceptions dans un pipeline, trois possibilités :

Terminaison dans l'ordre des instructions Une instruction multi-cycles oblige toutes les instructions suivantes à attendre pour se terminer, même
- s'il n'y a pas de dépendances de données
- si elles ne provoquent pas d'exceptions

Terminaison non ordonnées des instructions Sans dépendance de données, si on autorise les instructions suivantes à se terminer, elles vont modifier les registres (et la mémoire)
Des exceptions « propres » sont alors impossibles (les instructions suivant l'exception ne doivent pas terminer)

Exécution « *spéculative* » des instructions Les instructions peuvent s'exécuter, mais on attend pour modifier l'état du processeur ou de la mémoire.
- rangement temporaire des résultats des instructions
- on ne modifie les registres et la mémoire que lorsque l'on est sûr qu'il n'y aura pas d'exception (instructions « garanties »)

Renommage des registres

Le renommage des registres est aussi utile pour l'exécution spéculative.

- registre logique (dans le jeu d'instruction)
- registre physique (dans le processeur et en nombre supérieur au nombre des registres logiques)

Deux mécanismes :

- on duplique le banc de registres : registres *futurs* / registres *présents*. Une instruction écrit dans un registre futur et on recopie dans un registre présent quand l'instruction est garantie
- au lancement de l'instruction, on associe un registre physique à un registre logique avec une table de correspondance (renommage). La recopie du registre physique vers le registre logique se fait quand l'instruction est achevée et garantie.

Dépendance de données et boucles

Cas du SAXPY/DAXPY

```
const int n=10000;
double a, x[n], y[n];
for (int i=0; i<n; i++)
   y[i]= a*x[i] + y[i];
```

On suppose les latences suivantes des instructions :

Source / *Destination*	UAL	`ld`	+ FP	× FP
UAL	1	2		
`ld/st` (adresse)	1	2		
`st` (données)	1	1	4	4
op. FP		2	5	5

Dépendance de données et boucles (cont.)

Version 0 très inefficace

```
1  ;;;       r1, r2 adr. x(i), y(i) initialisés à &(x[0]) et &(y[0])
2  ;;;       r6 compteur de boucle initialisé à n-1
3  ;;;       f0 contient la constante a
4  boucle:   ld   f1,(r1)       ; charge x(i) dans f1
5            ld   f2,(r2)       ; charge y(i) dans f2
6            fmul f1,f0,f1      ; f1 <- a*x(i)
7                               ; 4 cycles attente
8                               ; pour fmul
9                               ; (latence 5 vers
10                              ; autre op flottante)
11           fadd f2,f2,f1      ; f2 <- a*x(i)+y(i)
12                              ; 3 cycles attente
13                              ; pour fadd
14                              ; (latence 4 vers store)
15           sd   f2,(r2)       ; range f2 dans y(i)
16           subi r6,r6,#1      ; decr. cmpt boucle
17           addi r1,r1,8       ; calcul adresse x(i+1)
18           addi r2,r2,8       ; calcul adresse y(i+1)
19           bnz  r6,boucle     ; si i<n-1, branch.
```

16 cycles/itération dont 7 perdus

On réarrange le code en remarquant que les instructions 16–18 n'ont pas (ou peu) de dépendance avec les instructions 11 ou 15 : permutation possible en position 12–14.

Dépendance de données et boucles (cont.)

réordonnancement de code

Version 1 améliorée par réordonnancement,
mais toujours non optimale

```
1  ;;;       r1 initialisé à &(x[0]) et r2 à &(y[0])
2  ;;;       r6 compteur de boucle initialisé à n-1
3  ;;;       f0 contient la constante a
4  boucle:   ld   f1,(r1)            ; charge x(i) dans f1
5            ld   f2,(r2)            ; charge y(i) dans f2
6            fmul f1,f0,f1           ; f1 <= a*x(i)
7                                    ; attente résultat
8                                    ; multiplication
9                                    ; latence 5
10                                   ; fmul -> fadd
11           fadd f2,f2,f1           ; f2<=a*x(i)+y(i)
12           subi r6,r6,#1           ; decr. cmpt boucle
13           addi r1,r1,8            ; adresse x(i+1)
14           addi r2,r2,8            ; adresse y(i+1)
15           sd   f2,-8(r2)          ; range f2 dans y(i); -8 à cause de ligne 14
16           bnz  r6,boucle          ; si i<n-1, branch.
```

13 cycles/itération dont 4 perdus

L'amélioration implique une réorganisation plus importante du code.

Optimisation par déroulage de boucle

On traite plusieurs données d'indices consécutifs par itération

```
1  boucle: ld   f1,(r1)   ;charge x(i)
2          ld   f3,8(r1)  ;charge x(i+1)
3          ld   f5,16(r1) ;charge x(i+2)
4          ld   f7,24(r1) ;charge x(i+3)
5          ld   f2,(r2)   ;charge y(i)
6          ld   f4,8(r2)  ;charge y(i+1)
7          ld   f6,16(r2) ;charge y(i+2)
8          ld   f8,24(r2) ;charge y(i+3)
9          fmul f1,f0,f1  ;a*x(i)
10         fmul f3,f0,f3  ;a*x(i+1)
11         fmul f5,f0,f5  ;a*x(i+2)
12         fmul f7,f0,f7  ;a*x(i+3)
13         subi r6,r6,#4  ;i-=4
14         fadd f2,f2,f1  ;a*x(i)+y(i)
15         fadd f4,f4,f3  ;a*x(i+1)+y(i+1)
16         fadd f6,f6,f5  ;a*x(i+2)+y(i+2)
17         fadd f8,f8,f7  ;a*x(i+3)+y(i+3)
18         sd   f2,(r2)   ;range y(i)
19         sd   f4,8(r2)  ;range y(i+1)
20         sd   f6,16(r2) ;range y(i+2)
21         sd   f8,24(r2) ;range y(i+3)
22         addi r1,r1,32  ;adresse x(i+4)
23         addi r2,r2,32  ;adresse y(i+4)
24         bnz  r6,boucle;si i<n-1,brcht
```

```
for(i=0; i<n; i+=4){
  y[i]  +=a*x[i];
  y[i+1]+=a*x[i+1];
  y[i+2]+=a*x[i+2];
  y[i+3]+=a*x[i+3];
}
```

24 cycles/4 itérations (6/itération au lieu de 13)

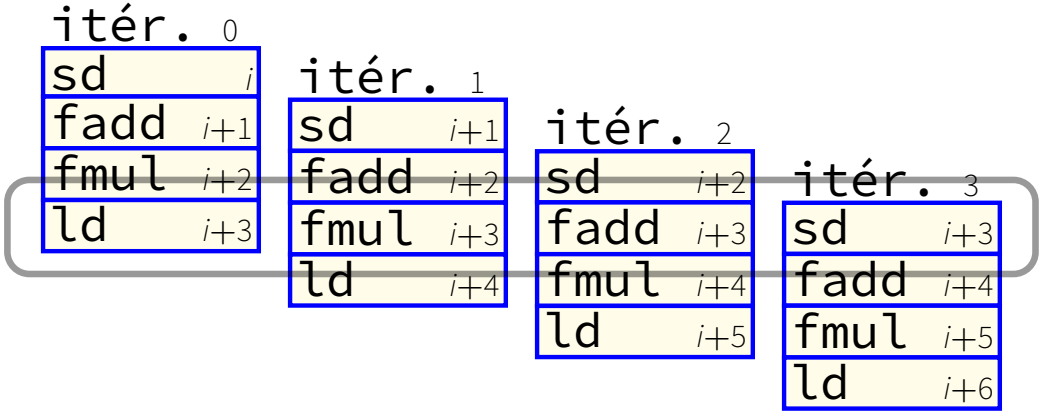
- plus de suspensions dues à la latence des opérateurs
- 1 seul cycle de gestion de boucle : 4 inst/4cy au lieu de 4/1cy

Optimisation par *pipeline logiciel*

Chaque itération traite des données correspondant à des indices différents.

```
prologue;
for (i=2; i<N-2; i++){
  y[i]=t;       /* range Y(i)       */
  t = u + v;    /* calcule Y(i+1)   */
  u = a * w;    /* calcule a*X(i+2)*/
  v = y[i+2];   /* charge y(i+2)    */
  w = x[i+3];   /* charge x(i+3)    */
}
epilogue;
```

```
1 boucle:  sd   f4,0(r2) ;range y(i)
2          fadd f4,f2,f3 ;a*x(i+1)+y(i+1)
3          fmul f3,f0,f1 ;a*x(i+2)
4          ld   f2,16(r2) ;charge y(i+2)
5          ld   f1,24(r1) ;charge x(i+3)
6          subi r6,r6,#1 ;decr. cmpt boucle
7          addi r1,r1,8  ;adresse x(i+2)
8          addi r2,r2,8  ;adresse y(i+1)
9          bnz  r6,boucle;si i<n-1, branch.
```



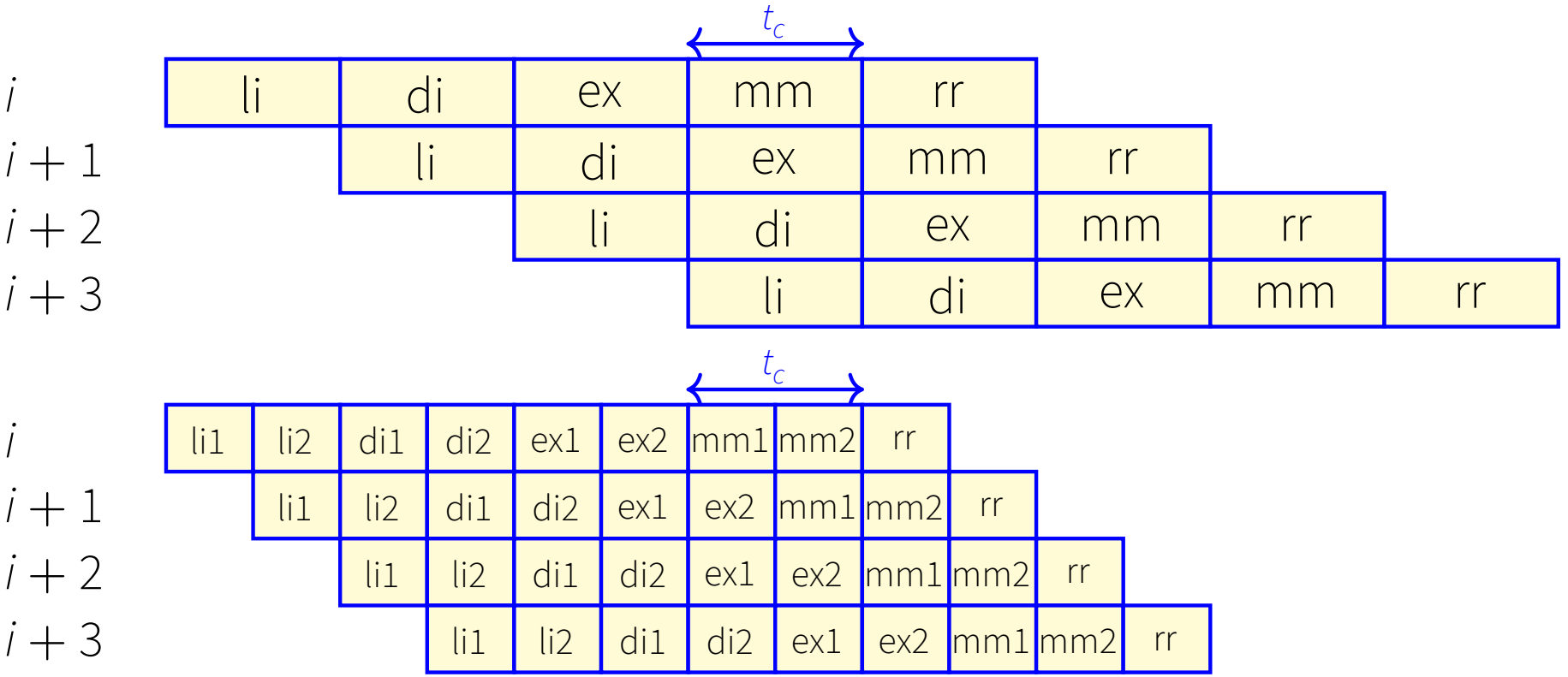
Aucune dépendance RAW dans la boucle

9 cycles par itération

+ surcoût prologue et épilogue

Augmentation de la profondeur des pipelines

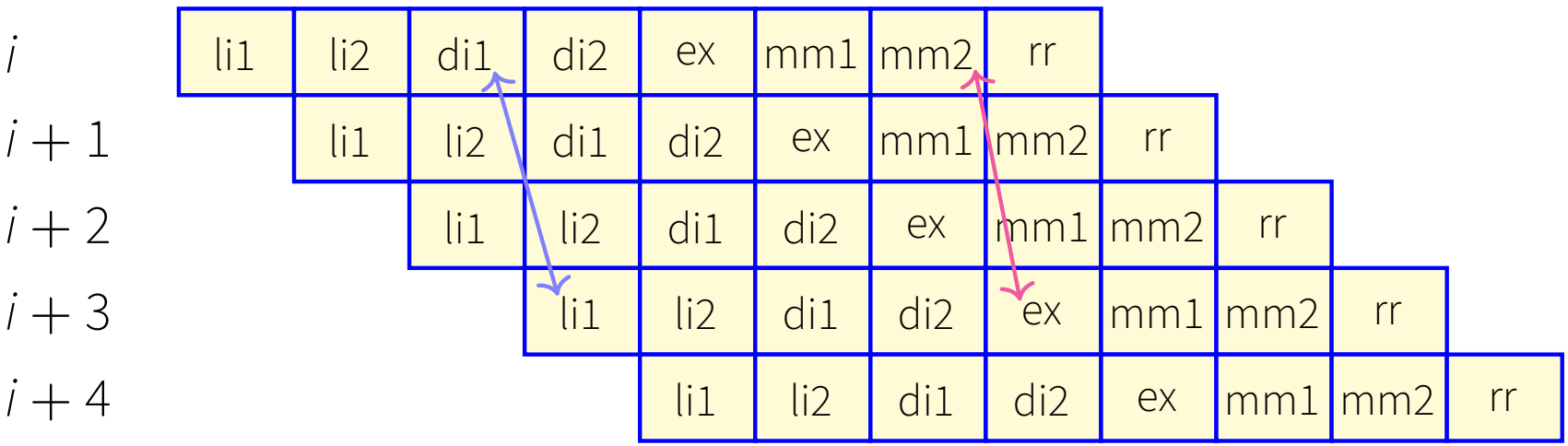
En découpant plus finement les étages d'un pipeline, on peut augmenter la fréquence d'horloge (*superpipeline*).



Duplication des étages critiques
Accroissement de la profondeur du pipeline.
Permet de diviser par deux le temps de cycle

Augmentation de la profondeur des pipelines (cont.)

Exemple de superpipeline



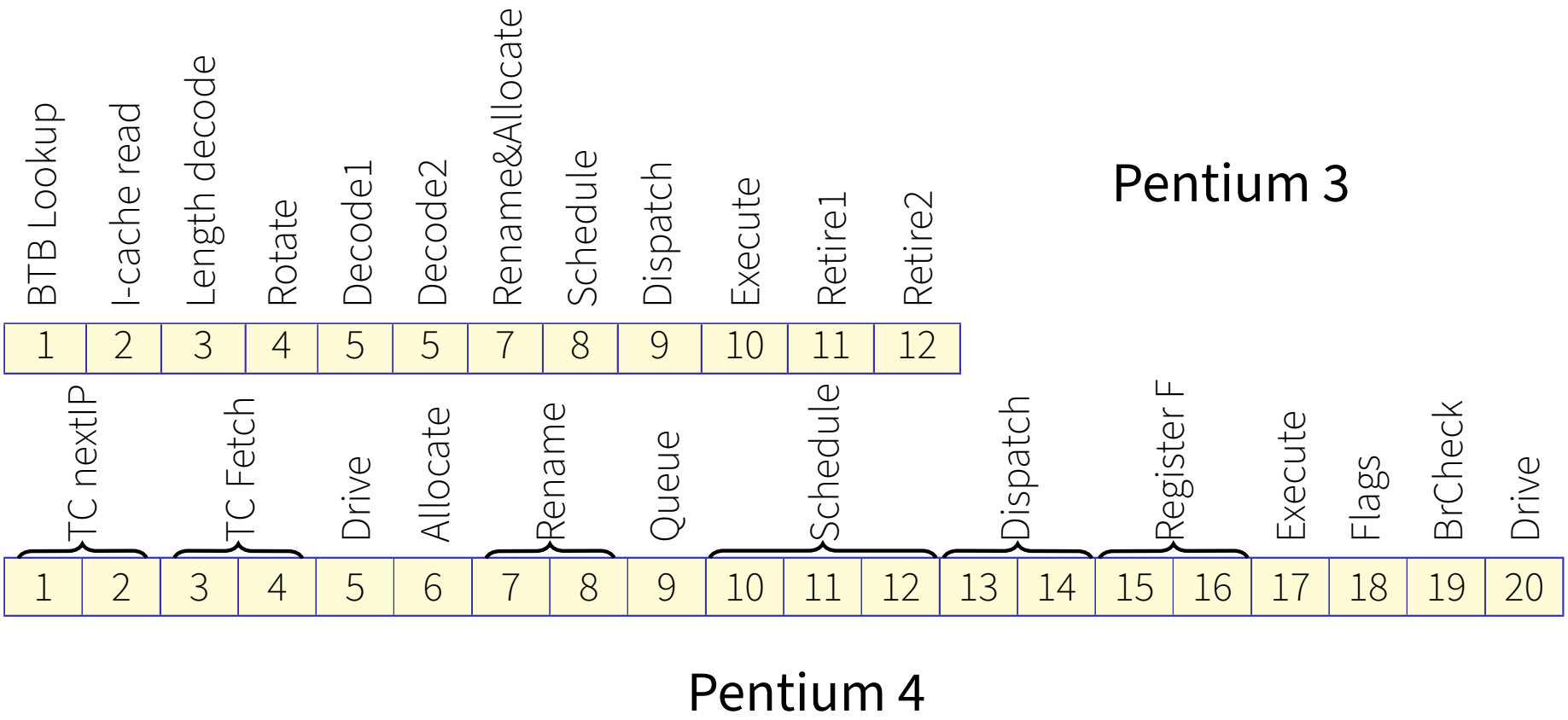
Pipeline de 9 cycles.

Délai de branchement ↔ 2 Délai de chargement ↔ 2

Le superpipeline augmente les délais de chargement et de branchement (en cycles)
mais permet de réduire le temps de cycle.

Augmentation de la profondeur des pipelines (cont.)

Pipeline dans différentes générations de pentium



Augmentation de la profondeur des pipelines (cont.)

Le nombre d'étages de pipeline dépend beaucoup des choix de mise en oeuvre architecturale et de la technologie :

Versions successives du pentium

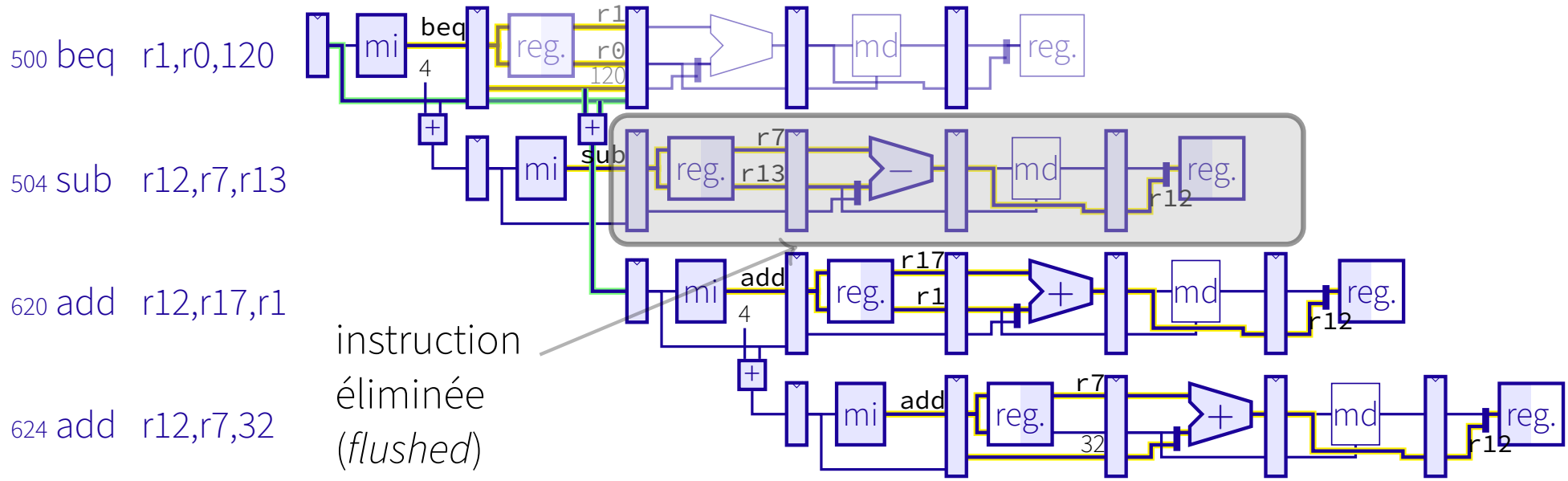
Architecture	pipeline
P5 (Pentium)	5
P6 (Pentium 3)	10
P6 (Pentium Pro)	14
NetBurst (Willamette)	20
NetBurst (Northwood)	20
NetBurst (Prescott)	31
NetBurst (Cedar Mill)	31
Core	14
Bonnell	16
Haswell/Skylake/Kabylake	14

Différentes versions Arm

Architecture	pipeline
ARM 1-7 (v1-v3)	3
ARM 8-9 (v4-v5)	5
ARM 11 (v6)	8
Cortex R (v7)	8-11
Cortex A7 (v7)	8-10
Cortex A8 (v7)	13
Cortex A15 (v7)	15-24
Cortex A77 (v8)	13

Aléas de branchement

Une rupture de séquence dans un programme provoque un *délai de branchement*.
On essaie de faire le branchement (calcul adresse branchement + comparaison) au plus tôt : en phase **`di`**.

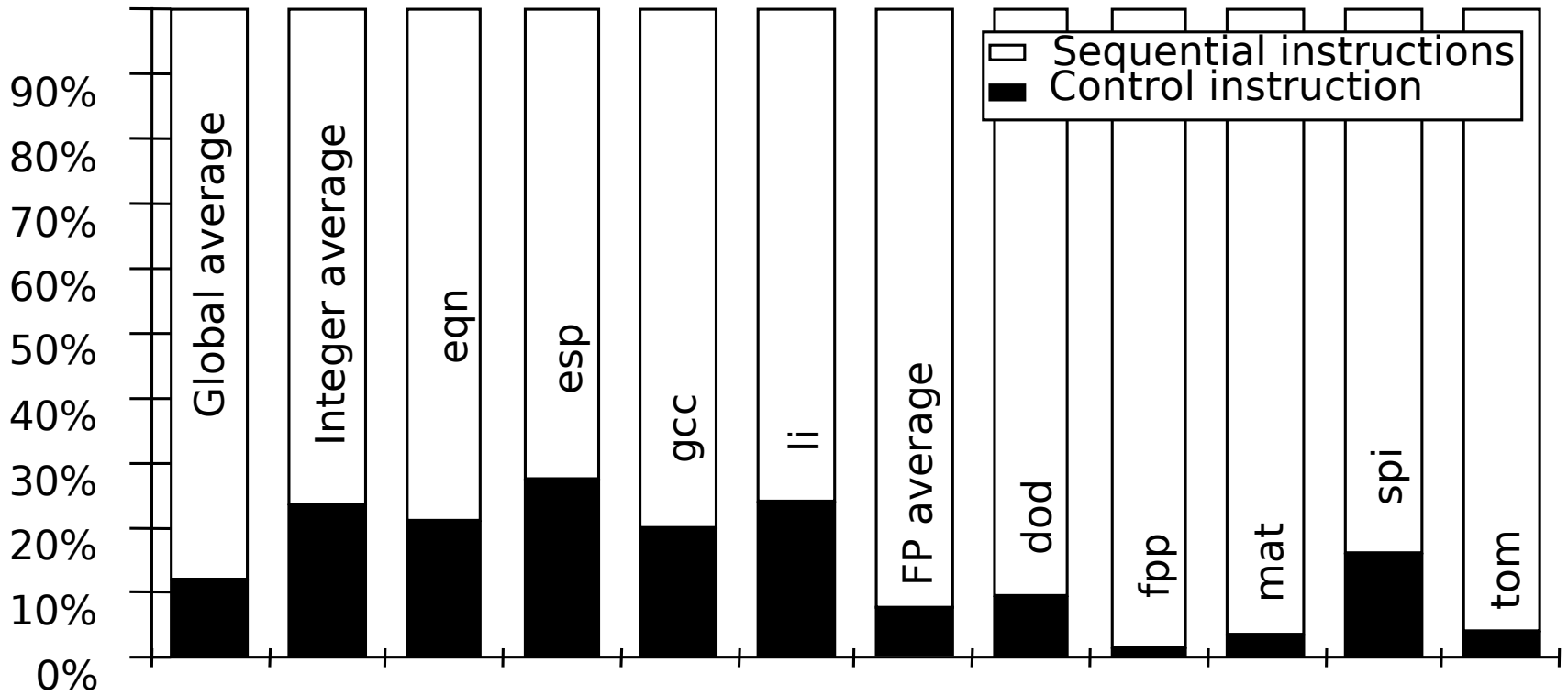


Pour supprimer la pénalité de branchement :

- prédiction matérielle de branchement
- instructions de transfert conditionnel
- ordonnancement des instructions

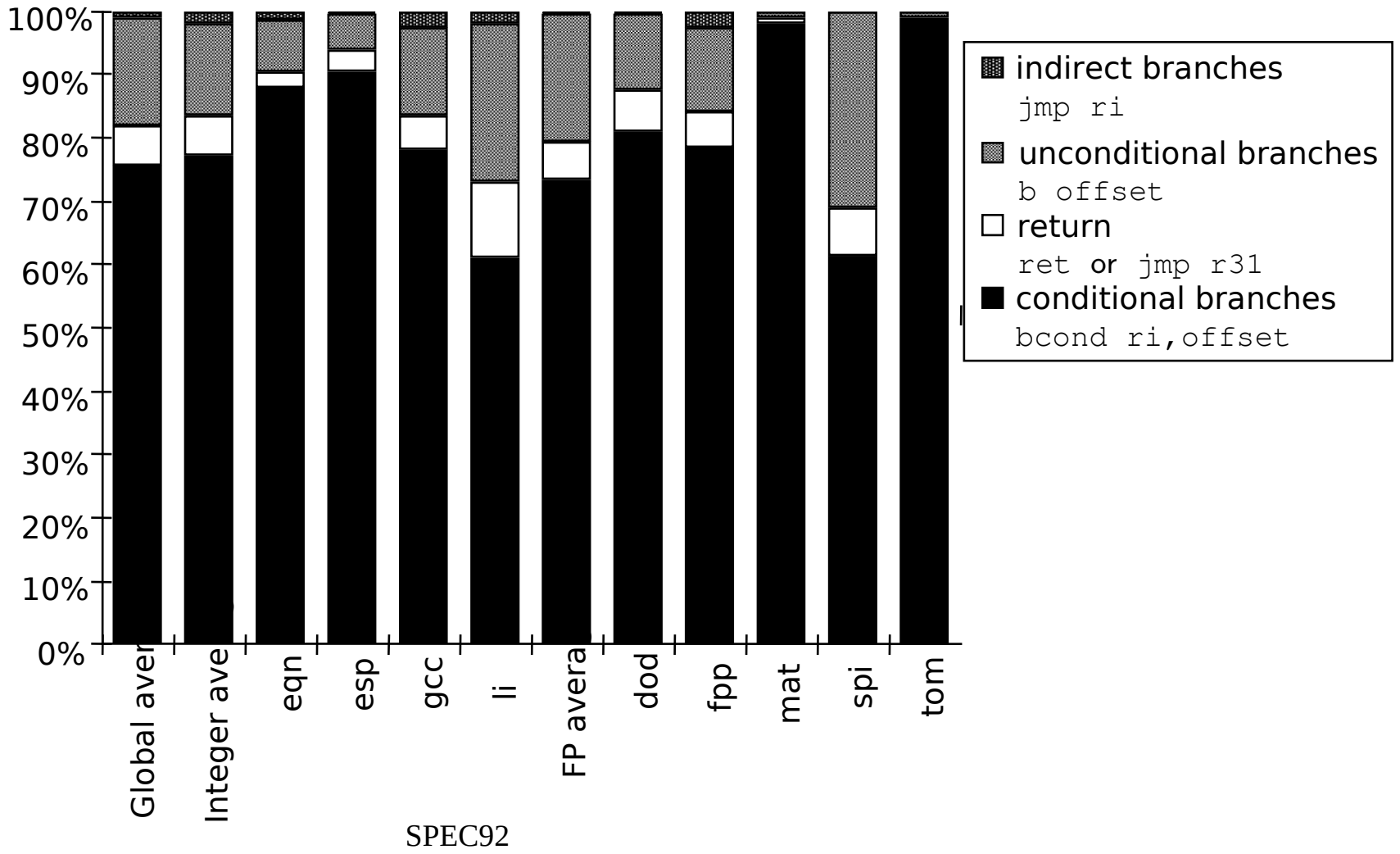
Aléas de branchement (cont.)

Importance des branchements dans le temps d'exécution d'un programme



Evalué sur SPEC92 (Yeh et Patt)

Aléas de branchement (cont.)



Utilisation relative des différents types de branchement

Prédiction de branchement

Cache d'adresse de branchement *branch target buffer* (BTB) (ou *branch target cache* BTC)

Si on attend d'avoir décodé l'instruction de branchement, la pénalité de branchement est *inévitable*.

Hypothèse : une instruction de branchement a un comportement (souvent) répétitif.

On mémorise dans une petite mémoire le comportement des instructions de branchement (pris/non pris + adresse cible).

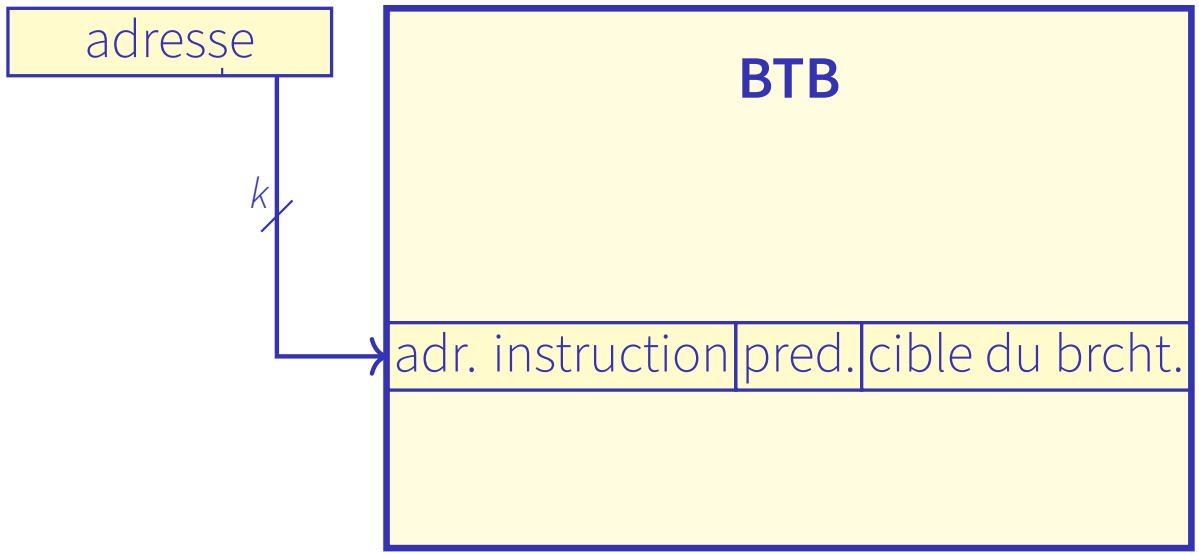
On utilise *l'adresse* de l'instruction pour déterminer si c'est un branchement et prédire son comportement.

Aléas de branchement (cont.)

prédiction de branchement

Si le BTB comprend 2^k entrées, on utilise les k bits de poids faible significatifs de l'adresse du branchement pour indéxer la table.
La table comprend :

- l'adresse du branchement pour éviter des confusions entre branchements dont les k bits de poids faible sont identiques (*étiquette*).
- les bits utilisés pour déterminer la prédiction pris/non pris du branchement
- l'adresse cible quand le branchement est pris



Aléas de branchement (cont.)

prédiction de branchement

Pour chaque instruction de branchement rencontrée :

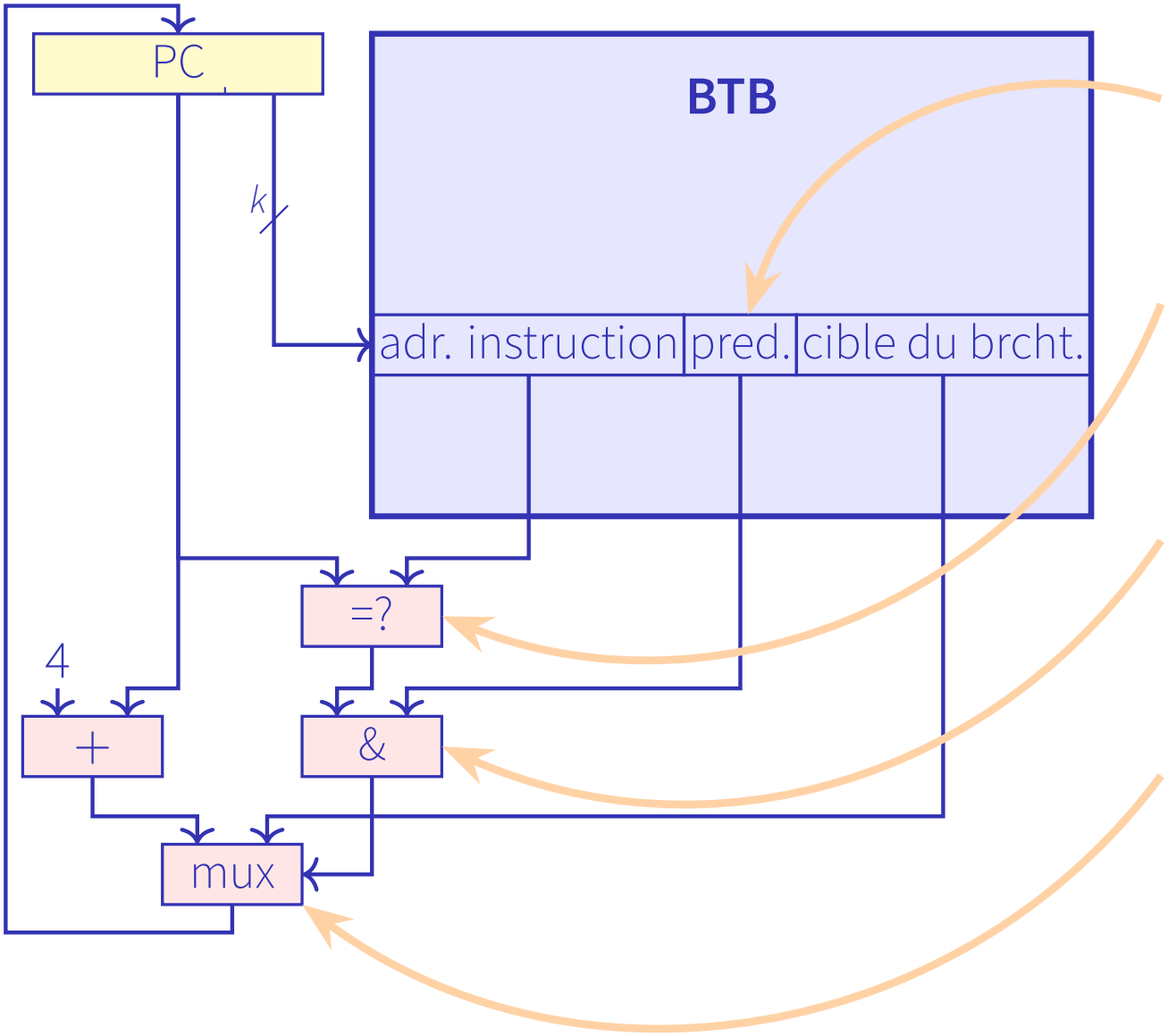
- entrer l'adresse de l'instruction de branchement
- entrer l'adresse vers laquelle a conduit le branchement (*adresse cible*)
- entrer des informations pour la prédiction (bits de prédiction)

Quand on accède à une instruction :

chercher si l'instruction est dans le BTB
si oui
récupèrer l'adresse cible
utiliser les bits de prédiction pour prédire si :
pc ← pc+4 ??
ou
pc ← adresse cible ??
aller à cette adresse pour l'instruction suivante

Si la prédiction est bonne, on évite la pénalité de branchement
Si la prédiction est fausse, on est ramené au cas sans prédiction

Aléas de branchement (cont.)



Lire l'étiquette, prédiction et cible dans le BTB

Déterminer si étiquette==PC

Le branchement est pris si prédiction vraie **et** étiquette==PC

Nouveau PC = soit PC+4, soit adresse cible prédite

Prédiction statique (à la compilation)

- informations connues à la compilation (boucles, par exemple)
- profilage des programmes

Prédictions dynamiques (à l'exécution)

- prédicteurs locaux
  - prédicteurs 1 bit et 2 bits
- prédicteurs globaux
  - historique des branchements

Prédiction statique

Suppose le même comportement pour un branchement dans un programme
Peut être défini par le matériel ou prédit par le compilateur
Exemples :

- toujours pris
- toujours non-pris
- branchements arrière pris (boucles `for` et `while`),
  branchements avant non pris (`if-then-else`)
  backward taken, forward not taken (BTFNT)

Dans certaines architectures, un bit généré par le compilateur (ou une analyse par un *profileur*) peut décider de la direction de la prédiction.

Prédiction dynamique

Prédicteur 1 bit : on mémorise l'état *pris/non pris* pour chaque branchement.

Suppose le comportement d'un branchement suffisamment répétitif.

Marche bien pour des boucles importantes.

```
for(i=0;i<100000;i++){
  y[i]=t;
}
```

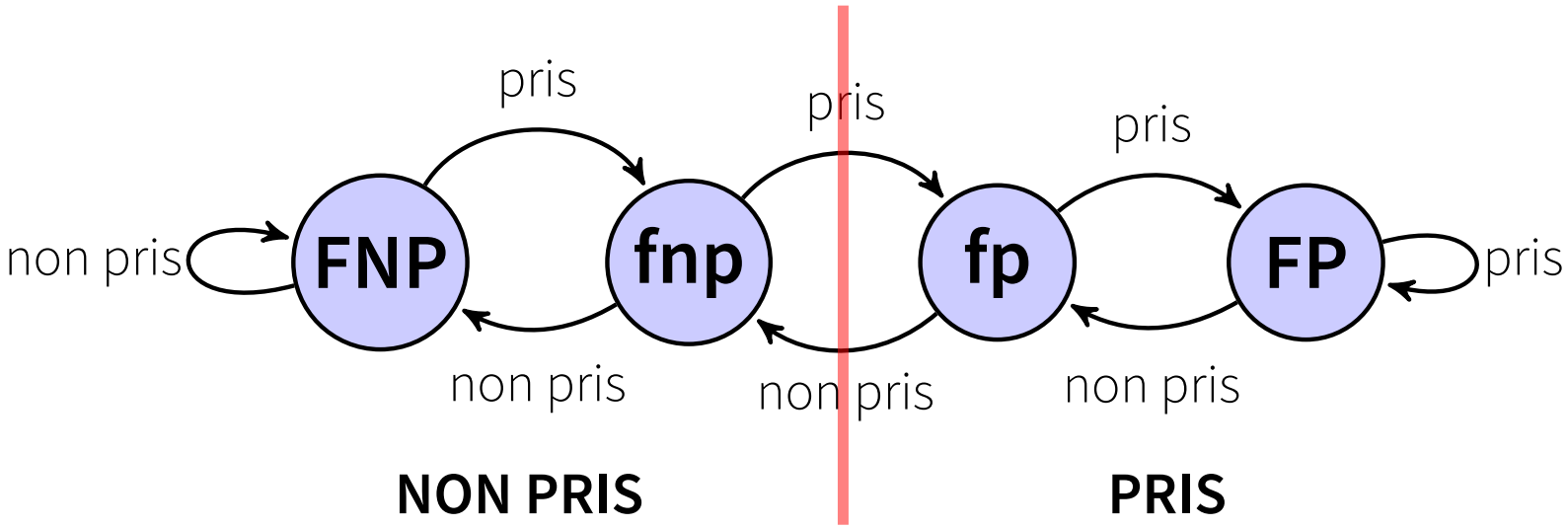
Une seule mauvaise prédiction en sortie de boucle.
(+ éventuellement lors de la 1ère itération, suivant l'état initial du prédicteur)

Moins efficace pour des boucles imbriquées.

Aléas de branchement (cont.)

prédicteurs 2 bits

Prédicteur 2 bits : on mémorise l'historique sous forme d'un automate à 4 états : *Fortement-Pris*, *faiblement-pris*, *Fortement-Non-Pris*, *faiblement-non-pris*.



Il faut deux mauvaises prédictions successives pour changer le sens de la prédiction (à partir d'un état fortement pris/non pris)
Prédicteur *à hystéresis*

Permet d'améliorer certaines prédictions

Aléas de branchement (cont.)

prédicteurs 2 bits

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

```
Lk:   ....
Lj:   ....
      ....
      bne rj,r0,Lj
      bne rk,r0,Lk
```

Le branchement externe (sur k) est pris $m-1$ fois et non-pris 1 fois.
Quel que soit le prédicteur : 1 mauvaise prédiction
Chaque branchement de la boucle intérieure (sur j) est pris $(n-1)$ fois et non pris 1 fois.
Prédicteur 1 bit : 2 mauvaises prédictions/n ($j=1$ et $j=n$), soit en tout 1+2n mauvaises prédictions.
Prédicteur 2 bits : en sortie de la boucle intérieure ($j=n$), le prédicteur passe de FP (fortement pris), à fp (faiblement pris).
1 seule mauvaise prédiction/n ($j=n$) ; en tout 1+n mauvaises prédictions.

Le prédicteur 2 bits divise par deux le nombre de mauvaises prédictions.

Historique local des branchements

Dans certains cas, la prédiction peut être très mauvaise.
Une solution est d'utiliser un *historique local*.

```
for(i=0;i<100000;i++){
  if(i&1) { // i impair
    ...
  }
}
```

Le `if` est pris une fois sur deux.

- prédicteur 1 bit : 100% de mauvaises prédictions.
- idem si prédicteur 2 bits initialisé à *fp*.
- prédicteur 2 bits initialisé à *fnp, FNP ou FP* : 50% de mauvaises prédictions (identique à pas de prédiction !).

Solution : ***historique local***.
Plusieurs prédicteurs (à 1 ou 2 bits), indexés par le comportement pris/non pris de la (ou des) dernières instances du branchement.

- prédicteur quand itération précédente prise : indique **toujours** FNP
- prédicteur quand itération précédente non prise : indique **toujours** FP

100% de bonnes prédictions.

historique global des branchements

La prédiction locale ne permet pas de gérer correctement la corrélation entre branchements différents.

```
if (x>0) {
  ...
}
if (x>5) {
 ...
}
```

Le deuxième `if` n'est *jamais* pris quand le premier est non-pris.

On peut résoudre ce problème par un *historique global*.

On utilise plusieurs prédicteurs indexés par les derniers branchements (quels qu'ils soient).

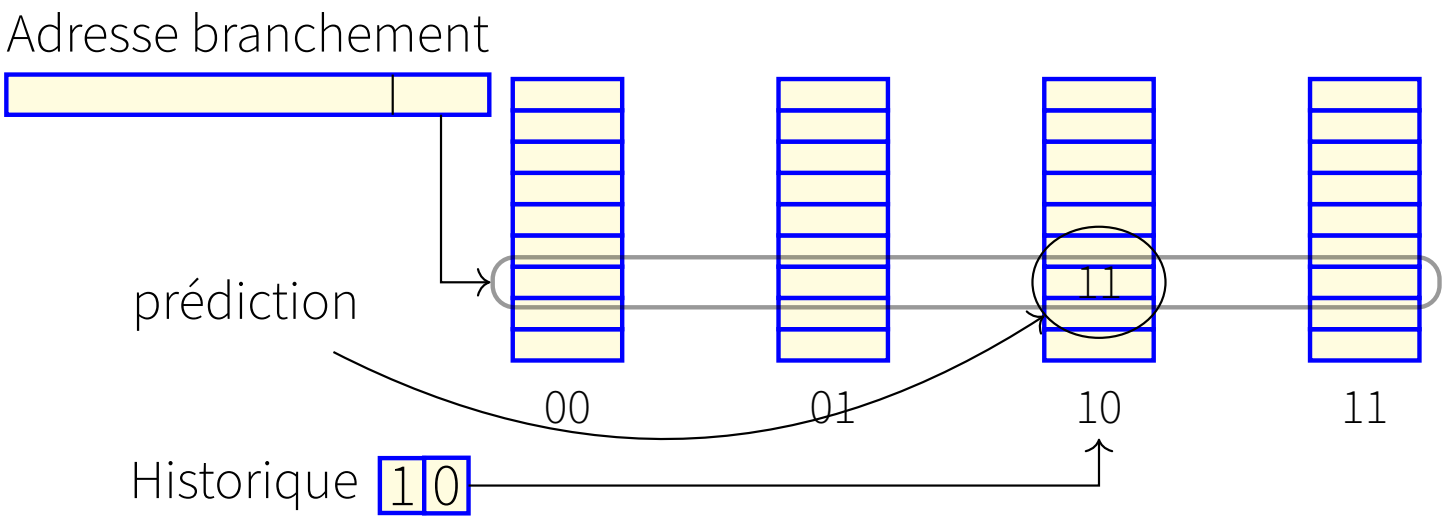
Peut également servir comme un prédicteur local

Aléas de branchement (cont.)

historique local

On mémorise dans un registre à décalage l'historique (*pris/non-pris*) des m derniers branchements (*global history buffer GHB*).
On utilise cet historique, pour sélectionner un prédicteur parmi 2^m. Ce prédicteur peut être à 1 ou 2 bits.

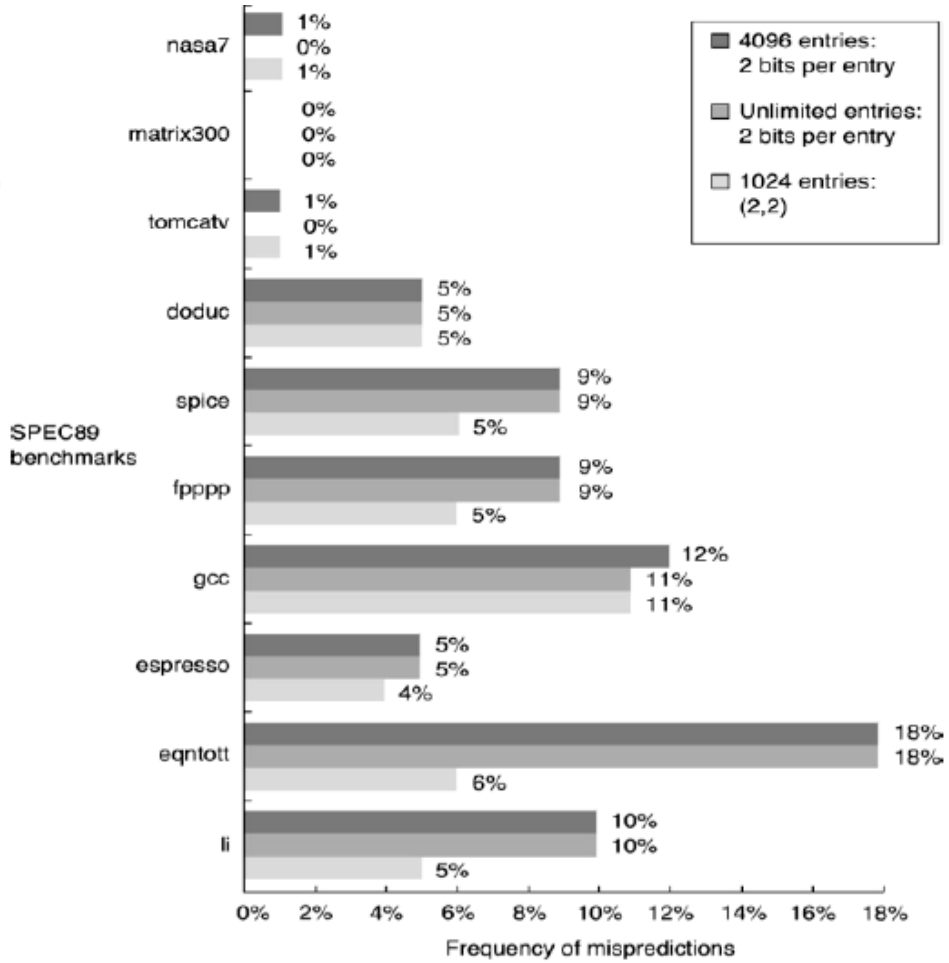
Exemple : prédicteur(2,2) (2 bits historique, 2 bits prédiction)



Performances comparées des prédicteurs

Prédicteur (2,2) meilleur que prédicteur 2bits pour la même taille de table (4k)

Meilleur qu'un prédicteur 2bits avec nombre d'entrées infini

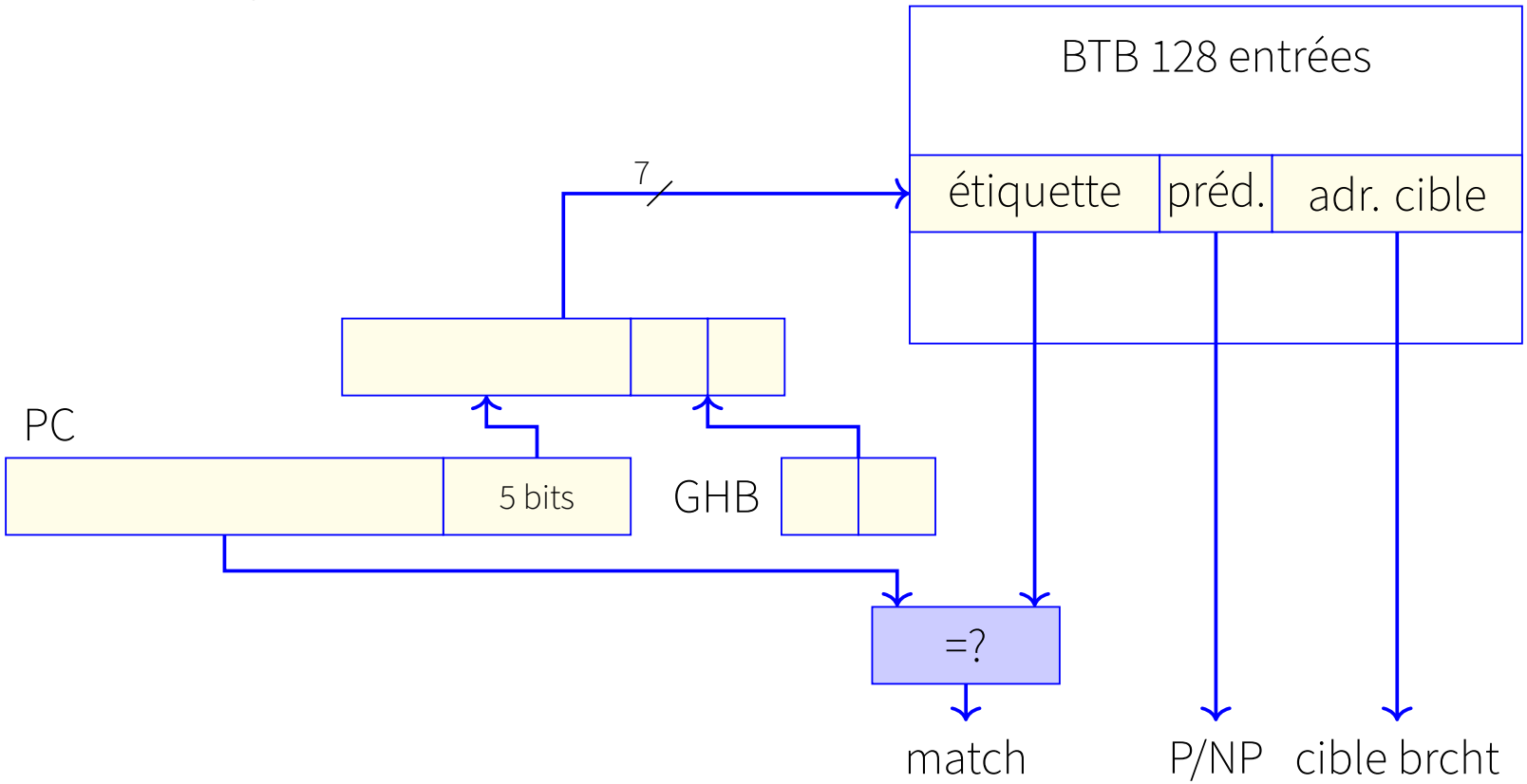


Aléas de branchement (cont.)

BTB

Le BTB fonctionne comme un cache.
Dans sa version la plus simple, il est à correspondance directe.

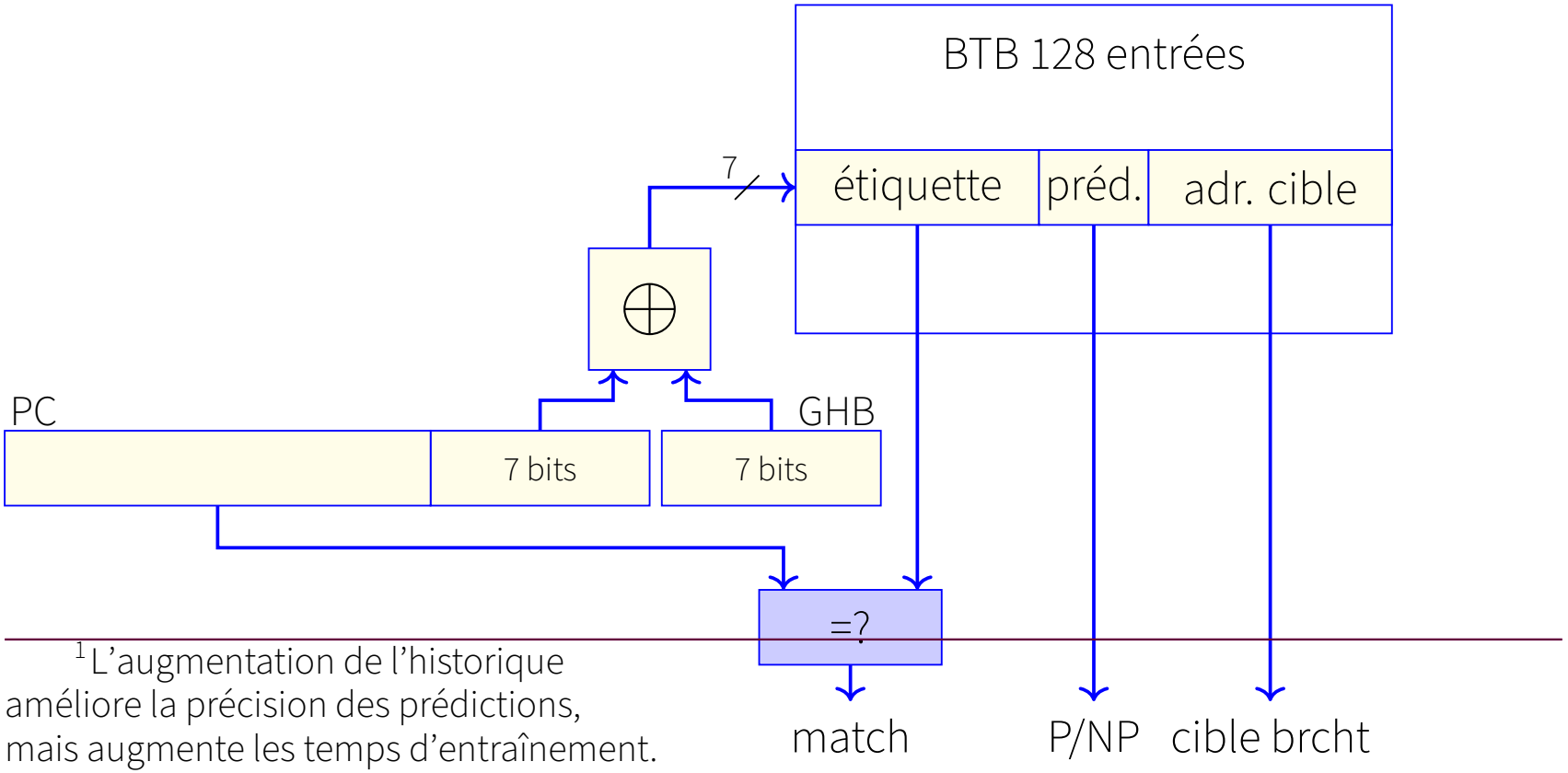
Exemple : 128 (2^7) entrées, 2 bits d'historique. On construit l'adresse dans le BTB en concaténant les 5 bits de poids faible du compteur de programme et les bits d'historique (**gselect**).



Aléas de branchement (cont.)

BTB

En pratique, historiques de branchements beaucoup plus importants[1].
→ utilisation d'une fonction de *hachage* : **ou** exclusif entre bits du GHB et poids faible de l'adresse pour construire l'adresse dans le BTB (**gshare**).
Exemple : BTB 127 entrées, 7 bits historique global.



[1] L'augmentation de l'historique améliore la précision des prédictions, mais augmente les temps d'entraînement.

Instructions conditionnelles

Les instructions conditionnelles présentes dans certaines architectures (ARM AArch32, C6x, IA64) permettent d'éviter des branchements.

```
if(r1)
  r3=r2;
```

Sans instructions conditionnelles

```
    bz r1,l1
    add r3,r0,r2
L1:...
```

Avec instructions conditionnelles

```
cmov r3,r2,r1
```

Marche bien pour de petits branchements

cmov présent dans x86-32 (depuis pentium pro)

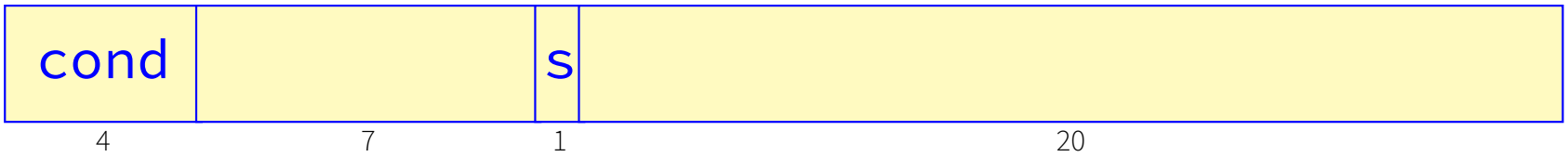
Aléas de branchement (cont.)

instructions conditionnelles

Jeu d'instruction ARM AArch32

Toute instruction

- peut être exécutée conditionnellement (au registre d'état) **cond**
- peut positionner le registre d'état (**s**)



0000	EQ	==	1000	HI	unsigned >
0001	NE	!=	1001	LS	unsigned <=
0010	CS / HS	unsigned >=	1010	GE	signed >=
0011	CC / LO	unsigned <	1011	LT	signed <
0100	MI	<0	1100	GT	signed >
0101	PL	>=0	1101	LE	signed <=
0110	VS	overflow	1110	AL	always
0111	VC	no overflow	0000	NV	never

Aléas de branchement (cont.)

instructions conditionnelles

Calcul du PGCD de deux nombres (d'après Euclide)

Programme en C

```
while (a != b) {
  if(a>b)
    a=a-b;
  else
    b=b-a;
}
```

Code assembleur ARM AArch32

```
gcd: cmp   ra,rb
     subgt ra,ra,rb
     sublt rb,rb,ra
     bne   gcd
```

Le **cmp** positionne le registre d'état (APSR) et conditionne les trois instructions suivantes.

Aléas de branchement (cont.)

Branchements retardés

Branchements retardés (*delayed branch*)
Historiquement utilisé dans les premiers processeurs RISC (mips)
Changement de la sémantique du branchement : *branch-and-execute* `brx`

- exécuter l'instruction qui suit le branchement (toujours)
- puis prendre (ou non) le branchement

Si on arrive à déplacer une instruction *utile* et sans dépendance après le `brx`, pas de pénalité de branchement.
Sinon, mettre un `nop` après le branchement.

```
if(! r1)
  r2+=5;
r1++;
```

Sans branchement retardé

```
    bz   r1,L1
    # délai brcht
    addi r2,r2,5
L1: addi r1,r1,1
```

Avec branchement retardé

```
    bzx  r1,L1
    addi r1,r1,1
    addi r2,r2,5
L1: ...
```

`addi r1,1` toujours exécuté

Inutile avec les fortes pénalités de branchement des processeurs actuels

Aléas structurels

Arrivent quand deux instructions différentes ont besoin de la même ressource matérielle.

- arrivait dans les premiers processeurs RISC sans mémoire cache séparée instructions/données
  La phase LI veut accéder à la mémoire unique I/D et peut entrer en conflit avec l'accès mémoire d'une instruction de chargement/rangement.
- arrive avec des opérateurs de calcul non pipelinés.
  Exemple :
  ```
  fdiv f3,f2,f1
  fdiv f6,f5,f4
  ```
- arrive dans les processeurs actuels quand certaines ressources nécessaires pour lancer une instruction sont occupées (stations de réservation, registres de renommage, entrée de LSQ, etc). (voir cours sur parallélisme d'instructions)

Aléas structurels (cont.)

Solutions :

- Supprimer ou limiter l'aléa structurel en dupliquant/augmentant dans l'architecture les ressources matérielles, pipelinant les opérateurs, etc.
- Détecter le conflit et procéder à une suspension du pipeline.

  La deuxième instruction est bloquée (ainsi que toutes les instructions après elle) jusqu'à ce que la première libère la ressource.

.