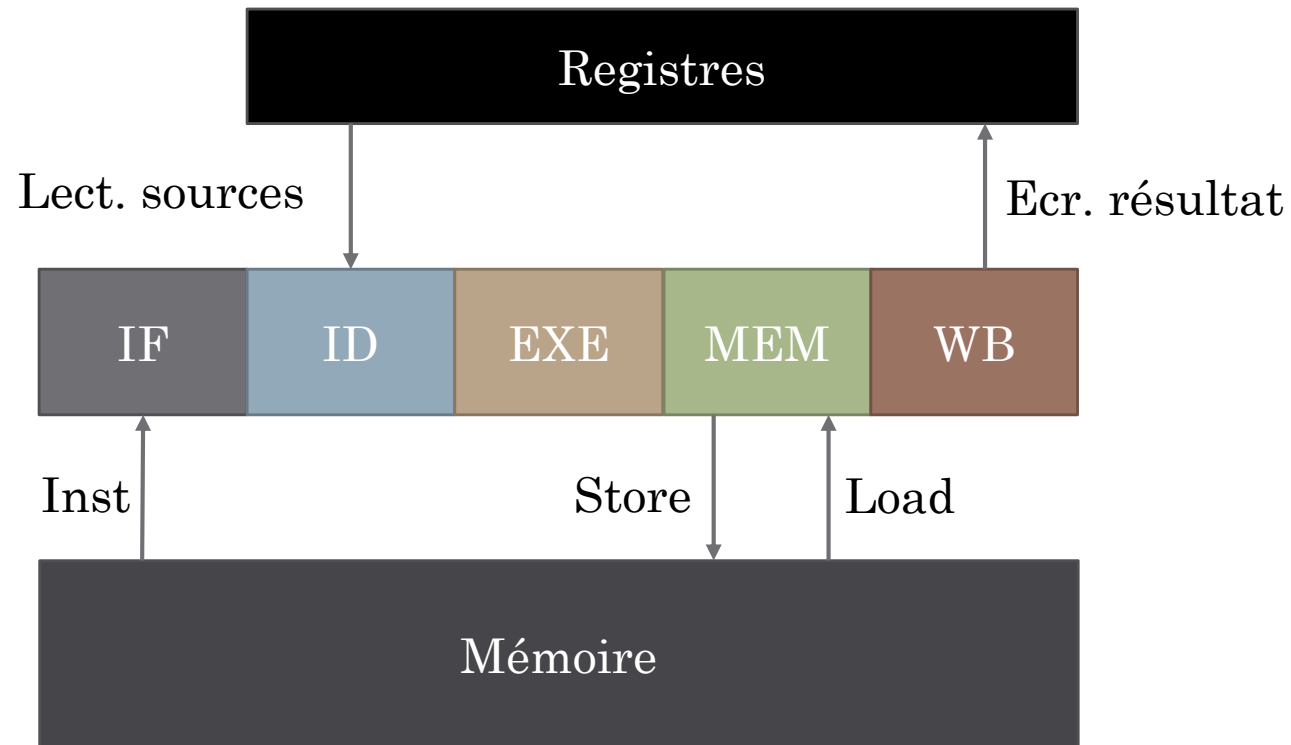


# Pipelining et prédiction de branchement

SEOC3A – CEAMC

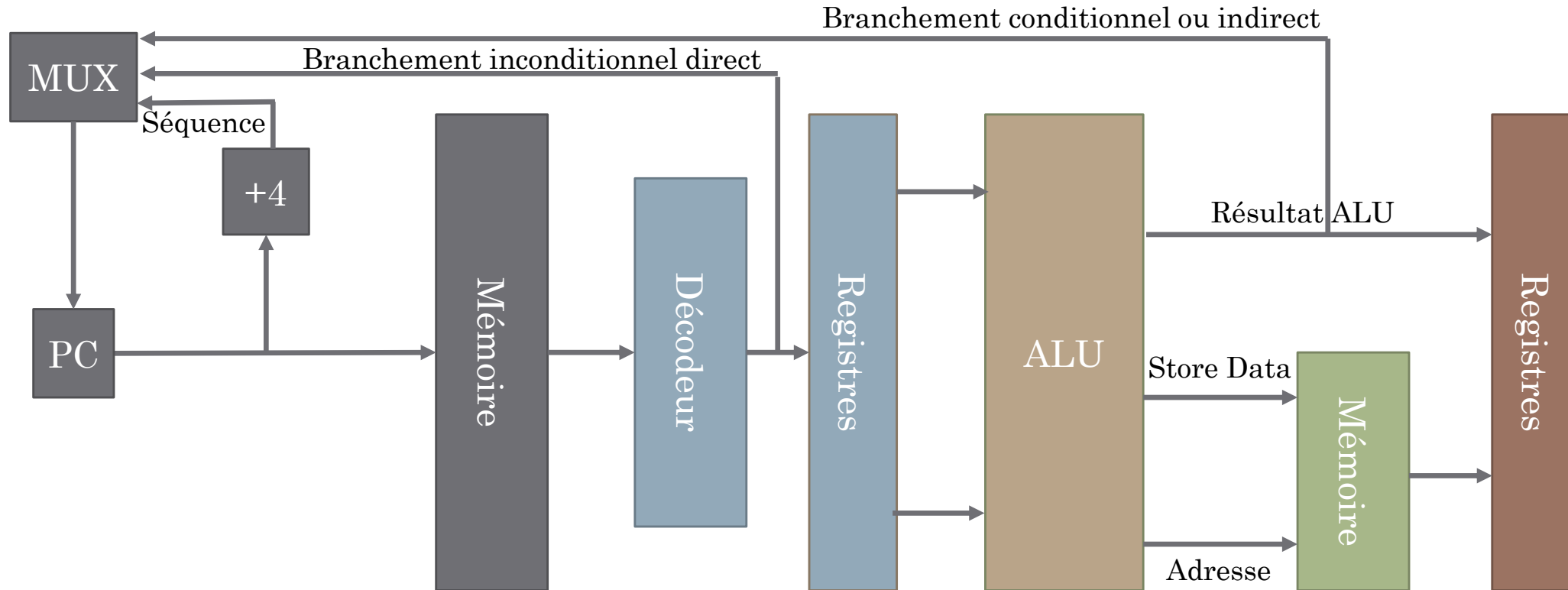
Arthur Perais (arthur.perais@univ-grenoble-alpes.fr)

# CPU minimaliste

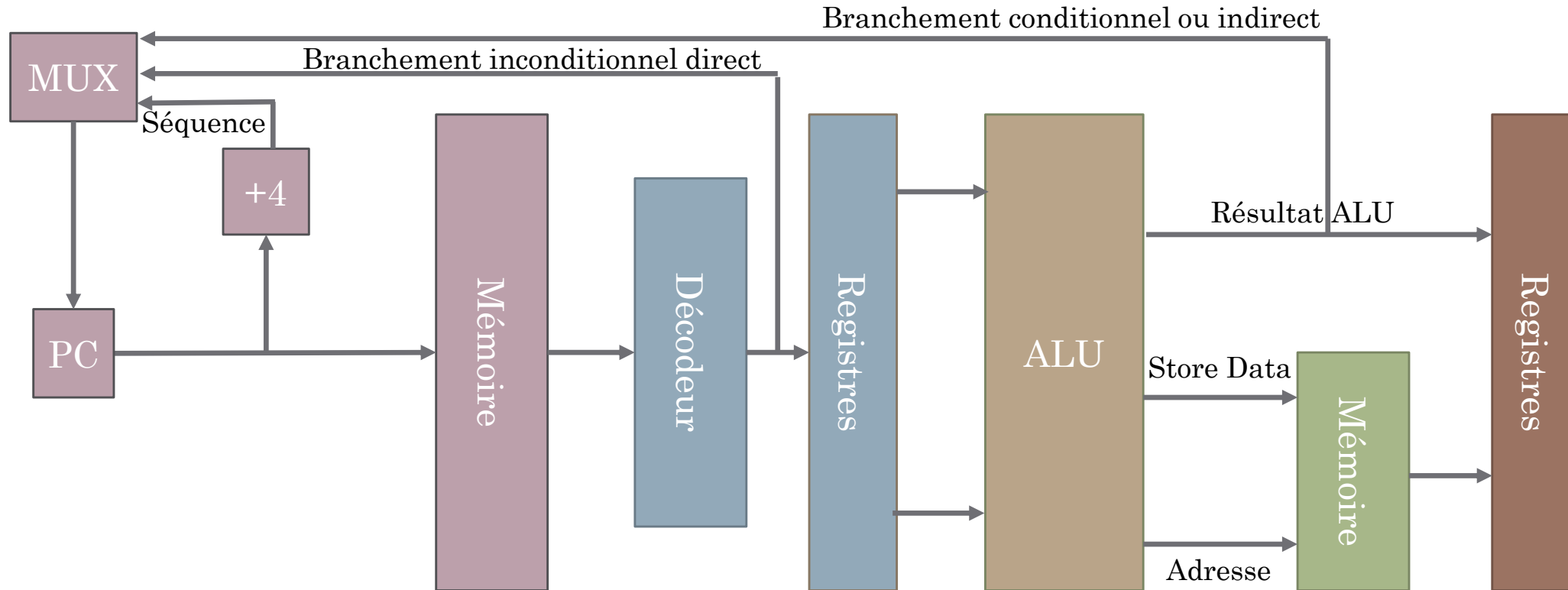


Latence : 5 CPI  
Débit : 1 IPC

# Exemple de CPU minimaliste



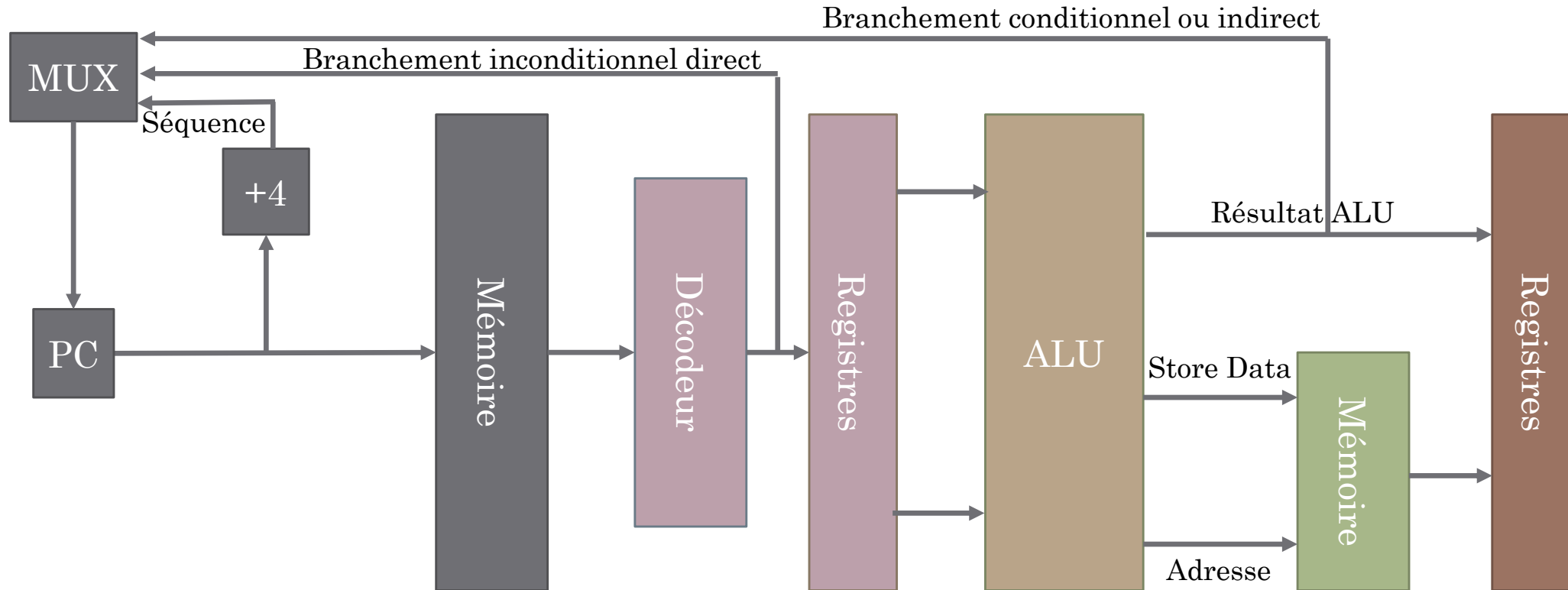
# Exemple de CPU minimaliste



## Instruction Fetch (IF)

1. Récupérer l'instruction machine depuis la mémoire
2. Calculer l'adresse de la prochaine instruction à récupérer

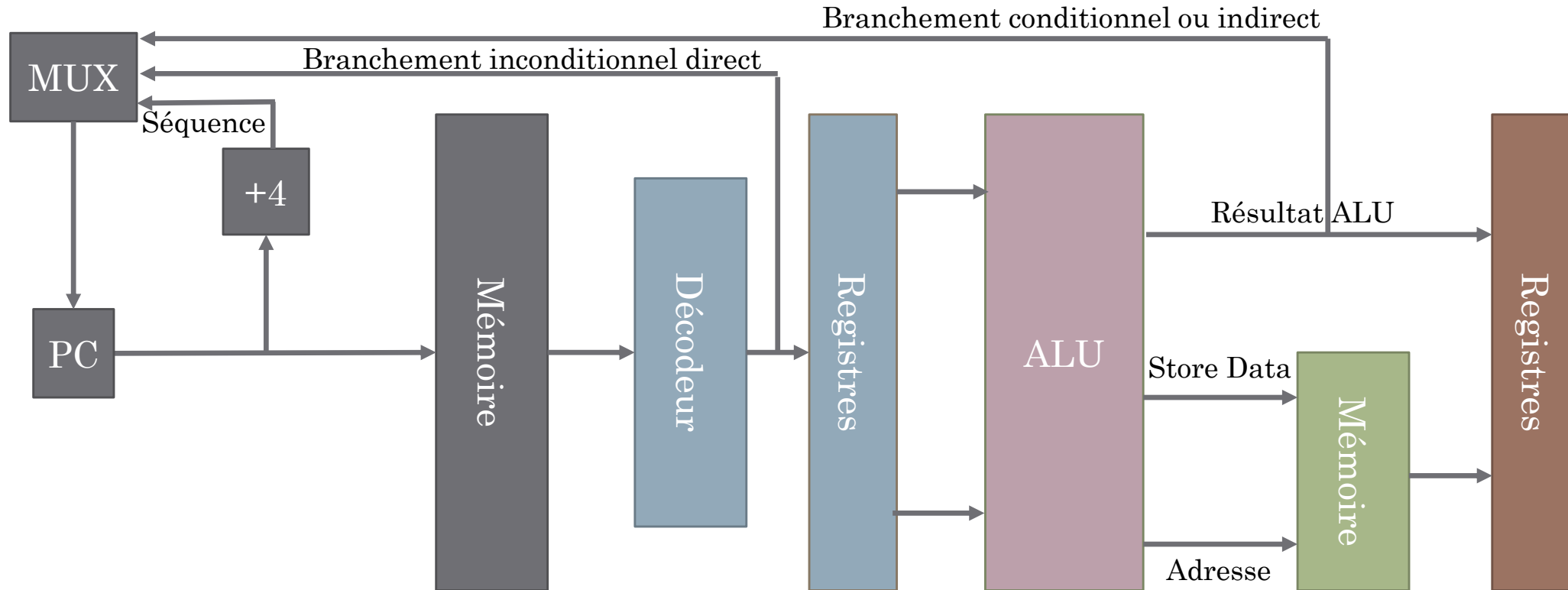
# Exemple de CPU minimaliste



## Instruction Decode (ID)

1. Décoder l'instruction pour former le vecteur de contrôle du reste de la machine
2. Gérer les branchements inconditionnels directs
3. Lecture des opérandes sources

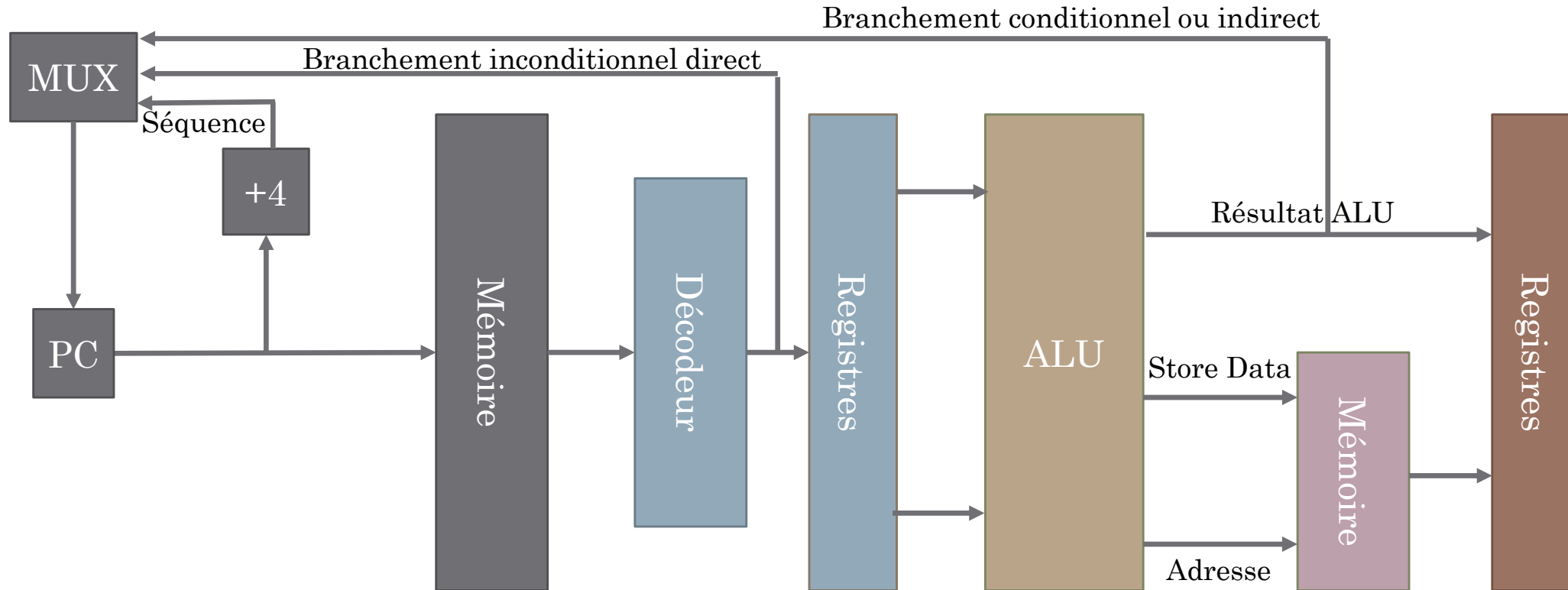
# Exemple de CPU minimaliste



## Execute (EXE)

1. Calcul du résultat de l'instruction (ALU)
2. Si opération mémoire, calcul de l'adresse

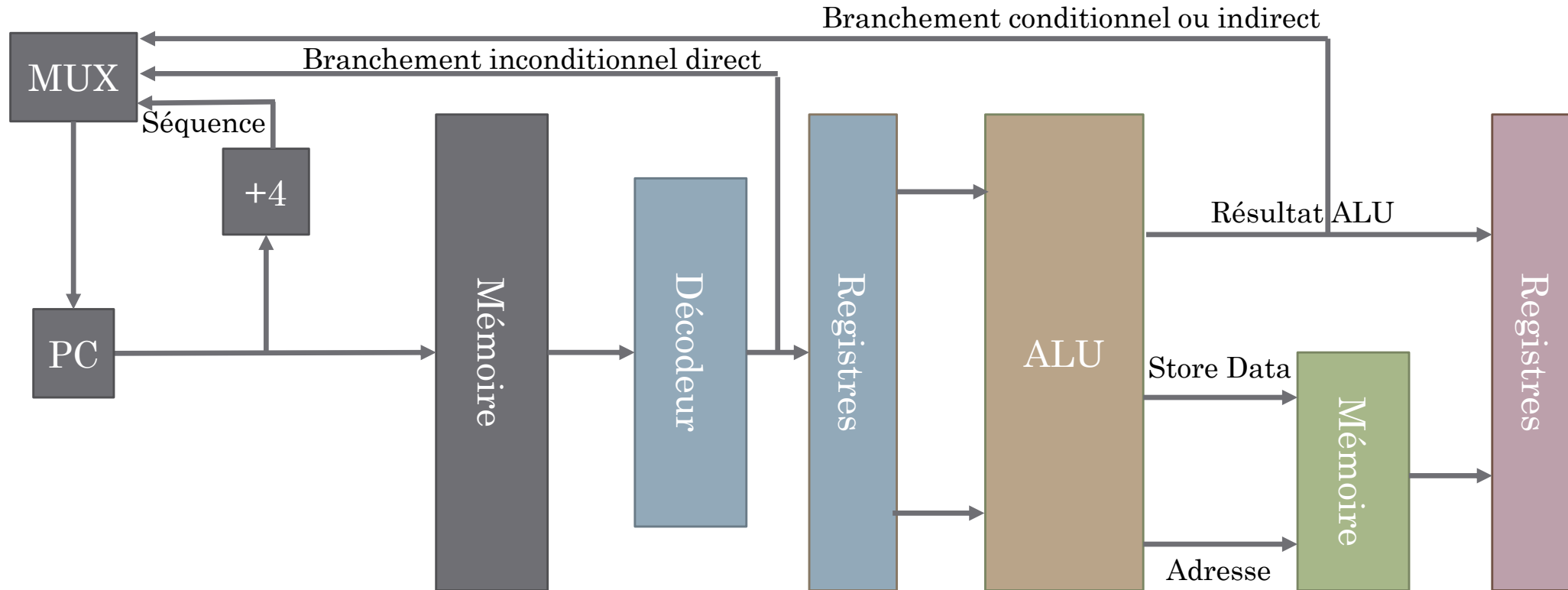
# Exemple de CPU minimaliste



## Accès mémoire (MEM)

1. Accès à la mémoire

# Exemple de CPU minimaliste



## Writeback (WB)

1. Ecriture du résultat dans le fichier de registres



# CPU minimaliste

- Ce pipeline est une *implémentation* matérielle possible
  - Invisible du logiciel -> microarchitecture
  - Pas la seule implémentation possible
- Quelles sont les limites à la performance ?

# CPU minimaliste

- Ce pipeline est une *implémentation* matérielle possible
  - Invisible du logiciel -> microarchitecture
  - Pas la seule implémentation possible
- Quelles sont les limites à la performance ?
  - Une instruction par cycle au maximum

# CPU minimaliste

- Ce pipeline est une *implémentation* matérielle possible
  - Invisible du logiciel -> microarchitecture
  - Pas la seule implémentation possible
- Quelles sont les limites à la performance ?
  - Une instruction par cycle au maximum
  - Dépendances
    - Entre instructions dans le programme
    - Structurelles (causées par le pipeline lui-même)

# Dépendances de données – Arch.

- Dépendances entre producteur et consommateur

```
{  
  a = b + c;  
  d = a + b;  
}
```

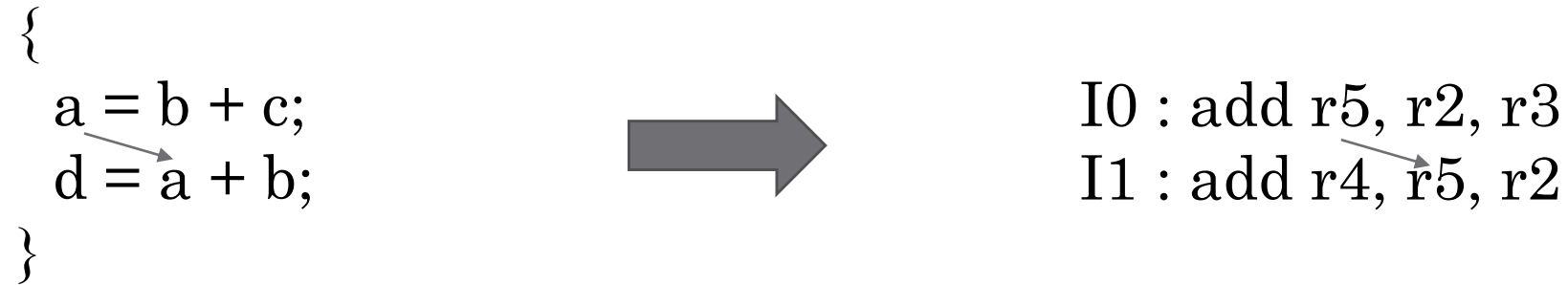


```
I0 : add r5, r2, r3  
I1 : add r4, r5, r2
```

Dépendances?

# Dépendances de données – Arch.

- Dépendances entre producteur et consommateur



Dépendance de donnée au niveau *architectural*

I1 consomme le résultat produit par I0, donc I1 est exécutée **après** I0

# Dépendances de données – $\mu$ arch.

- Comment la dépendance de donnée au niveau architectural est-elle traduite dans le pipeline ( $\mu$ arch) ?

I0 : add r5, r2, r3



I1 : add r4, r5, r2



# Dépendances de données – $\mu$ arch.

- Comment la dépendance de donnée au niveau architectural est-elle traduite dans le pipeline ( $\mu$ arch) ?

I0 : add r5, r2, r3



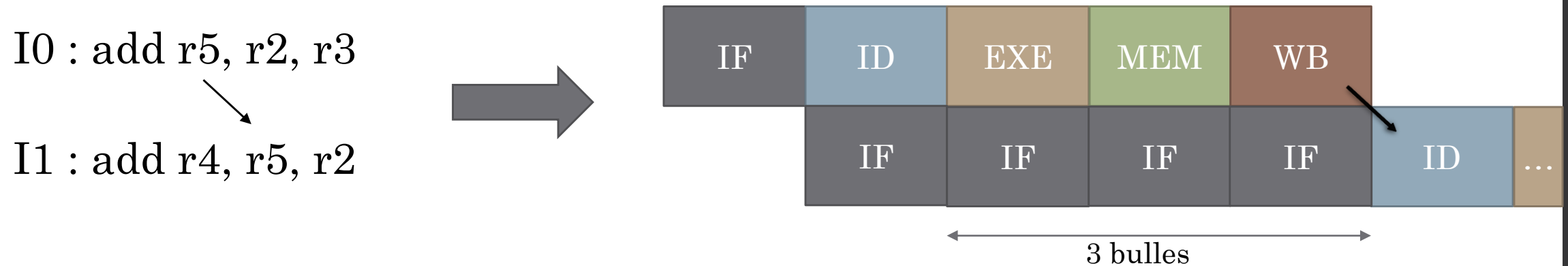
I1 : add r4, r5, r2



Architectural	Microarchitectural
Dépendance de donnée entre I0 et I1 sur r5, I1 s'exécute <b>après</b> I0	I1 doit lire r5 dans ID <b>après</b> que I0 ait écrit r5 dans WB
	=
	I1 quitte EXE au minimum 4 <b>cycles après</b> I0

# Dépendances de données – $\mu$ arch.

- Comment la dépendance de donnée au niveau architectural est-elle traduite dans le pipeline ( $\mu$ arch) ?

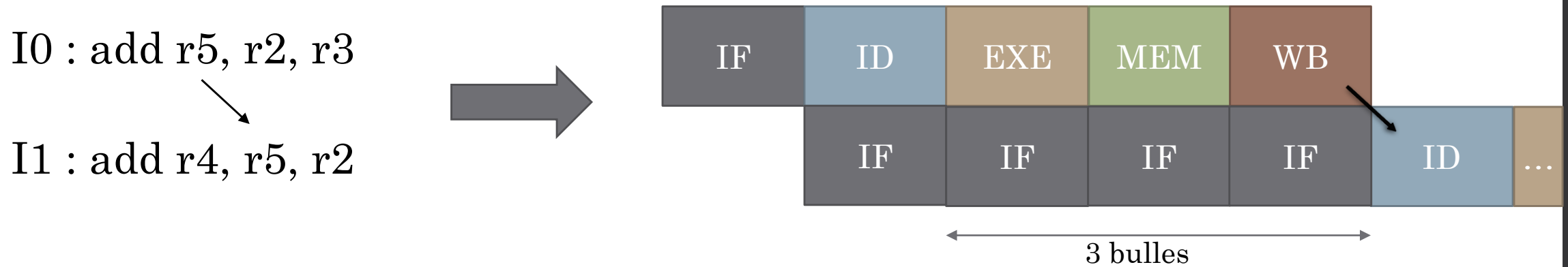


Architectural	Microarchitectural
Dépendance de donnée entre I0 et I1 sur r5, I1 s'exécute <b>après</b> I0	I1 doit lire r5 dans ID <b>après</b> que I0 ait écrit r5 dans WB
	=
	I1 quitte EXE au minimum 4 <b>cycles après</b> I0



# Dépendances de données – $\mu$ arch.

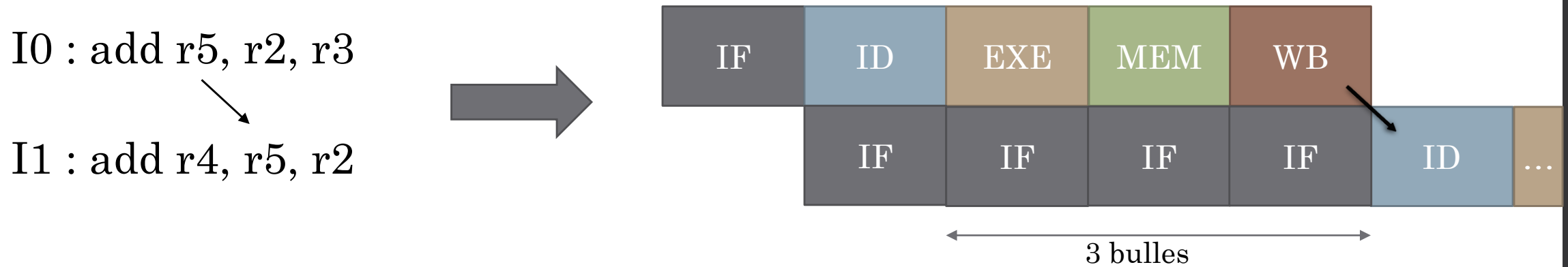
- Comment la dépendance de donnée au niveau architectural est-elle traduite dans le pipeline ( $\mu$ arch) ?



La structure du pipeline empire la dépendance de donnée via une dépendance **structurelle** = qui provient des limites de l'implémentation et non de la sémantique des instructions

# Dépendances de données – $\mu$ arch.

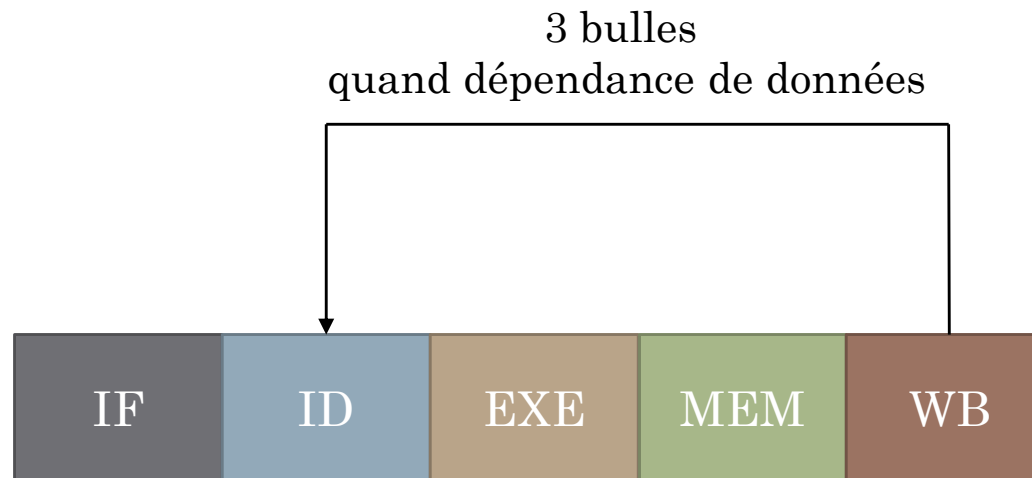
- Comment la dépendance de donnée au niveau architectural est-elle traduite dans le pipeline ( $\mu$ arch) ?



- Va empêcher d'atteindre la performance crête (1 IPC)
  - Au pire, toutes les instructions dos à dos sont dépendantes : 0,25 IPC max vs. 1 IPC max
  - Sauf à être capable de toujours avoir trois instructions indépendantes entre un producteur et un consommateur

# Dépendances de données – $\mu$ arch.

- Notion de boucle microarchitecturale
  - Plus la boucle est longue, plus on doit trouver des instructions indépendantes pour occuper le pipeline entre les deux instructions dépendantes
    - Pas toujours faisable



# Dépendances de données – $\mu$ arch.

- Les dépendances structurelles sont causées par la microarchitecture
  - On peut donc les supprimer en améliorant la microarchitecture
- Implémentation sans pipeline ? 1 cycle pour traiter une instruction
  - Pas de dépendance structurelle sur les données : prod cycle 1, cons cycle 2
  - Oui mais...Performance crête :
    - Sans pipeline : 1 IPC @ XGhz
    - Avec pipeline : 0,25 IPC @ **5X**Ghz (mieux que sans pipeline **sans rien faire**)
- Mieux vaut essayer de limiter les dépendances structurelles d'une implémentation pipelinée

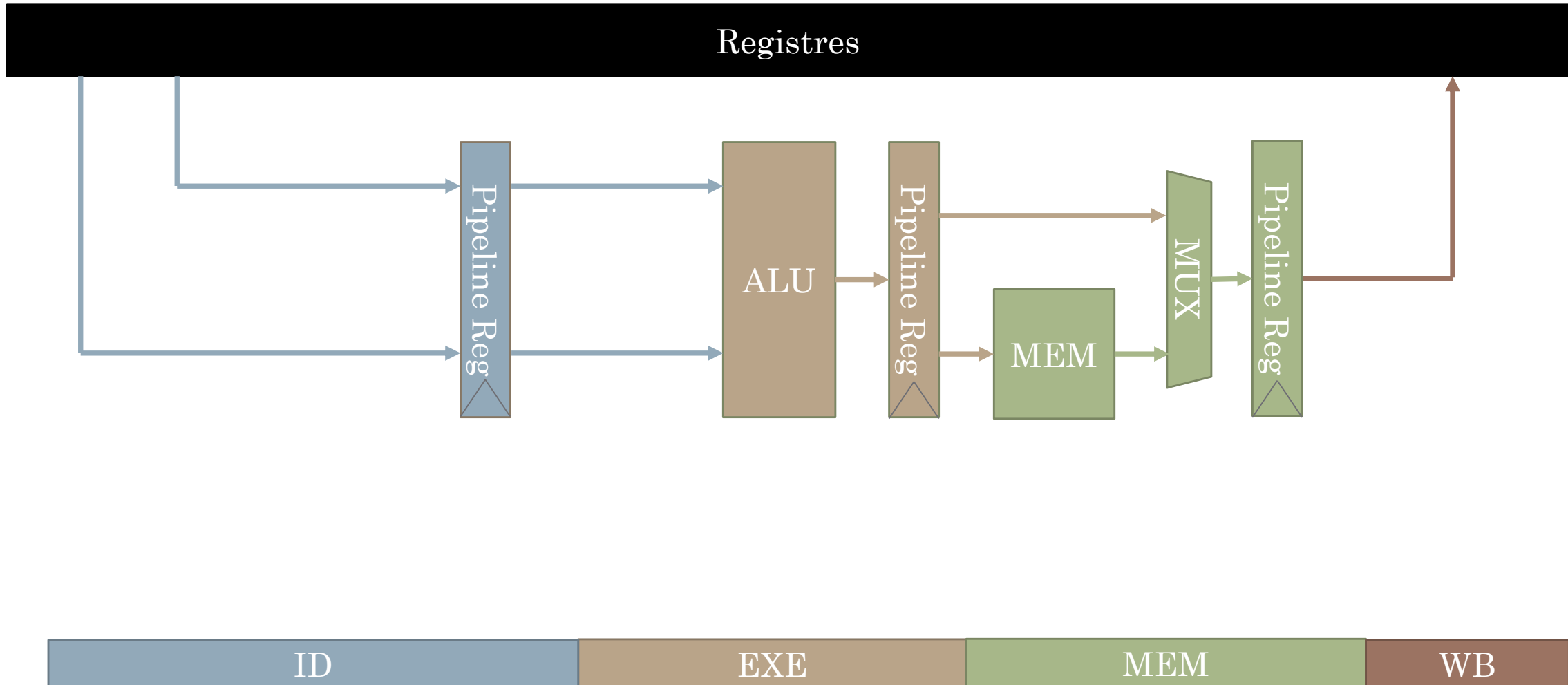
# Dépendances de données – $\mu$ arch.

- Les dépendances structurelles sont causées par la microarchitecture
  - On peut donc les supprimer en améliorant la microarchitecture
- Réseau de bypass pour supprimer les dépendances structurelles de notre pipeline sur les données

Architectural	$\mu$ arch sans bypass	$\mu$ arch avec bypass
Dépendance de donnée entre I0 et I1 sur r5, I1 s'exécute <b>après I0</b>	I1 quitte EXE au minimum <b>4 cycles après I0</b>	I1 quitte EXE au minimum <b>un cycle après I0</b> = le mieux que l'on puisse faire vu l'existence de la dépendance de donnée architecturale

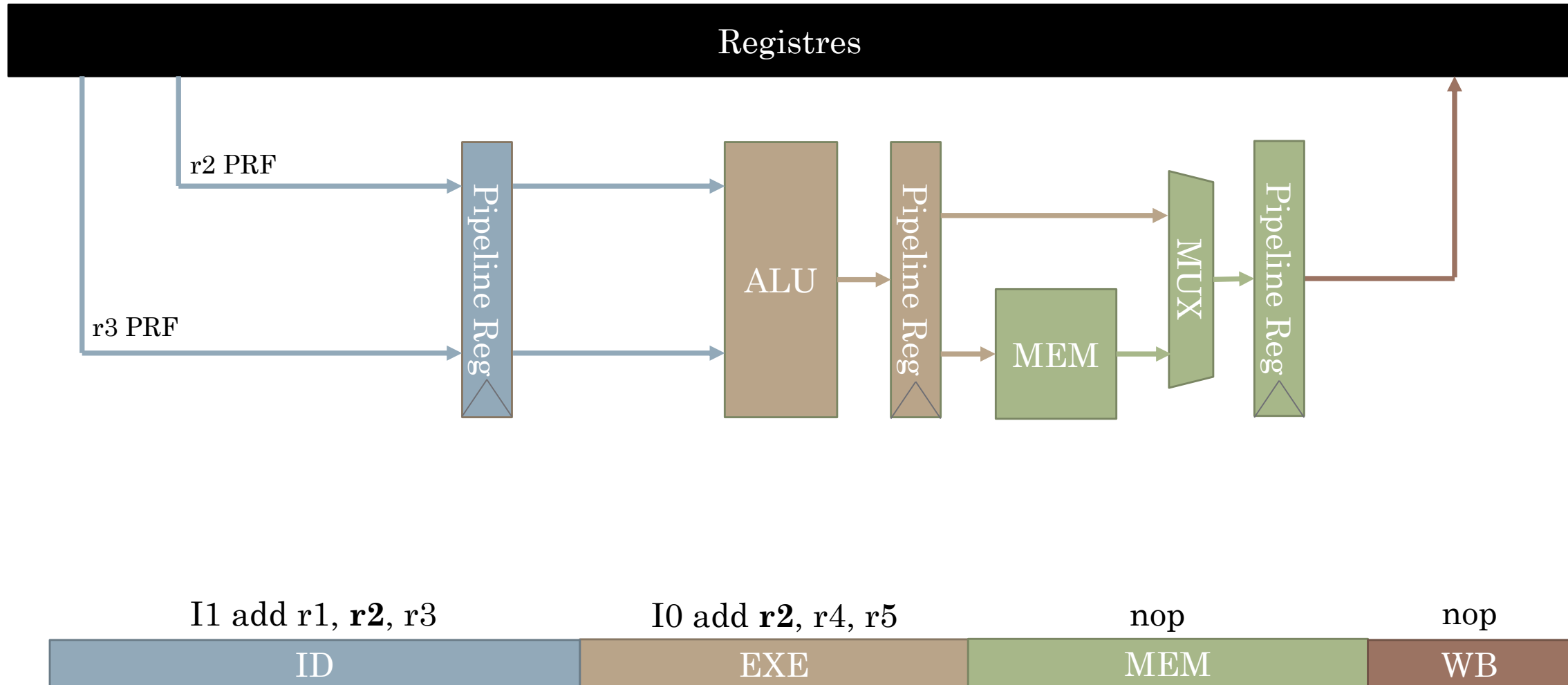
# Dépendances de données – Réseau de bypass

- On a trois bulles à cacher (distance entre ID et WB)



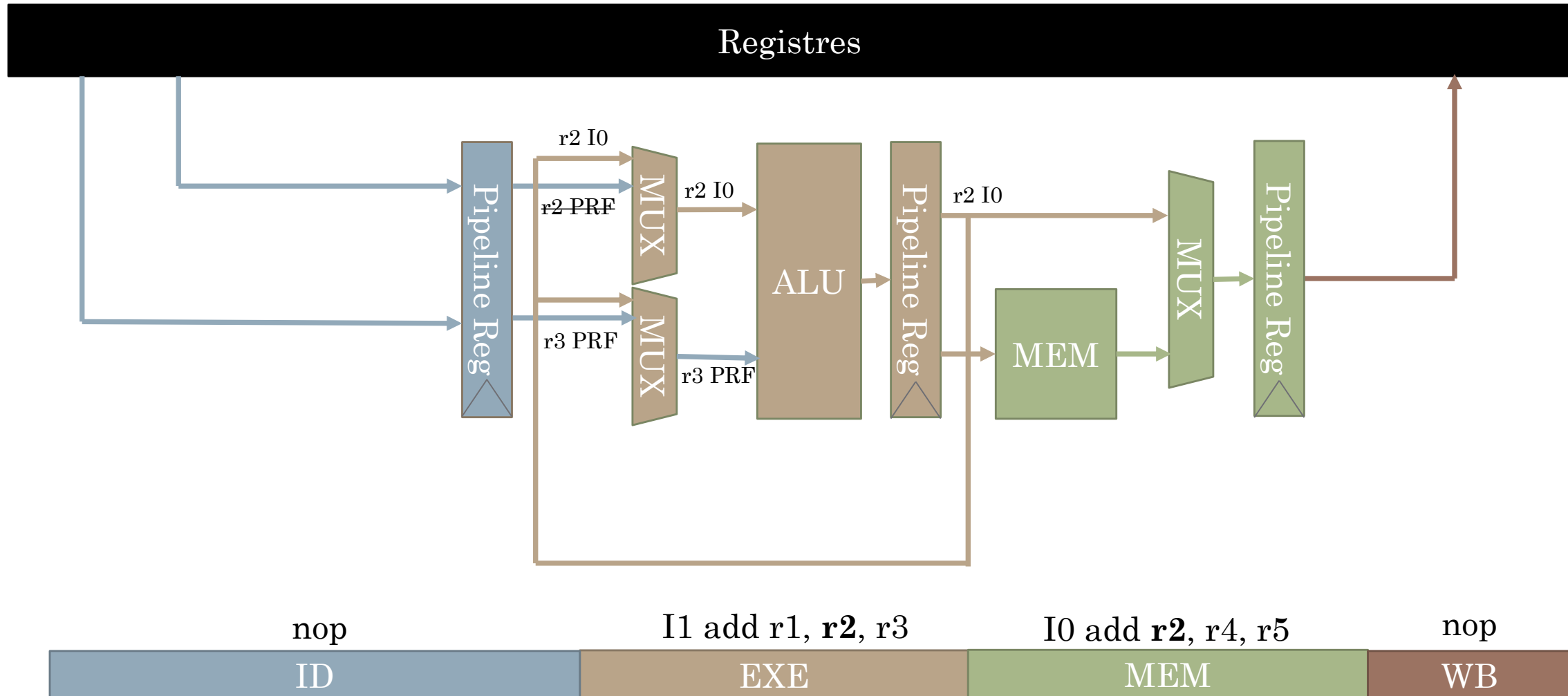
# Dépendances de données – Réseau de bypass

- Cas 1 : Producteur/consommateur dos à dos



# Dépendances de données – Réseau de bypass

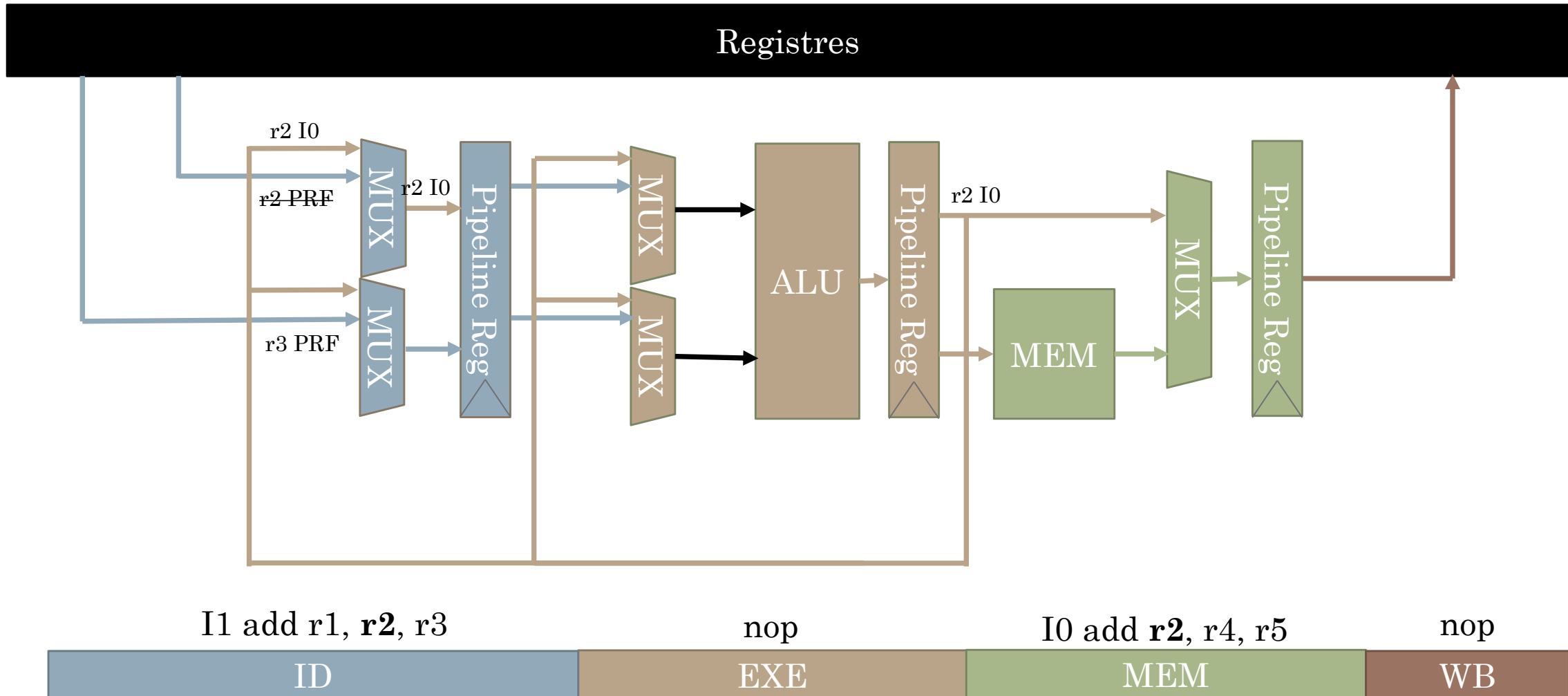
- Cas 1 : Producteur/consommateur dos à dos





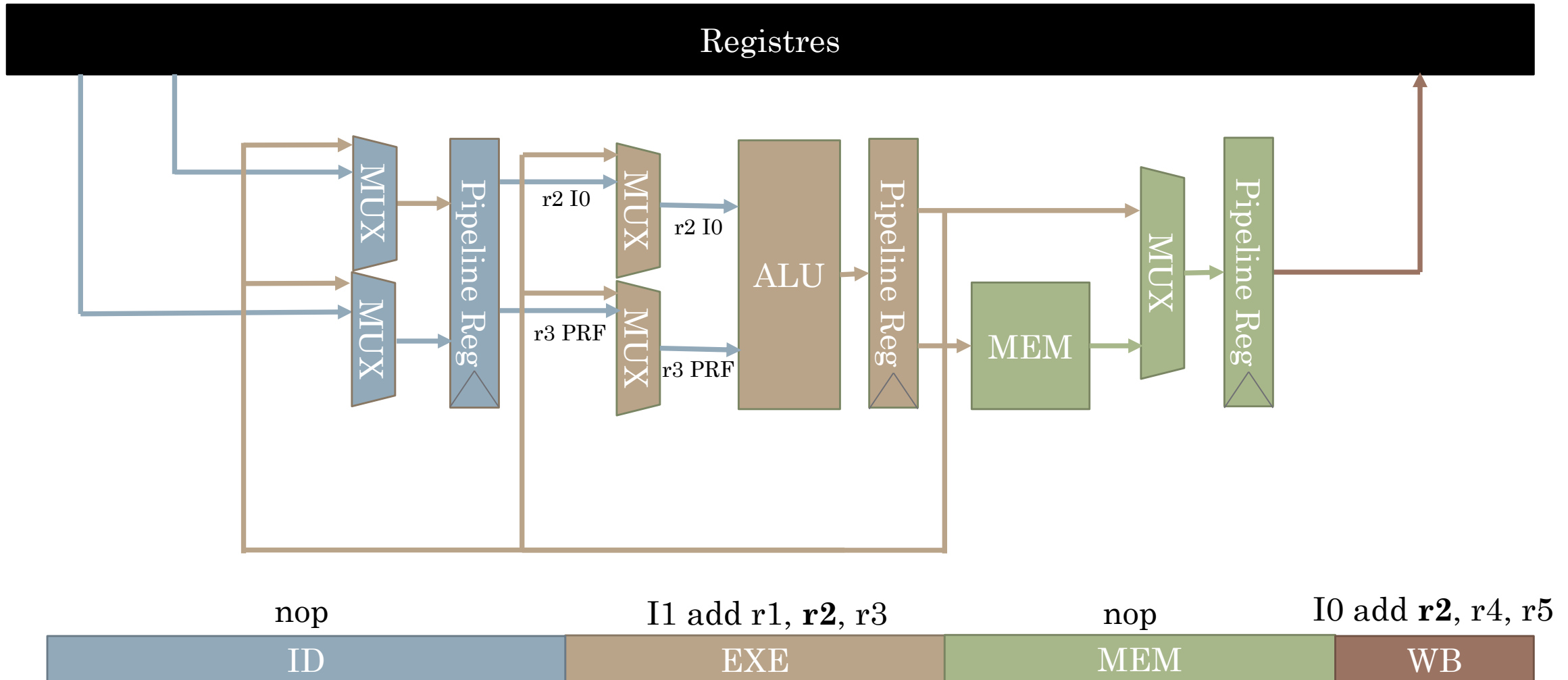
# Dépendances de données – Réseau de bypass

- Cas 2 : Producteur/consommateur avec un cycle d'écart



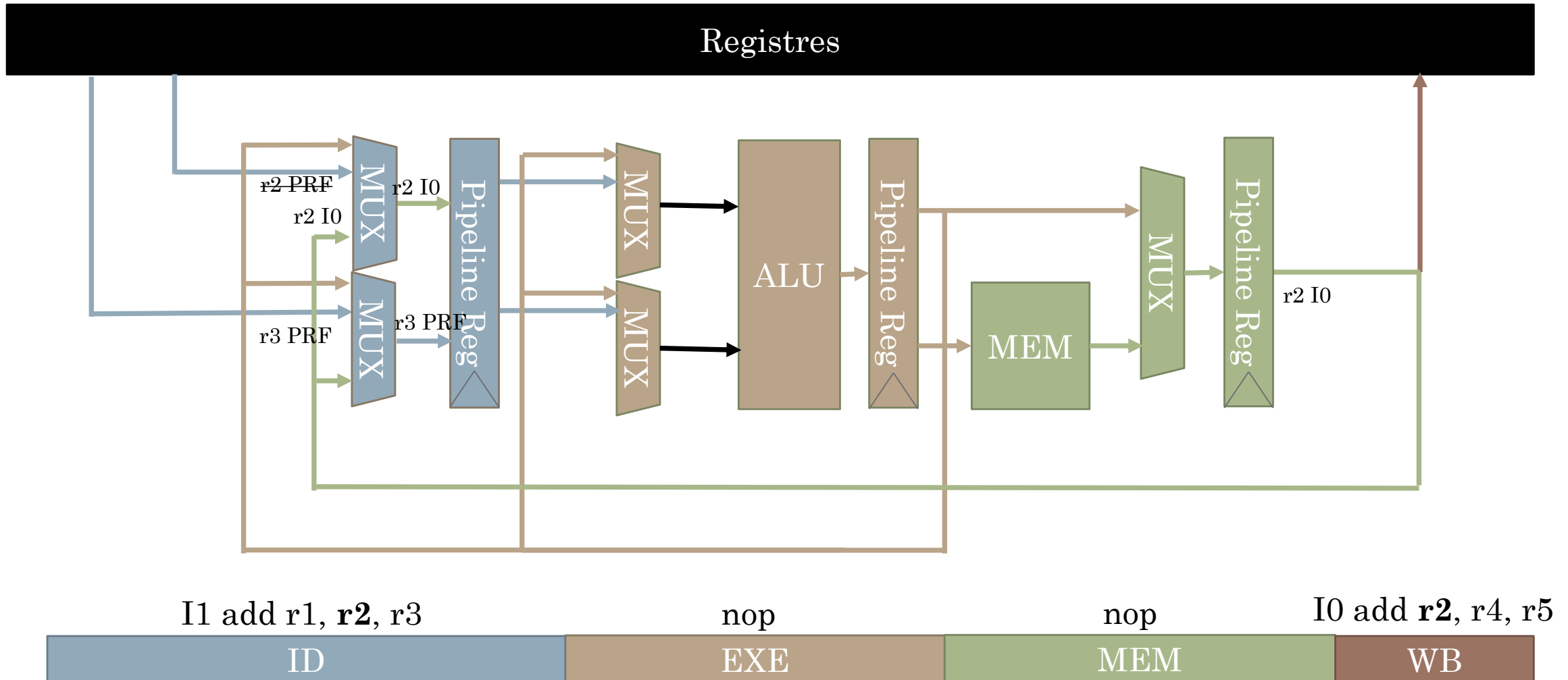
# Dépendances de données – Réseau de bypass

- Cas 2 : Producteur/consommateur avec un cycle d'écart



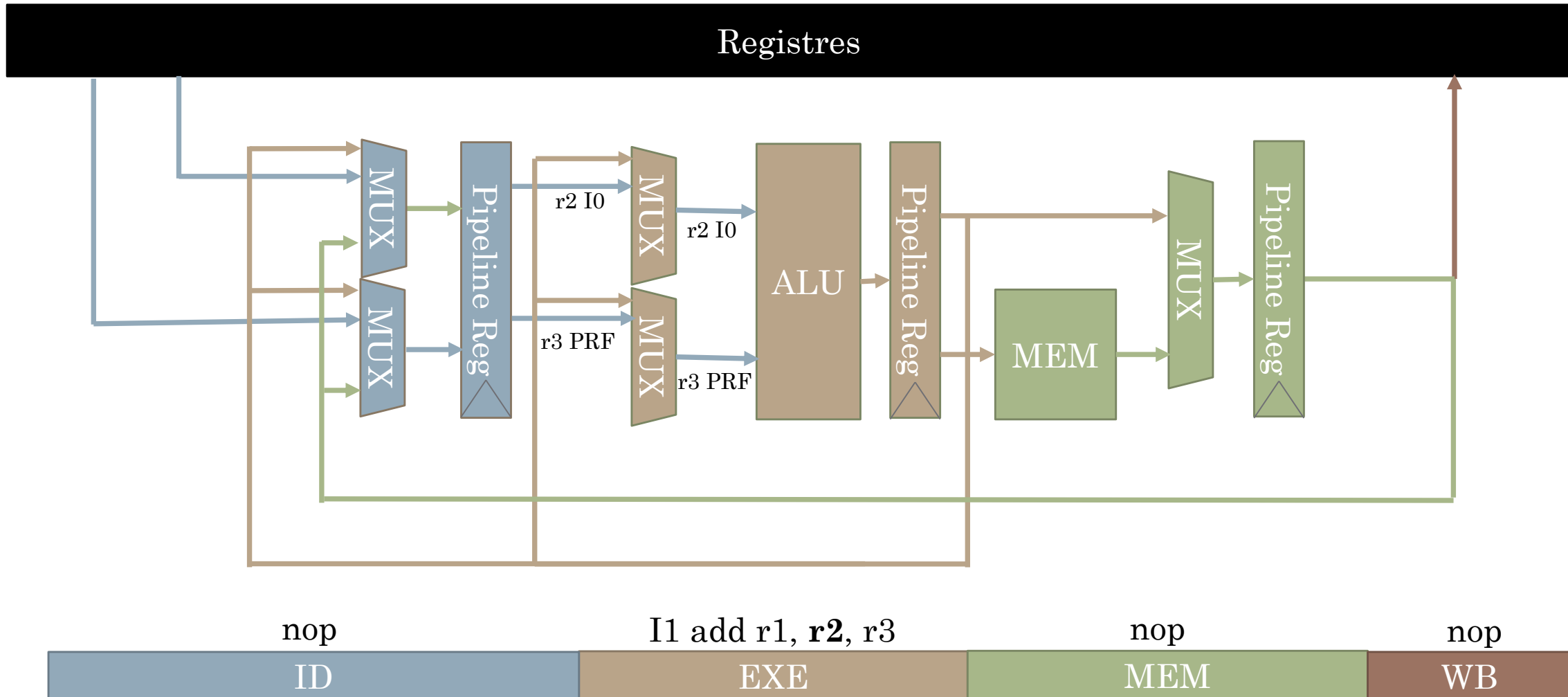
# Dépendances de données – Réseau de bypass

- Cas 3 : Producteur/consommateur avec deux cycles d'écart



# Dépendances de données – Réseau de bypass

- Cas 3 : Producteur/consommateur avec deux cycles d'écart



# Dépendances de données – Réseau de bypass

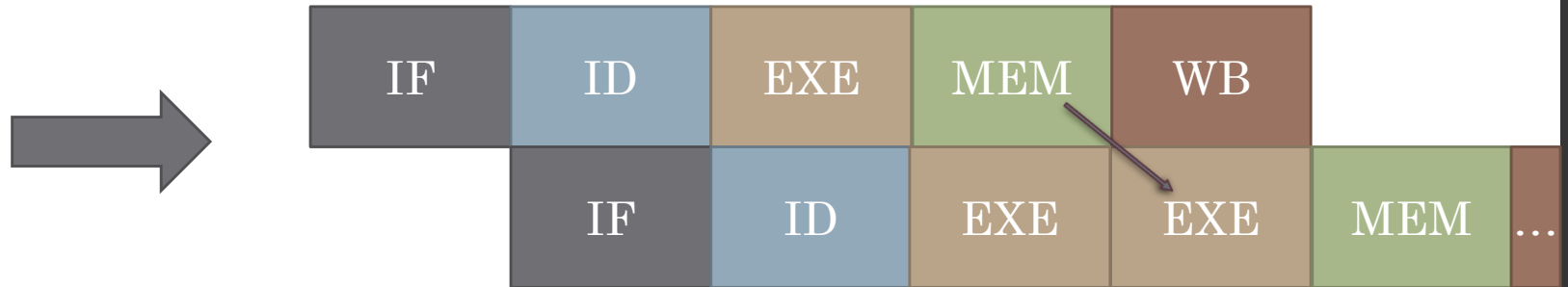
- A-t-on géré tous les cas ?

# Dépendances de données – Réseau de bypass

- A-t-on géré tous les cas ?
- Chargement mémoire !
  - EXE et MEM sont des étages différents
  - Une autre dépendance **structurelle** : I1 quitte EXE au minimum **deux cycles** après I0 si I0 est un chargement mémoire et il existe une dépendance de données entre I1 et I0

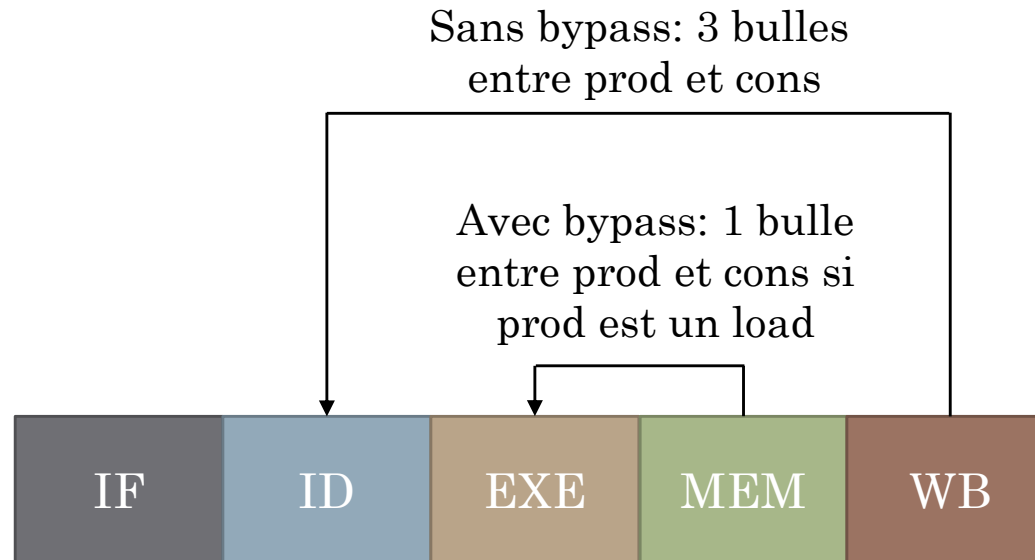
I0 : ld r5, 0(r2)

I1 : add r4, r5, r2



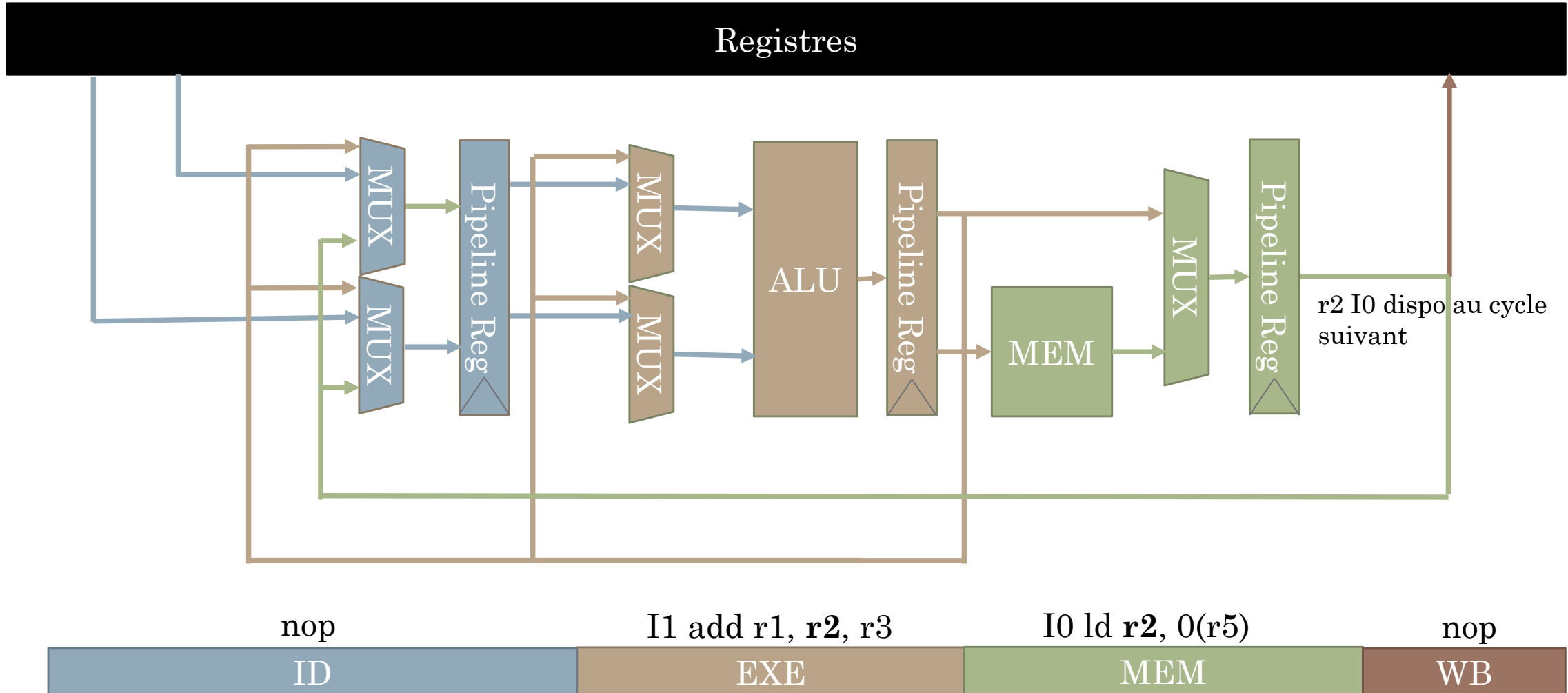
# Dépendances de données – Réseau de bypass

- On retrouve la notion de boucle microarchitecturale



# Dépendances de données – Réseau de bypass

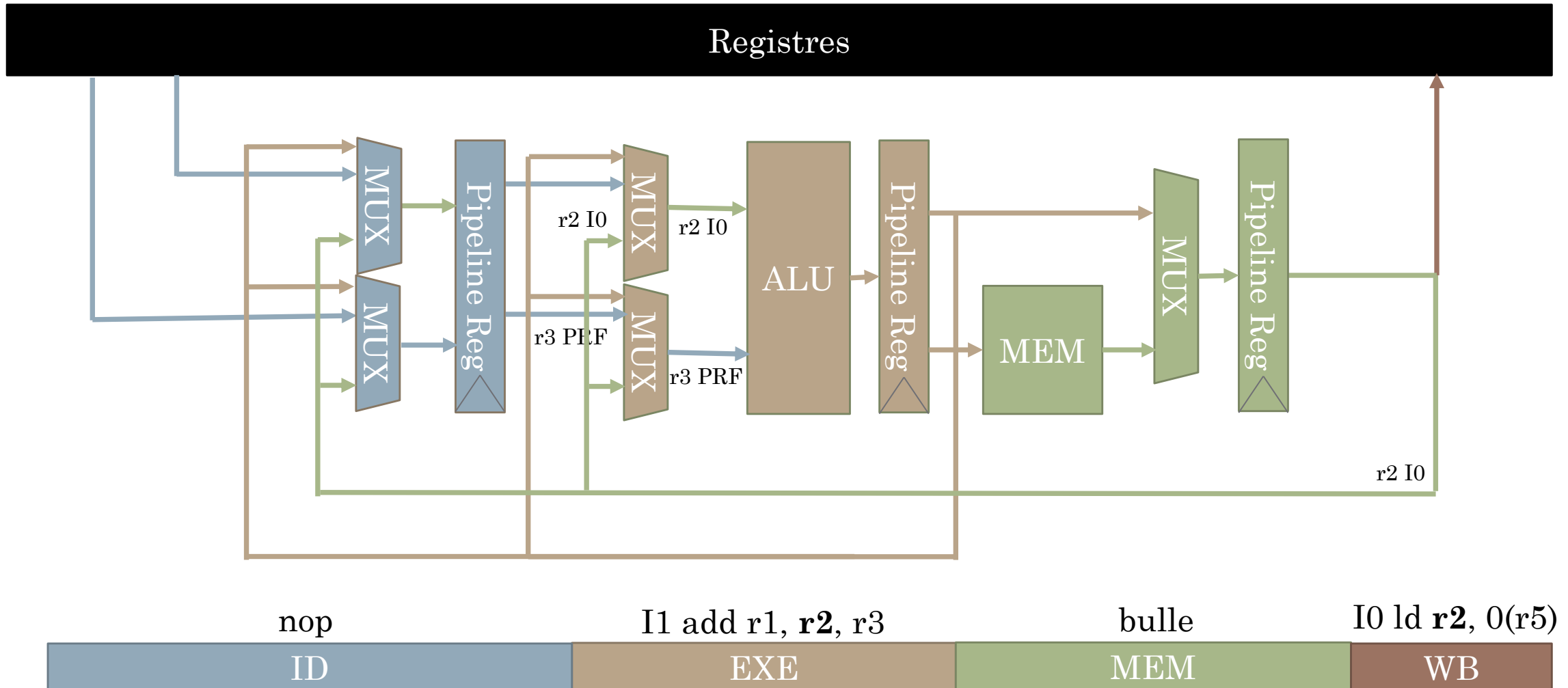
- Cas 4 : Chargement mémoire et consommateur dos à dos





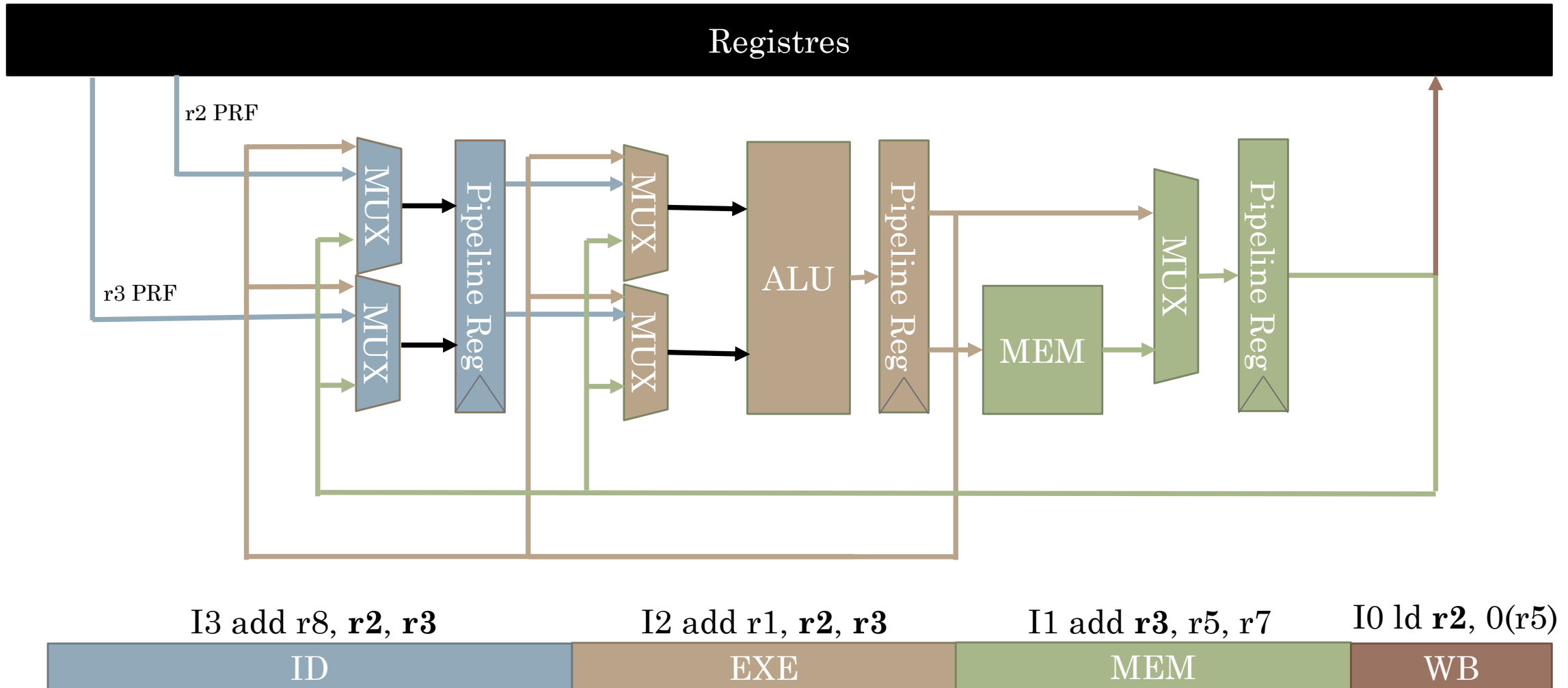
# Dépendances de données – Réseau de bypass

- Cas 4 : Chargement mémoire et consommateur dos à dos



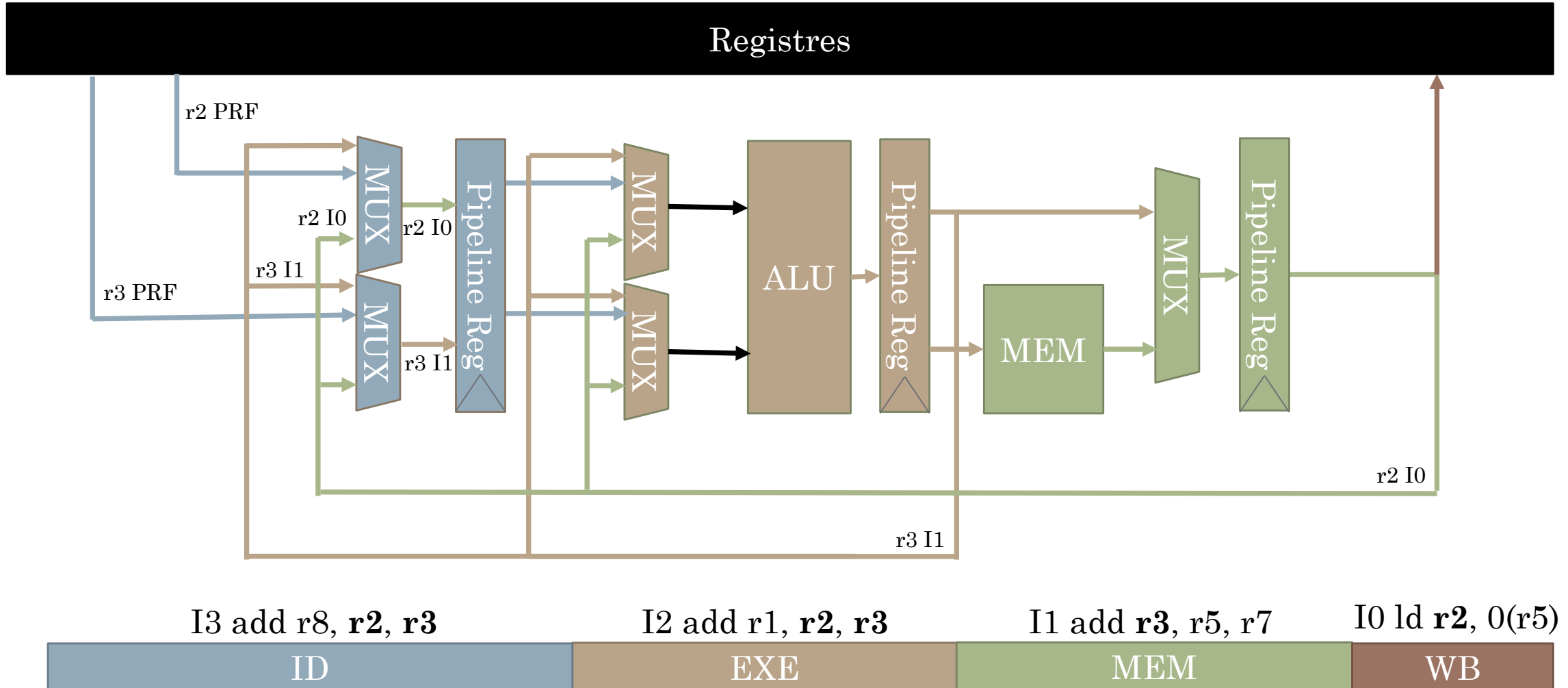
# Dépendances de données – Réseau de bypass

- Cas 5&6 : Quelle provenance pour les opérandes de I2 et I3 ?



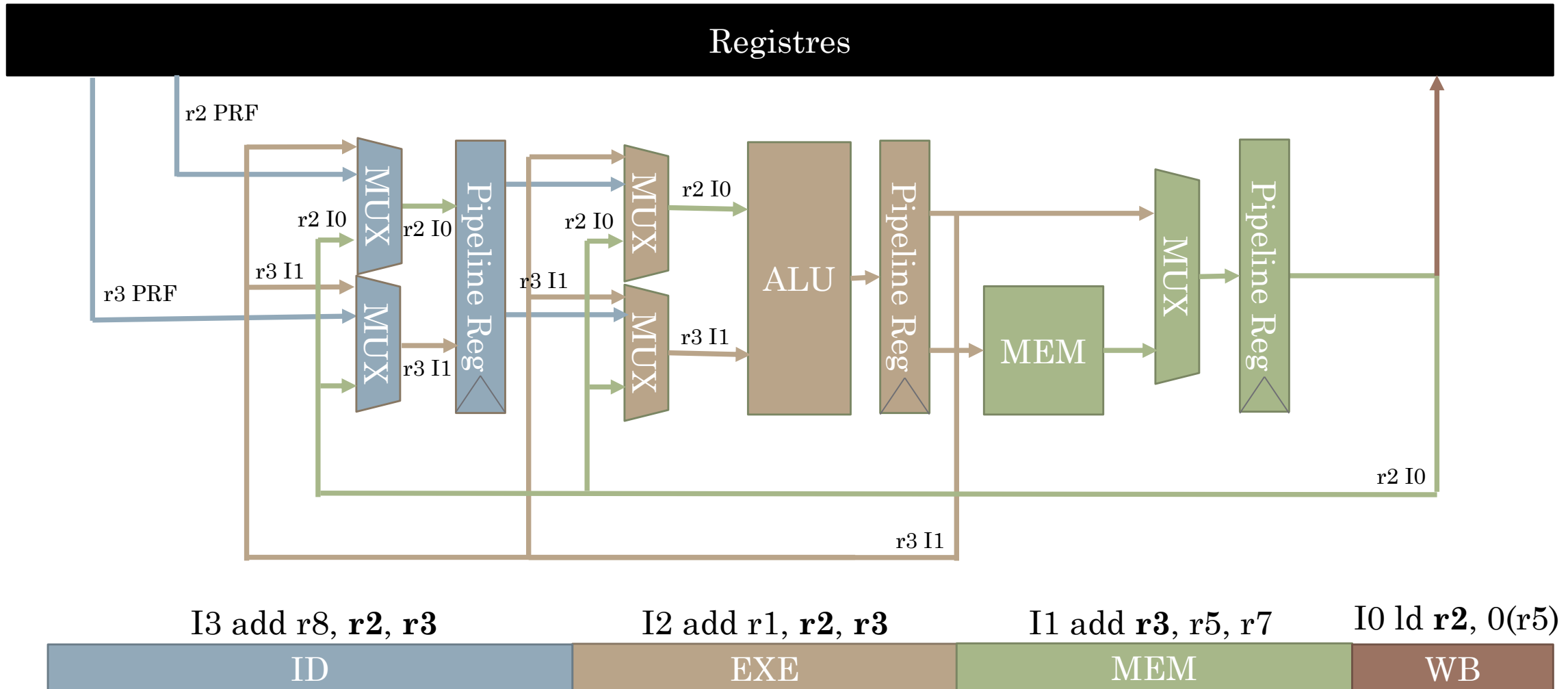
# Dépendances de données – Réseau de bypass

- Cas 5&6 (ld + consommateur avec un ou deux cycles d'écart) :  
couverts par les chemins existant



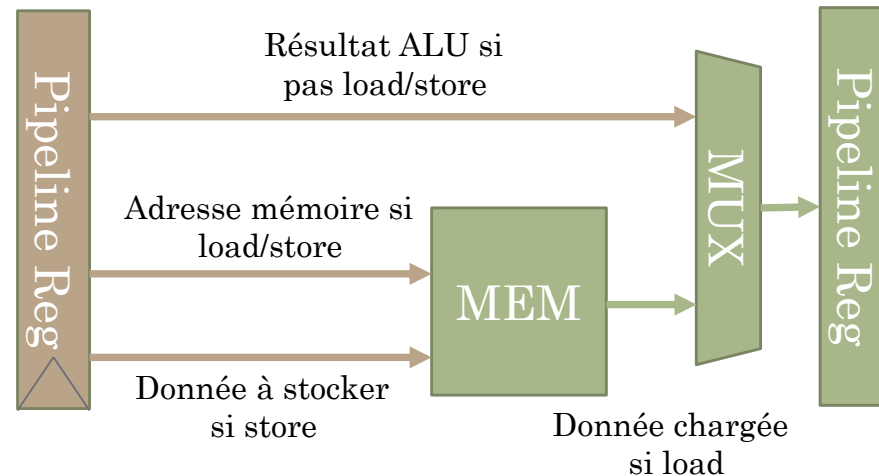
# Dépendances de données – Réseau de bypass

- Cas 5&6 (ld + consommateur avec un ou deux cycles d'écart) : couverts par les chemins existant



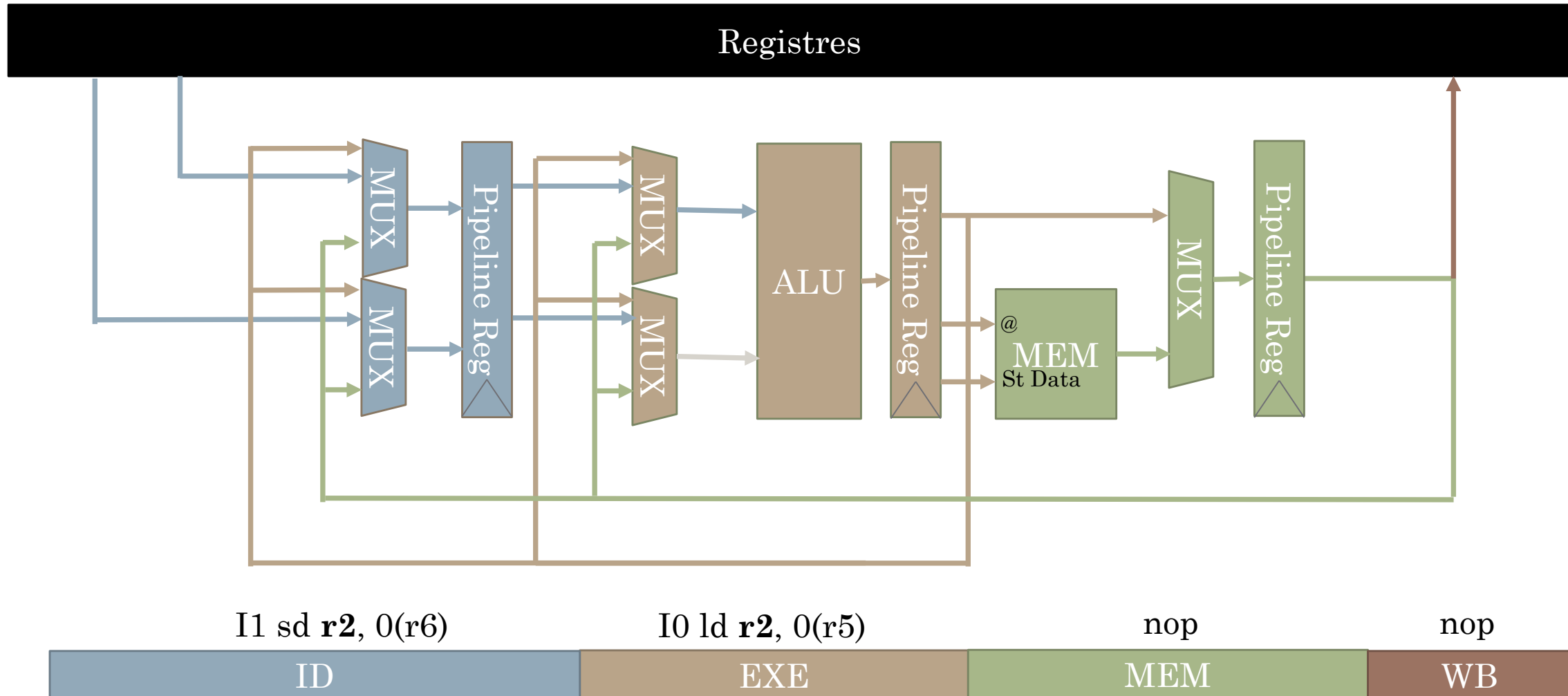
# Dépendances de données – Réseau de bypass

- Trois bulles à cacher via le bypass, mais seulement deux étages lisent le bypass : ID et EXE
- MEM n'aurait pas besoin de lire le bypass ? Quelles sources pour MEM :



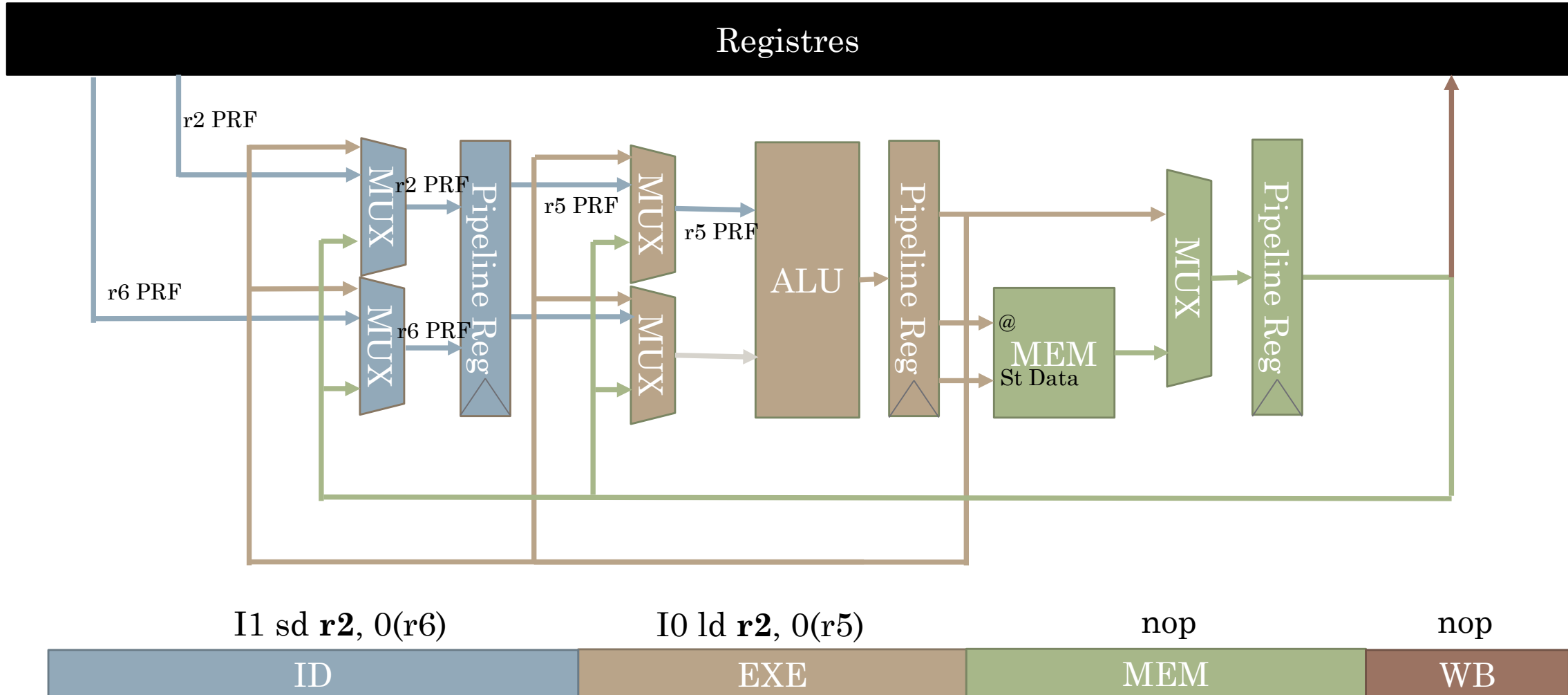
# Dépendances de données – Réseau de bypass

- Cas 7 : Load (producteur) puis store consommateur pour **donnée**, dos à dos



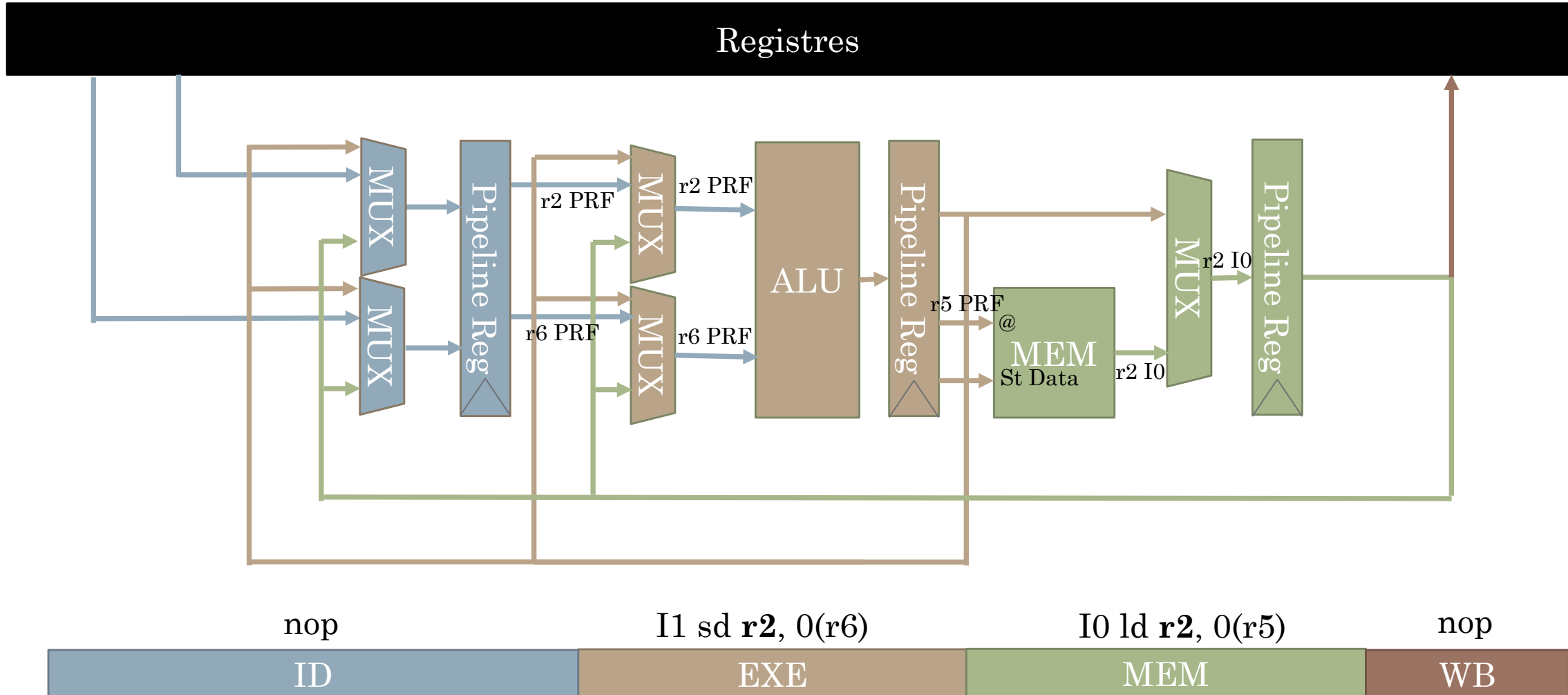
# Dépendances de données – Réseau de bypass

- Cas 7 : Load (producteur) puis store consommateur pour **donnée**, dos à dos



# Dépendances de données – Réseau de bypass

- Cas 7 : Load (producteur) puis store consommateur pour **donnée**, dos à dos







# Dépendances de données – Réseau de bypass

- Nouveau chemin WB vers MEM uniquement pour gagner un cycle sur certaines suites d'instructions
  - Utile ?

memcpy:

ld r2, 0(r1)

sd r2, 0(r3)

addi r1, 8

addi r3, 8

memcpy:

ld r2, 0(r1)

addi r1, 8

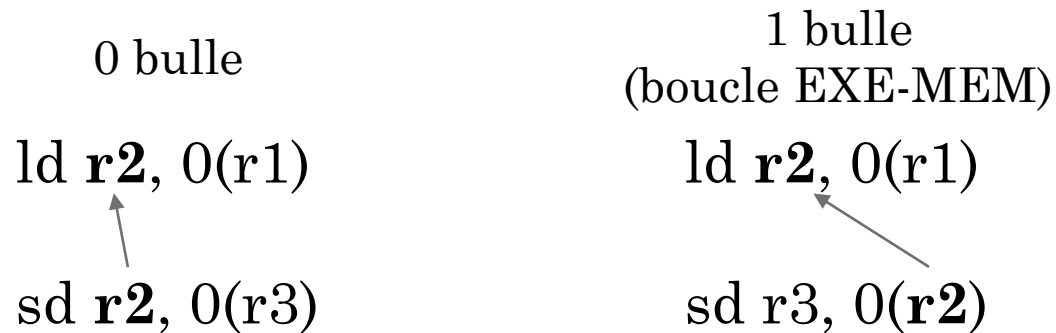
sd r2, 0(r3)

addi r3, 8

Combien de cycles **avec** et **sans** le nouveau chemin de bypass ?

# Dépendances de données – Réseau de bypass

- Nouveau chemin WB vers MEM uniquement pour gagner un cycle sur certaines suites d'instructions
- Ne fonctionne que si le store consomme le résultat du load comme registre de **donnée**
- Si consomme comme registre d'adresse, calcul d'adresse requis (fait par EXE, donc bulle d'après la boucle EXE-MEM ?)

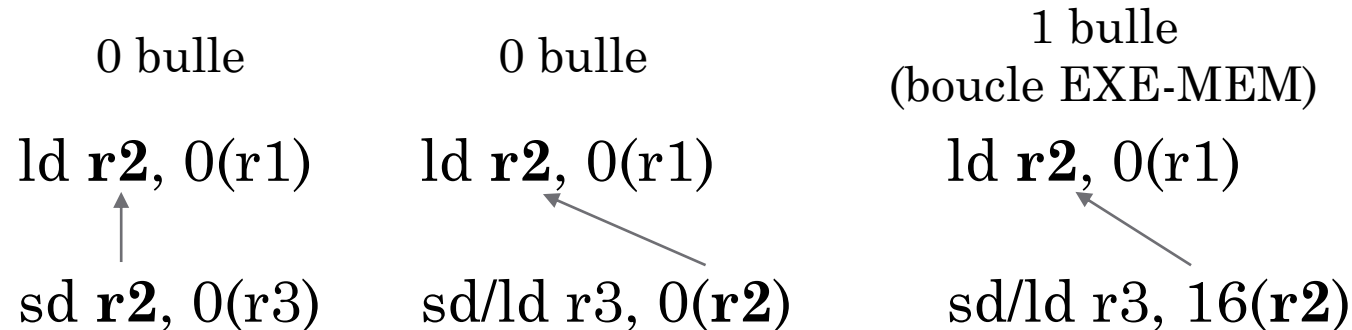


# Dépendances de données – Réseau de bypass

- Une idée d'optimisation microarchitecturale tirant partie de l'architecture ?
  - Par exemple dans RISC-V qui ne propose que  $\text{reg} + \text{imm}$  pour le calcul d'adresse...

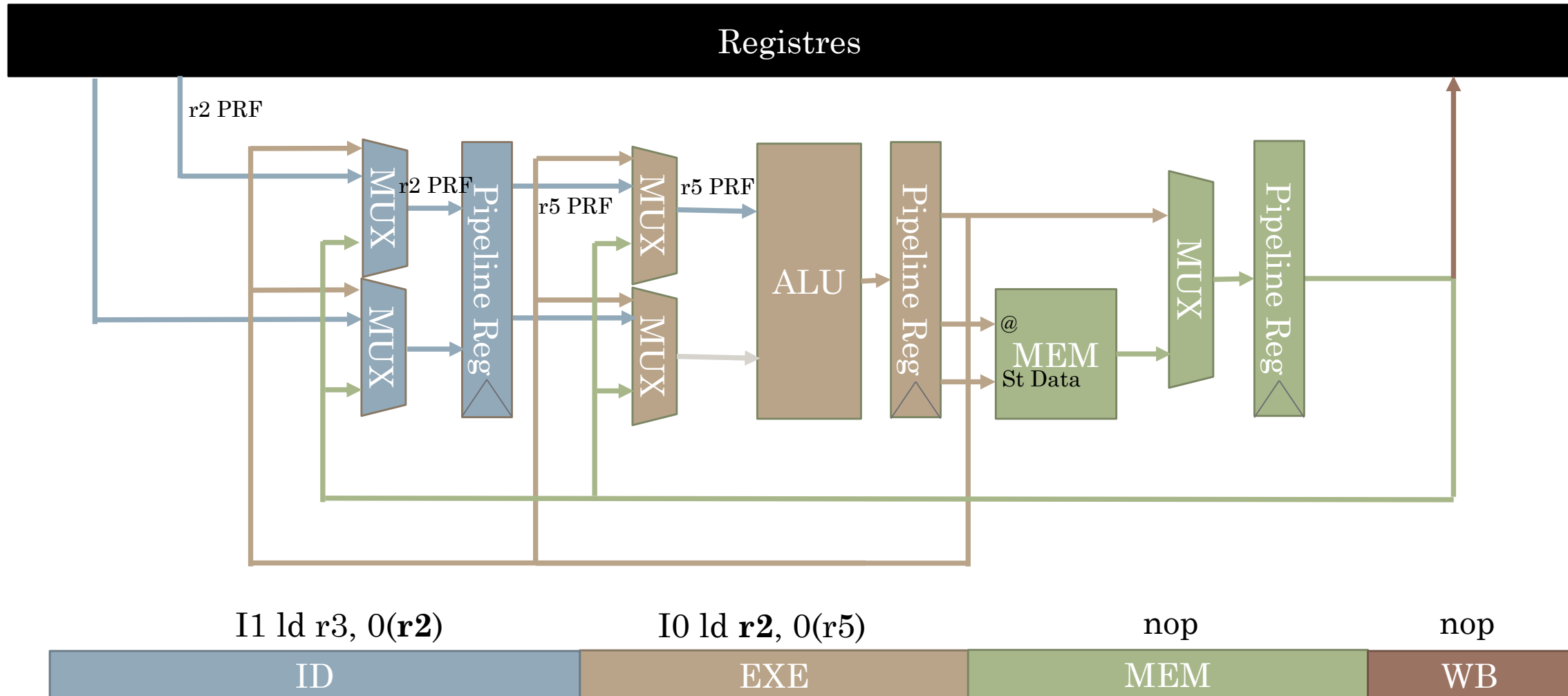
# Dépendances de données – Réseau de bypass

- Une idée d'optimisation microarchitecturale tirant partie de l'architecture ?
  - Par exemple dans RISC-V qui ne propose que `reg + imm` pour le calcul d'adresse...
- Si `imm` vaut 0, pas besoin d'addition pour calculer l'adresse, on peut utiliser le registre source directement
  - Déréférencement de pointeur sans bulle



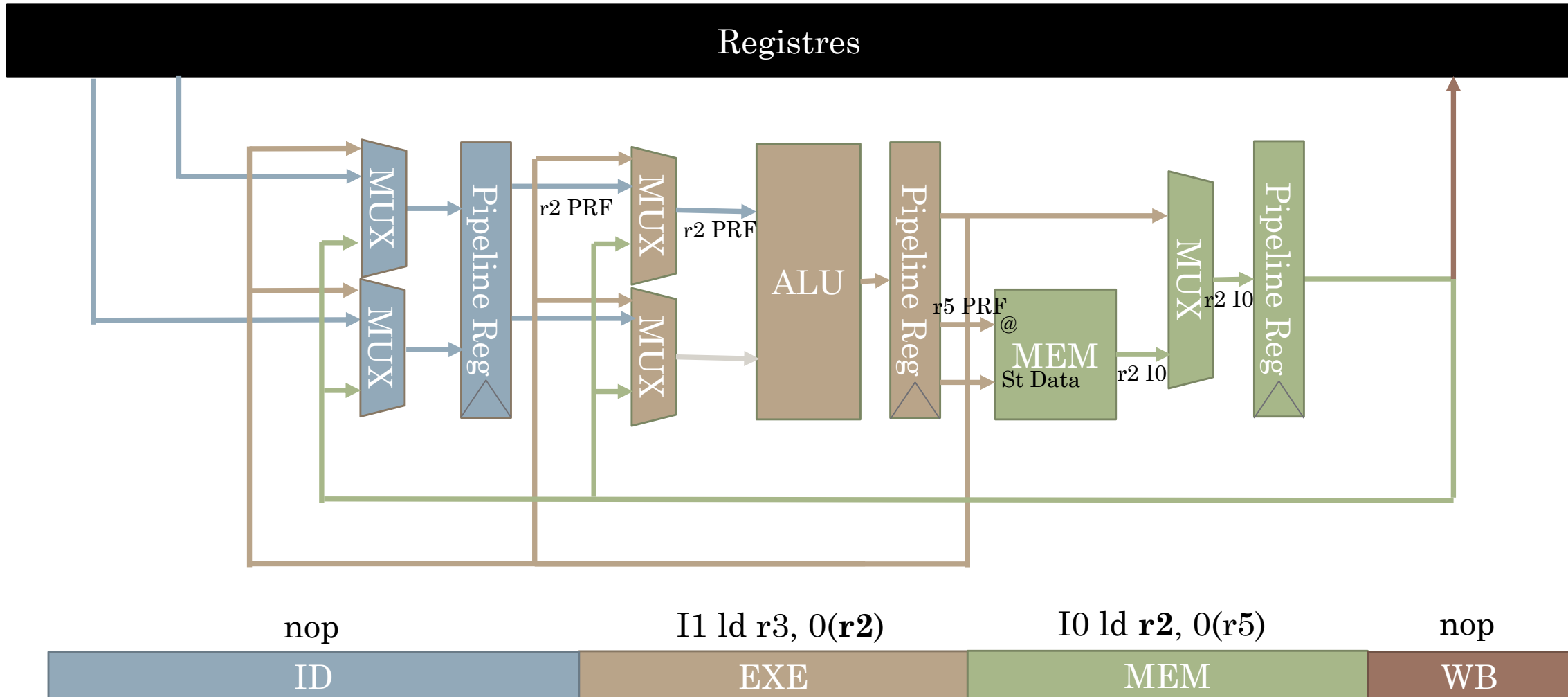
# Dépendances de données – Réseau de bypass

- Cas 8 : Load (producteur) puis load/store consommateur pour **adresse avec immédiat 0**, dos à dos



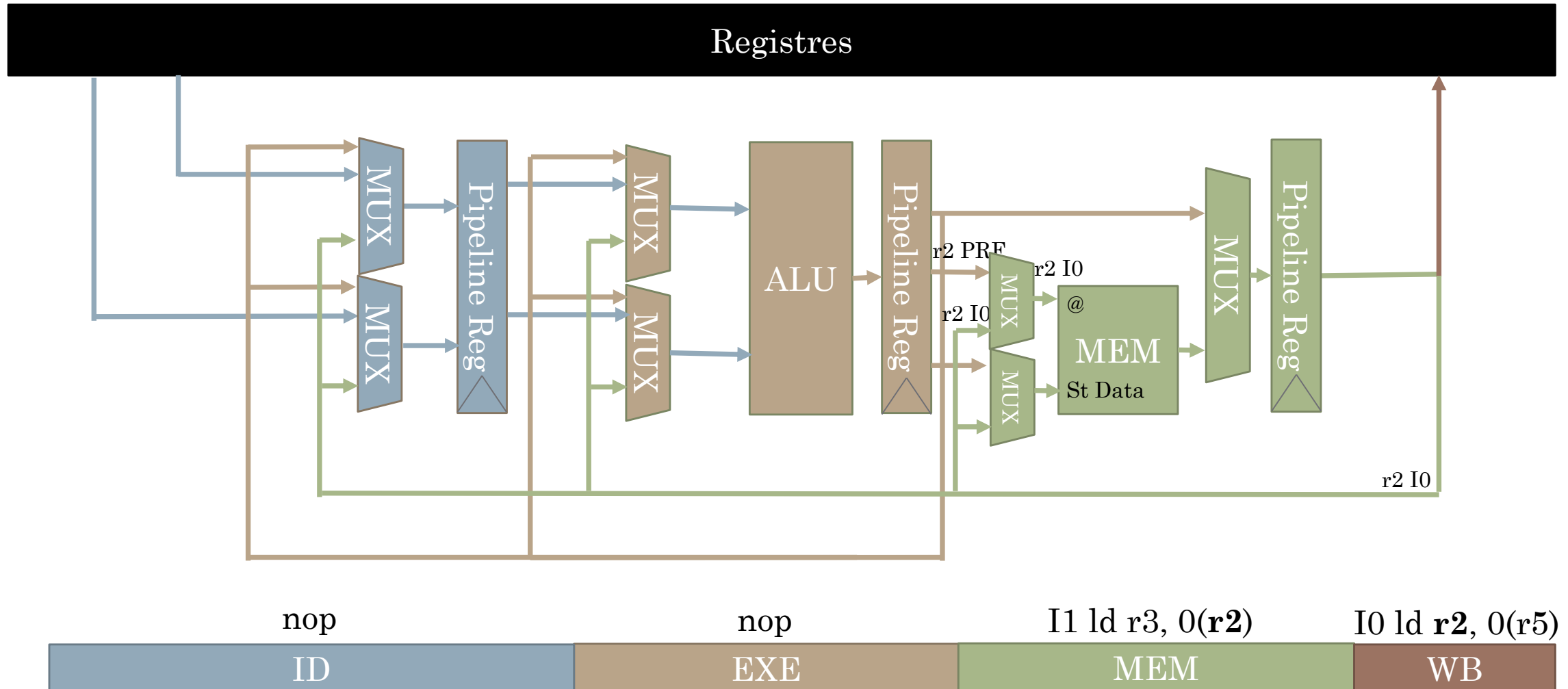
# Dépendances de données – Réseau de bypass

- Cas 8 : Load (producteur) puis load/store consommateur pour **adresse avec immédiat 0**, dos à dos



# Dépendances de données – Réseau de bypass

- Cas 8 : Load (producteur) puis load/store consommateur pour **adresse** avec **immédiat 0**, dos à dos





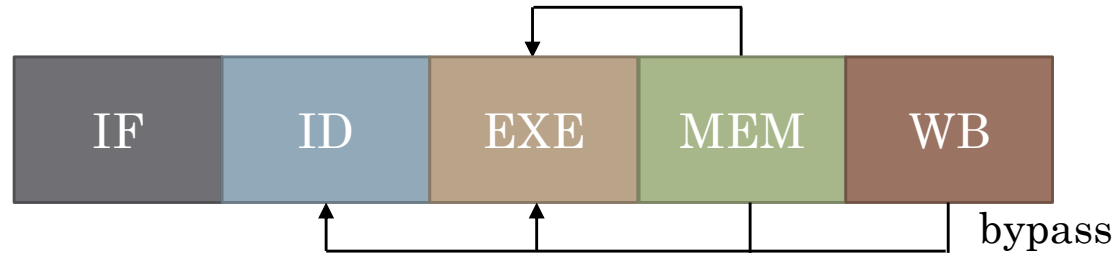
# Dépendances de données – Réseau de bypass

- Dépendances structurelles causées par les dépendances de données réduites en ajoutant des chemins de bypass

Sans bypass: 3 bulles  
entre prod et cons



Avec bypass 1<sup>ère</sup> version : 1 bulle  
entre prod et cons si prod est un  
load



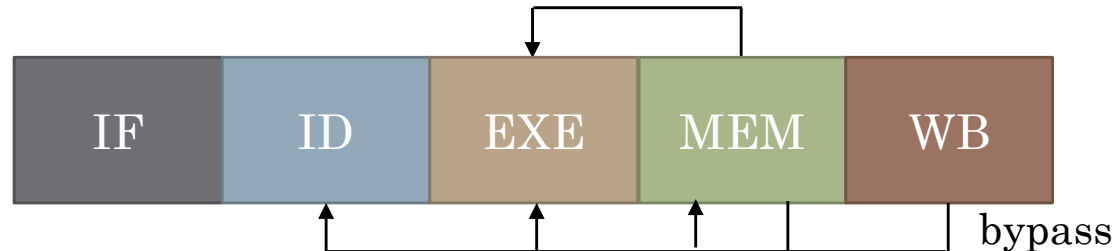
Avec bypass 2<sup>ème</sup> version : 1 bulle  
entre prod et cons si  
prod est un load

ET

(cons n'est pas un load/store

OU

cons est un load/store qui utilise le  
résultat de prod pour faire un calcul  
d'adresse)



# CPU minimaliste

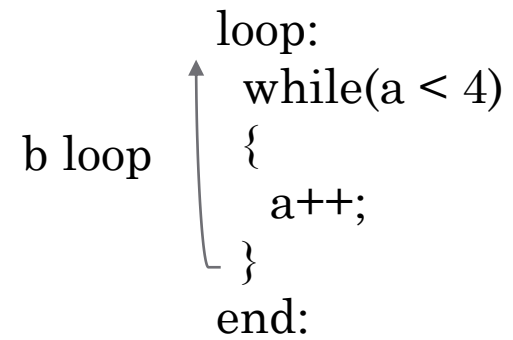
- Ce pipeline est une *implémentation* matérielle possible
  - Invisible du logiciel -> microarchitecture
  - Pas la seule implémentation possible
- Quelles sont les limites à la performance ?
  - Une instruction par cycle au maximum
  - Dépendances
    - Entre instructions dans le programme
    - Structurelles (causées par le pipeline lui-même)

# Branchements

- Rappel
  - Branchement non conditionnel : Toujours pris

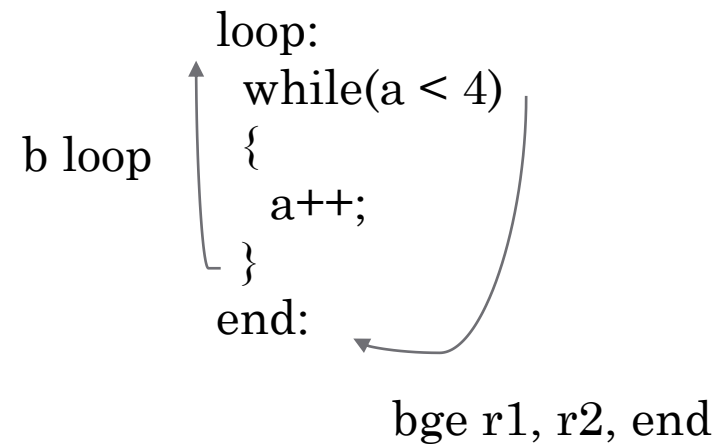
```
loop:
  while(a < 4)
  {
    a++;
  }
end:
```

b loop

A diagram illustrating a loop structure. On the left, the text "b loop" is positioned. A curved arrow originates from the "b loop" text and points upwards to the "loop:" label at the beginning of the code block on the right. The code block consists of a label "loop:", followed by a "while(a < 4)" condition, a block of code enclosed in curly braces containing "a++;", and finally an "end:" label.

# Branchements

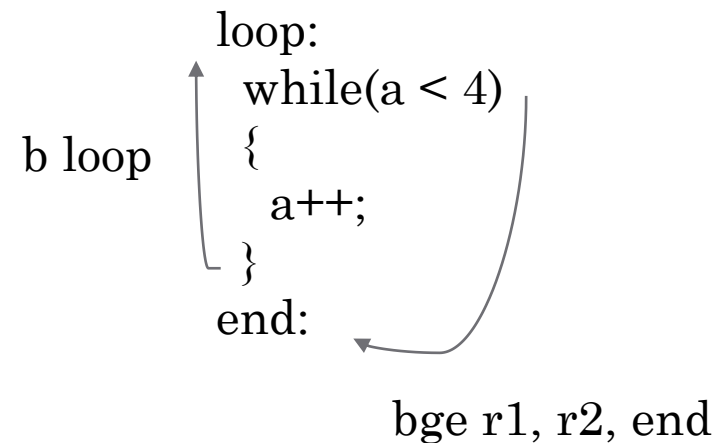
- Rappel
  - Branchement non conditionnel : Toujours pris
  - Branchement conditionnel : Pris si condition vraie



# Branchements

- Rappel

- Branchement non conditionnel : Toujours pris
- Branchement conditionnel : Pris si condition vraie
- Branchement direct : Cible est PC +/- déplacement fixe (encodé dans l'instruction)

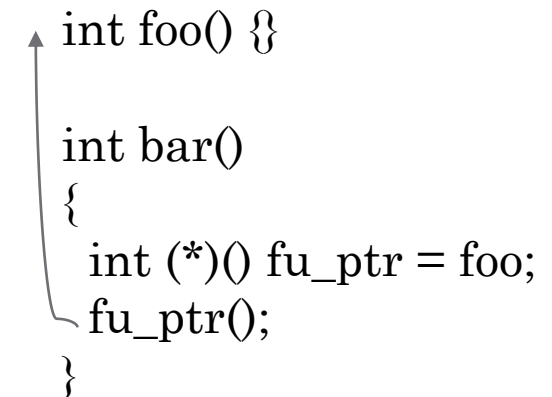


# Branchements

- Rappel

- Branchement non conditionnel : Toujours pris
- Branchement conditionnel : Pris si condition vraie
- Branchement direct : Cible est PC +/- déplacement fixe (encodé dans l'instruction)
- Branchement indirect : Cible est dans un registre

jalr r1



The diagram illustrates an indirect branch instruction. On the left, the instruction `jalr r1` is shown. A curved arrow originates from the right side of this instruction and points to the opening curly brace of the `foo` function definition on the right. The `foo` function is defined as `int foo() { int bar() { int (*)() fu_ptr = foo; fu_ptr(); } }`.

```
int foo() {  
    int bar()  
    {  
        int (*)() fu_ptr = foo;  
        fu_ptr();  
    }  
}
```

# Dépendances de contrôle – Arch.

- Dépendances causées par une divergence dans le programme

**before\_if:**

if(a != 0)

{

  a = 4;

}

**after\_if:**

a = a + 2;



**before\_if:**

I0 : beqz r1, after\_if

I1 : lui r1, 4

**after\_if:**

I2 : addi r1, r1, 2

Dépendance de contrôle au niveau *architectural*

L'exécution de I0 détermine si on exécute I1 ou I2

# Dépendances de contrôle – $\mu$ arch.

- Comment la dépendance de contrôle au niveau architectural est-elle traduite dans le pipeline ( $\mu$ arch) ?

I0 : beqz r1, after\_if

↓

I1 : lui r1, 4

I2 : addi r1, r1, 2



?



# Dépendances de contrôle – $\mu$ arch.

- Comment la dépendance de contrôle au niveau architectural est-elle traduite dans le pipeline ( $\mu$ arch) ?

I0 : beqz r1, after\_if

↓

I1 : lui r1, 4

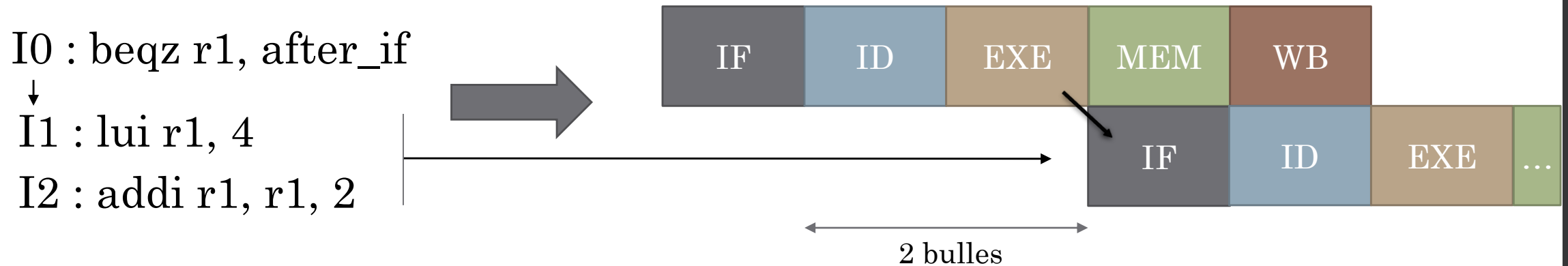
I2 : addi r1, r1, 2



Architectural	Microarchitectural
L'exécution de I0 détermine si on exécute I1 ou I2	L'instruction à récupérer dans IF est connue lorsque I0 quitte EXE = I1 ou I2 entre IF au minimum <b>3 cycles après I0</b>

# Dépendances de contrôle – $\mu$ arch.

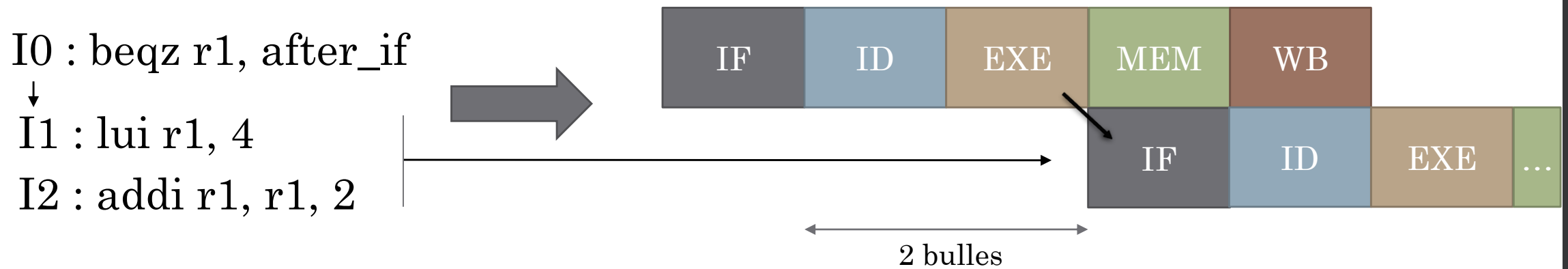
- Comment la dépendance de contrôle au niveau architectural est-elle traduite dans le pipeline ( $\mu$ arch) ?



Architectural	Microarchitectural
L'exécution de I0 détermine si on exécute I1 ou I2	L'instruction à récupérer dans IF est connue lorsque I0 quitte EXE = I1 ou I2 entre IF au minimum <b>3 cycles après I0</b>

# Dépendances de contrôle – $\mu$ arch.

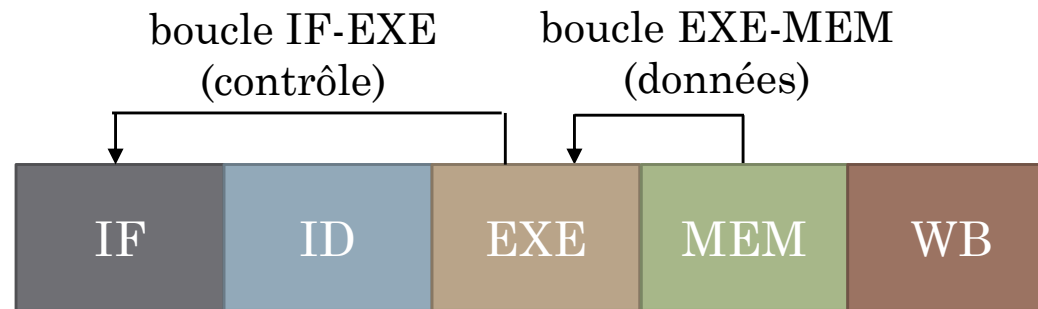
- Comment la dépendance de contrôle au niveau architectural est-elle traduite dans le pipeline ( $\mu$ arch) ?



La structure du pipeline empire la dépendance de donnée via une dépendance **structurelle** = qui provient des limites de l'implémentation et non de la sémantique des instructions

# Dépendances de contrôle – $\mu$ arch.

- 2 bulles pour chaque branchement conditionnel
  - En moyenne un branchement conditionnel toutes les 8 à 10 instructions = max 0,83 IPC soit 83% de la performance crête

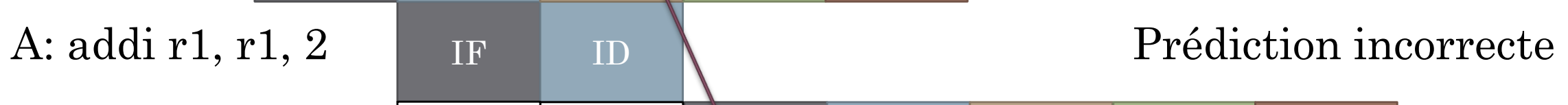
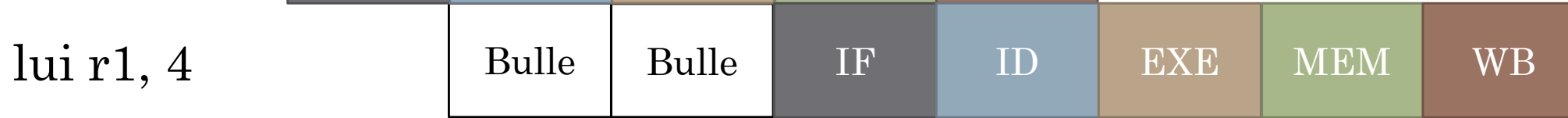


- Pourtant, le résultat du branchement est consommé « au plus tôt »
  - Une idée pour faire mieux ?

# Dépendances de contrôle – Exécution spéculative

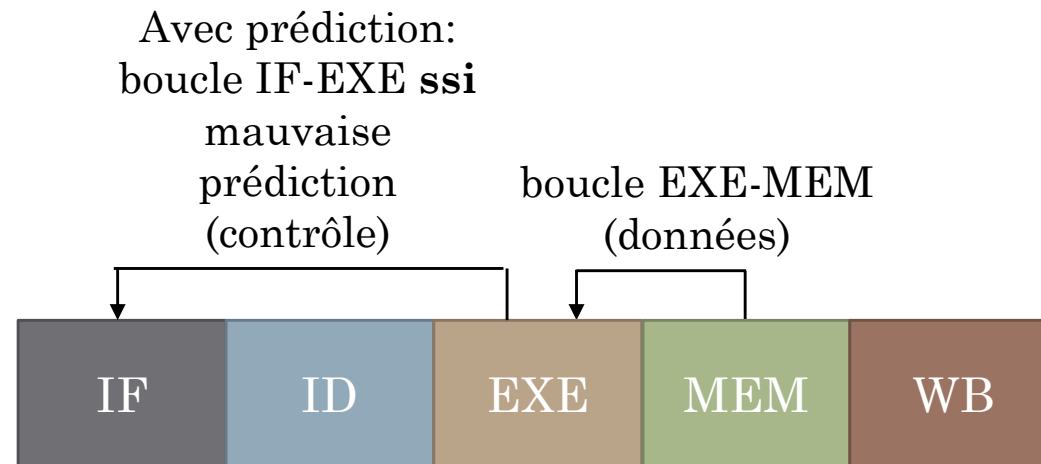
- La direction du branchement n'est pas calculée assez tôt
  - Plutôt qu'attendre, tenter de deviner la direction :
- **Prédiction de branchement**
  - Permet de connaître la direction du branchement (prédite) quand le branchement est dans IF
  - Très efficace car les programmes sont réguliers

# Dépendances de contrôle – Exécution spéculative



# Dépendances de contrôle – Exécution spéculative

- 2 bulles pour chaque branchement conditionnel
  - En moyenne un branchement conditionnel toutes les 8 à 10 instructions = max 0,83 IPC soit 83% de la performance crête



- 90% de prédictions correctes = max 0,98 IPC (98% crête)
- 99% de prédictions correctes = max 0,998 IPC (99,8% crête)

# Dépendances de contrôle – Exécution spéculative

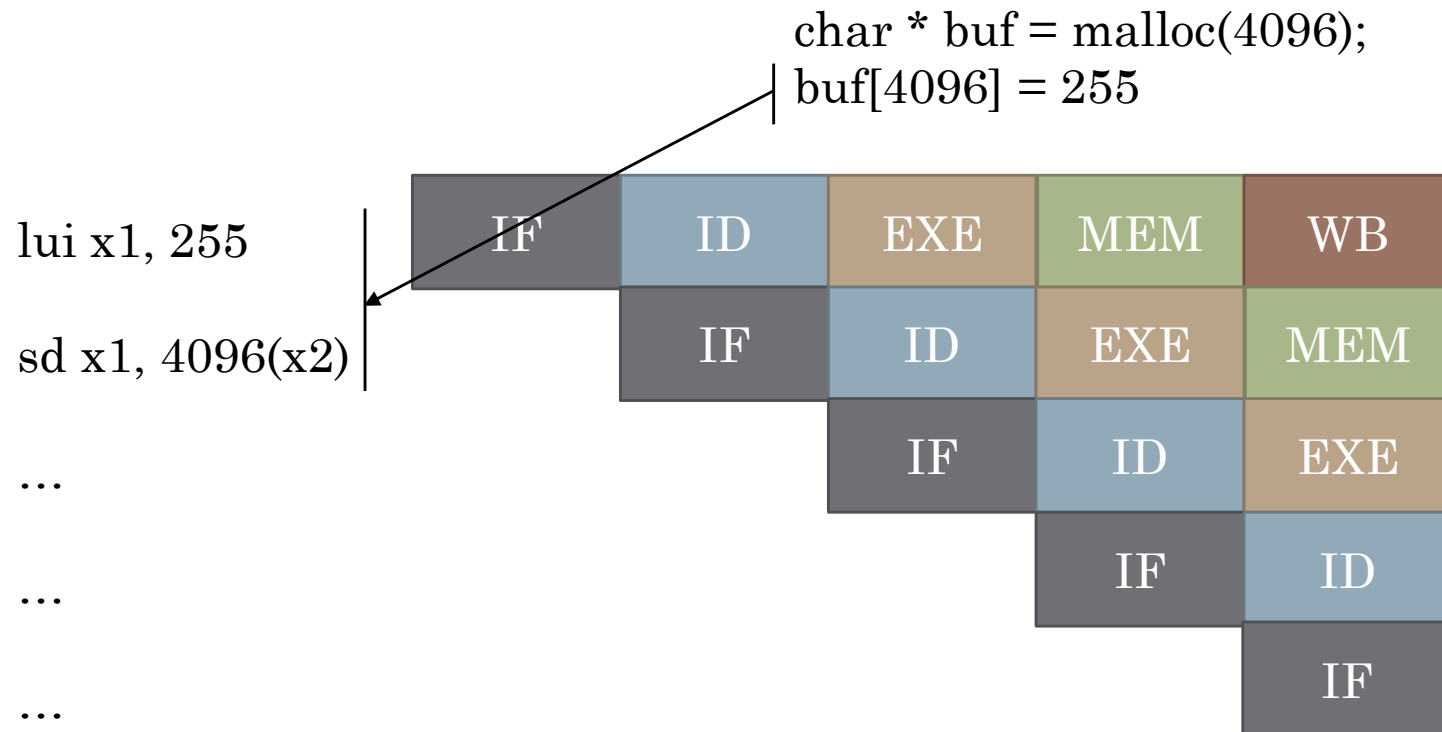
- Même sans prédiction de branchement, on fait déjà de l'exécution spéculative : **Exceptions**
  - Par exemple, erreur de segmentation quand on accède à une case mémoire non allouée

```
char * buf = malloc(4096);  
buf[4096] = 255
```



# Dépendances de contrôle – Exécution spéculative

- Même sans prédiction de branchement, on fait déjà de l'exécution spéculative : **Exceptions**
  - Par exemple, erreur de segmentation quand on accède à une case mémoire non allouée



# Dépendances de contrôle – Exécution spéculative

- On vide le début du pipeline et on saute vers le gestionnaire d'exception
  - Comme une mauvaise prédiction de branchement

```
char * buf = malloc(4096);  
buf[4096] = 255
```



# Dépendances de contrôle – Exécution spéculative

- Même sans prédiction de branchement, on fait déjà de l'exécution spéculative : **Exceptions**
  - Par exemple, erreur de segmentation quand on accède à une case mémoire non allouée
  - On vide le pipeline et on saute vers le gestionnaire d'exception
- Pourtant, on ne bloque pas le pipeline en attendant de savoir si un load/store va causer une exception
  - On *spécule* qu'il n'y aura pas d'exception (raisonnable car exception rare)

# Prédiction de Branchement Statique

- Naïve :
  - Toujours prédire « pris » : 34% de mauvaises prédictions dans la suite de benchmarks SPEC CPU 2000
- Heuristiques basés sur comment sont construits les programmes :
  - Exemple : Si un branchement saute vers une adresse plus faible, c'est probablement le saut vers le début du corps de boucle, donc il est probablement pris
- « *Educated guess* » :
  - Le programmeur ou compilateur insère un indice (instruction machine dédiée)
  - Profiling du programme à l'exécution puis recompiler une version optimisée utilisant ces indices
- Assez peu précis car **binaire** : On exprime seulement « plutôt pris » et « plutôt pas pris »

# Prédiction de Branchement Dynamique

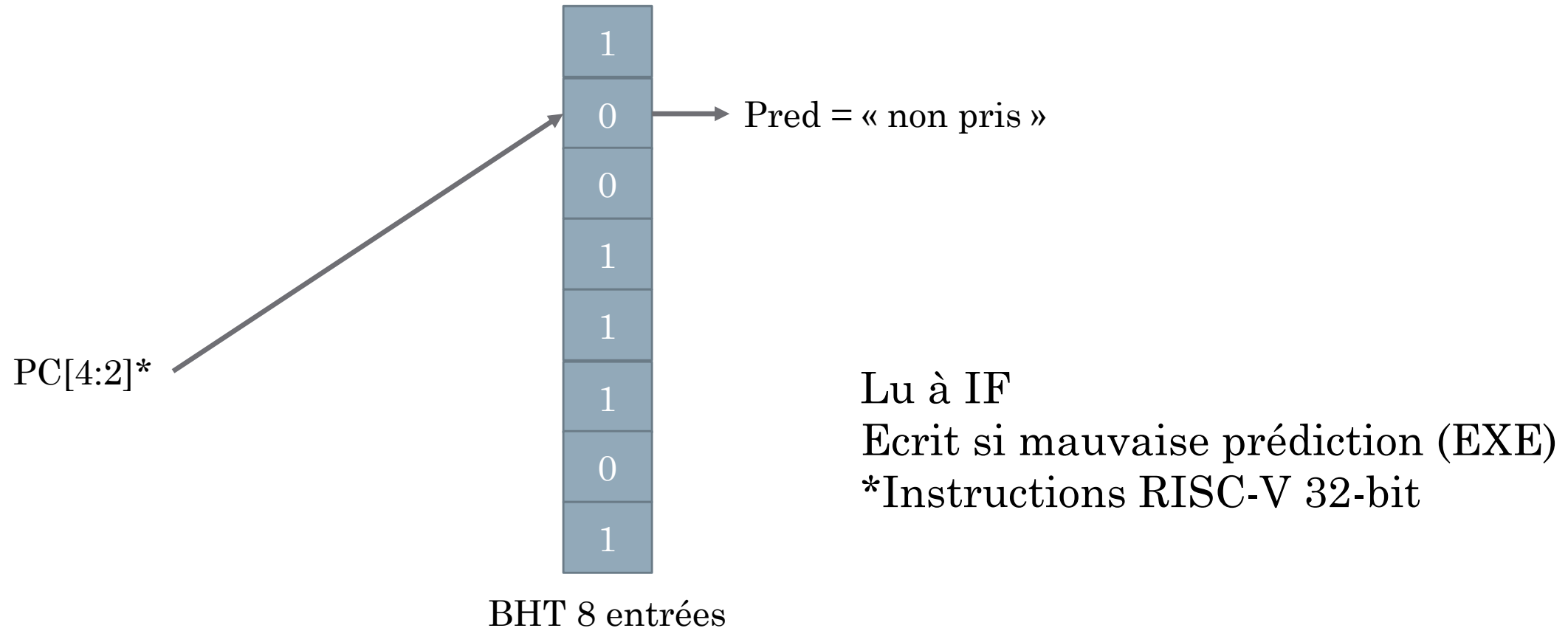
- Structure matérielle implémentant un algorithme spécifique

$$Pred = f(PC_{branchement}, \dots)$$

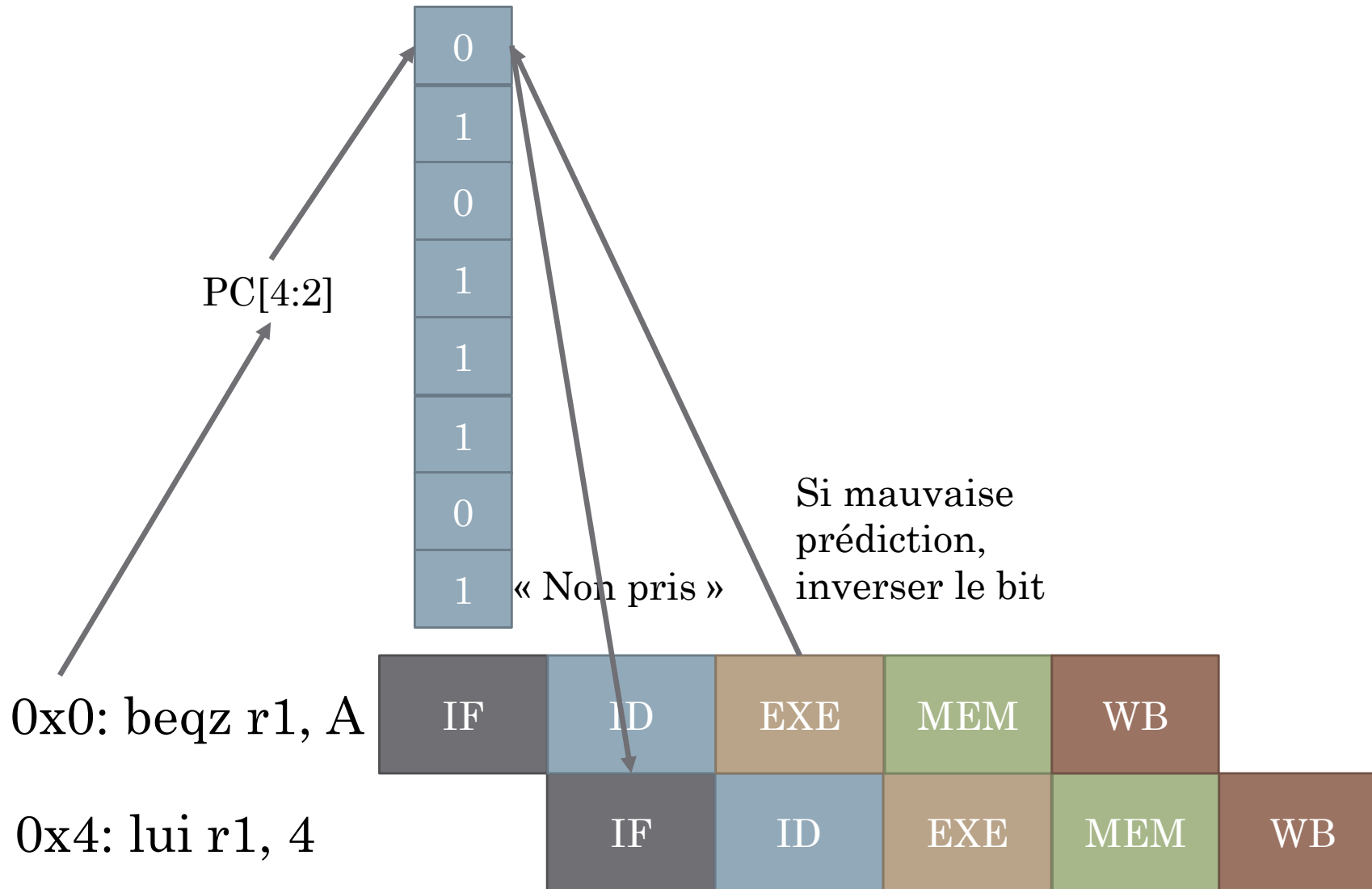
- 80-99% de bonnes prédictions
- « Règle du pouce » 10-90 (ou 20-80) : 10% des branchements du programme responsables de 90% des mauvaises prédictions de branchements

# Prédicteur simple : Branch History Table (BHT)

- Le passé donne souvent une bonne idée du futur
- Prédiction = direction prise par l'instance précédente du branchement



# Prédicteur simple : Branch History Table (BHT)



# Prédicteur simple : Performance

- Hypothèse : Bits du BHT initialisés à 0

```
for(int i = 0; i < 1024; i++) {  
    int j = 0;  
    do { j++; } while(j < 4);  
}
```

Comportement : 1110 1110 1110 (1: pris 0: non pris)

Prédictions BHT : A vos stylos

```
for(int i = 0; i < 1024; i++) {  
    if((i % 2) == 0) {...}  
}
```

Comportement : 1010 1010 1010

Prédictions BHT : A vos stylos



# Prédicteur simple : Performance

- Hypothèse : Bits du BHT initialisés à 0

```
for(int i = 0; i < 1024; i++) {  
    int j = 0;  
    do { j++; } while(j < 4);  
}
```

**50% correct**

Comportement : **1110 1110 1110** (1: pris 0: non pris)  
Prédictions BHT : **0111 0111 0111**

```
for(int i = 0; i < 1024; i++) {  
    if((i % 2) == 0) {...}  
}
```

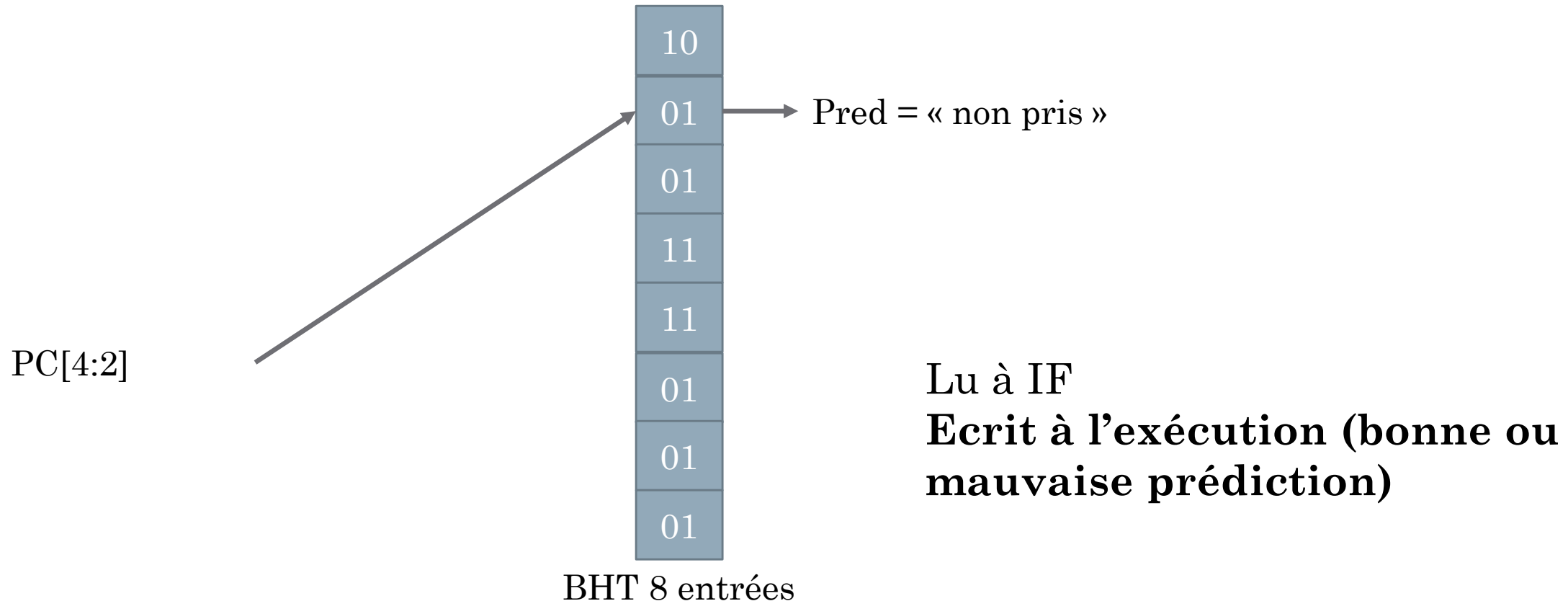
Comportement : **1010 1010 1010**  
Prédictions BHT : **0101 0101 0101**

**0% correct**

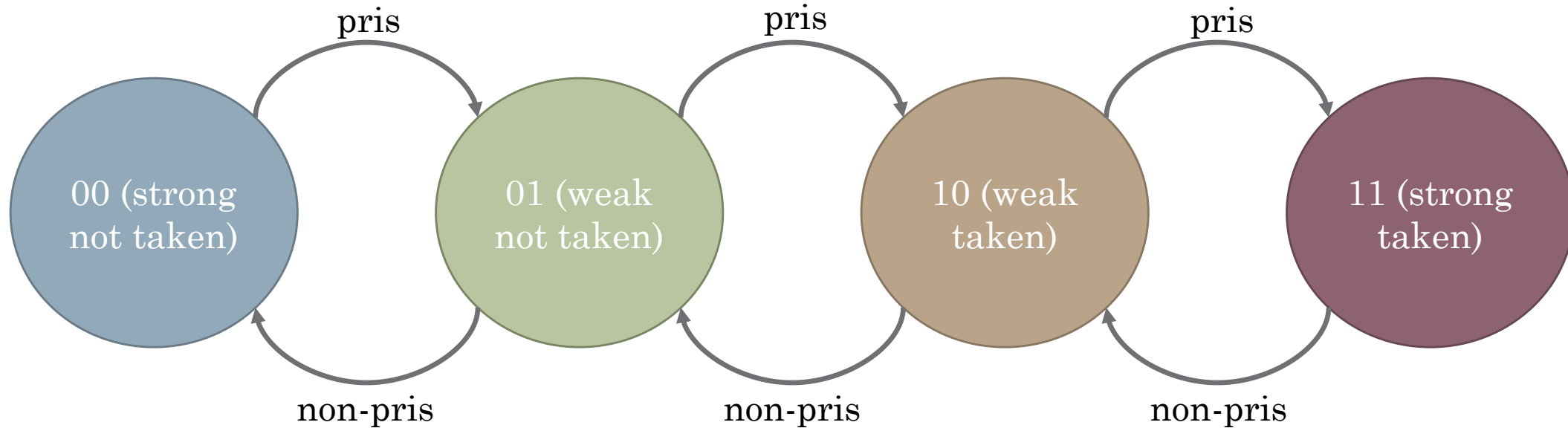
- Peut mieux faire.

# Prédicteur simple : 2-bit bimodal

- Compteur 2-bit au lieu de 1-bit
  - Bit 1 donne la direction
  - Bit 0 ajoute de l'*hysteresis* (de « l'inertie »)



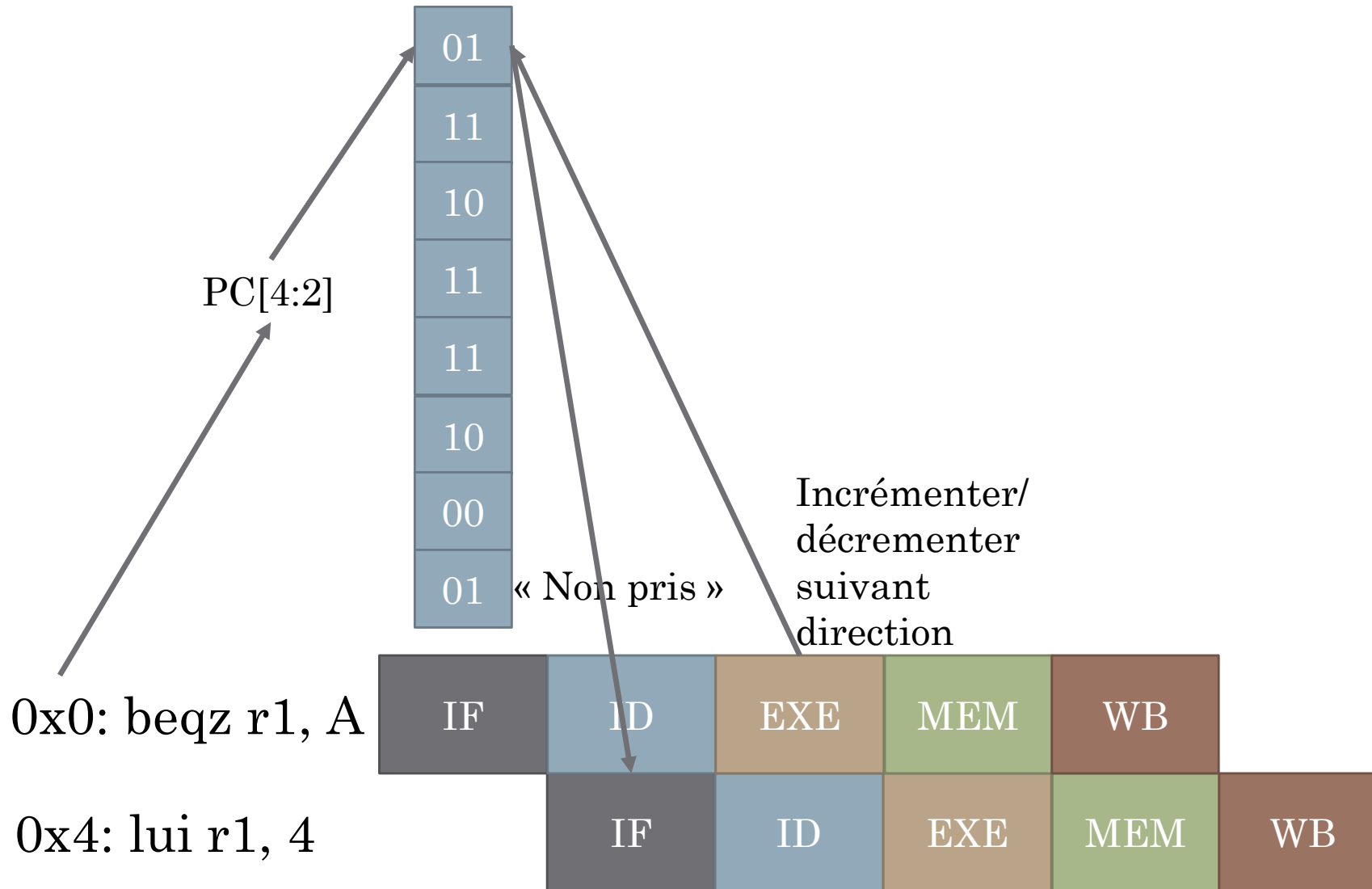
# Prédicteur simple : 2-bit bimodal



- Hysteresis utile, par exemple :

```
for(int i = 0; i < 1024; i++) {  
    int j = 0;  
    do { j++; } while(j < 64);  
}
```

# Prédicteur simple : 2-bit bimodal



# Prédicteur 2-bit : Performance

- Hypothèse : Bits du BHT initialisés à 01b

```
for(int i = 0; i < 1024; i++) {  
    int j = 0;  
    do { j++; } while(j < 4);  
}
```

Comportement : 1110 1110 1110 (1: pris 0: non pris)

Prédictions BHT : A vos stylos

```
for(int i = 0; i < 1024; i++) {  
    if((i % 2) == 0) {...}  
}
```

Comportement : 1010 1010 1010

Prédictions BHT : A vos stylos

# Prédicteur 2-bit : Performance

- Hypothèse : Bits du BHT initialisés à 01b

```
for(int i = 0; i < 1024; i++) {  
    int j = 0;  
    do { j++; } while(j < 4);  
}
```

**75% correct**

Comportement : **1110 1110 1110** (1: pris 0: non pris)

Prédictions BHT : **0111 1111 1111**

```
for(int i = 0; i < 1024; i++) {  
    if((i % 2) == 0) {...}  
}
```

**0% correct**  
**(50% si compteur**  
**était initialisé à**  
**11b)**

Comportement : **1010 1010 1010**

Prédictions BHT : **0101 0101 0101**

- Mieux ! Mais peut mieux faire.

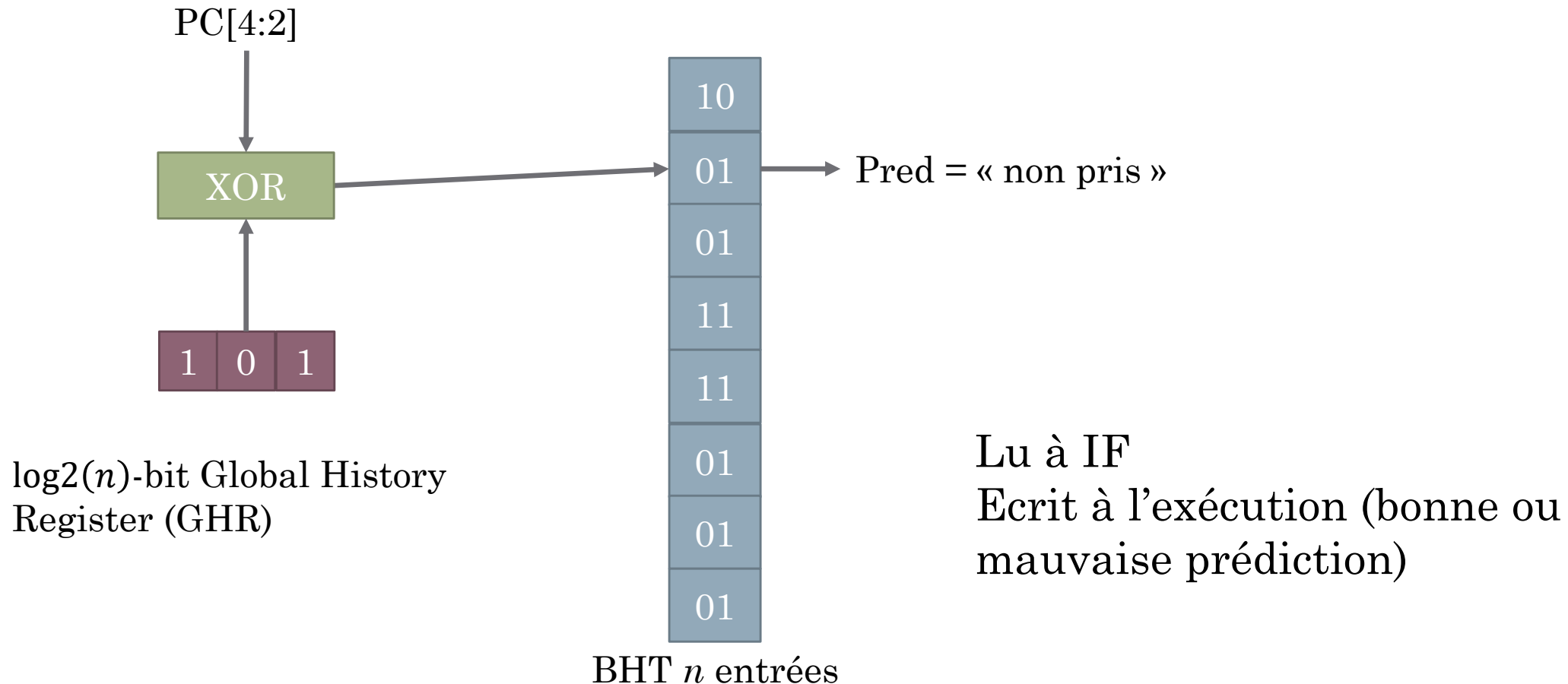
# Reconnaissance de motifs

- Notion d'**historique de branchement**
- Vecteur binaire: « 0 » = non pris, « 1 » = pris
- Historique de n-bit contient la direction des n branchements les plus récents (LSB le plus récent)
- Historique **local**
  - Un historique pour chaque branchement statique (par PC)
- Historique **global**
  - Un historique commun à tous les branchements conditionnels

$$pred = f(PC, historique)$$

# Corrélation globale : *gshare*

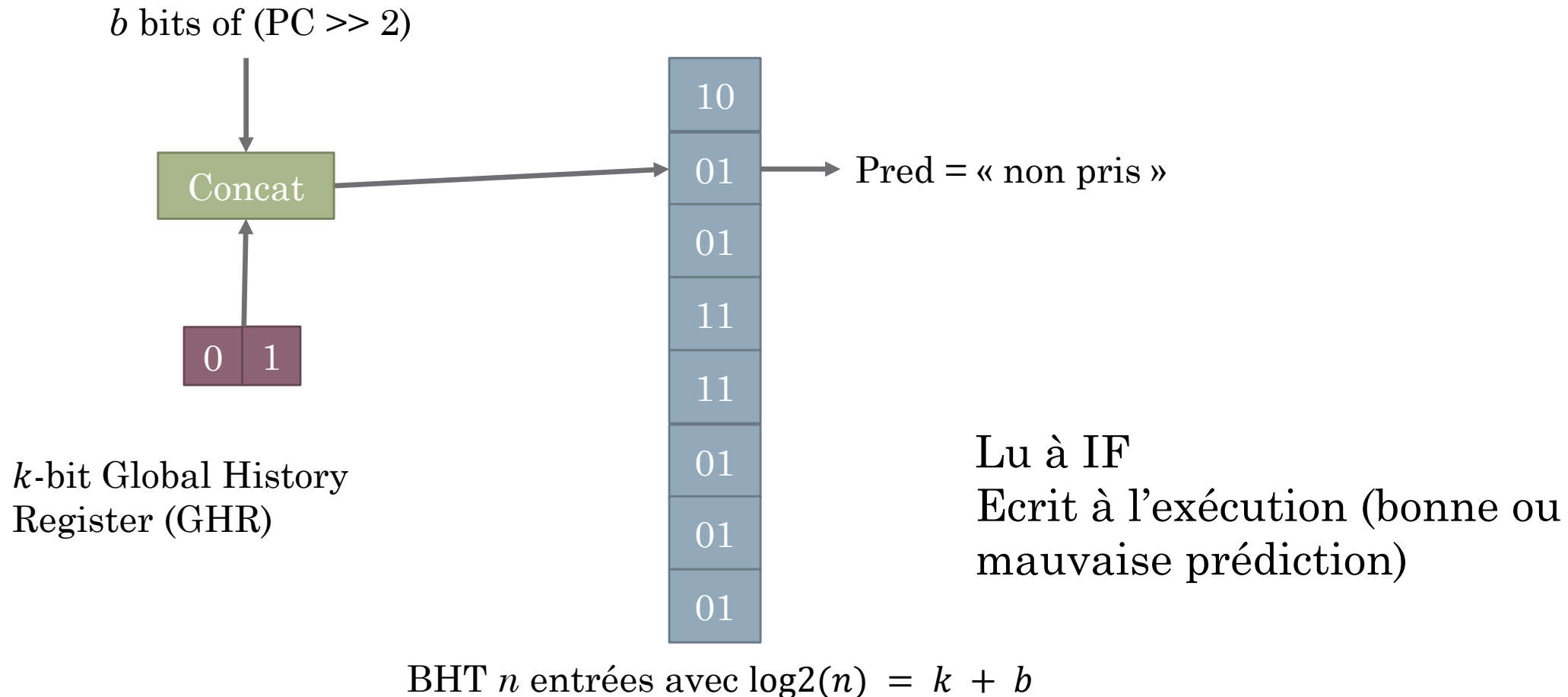
- Comme 2-bit bimodal, mais avec une fonction d'index différente





# Corrélation globale : *gselect*

- Comme 2-bit BHT, mais avec une fonction d'index différente



# Prédicteur *gshare* : Performance

- Hypothèse : Bits du BHT initialisés à 01b, 4-bit GHR (0000)

```
for(int i = 0; i < 1024; i++) {  
    int j = 0;  
    do { j++; } while(j < 4);  
}
```

Comportement : 1110 1110 1110 1110

Prédictions BHT : A vos stylos

```
for(int i = 0; i < 1024; i++) {  
    if((i % 2) == 0) {...}  
}
```

Comportement : 1010 1010 1010

Prédictions BHT : A vos stylos

- A noter, le GHR contiendra aussi le résultat du branchement du for extérieur

# Prédicteur *gshare* : Performance

- Hypothèse : Bits du BHT initialisés à 01b, 4-bit GHR (0000)

```
for(int i = 0; i < 1024; i++) {  
    int j = 0;  
    do { j++; } while(j < 4);  
}
```

**Tends vers 100%  
correct**

Comportement : 1110 1110 1110 1110  
Prédictions BHT : 0000 0000 1100 1110

```
for(int i = 0; i < 1024; i++) {  
    if((i % 2) == 0) {...}  
}
```

**Tends vers 100%  
correct**

Comportement : 1010 1010 1010  
Prédictions BHT : 0000 1010 1010

- Beaucoup mieux que bimodal

# Corrélation globale : *gshare*

- Peut discerner des corrélation entre plusieurs branchements différents
  - « Si A est pris alors B est pris »
- Seulement si les deux branchements sont dans l'historique global : Si historique trop petit, ne peut pas corrélérer des branchements éloignés
  - « Si A est pris alors B est pris » avec des branchements A – C – D – E – B durant l'exécution

D	E
0	1

2-bit GHR :  
corrélation entre A  
et B non identifiable

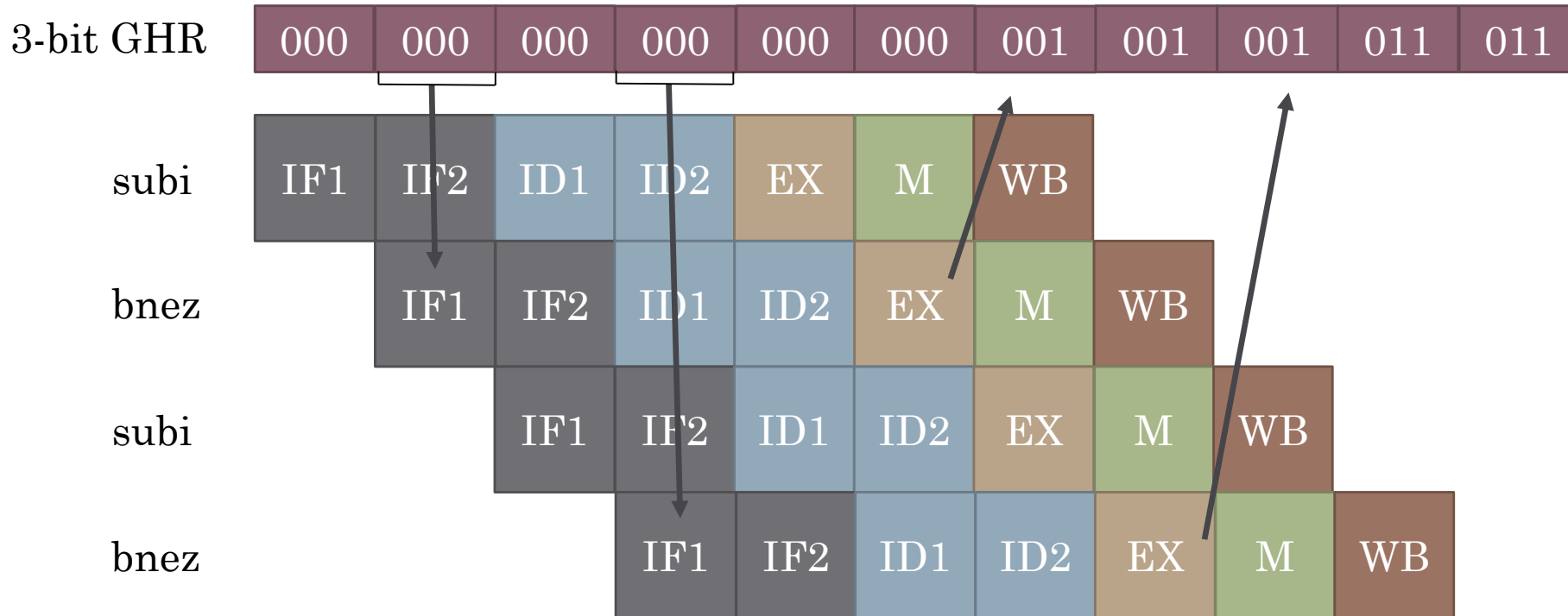
A	C	D	E
1	1	0	1

4-bit GHR :  
corrélation entre A  
et B identifiable

# Gestion de l'historique global

- Notre processeur est pipeliné, admettons 3 étages entre IF et EXE, prédiction de branchement dans IF1

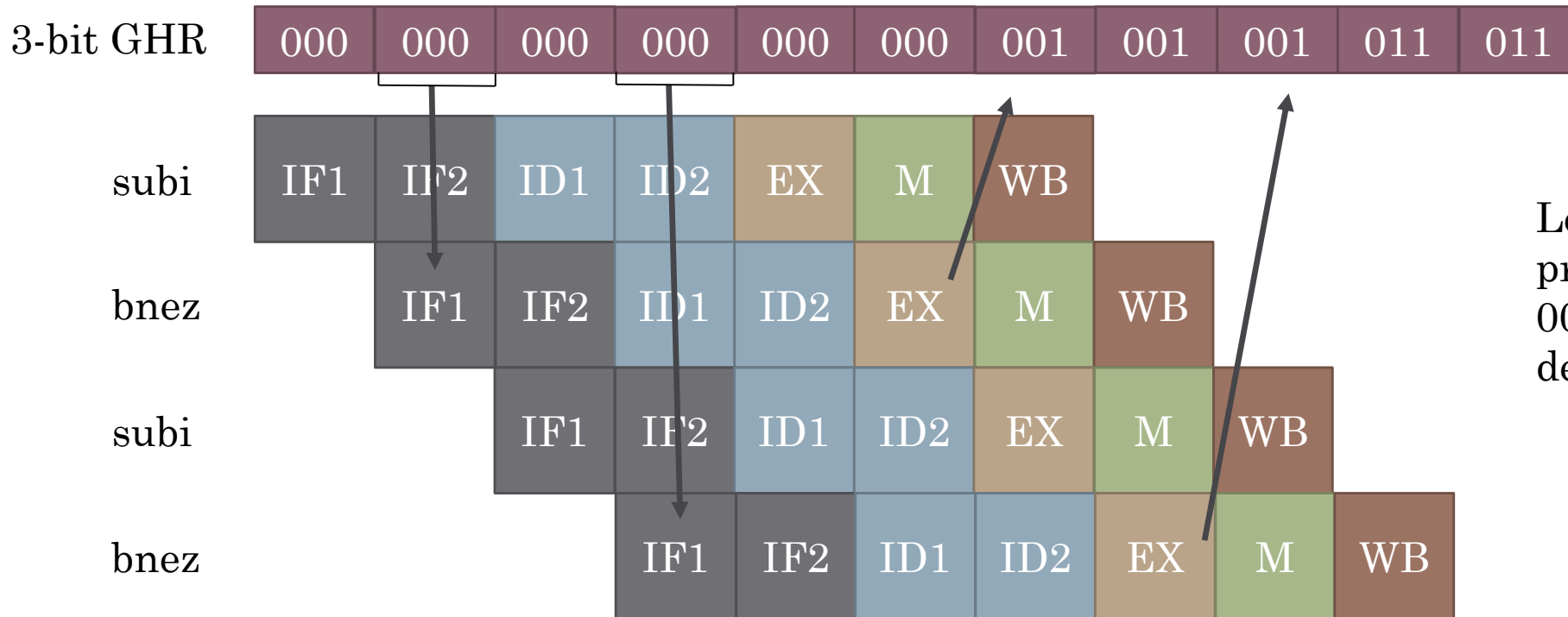
loop:  
subi r1, r1, 0x1  
bnez r1, loop



# Gestion de l'historique global

- Notre processeur est pipeliné, admettons 3 étages entre IF et EXE, prédiction de branchement dans IF1

loop:  
subi r1, r1, 0x1  
bnez r1, loop

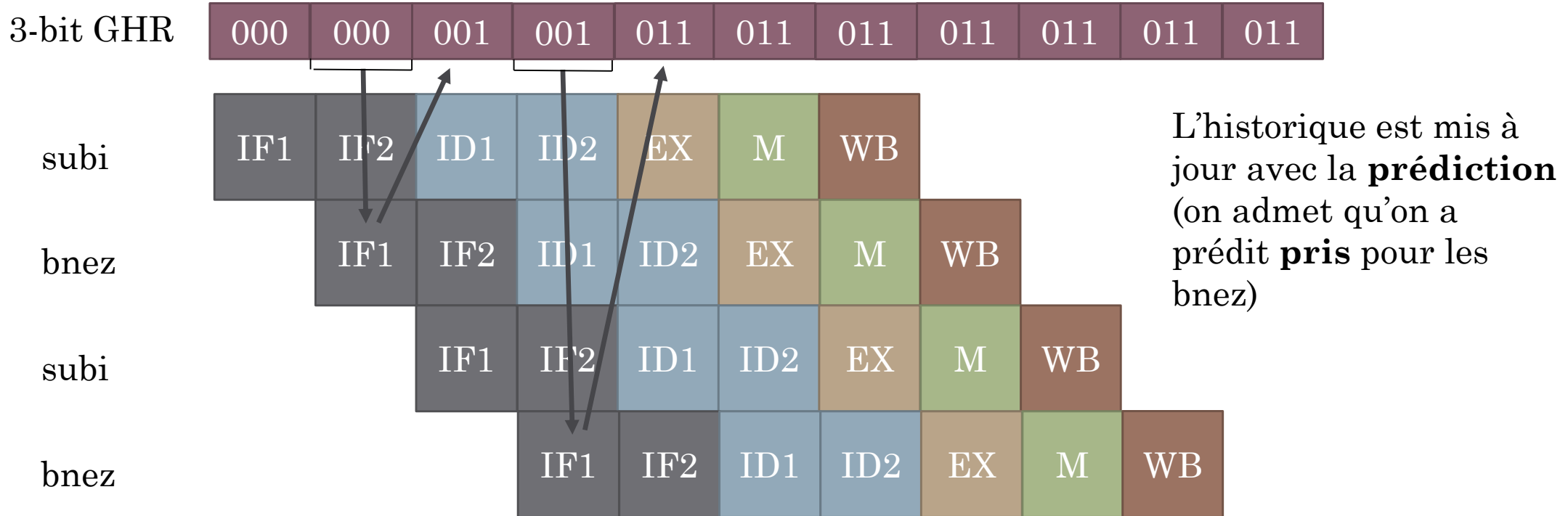


Le deuxième jnz est  
prédit avec GHR =  
000b alors que cela  
devrait être 001b !

# Gestion de l'historique global

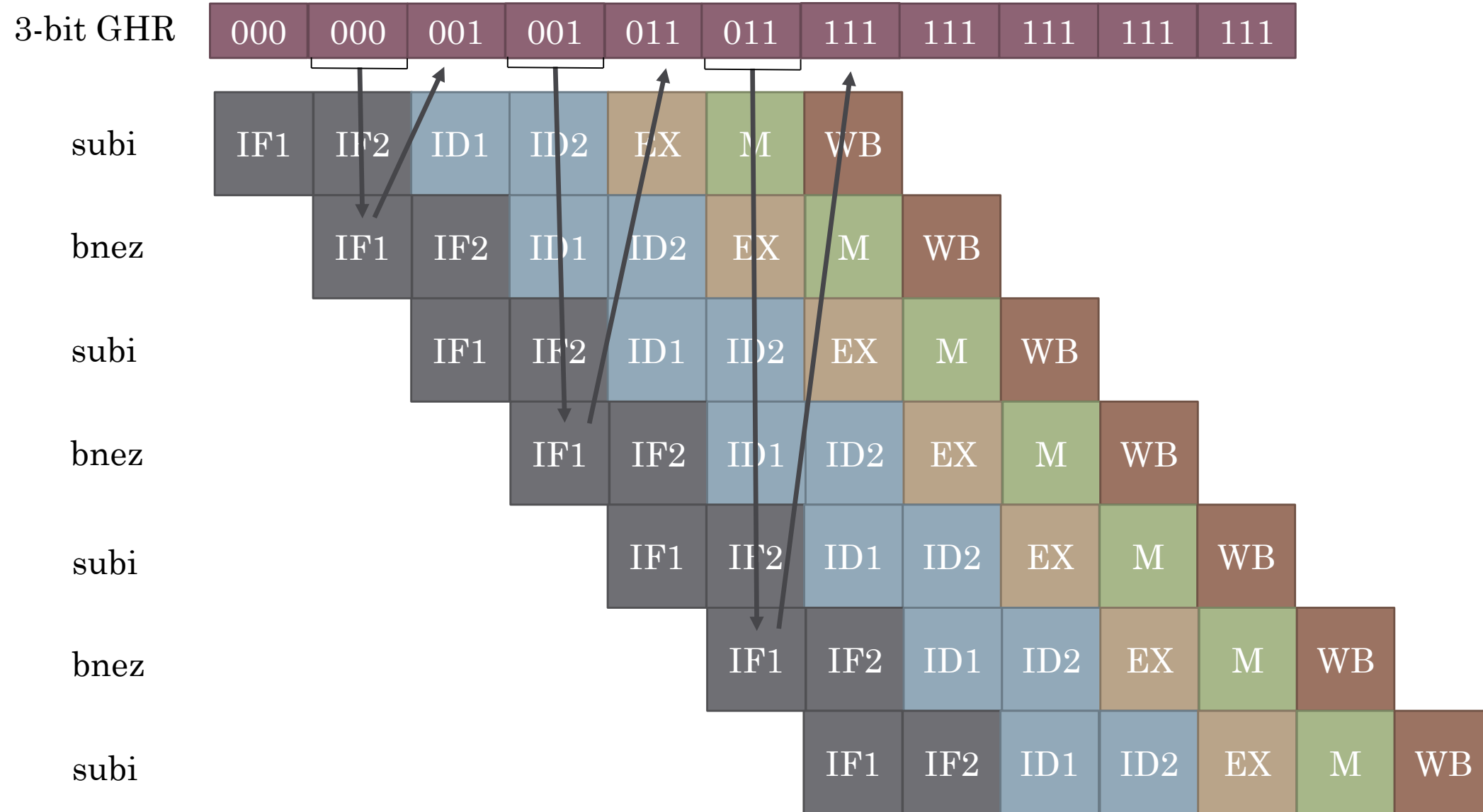
- Grave ? Oui !
- Historique et flot d'instructions désynchronisés
  - 30% de performance en moins d'après Skadron et al., « Speculative updates of local and global branch history: A quantitative analysis », JILP, 2000
- Solution : Ne pas attendre d'avoir calculé la direction du branchement avant de mettre à jour le GHR
  - Mise à jour **spéculative**

# Mise à jour spéculative



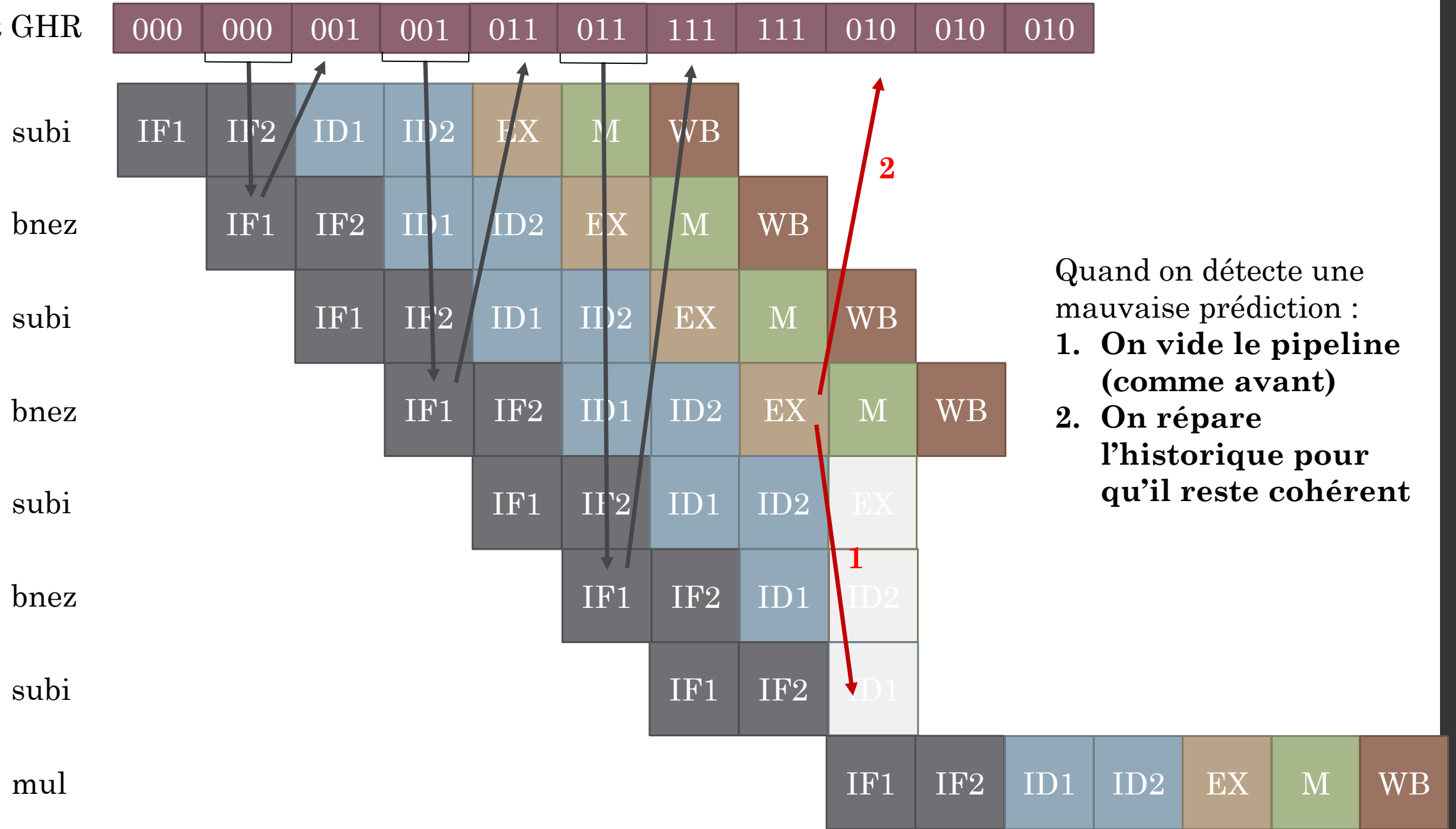


# Mauvaise prédiction



# Mauvaise prédiction

3-bit GHR

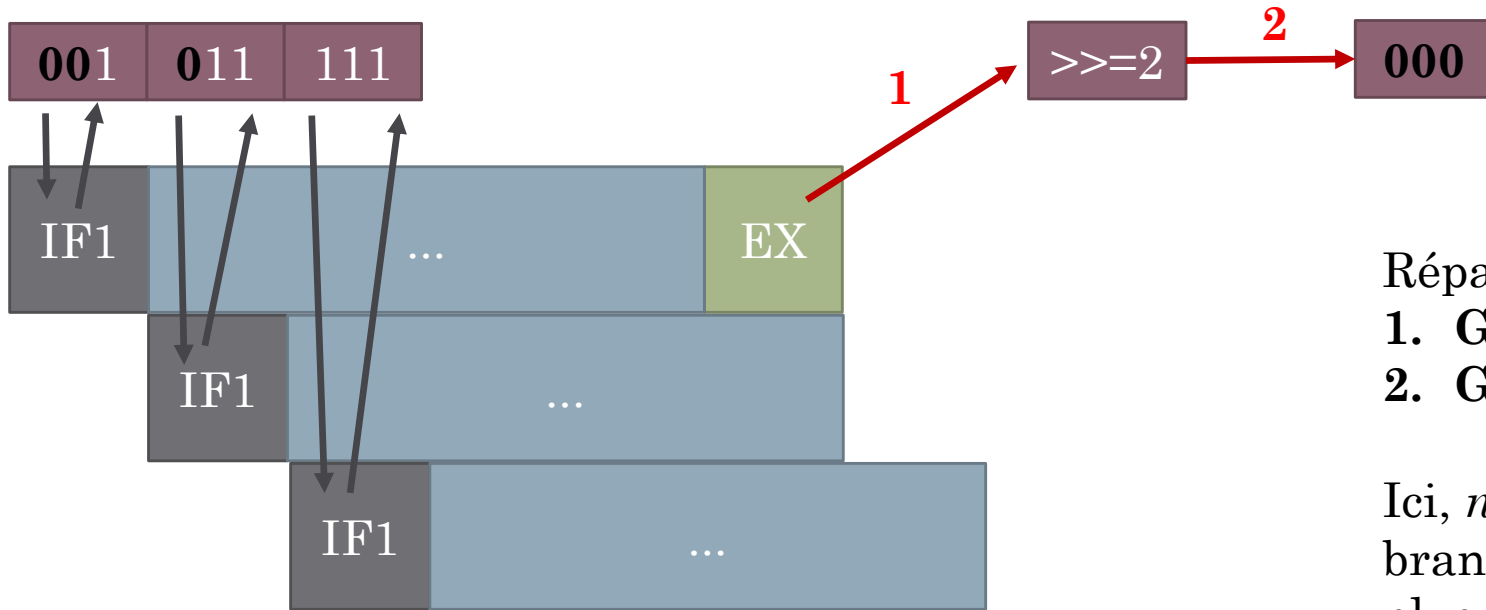


Quand on détecte une mauvaise prédiction :

1. On vide le pipeline (comme avant)
2. On répare l'historique pour qu'il reste cohérent

# Réparer l'historique global – Solution naïve

GHR initial = 000b (en noir)



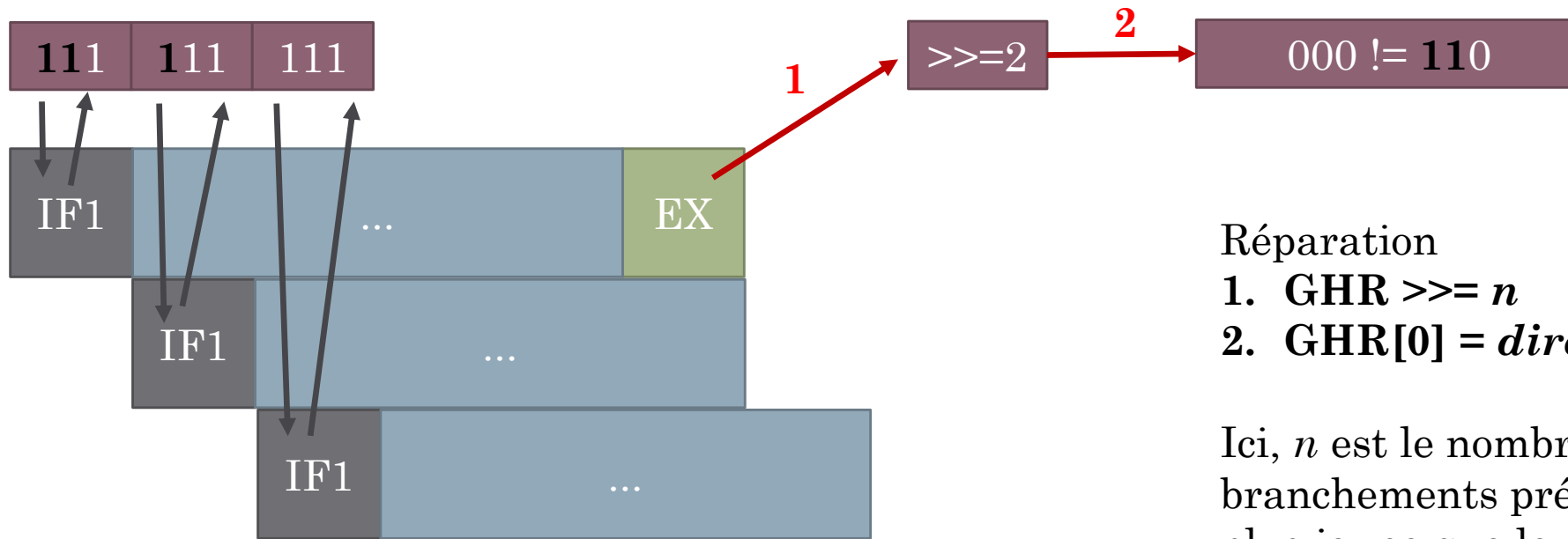
# Réparation

1.  $GHR \geq n$
2.  $GHR[0] = direction$

Ici,  $n$  est le nombre de  
branchements prédits  
plus jeune que le  
branchement mal prédit  
(ici, 2)

# Réparer l'historique global – Solution naïve

GHR initial = 111b (en noir)



Réparation

1. **GHR**  $\gg= n$
2. **GHR**[0] = *direction*

Ici,  $n$  est le nombre de  
branchements prédits  
plus jeune que le  
branchement mal prédit  
(ici, 2)

**On a perdu des bits du GHR !**

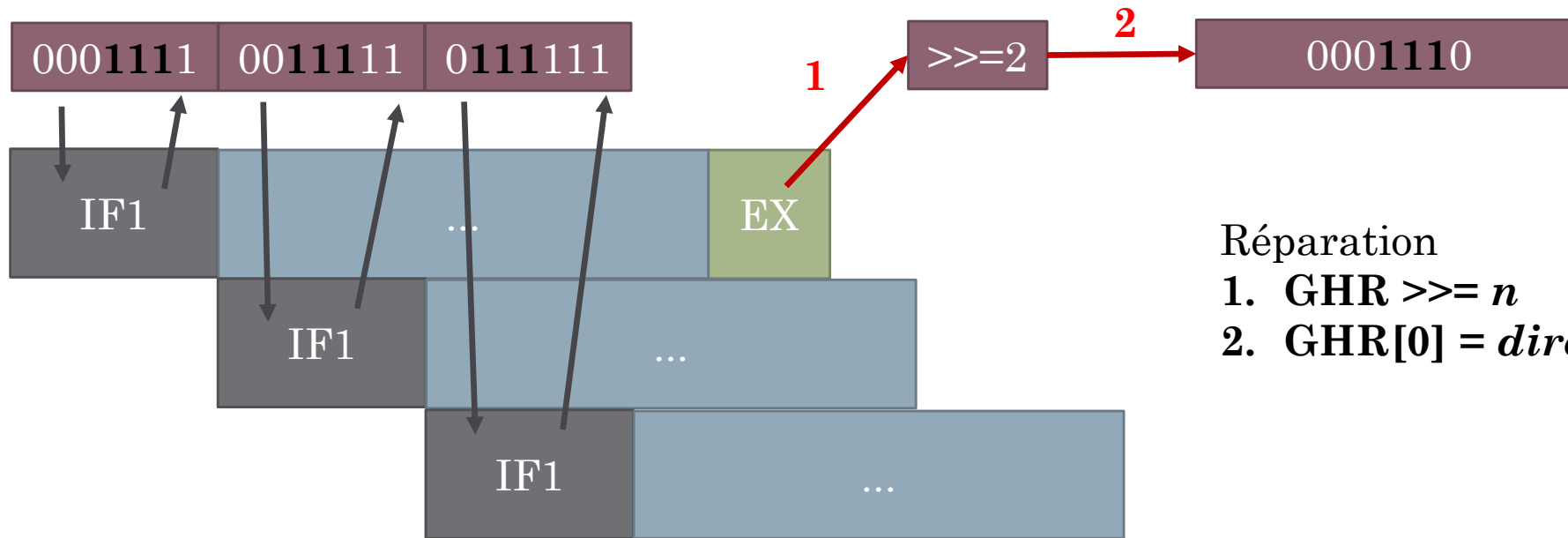
# Réparer l'historique global – Solutions

- Avoir un GHR assez grand pour pouvoir décaler à droite et modifier un seul bit lors d'une mauvaise prédiction **sans perdre d'information**
  - Assez grand = ?
  - Registre matériel avec  $n + s$  bits
    - $n$  les bits utilisés pour la fonction de hash (3 dans l'exemple), c'est le GHR logique
    - $s$  le nombre de mises à jour spéculative possibles avant la résolution d'un branchement (3 ou 4 dans notre exemple, suivant les détails de la microarchitecture)

# Réparer l'historique global – Solution naïve

GHR logique initial = 111b ( $n = 3$ , en noir)

GHR physique initial = 0000111b ( $s = 4$ )



On a bien  $GHR = 110b$

# Réparer l'historique global – Solutions

- Dans les processeurs modernes,  $s$  (nombre de mises à jours spéculatives avant la résolution d'un branchement) est grand (plusieurs dizaines), donc il faut un GHR physique significativement plus grand que le GHR logique
- Une autre solution est de transporter une copie du GHR avec chaque branchement dans le pipeline, et restaurer la copie au lieu de faire un décalage lors d'une mauvaise prédiction.
  - Problème si le GHR est grand
- On peut aussi gérer le GHR comme un registre circulaire avec un pointeur de tête et de queue, et transporter une copie du pointeur de tête, plutôt que du GHR entier

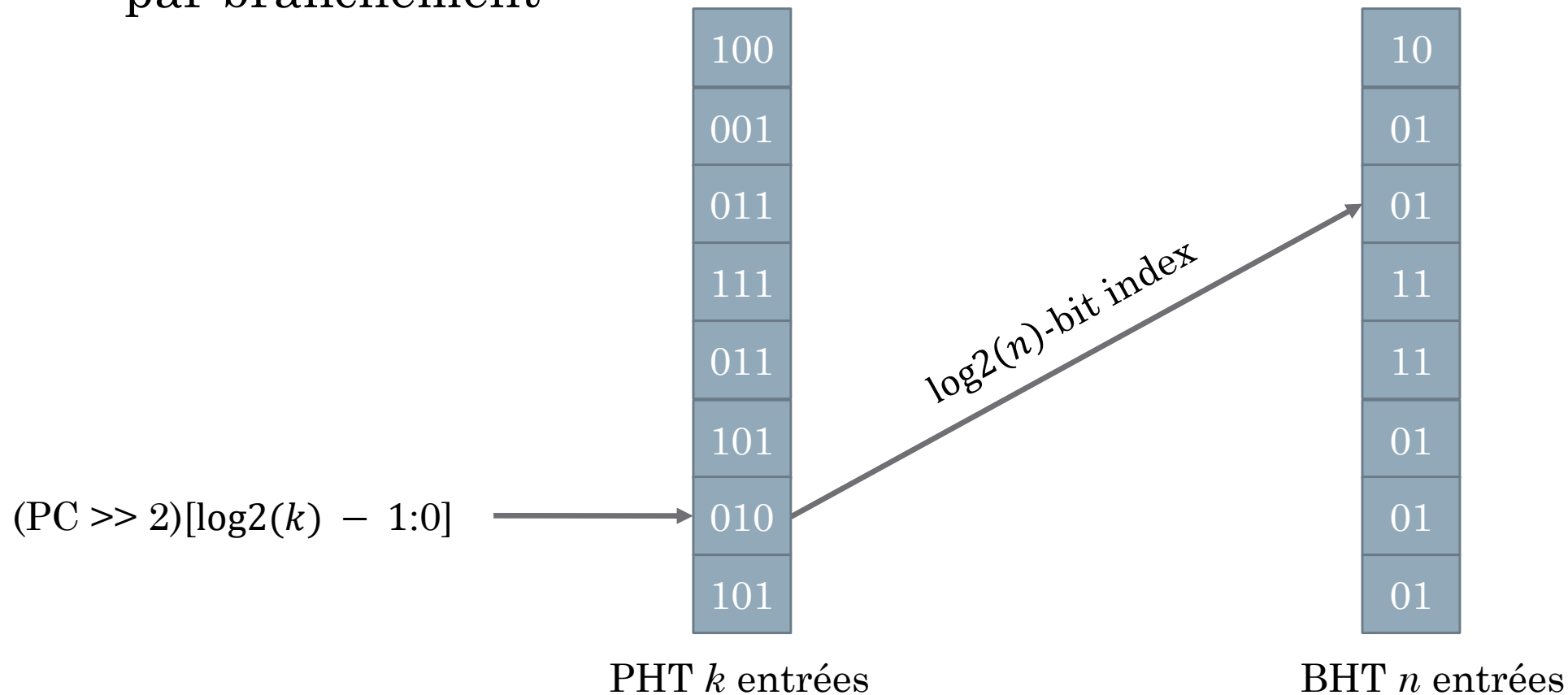
# *gshare* - Inconvénients

- Plus lent à entraîner
  - $2^{GHRBits}$  compteurs utilisables par une instruction vs. 1 seul compteur par instruction avec un BHT simple
  - Corollaire : Plus sensible aux interférences que bimodal pour le même nombre d'entrées
- Plus complexe (fonction d'index, gestion de l'historique)
- Mais bien meilleur que bimodal (voir TP)



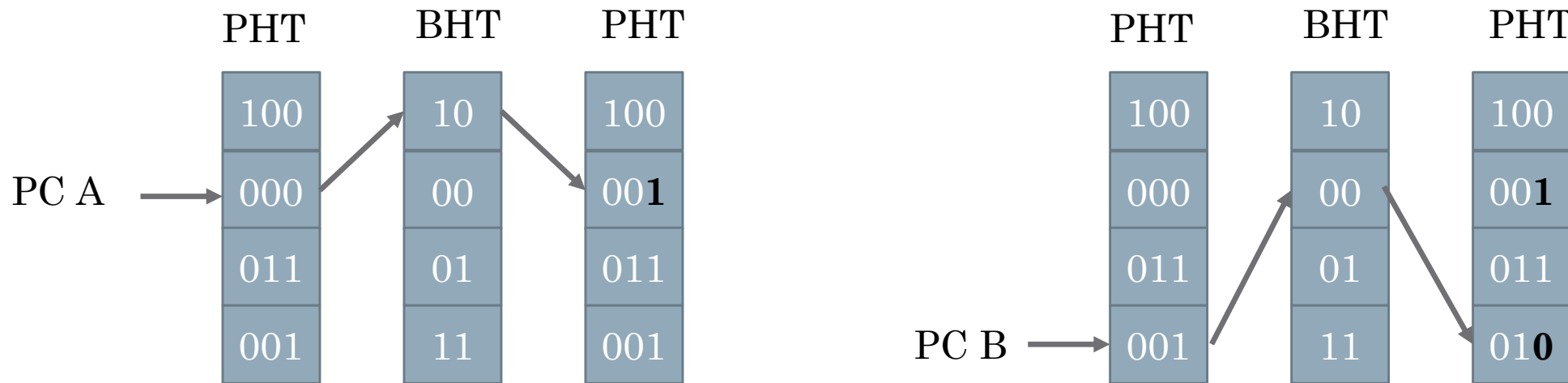
# Corrélation locale : 2-level adaptive

- BHT + Pattern History Table (PHT) qui contient un historique par branchement



# 2-level adaptive - Inconvénients

- Similaire à *gshare*, plus:
  - Gestion spéculative de l'historique local difficile :
    - Plusieurs historiques différents modifiés spéculativement pour garder de bonnes performances
    - Chaque historique modifié doit être réparé lors d'un vidage de pipeline



- Si A est mal prédit, il faut restaurer les historiques de A et B, difficile à faire rapidement (processus itératif)

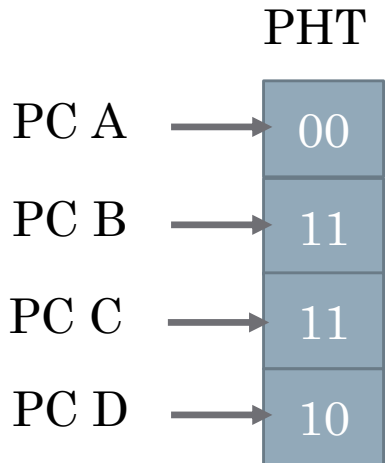
# 2-level adaptive - Inconvénients

- Similaire à *gshare*, plus:
  - Gestion spéculative de l'historique local difficile
  - Plus lent (deux structures à indexer de manière séquentielle)
  - Performance pas forcément meilleure que les prédicteurs à historique global
    - En théorie, avec un historique global suffisamment long, on peut corréler un branchement avec les instances précédentes du même branchement

# 2-level adaptive - Inconvénients

- Local vs. global
  - Séquence de branchements (PC, direction)

A(0) B(1) C(1) D(1) A(0) B(1) C(1) D(0)



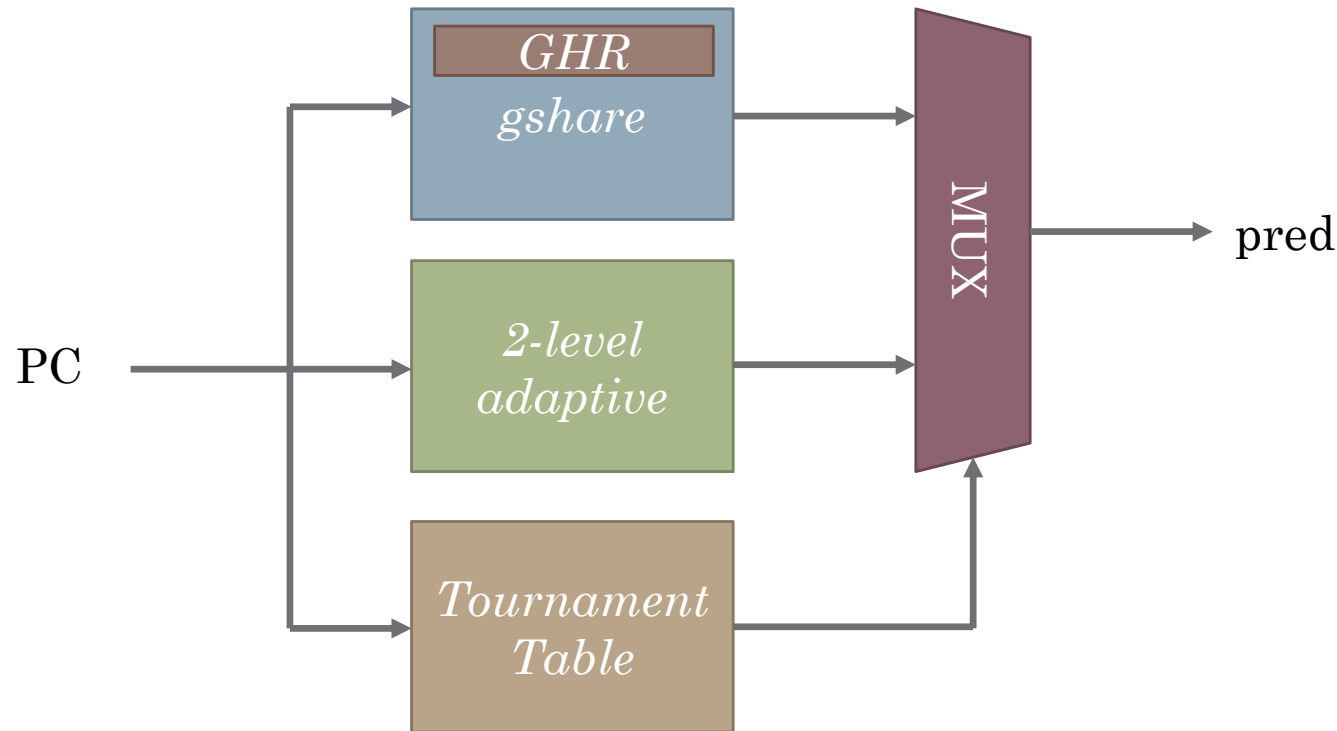
- Dans ce cas, la même information est disponible, mais encodée différemment. Cela étant :
  - Historique local peut retenir l'information plus longtemps (PC garanti d'avoir toujours 2 bits d'historique avec le PHT, 0 bits avec le GHR)
  - Historique local moins sujet au « bruit »

# 2-level adaptive - Inconvénients

- Similaire à *gshare*, plus:
  - Gestion spéculative de l'historique local difficile
  - Plus lent (deux structures à indexer de manière séquentielle)
  - Performance pas forcément meilleure que les prédicteurs à historique global
    - En théorie, avec un historique suffisamment long, on peut corrélérer un branchement avec les instances précédentes du même branchement
- Toujours bien meilleur que bimodal (voir TP)

# Métaprédicteur : Tournoiement

- Certains prédicteurs sont meilleurs pour certains types de codes, ou certains branchements spécifiques
  - Tournoiement : un prédicteur pour prédire quel prédicteur utiliser (métaprédicteur)



# Métaprédicteur : Tournaement

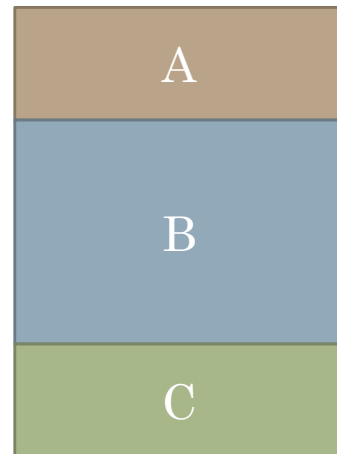
- Tournaement Table contient des compteurs  $n$ -bit
  - Par ex., 2-bit: bit1 donne le prédicteur « gagnant » (0 = 2-level, 1 = gshare), bit0 donne de l'hysteresis
  - Si gshare a bien prédit et 2-level non, incrémenter le compteur
  - Si 2-level a bien prédit et gshare non, décrémenter le compteur
  - Sinon, ne rien faire
- On met à jour les deux prédicteurs pour chaque branchement conditionnel

# Alternative architecturale : Prédication

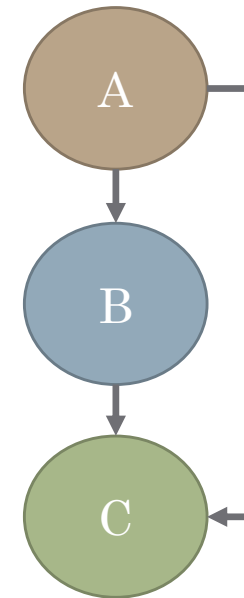
- Idée : transformer le flot de contrôle en flot de données via l'exécution conditionnelle

```
int x = 1;  
cond = ...  
if (cond)  
{  
    x = (x * 3) + 2;  
}  
y = x;  
...
```

Blocs de base



Graphe de flot de contrôle (CFG)





# Alternative : Prédication

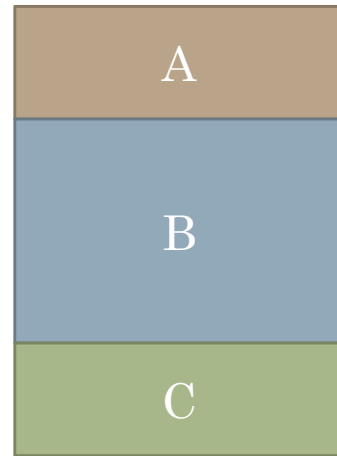
- Idée : transformer le flot de contrôle en flot de données via l'exécution conditionnelle

```
int x = 1;  
cond = ...
```

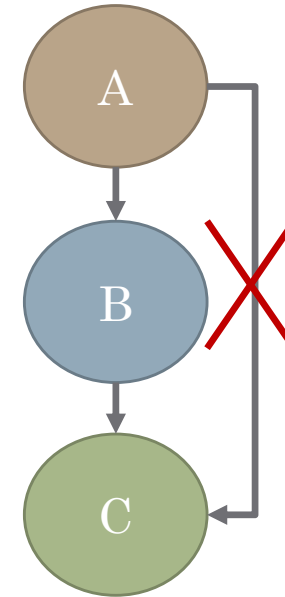
```
x = (x * 3) + 2 si cond;
```

```
y = x;  
...
```

Blocs de base



Graphe de flot de contrôle (CFG)



# Prédication via cmov (x86)

- `cmovxx dst, src` : Copie `src` dans `dst` si `xx` est vrai
  - Calculer plusieurs versions de `x` puis choisir
  - Le calcul du `x` du « if » n'est pas prédiqué (seul le `CMOV` l'est)

```
int x = 1;  
cond = ...
```

```
int x_tmp = (x * 3 + 2);  
x = x_tmp si cond;
```

```
y = x;  
...
```



```
mov r0, 0x1 // x = 1  
ld r1, [@] // Load cond
```

```
mov r2, r0 // temporaire  
mul r2, 0x3 // x_tmp = x * 3  
add r2, 0x2 // x_tmp += 2  
cmp r1, 0x0 // Set flags  
cmovnz r0, r2 // si !Zero Flag  
                // (cond != 0x0)  
                // x = x * 3 sinon nop
```

```
mov r3, r0 // y = x
```

# Prédication généraliste (armv7)

- Chaque instruction peut être prédiquée
  - On prédique toutes les instructions qui servent à calculer la version de x dans le « if »

```
int x = 1;  
cond = ...
```

```
x = (x * 3) + 2 si cond;
```

```
y = x;  
...
```



```
mov r0, 0x1 // x = 1  
ld r1, [@] // Load cond
```

```
cmp r1, 0x0 // Set flags  
mulne r0, r0, 0x3 // x = x * 3 si  
// cond != 0x0  
addne r0, r0, 0x2 // x += 2 si cond != 0
```

```
mov r2, r0 // y = x
```

# Prédication : Inconvénients

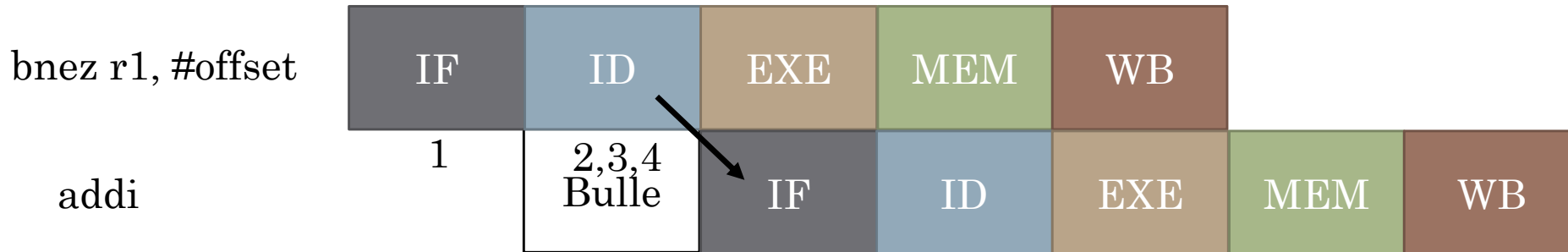
- Les instructions dont le prédicat se révèle faux consomment quand même des ressources puisqu'on les exécute
  - Si le corps du if est gros, beaucoup d'instructions « inutiles » si la condition s'avère fausse
  - Si la version avec branchement se prédit facilement, on risque surtout de perdre de la performance
- Difficile de jauger l'utilité à la compilation
  - Le compilateur ne sait pas vraiment si le branchement est prédictible
  - Le compilateur ne sait pas vraiment quel est le coût d'une mauvaise prédiction de branchement vs. prédication (dépend de la microarchitecture)

# Prédication : Inconvénients

- De manière générale, moins intéressante pour la haute performance que pour l'embarqué (sauf exception)
  - Principalement car un processeur embarqué a rarement un très bon prédicteur de branchement
- ISA modernes ont seulement quelques instructions prédiquées
  - x86 : cmov
  - armv8 : csel/csinc/csneg
  - RISC-V : cmov (extension bitmanip du jeu d'instruction de base)

# Revenons-en à nos branchements

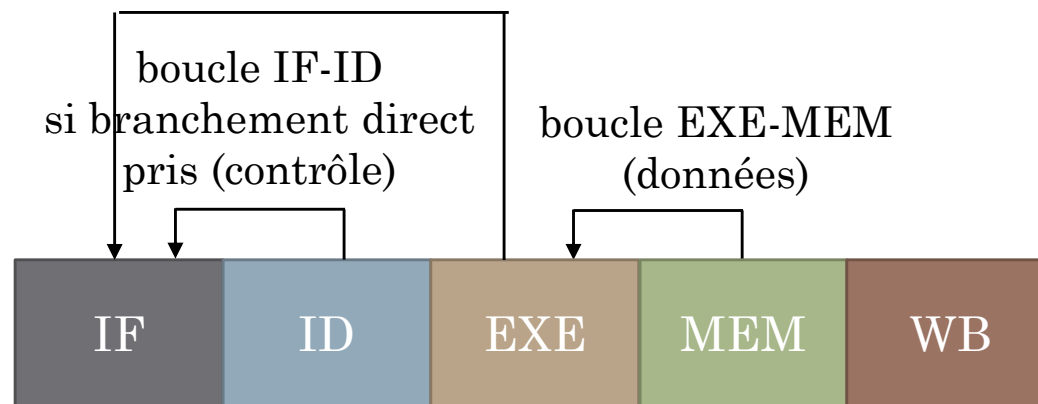
- Prédiction de branchement = prédire la direction ?
- Calcul du prochain PC pour un branchement direct requiert :
  1. D'avoir lu l'instruction dans la mémoire d'instructions
  2. D'avoir décodé l'instruction pour savoir qu'il s'agit d'un branchement
  3. D'avoir étendu le signe du déplacement encodé dans l'instruction (#offset)
  4. De faire une addition 64-bit pour calculer  $\text{NextPC} = \text{PC} \pm \text{déplacement}$



# Revenons-en à nos branchements

- Notion de pénalité de branchement pris (ou non conditionnel) :  
boucle IF-ID
- Idéalement, IF doit récupérer une instruction à **chaque cycle**

boucle IF-EXE si mauvaise prédiction ou  
branchement indirect  
(contrôle)

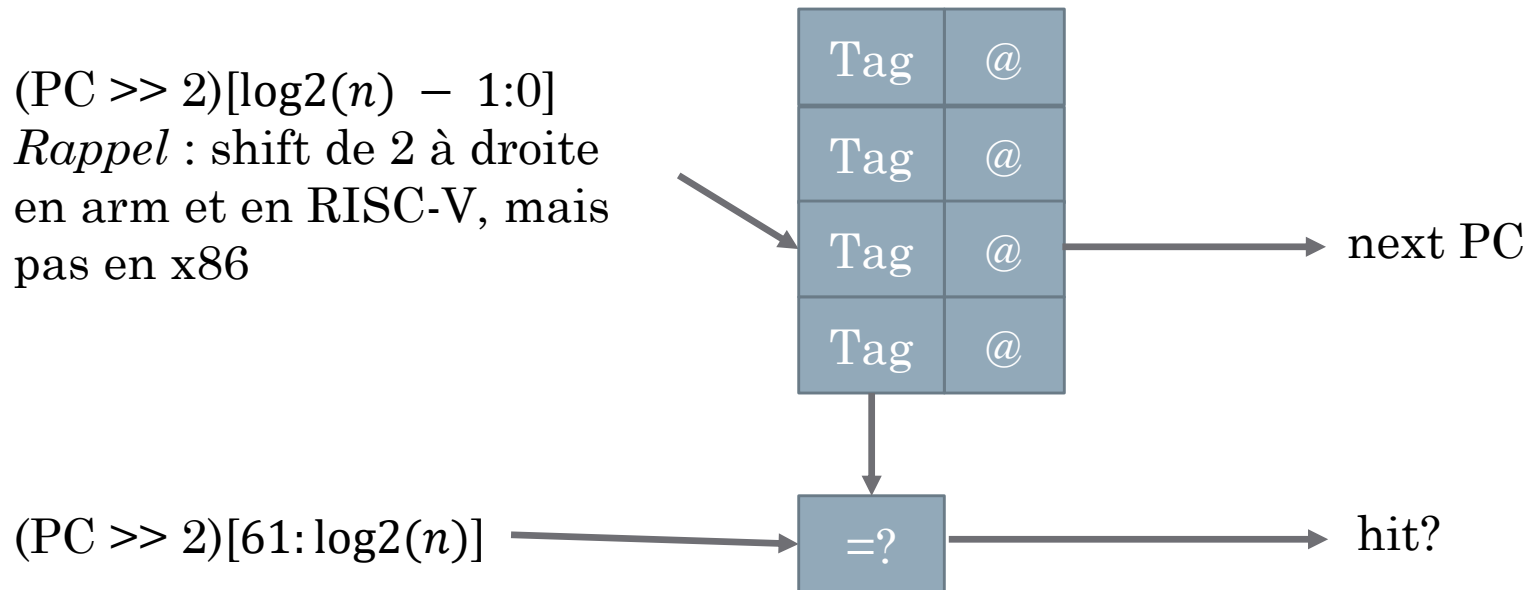


# Branch Target Buffer (BTB)

- Table matérielle qui contient les adresses cibles déjà calculées
  - En général on utilise un tag pour éviter qu'une instruction qui n'est pas un branchement obtienne une adresse

$(PC \gg 2)[\log_2(n) - 1:0]$

*Rappel* : shift de 2 à droite  
en arm et en RISC-V, mais  
pas en x86





# Branch Target Buffer (BTB)

- Le BTB fournit une **prédiction**, qui sera validée en calculant l'adresse dans ID (branchement direct) ou EXE (branchement indirect)
- Pourquoi une prédiction ?
  - *Aliasing* : plusieurs branchements peuvent avoir le même index dans le BTB si le tag est partiel
  - *Code auto-modifiant* : Même si le tag est complet (tout le reste des bits du PC non utilisé pour l'index), certains paradigmes logiciels peuvent remplacer le code à la volée. Un branchement au PC A sautant vers B peut être remplacé par un branchement au PC A sautant vers C
- Une mauvaise prédiction par le BTB est aussi appelée « misfetch » dans le cas d'un branchement direct, « mistarget » dans le cas d'un branchement indirect

# Calcul prochain PC jusqu'ici

Adresse autre chemin ou indirecte calculée

de EXE

Adresse branchement direct calculée

Adresse prédite si prédit « pris » ou inconditionnel

Adresse séquentielle

Mauvaise prédiction de direction ou  
d'adresse branchement indirect

de EXE

Mauvaise prédiction d'adresse  
branchement direct

pris/non pris

+4

Branch  
Target Buffer

Prédicteur de  
branchements

Cache  
Instructions

Décodeur

dépl.

SEXT

ADD

=?

vers EXE

IF

ID

XNW

PC

# Calcul prochain PC jusqu'ici

- A noter que dans la figure précédente, on récupère une prédiction de branchement avant même de savoir si l'instruction qu'on récupère est un branchement
- On peut donc utiliser le hit/miss du tag dans le BTB pour éviter de prédire « pris » pour une instruction qui n'est pas un branchement
  - direction = output(prédicteur de branchement) & BTB hit
  - Plus généralement, le BTB contient aussi des métadonnées sur les branchements, notamment le type (cond/non cond, direct/indirect)

# Revenons-en à nos branchements

- Prédiction de branchement = prédire la direction ?
  - Aussi l'adresse ! BTB pour branchements directs.
- Calcul du prochain PC pour un branchement indirect ? On distingue :
  - *ret* saute vers l'adresse de retour de fonction contenue dans un registre (arm, RISC-V) ou la pile (x86)
  - *jr/jalr* qui sautent vers l'adresse contenue dans un registre
- On peut utiliser le BTB, mais le BTB est plutôt fait pour les branchements qui ne changent jamais de cible (directs)
- Structures de prédictions dédiées

# Return Address Stack (RAS)

- Les programmes sont organisés en *fonctions* qui utilisent la pile d'appel pour gérer leur contexte
- Au niveau du langage machine, deux primitives *call* et *return*
  - *call* (a.k.a. *jump & link*) peut être direct ou indirect
  - *return* est indirect, mais sautera en général vers le PC de l'instruction qui suit le *call* le plus récent
- Idée : Garder les adresses de retour futures dans une structure matérielle type pile

# Return Address Stack (RAS)

call A // IFetch : push  $PC(\text{call A}) + 4$

A:

...

call B

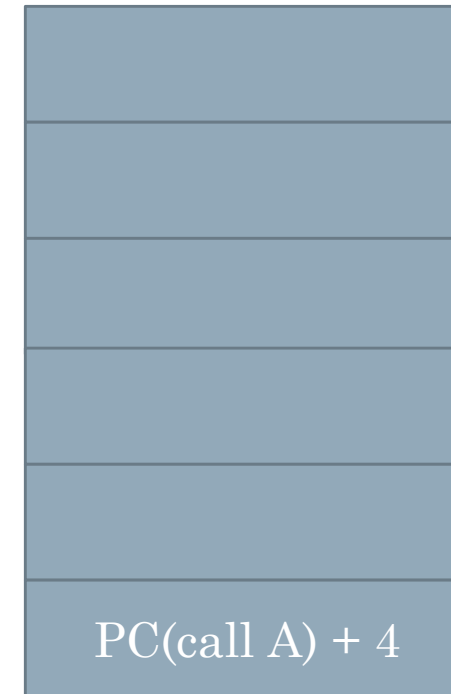
...

return

B:

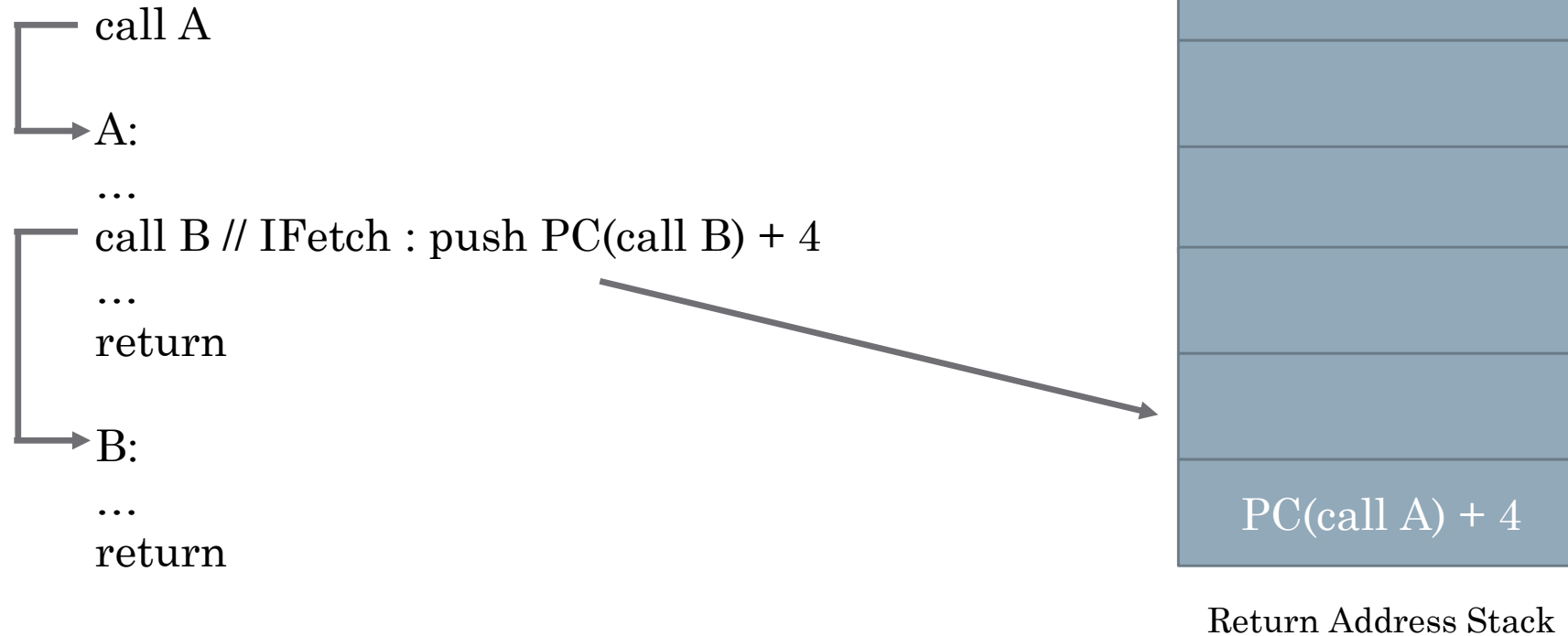
...

return

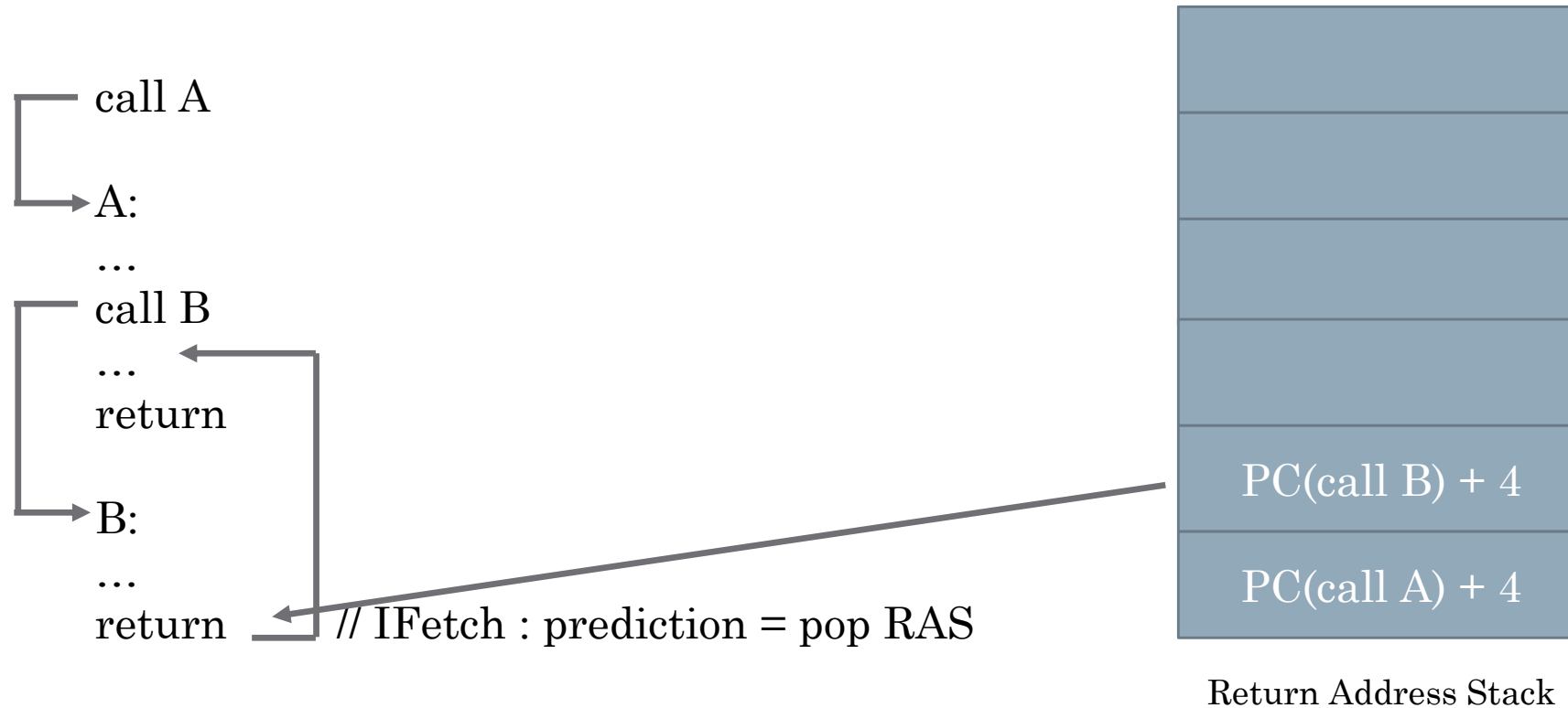


Return Address Stack

# Return Address Stack (RAS)

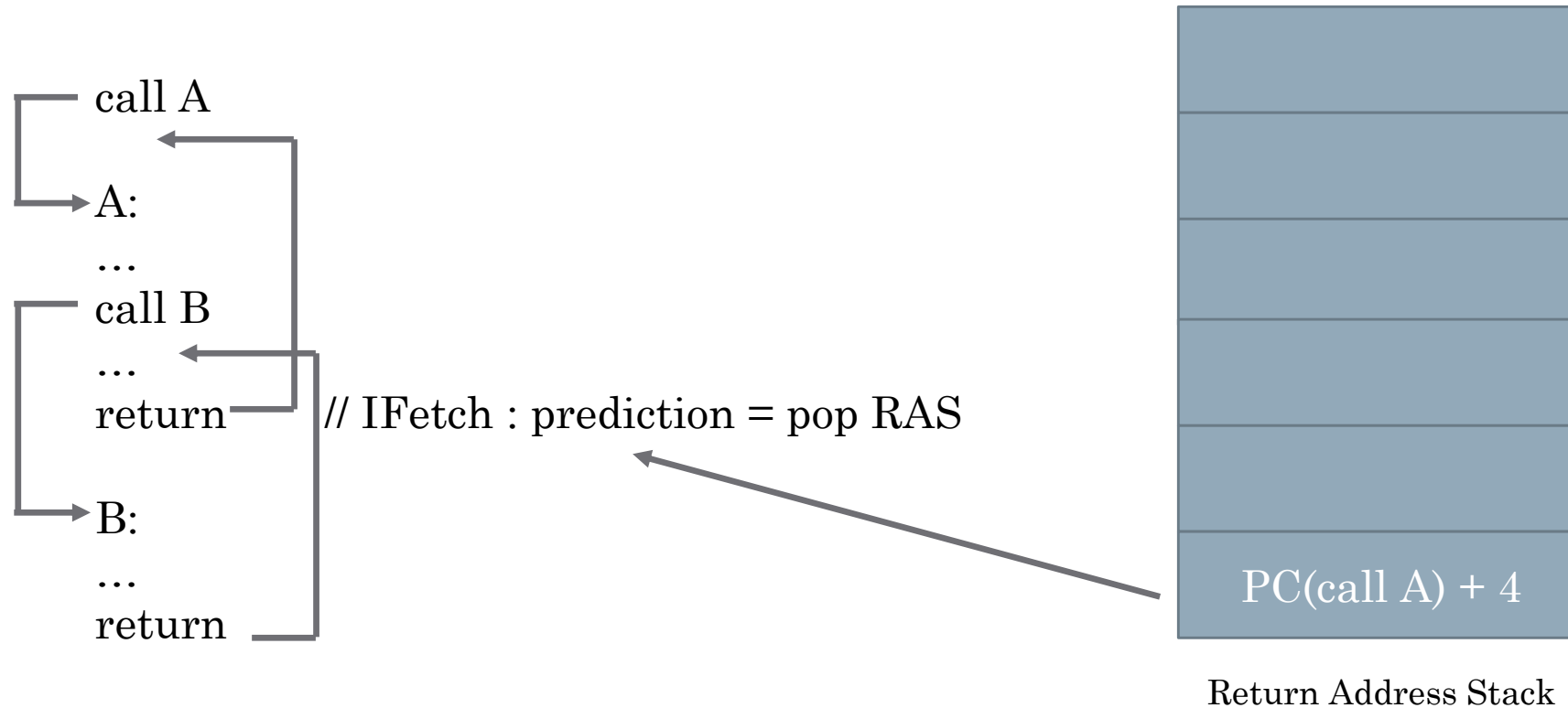


# Return Address Stack (RAS)





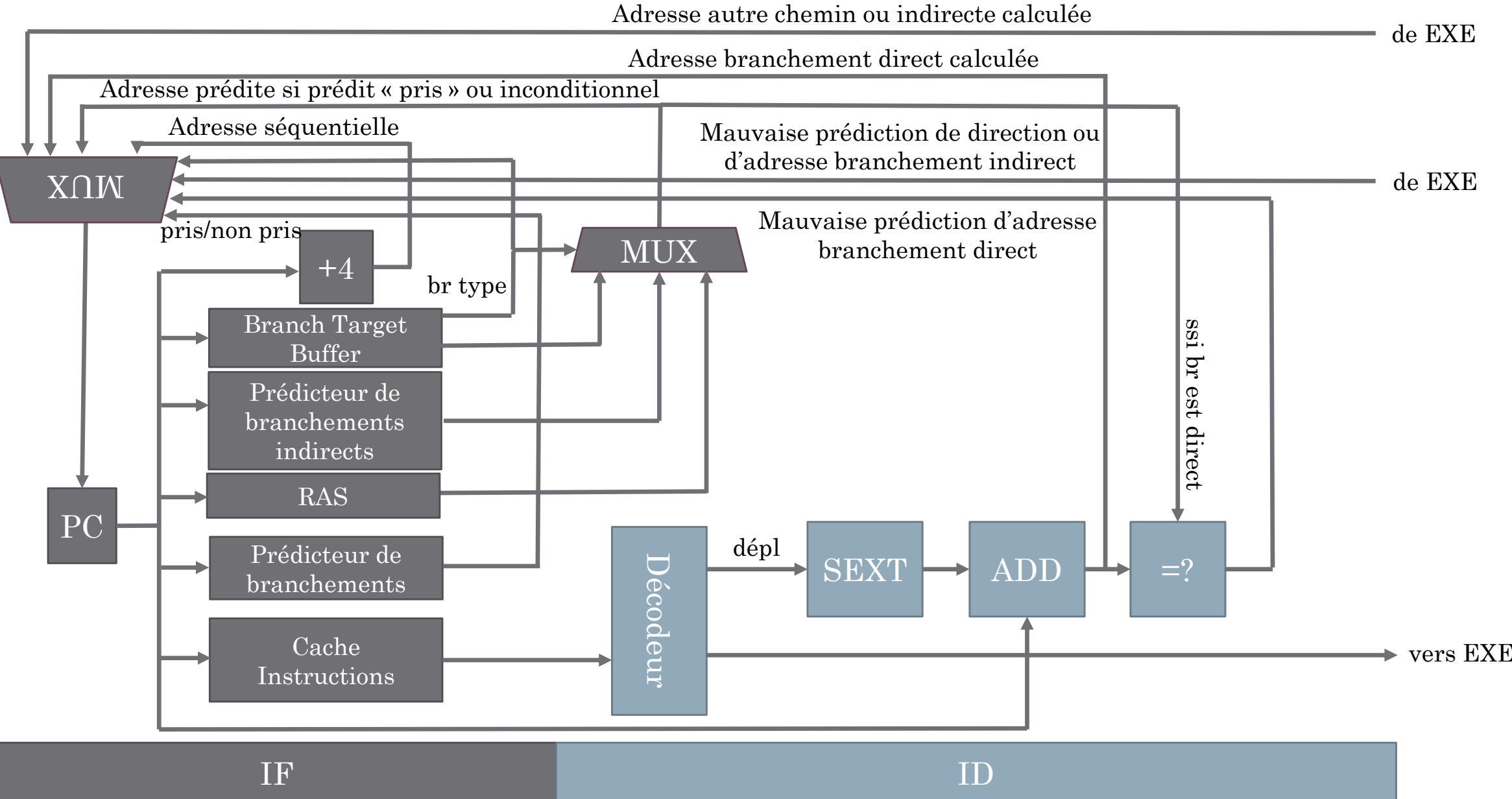
# Return Address Stack (RAS)



# Return Address Stack (RAS)

- RAS obtient > 99% de précision pour les *return*
- Quelques problèmes intéressants
  - Quelle taille pour la RAS ?
  - Appels récursifs ?
  - Tout comme l'historique de branchement, on doit gérer la RAS de manière spéculative. Comment gérer les push/pop sur le mauvais chemin ?

# Calcul prochain PC : Pas si simple



IF

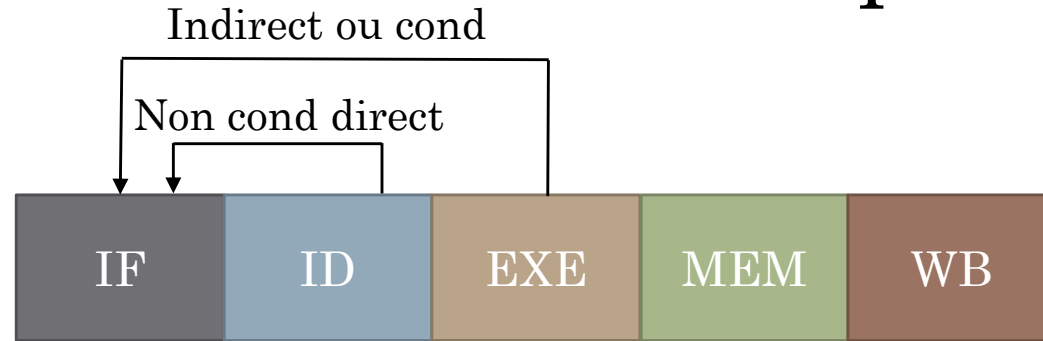
ID

# Pour résumer

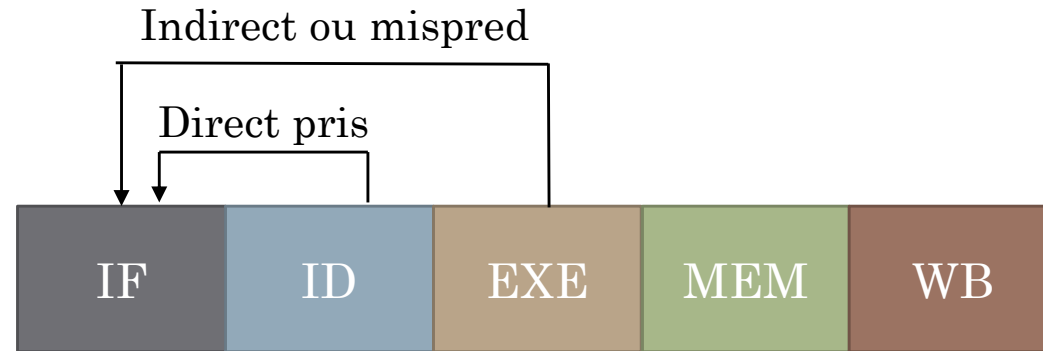
- L'étage Fetch doit minimiser les bulles dues aux branchements = calcul du prochain PC à chaque cycle, idéalement
- Implique de multiples structures et prédictions
  - Prédicteur de branchement pour les branchements conditionnels
  - BTB pour les adresses des branchements directs (cond et non cond)
  - RAS pour les *return*
  - Prédicteur de branchement indirect pour les autres branchements indirects
- PC à chaque cycle implique que toutes ces structures soient accessibles **en moins d'un cycle**
  - Pas toujours possible à 3+Ghz car les structures sont grandes (milliers d'entrées)
    - Certains designs ont une pénalité de branchement pris (= boucle  $\mu$ arch) malgré la présence de prédicteurs

# Dépendances de contrôle – Exécution spéculative

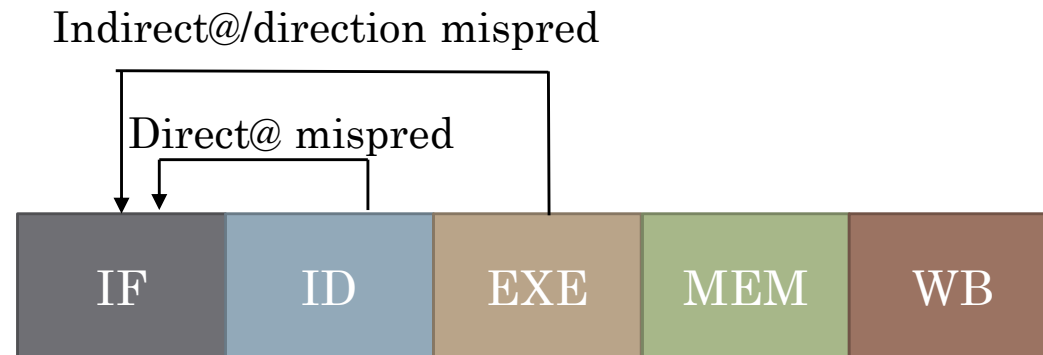
Sans prédiction: 2 bulles  
à chaque branchement  
conditionnel/indirect  
1 bulle à chaque branchement non  
cond direct



Avec prédiction de direction: 2  
bulles quand mauvaise prédiction  
de branchement, 1 bulle quand  
branchement direct pris, 2 bulles  
quand branchement indirect



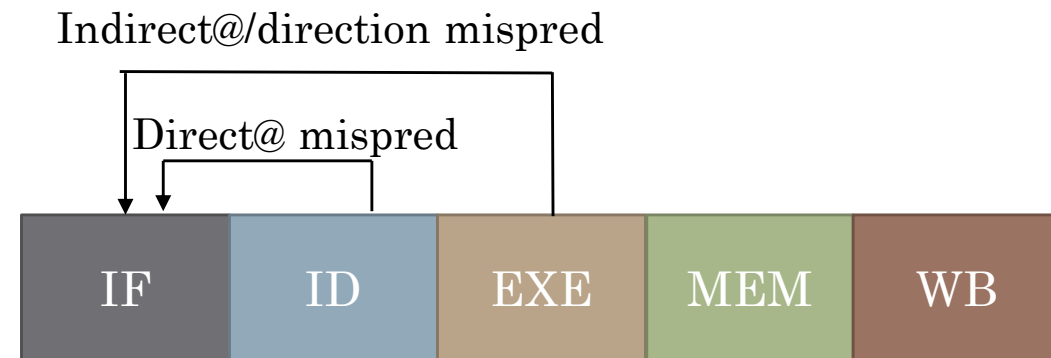
Avec prédiction de direction et  
adresse: 2 bulles quand mauvaise  
prédiction direction/adresse  
indirecte  
1 bulle quand mauvaise prédiction  
adresse direct



# Dépendances de contrôle – Exécution spéculative

- Dépendances structurelles causées par les dépendances de contrôle réduites avec l'exécution spéculative
- Efficacité dépend du nombre de prédictions correctes
  - Prédicteurs +gros,+complexe -> moins de mauvaises prédictions
  - Prédicteurs +gros,+complexe -> plus lent, potentiel pour réintroduire des boucles microarchitecturales même lorsque la prédiction est correcte

Avec prédiction de direction et  
adresse: 2 bulles quand mauvaise  
prédiction direction/adresse  
indirecte  
1 bulle quand mauvaise prédiction  
adresse directe



# Questions ?

## Pipelining et prediction de branchement

SEOC3A – CEAMC

Arthur Perais (arthur.perais@univ-grenoble-alpes.fr)