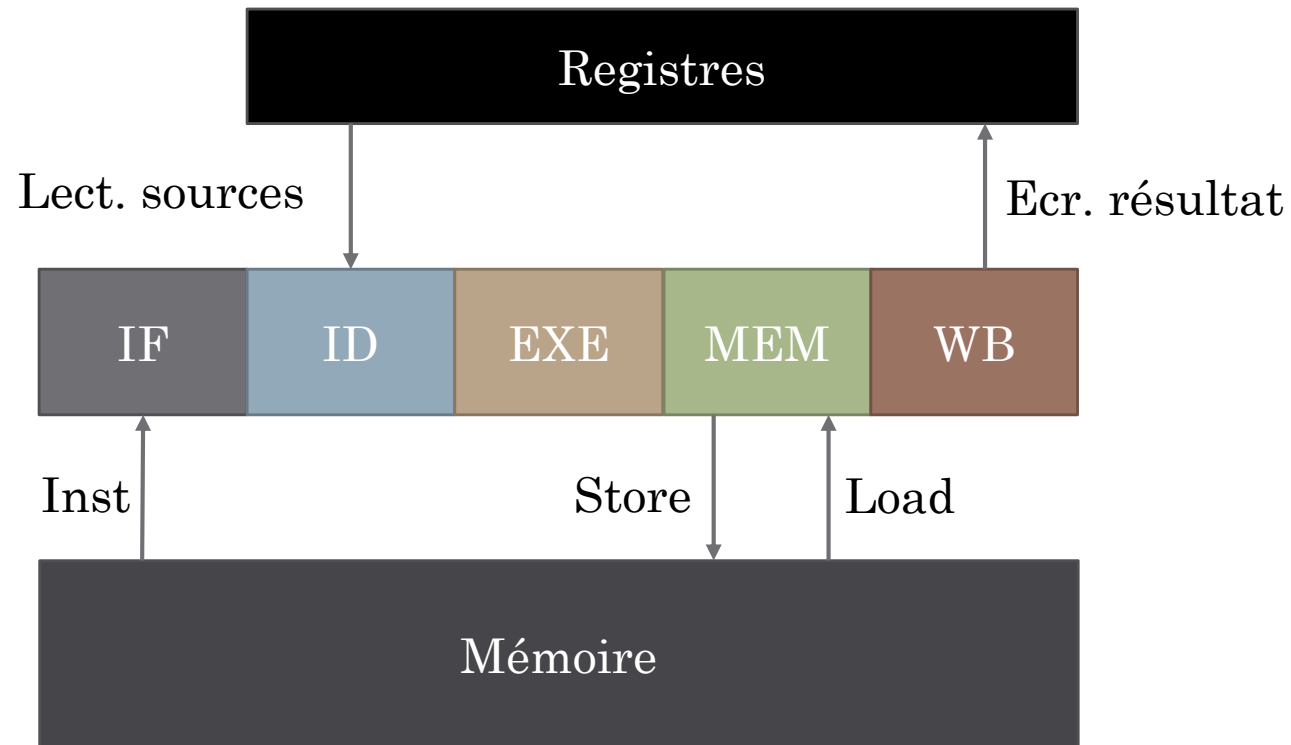


# Pipelining et prédiction de branchement

SEOC3A – CEAMC

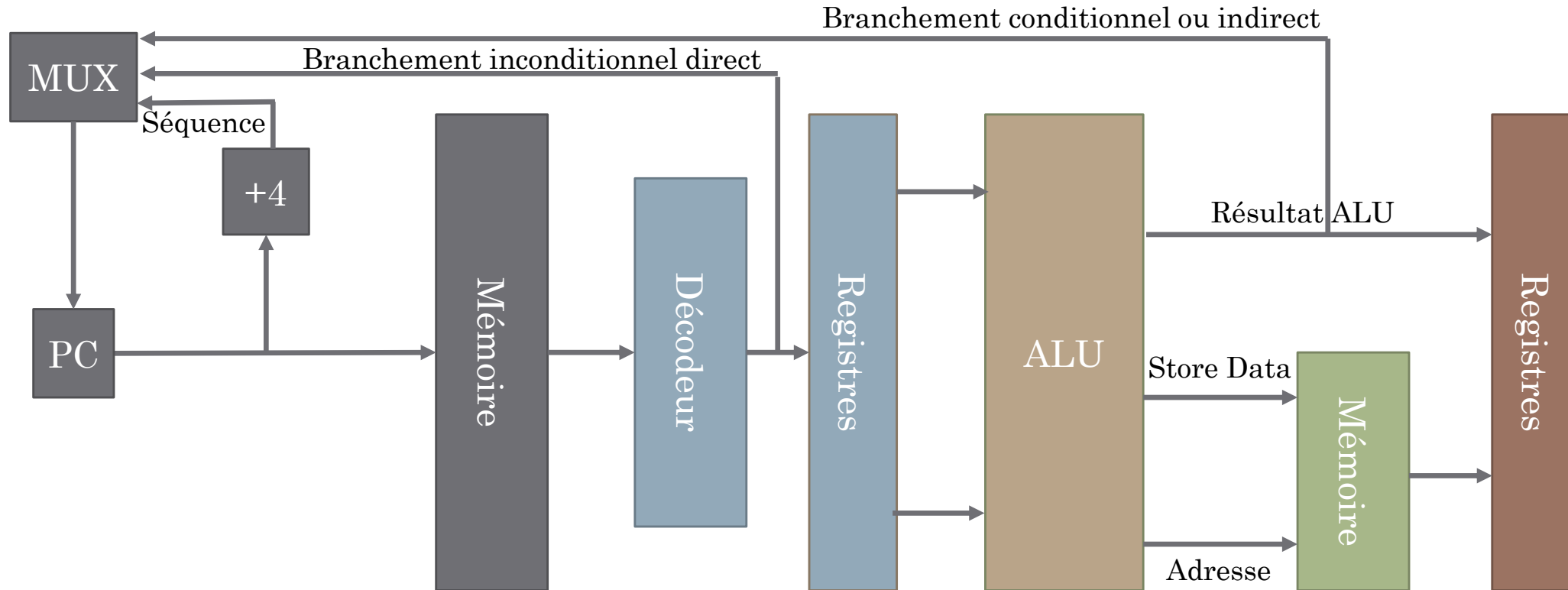
Arthur Perais (arthur.perais@univ-grenoble-alpes.fr)

# CPU minimaliste

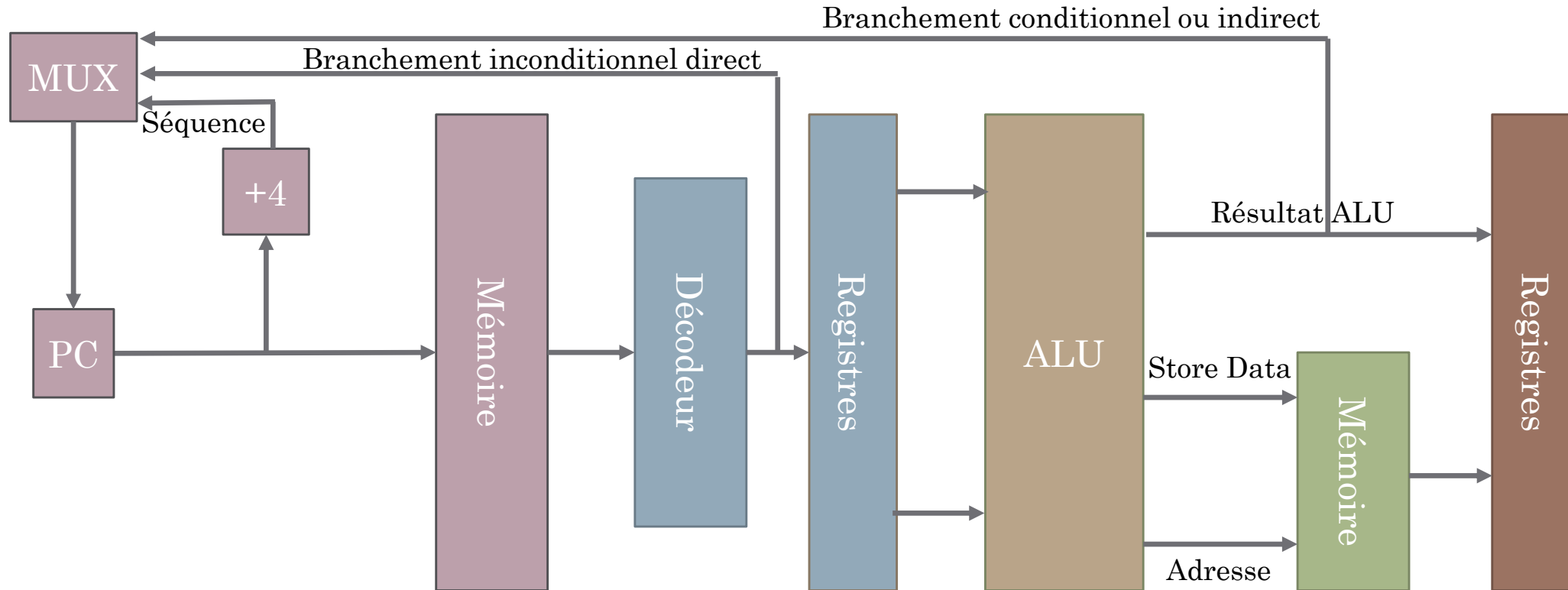


Latence : 5 CPI  
Débit : 1 IPC

# Exemple de CPU minimaliste



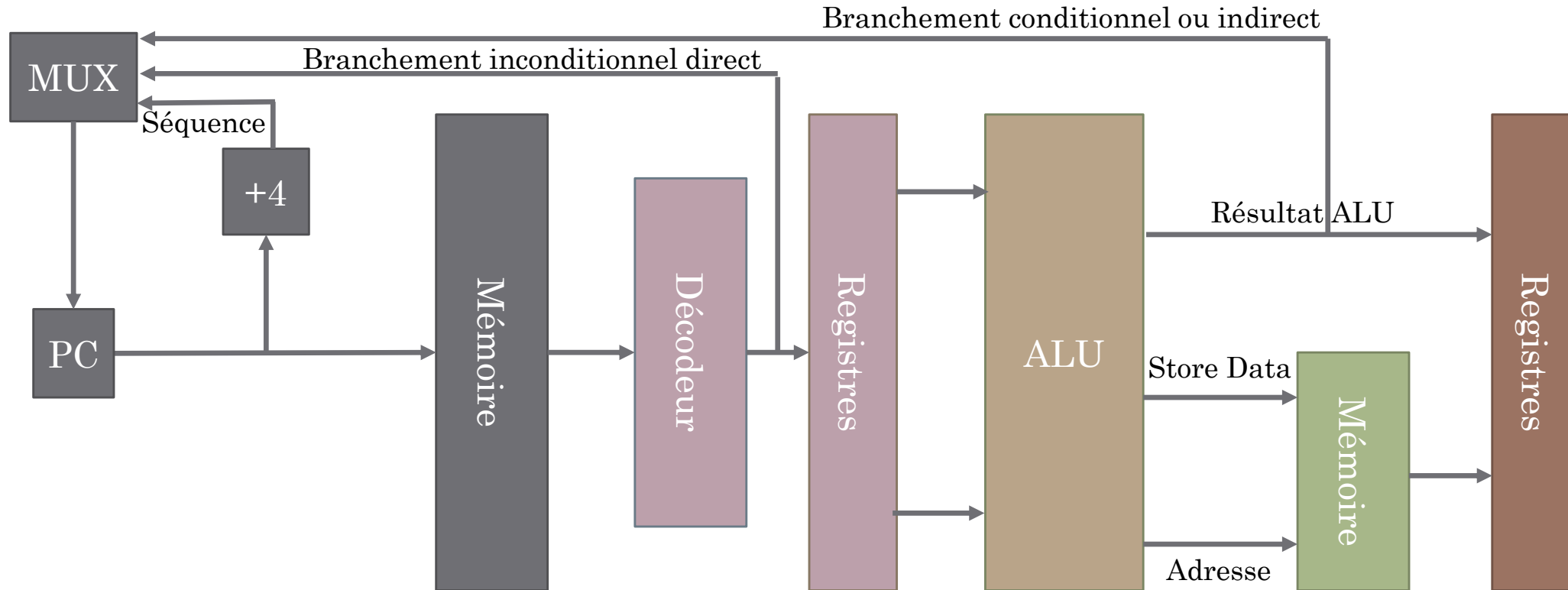
# Exemple de CPU minimaliste



## Instruction Fetch (IF)

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

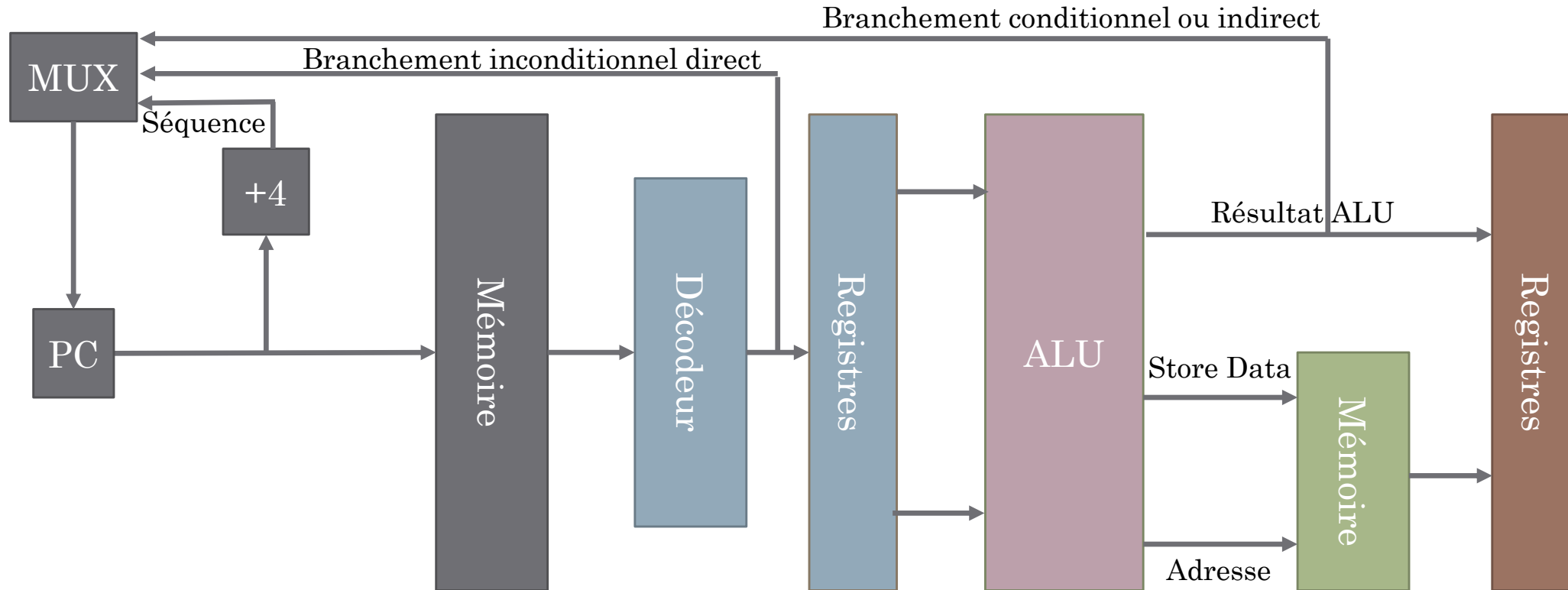
# Exemple de CPU minimaliste



## 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

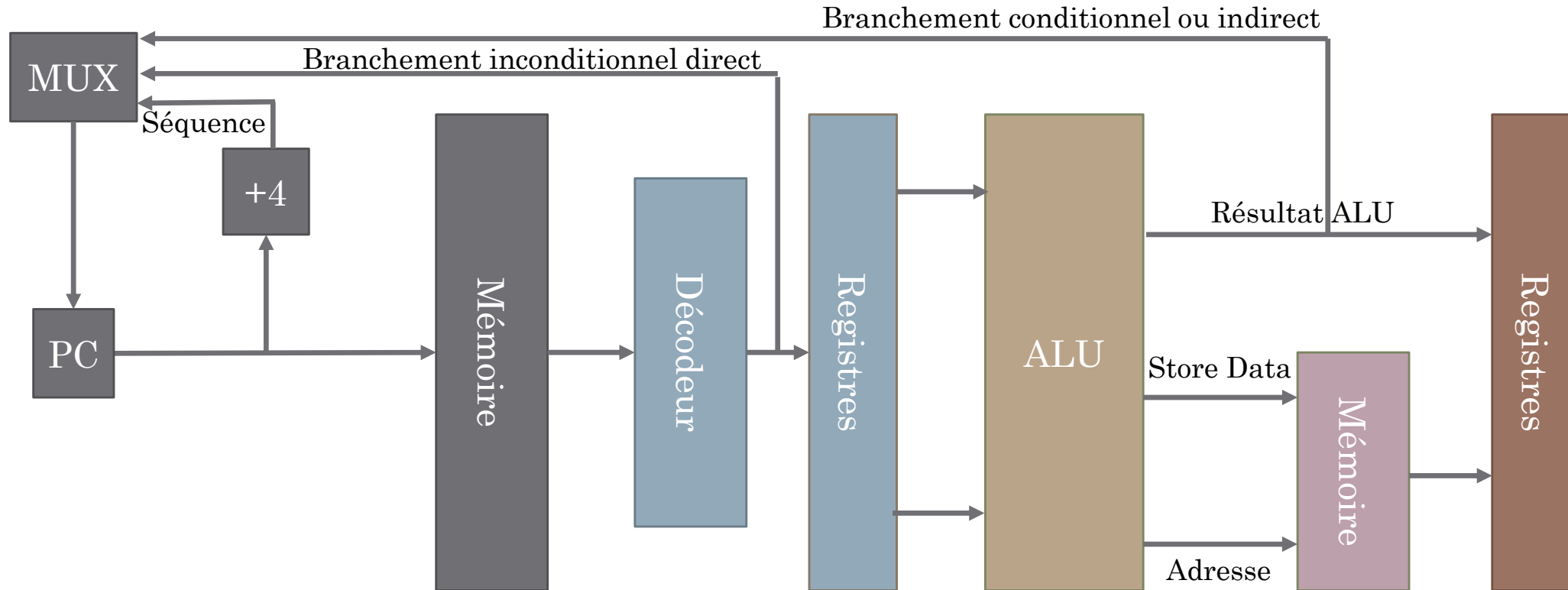
# Exemple de CPU minimaliste



## Execute (EXE)

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

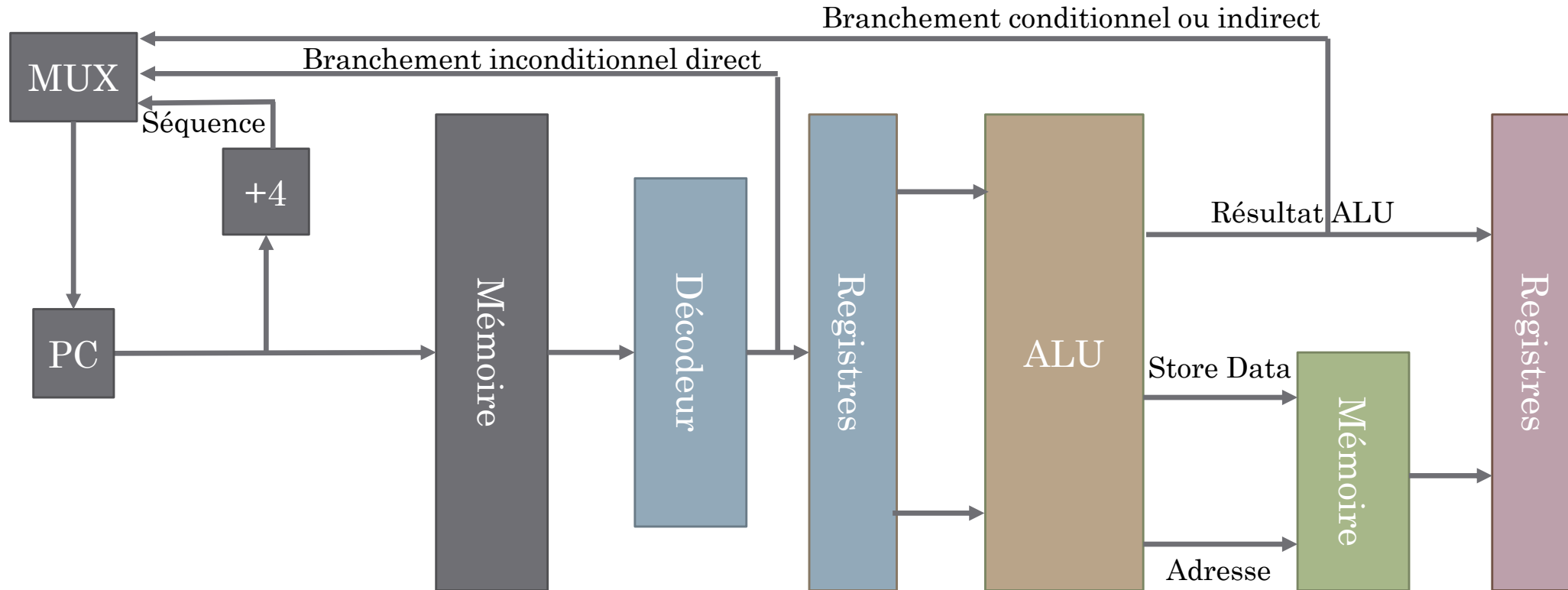
# Exemple de CPU minimaliste



## Accès mémoire (MEM)

1. Accès à la mémoire (stockage/chargement)

# 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
    - Une instruction s'exécute lorsque toutes ses dépendances sont satisfaites

# Dépendances architecturales

- Initialement, une seule condition est nécessaire pour une exécution correcte : **Si  $I_y$  apparaît après  $I_x$  dans la trace d'exécution dynamique du programme, alors  $I_y$  a été exécutée après  $I_x$**
- Couvre notamment les dépendances producteur-consommateur (données)
  - Si  $I_y$  utilise le résultat produit par  $I_x$ , exécuter  $I_y$  après  $I_x$  garanti que le résultat de  $I_x$  a été produit
- Couvre aussi les dépendances liées aux branchements conditionnels (contrôle)
  - Si  $I_x$  est un branchement conditionnel, il faut exécuter  $I_x$  pour savoir si la prochaine instruction sera  $I_y$  (non pris) ou  $I_z$  (pris).  $I_x$  s'exécutant avant la prochaine instruction, on sait quelle sera la prochaine instruction
- Condition spécifiée par tous les jeux d'instructions généralistes modernes (à delta près)

# Dépendances architecturales

- Initialement, une seule condition est nécessaire pour une exécution correcte : **Si  $I_y$  apparaît après  $I_x$  dans la trace d'exécution dynamique du programme, alors  $I_y$  a été exécutée après  $I_x$**
- Une implémentation (microarchitecture) qui exécute une instruction par cycle respecte cette contrainte, par construction
- A mesure que l'on va complexifier la microarchitecture, on va faire apparaître des dépendances architecturales plus spécifiques sur les données et le contrôle
  - La microarchitecture devra explicitement respecter ces dépendances

# Dépendances de données

- Par exemple, dépendances entre producteur et consommateur

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

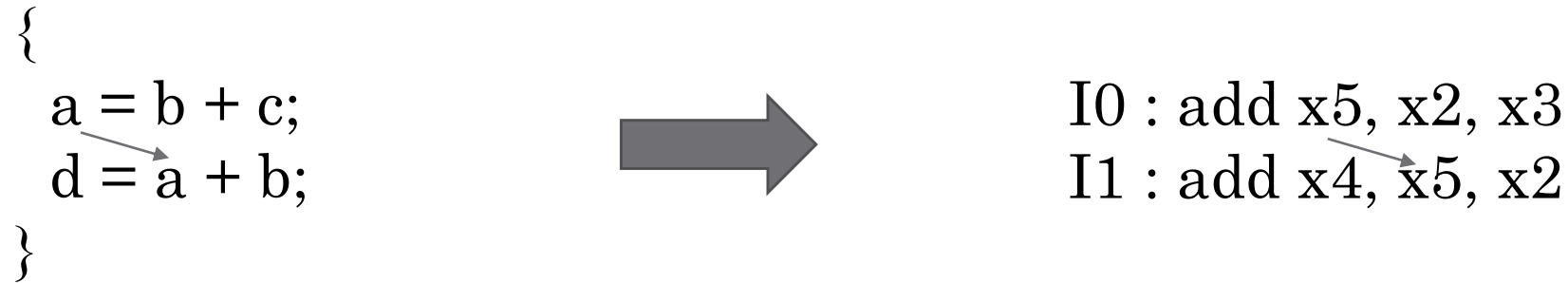


```
I0 : add x5, x2, x3  
I1 : add x4, x5, x2
```

Dépendances?

# Dépendances de données

- Dépendances entre producteur et consommateur



Dépendance de donnée *architecturale* : I1 consomme le résultat produit par I0, donc I1 est exécutée **après** I0

Déjà contenue dans la dépendance d'ordre qui dicte que I1 est exécutée après I0 **même en l'absence** d'une dépendance de donnée

# Dépendances de données

- Le pipeline exécute les instructions dans l'ordre
  - Est-ce suffisant pour respecter les dépendances de données producteur-consommateur ?

I0 : add x5, x2, x3



I1 : add x4, x5, x2



?



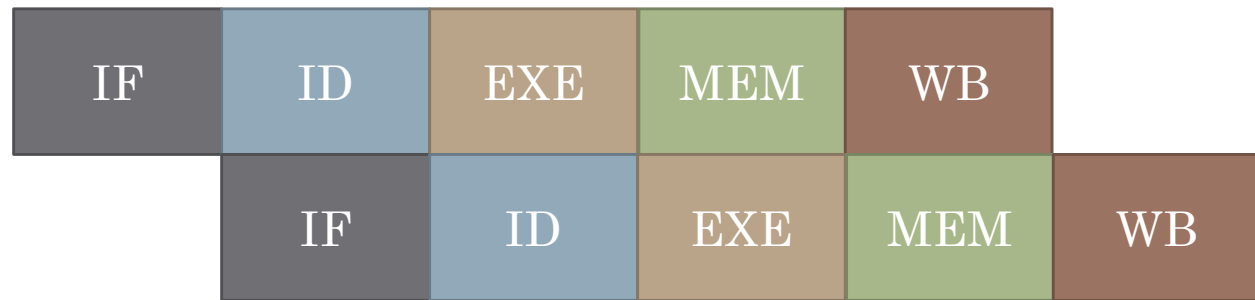
# Dépendances de données

- Le pipeline exécute les instructions dans l'ordre
  - Est-ce suffisant pour respecter les dépendances de données producteur-consommateur ?

I0 : add x5, x2, x3

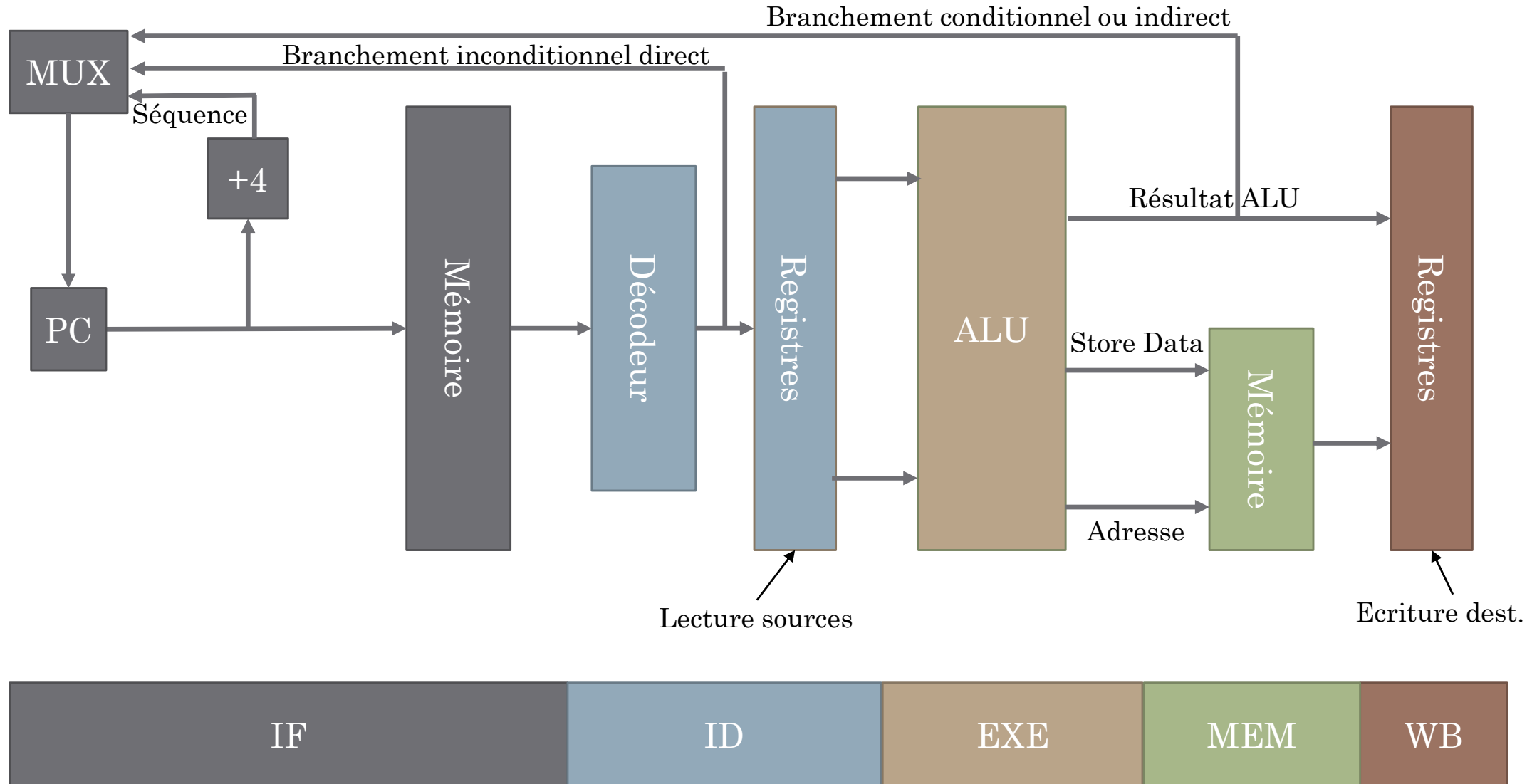


I1 : add x4, x5, x2



- **Exercice:** Quelle valeur pour x4 (x2 vaut 10, x3 vaut 15, x5 vaut initialement 42)

# Rappel du Pipeline



# Dépendances de données

- Le pipeline exécute les instructions dans l'ordre
  - Est-ce suffisant pour respecter les dépendances de données producteur-consommateur ?

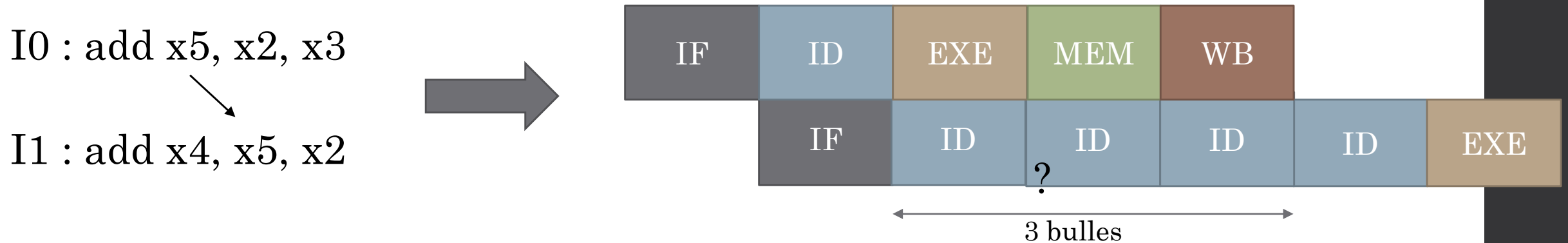


- Malgré l'exécution dans l'ordre, l'organisation en pipeline révèle la dépendance producteur-consommateur

I1 doit lire x5 dans ID **après** que I0 ait écrit x5 dans WB

# Dépendances de données

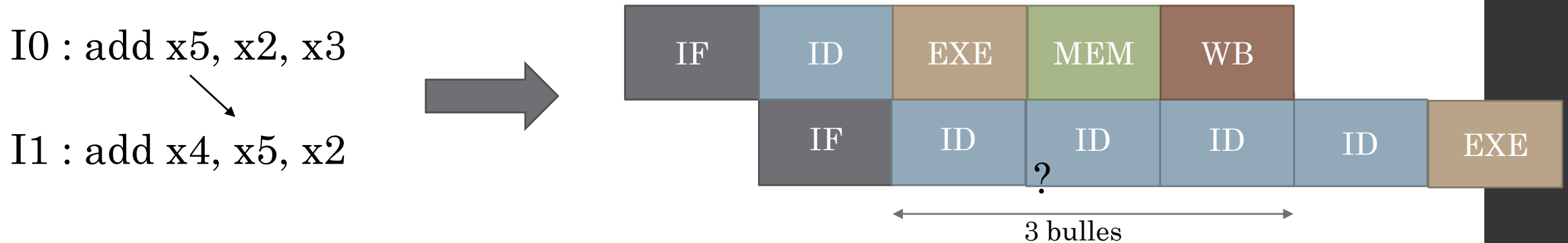
- Le pipeline exécute les instructions dans l'ordre
  - Est-ce suffisant pour respecter les dépendances de données producteur-consommateur ?



- Non ! On bloque le pipeline pour respecter la dépendance
  - On découvrira d'autres dépendances à mesure des améliorations du pipeline

# Dépendances de données

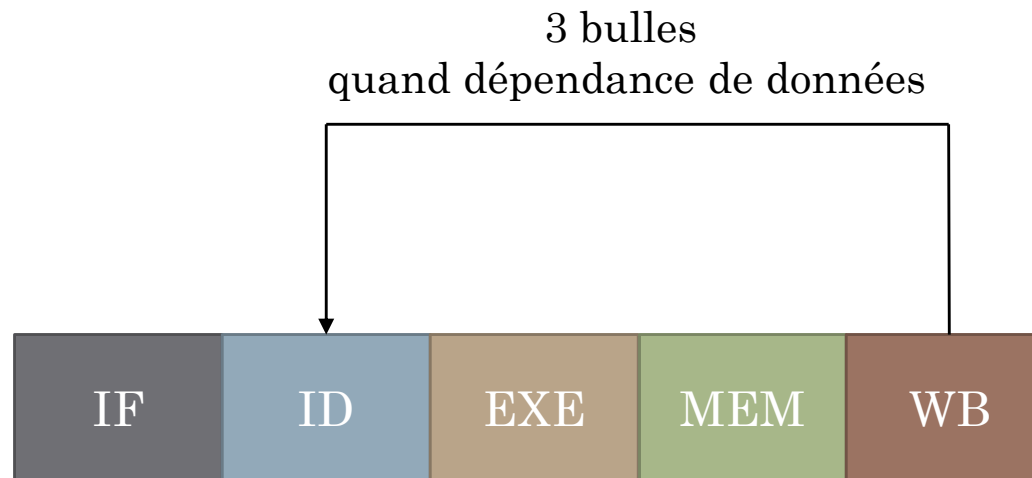
- Le pipeline exécute les instructions dans l'ordre
  - Est-ce suffisant pour respecter les dépendances de données producteur-consommateur ?



- Au pire, toutes les instructions dos à dos sont dépendantes :  
0,25 IPC max vs. 1 IPC théorique

# Dépendances de données

- 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

- Si la microarchitecture, de par sa conception, met en lumière une dépendance, elle doit s'assurer que la dépendance est respectée
  - De manière efficace, si possible
- Beaucoup de microarchitectures différentes possibles
  - Objectif n'est pas de toutes les connaître par cœur
  - Pour une uarch donnée, savoir identifier quels *chemins* matériels font apparaître des dépendances architecturales à respecter

# Dépendances de données

- Implémentation sans pipeline ? 1 cycle pour traiter une instruction
  - Dépendance prod-cons couverte par l'exécution dans l'ordre : prod cycle 1, cons cycle 2
  - Oui mais...Performance théorique limitée :
    - Sans pipeline : 1 IPC @ freq
    - Avec pipeline : 0,25 IPC @ **5 \* freq**
- Mieux vaut essayer de limiter l'impact des ces « nouvelles » dépendances dans l'implémentation pipelinée



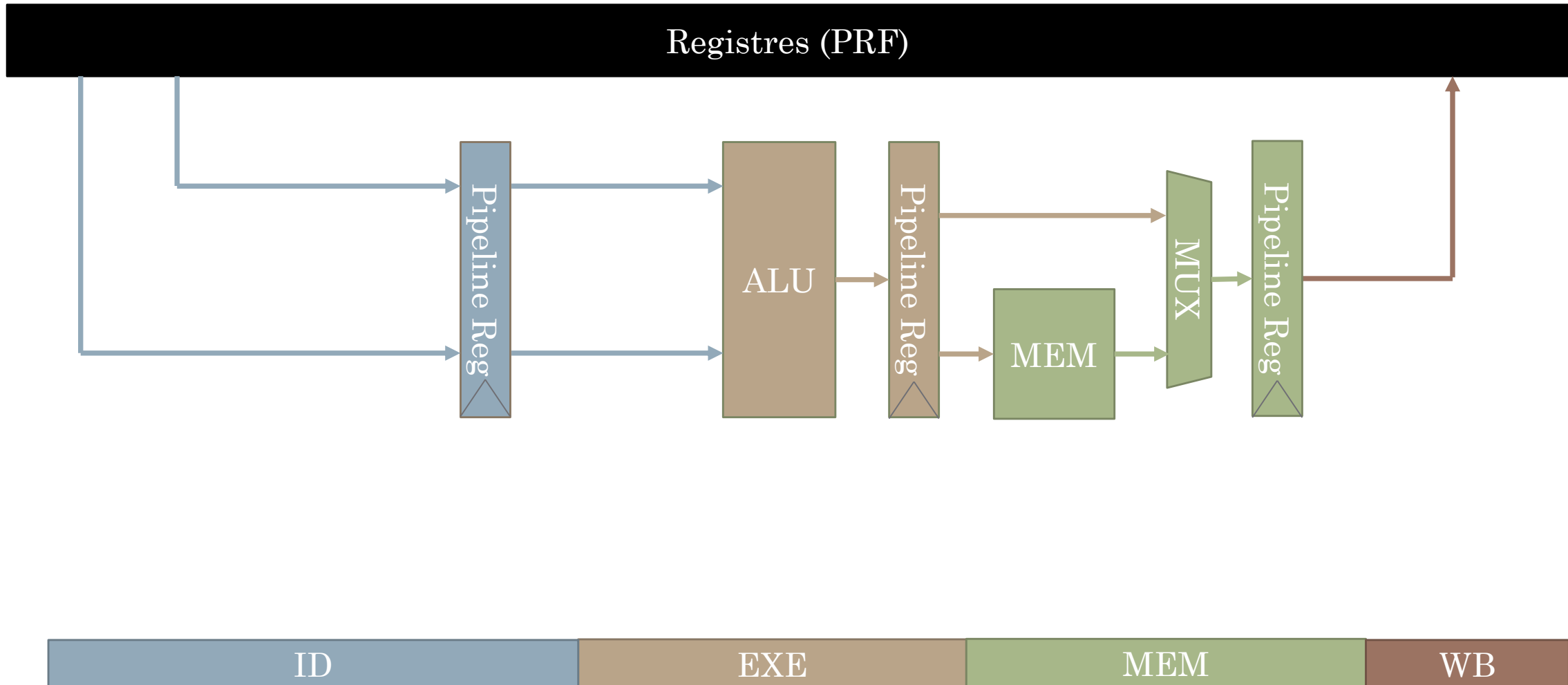
# Dépendances de données – $\mu$ arch.

- Si la microarchitecture, de par sa conception, met en lumière une dépendance, elle doit s'assurer que la dépendance est respectée
  - De manière efficace, si possible
- Réseau de bypass pour permettre à deux instructions de s'exécuter dos à dos

$\mu$ arch sans bypass	$\mu$ arch avec bypass
consommateur doit lire <i>reg</i> dans ID <b>après</b> que producteur ait écrit <i>reg</i> dans WB = 3 bulles entre producteur et consommateur	Un mécanisme stocke <i>reg</i> pendant trois cycles dès qu'il est produit par consommateur dans EXE. Consommateur peut lire <i>reg</i> depuis ce mécanisme à son entrée dans EXE. = 0 bulle entre producteur et consommateur

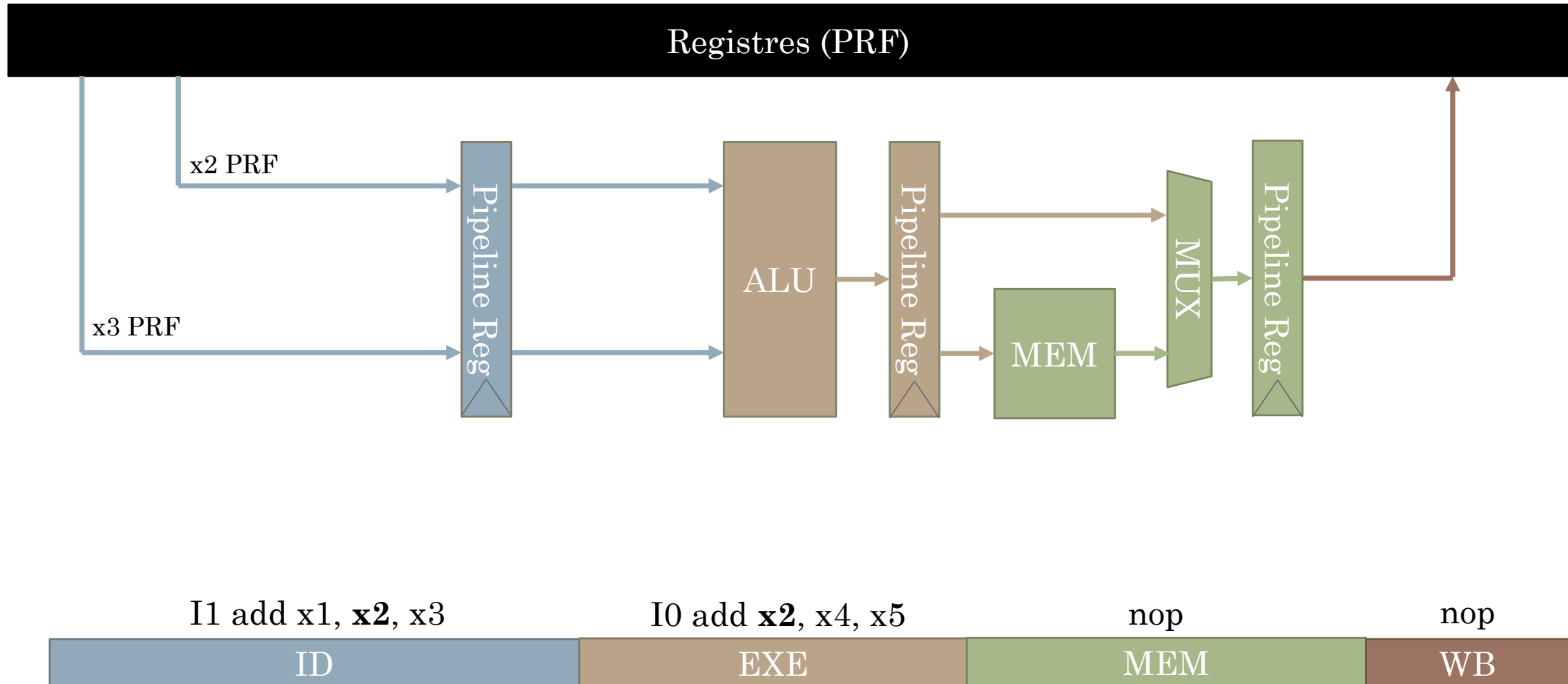
# Dépendances de données – Réseau de bypass

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



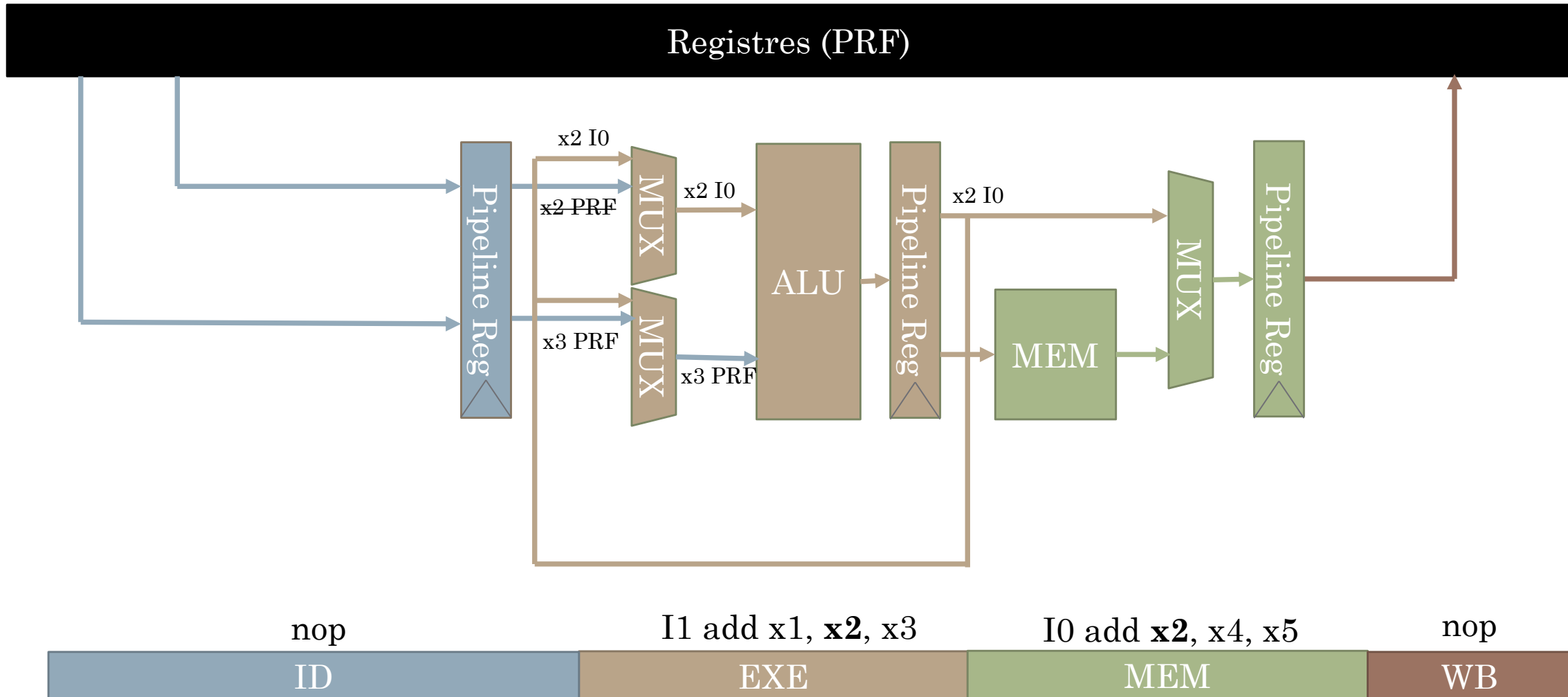
# Dépendances de données – Réseau de bypass

- Cas 1 : Producteur/consommateur dos à dos



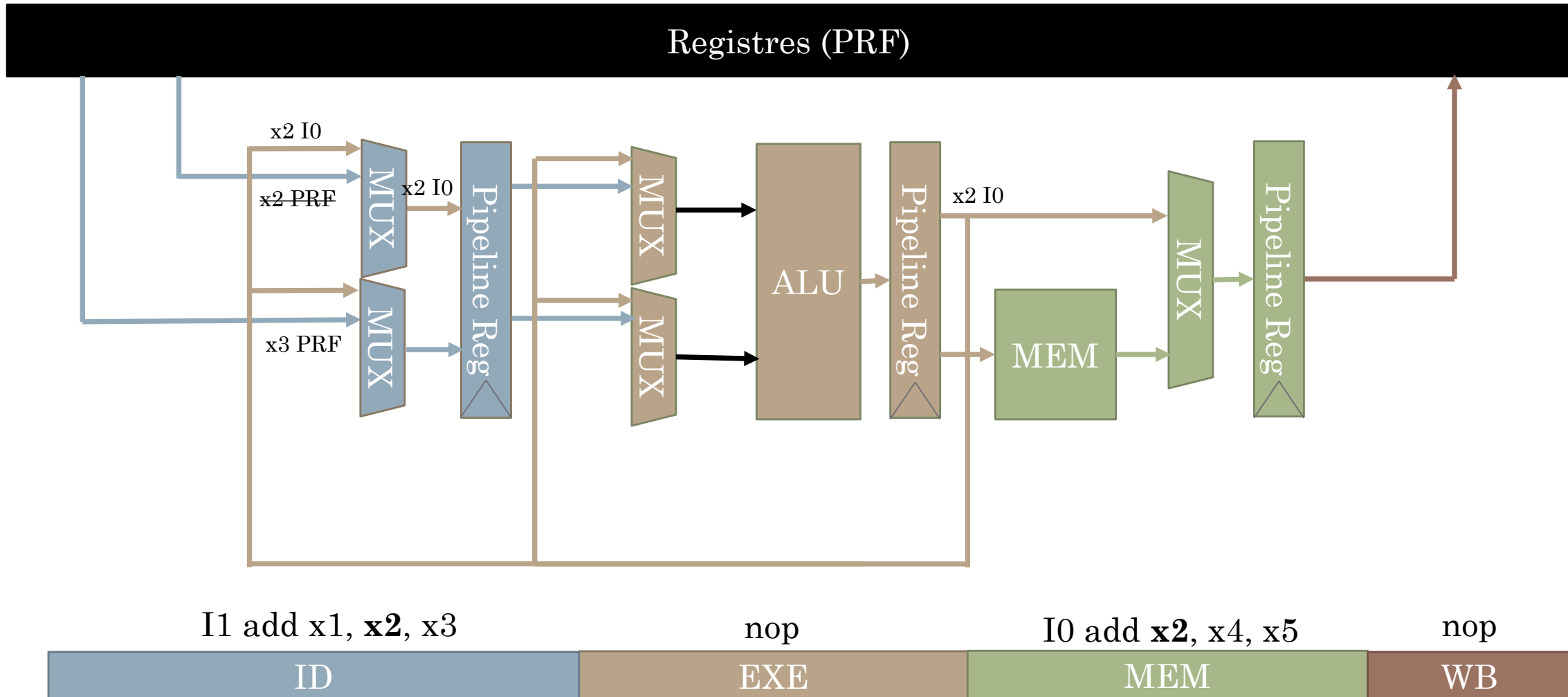
# Dépendances de données – Réseau de bypass

- Cas 1 : Producteur/consommateur dos à dos



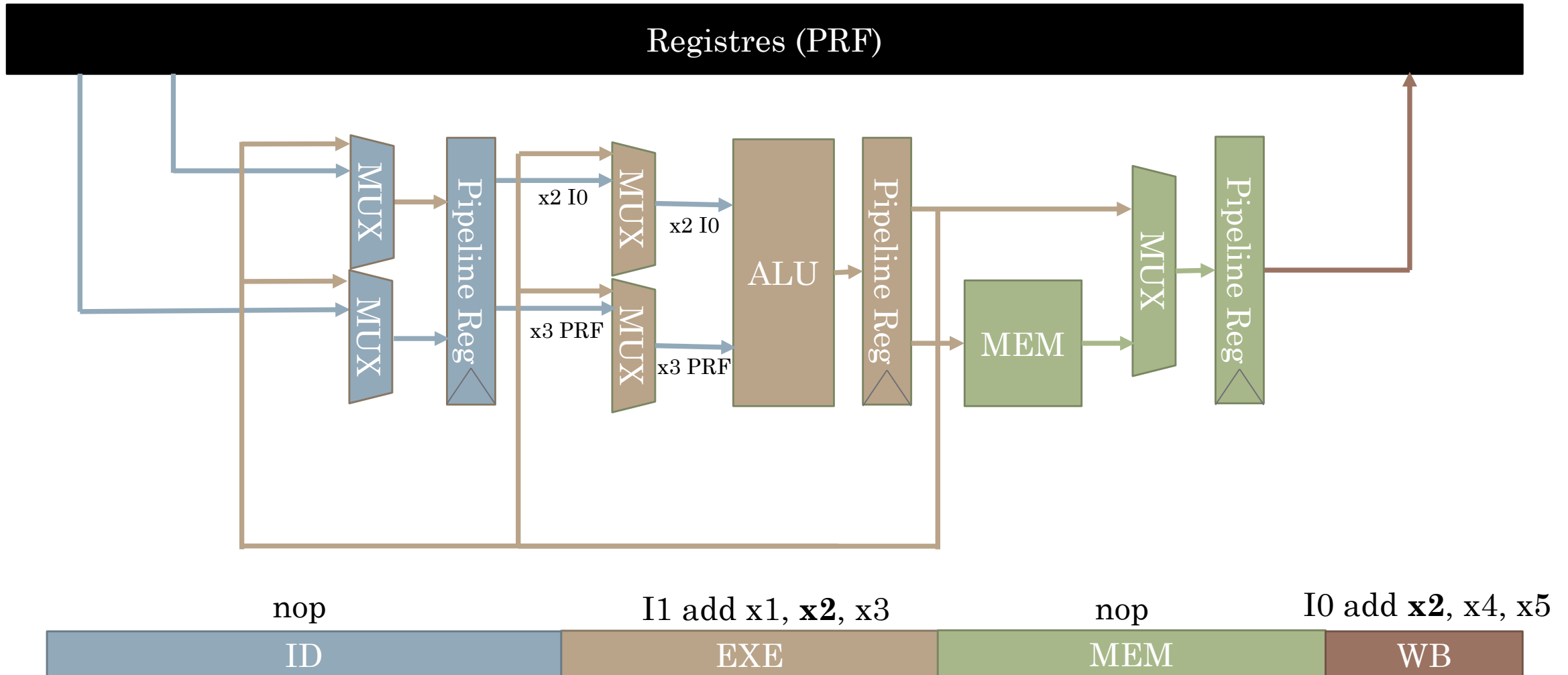
# Dépendances de données – Réseau de bypass

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



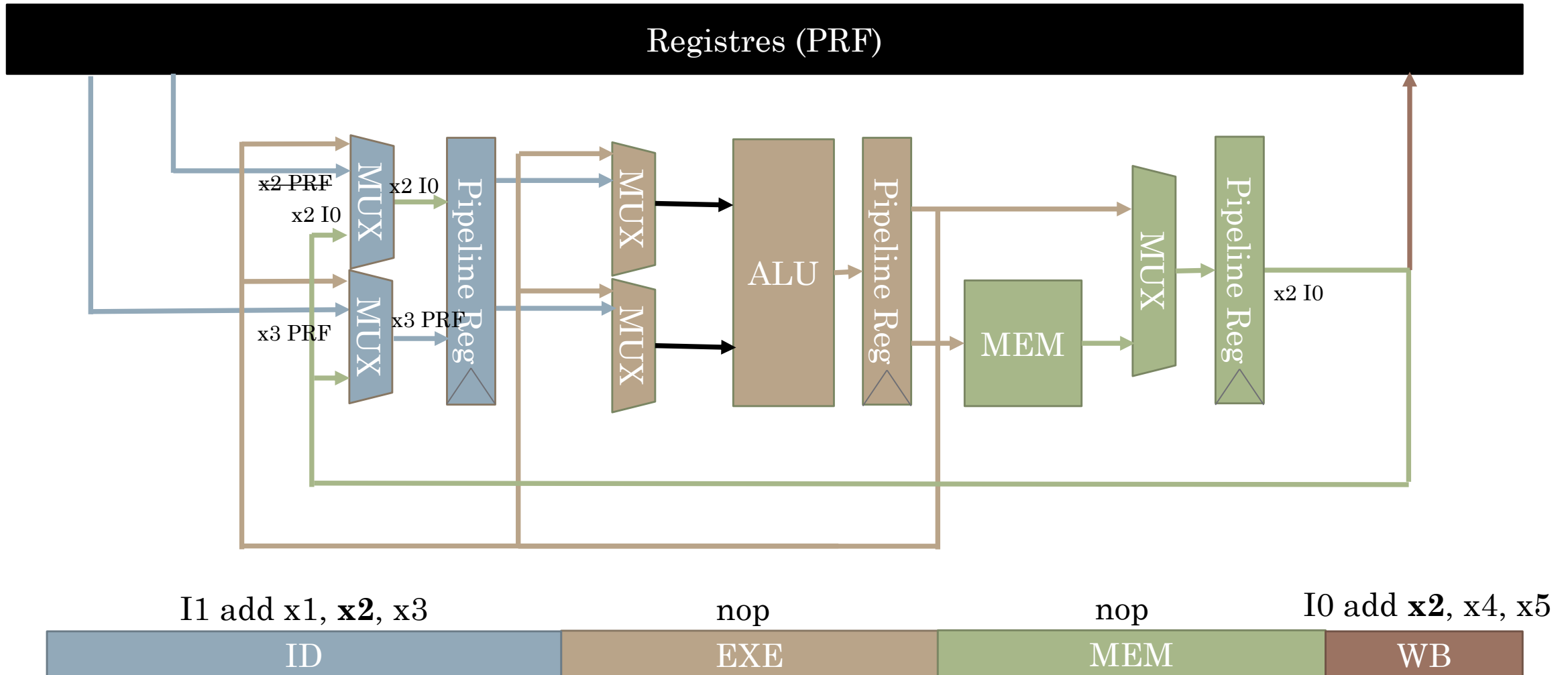
# Dépendances de données – Réseau de bypass

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



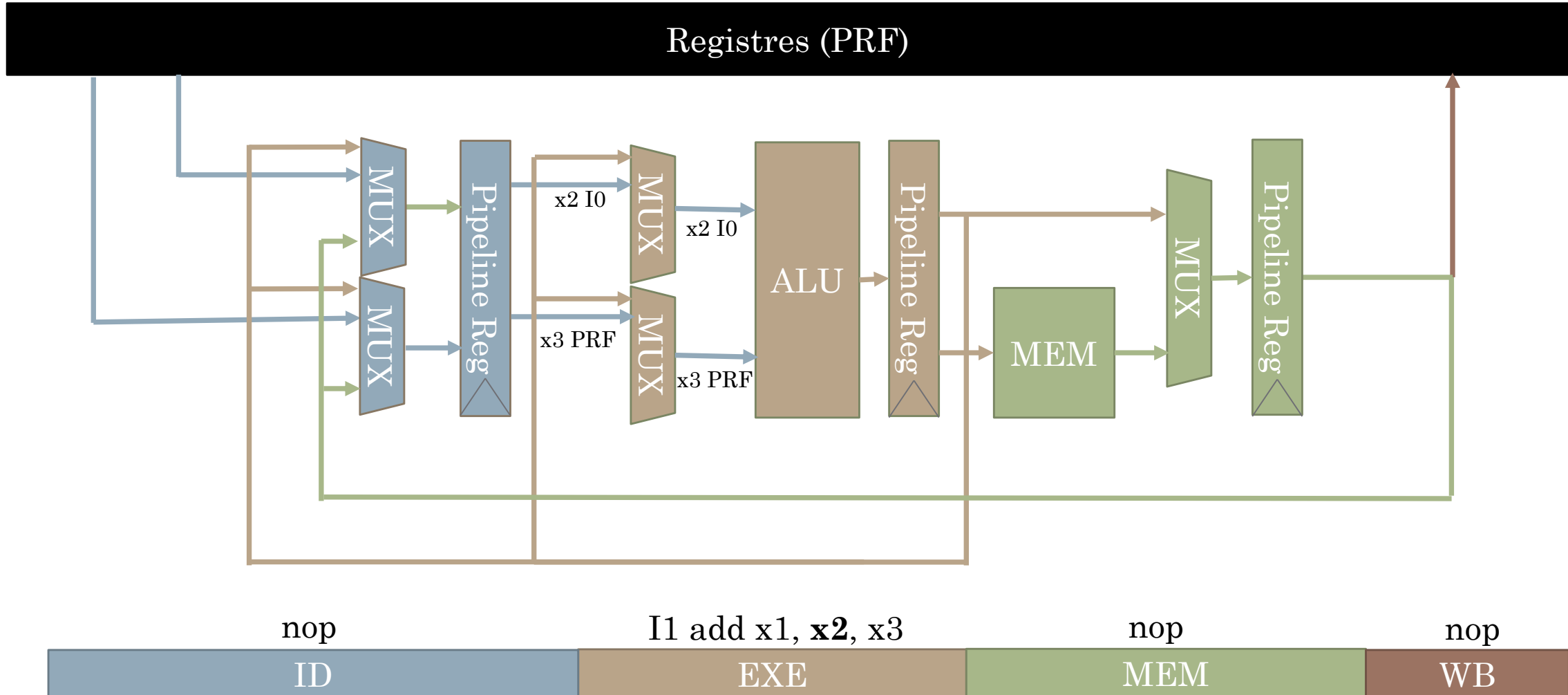
# Dépendances de données – Réseau de bypass

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



# Dépendances de données – Réseau de bypass

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





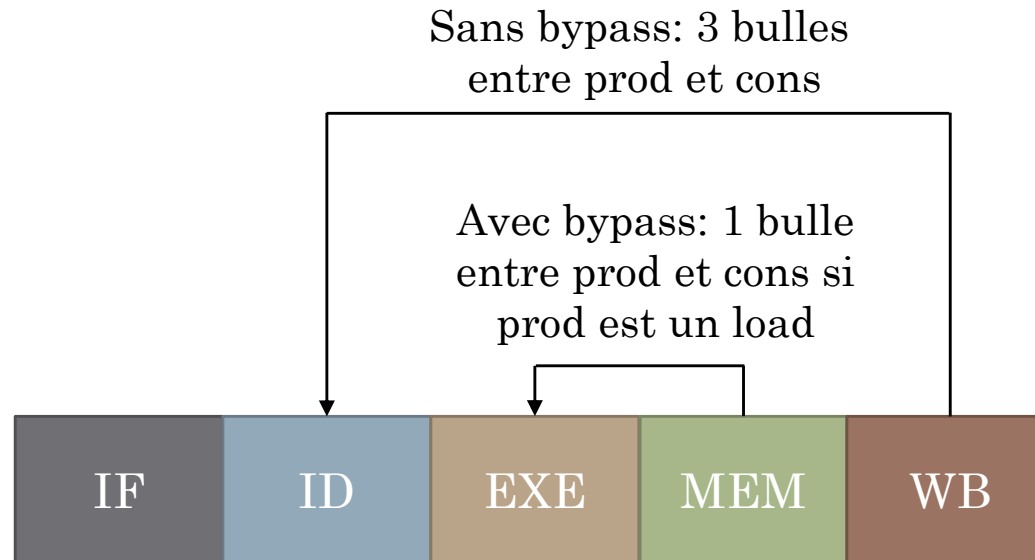
# Dépendances de données – Réseau de bypass

- A-t-on géré tous les cas ?



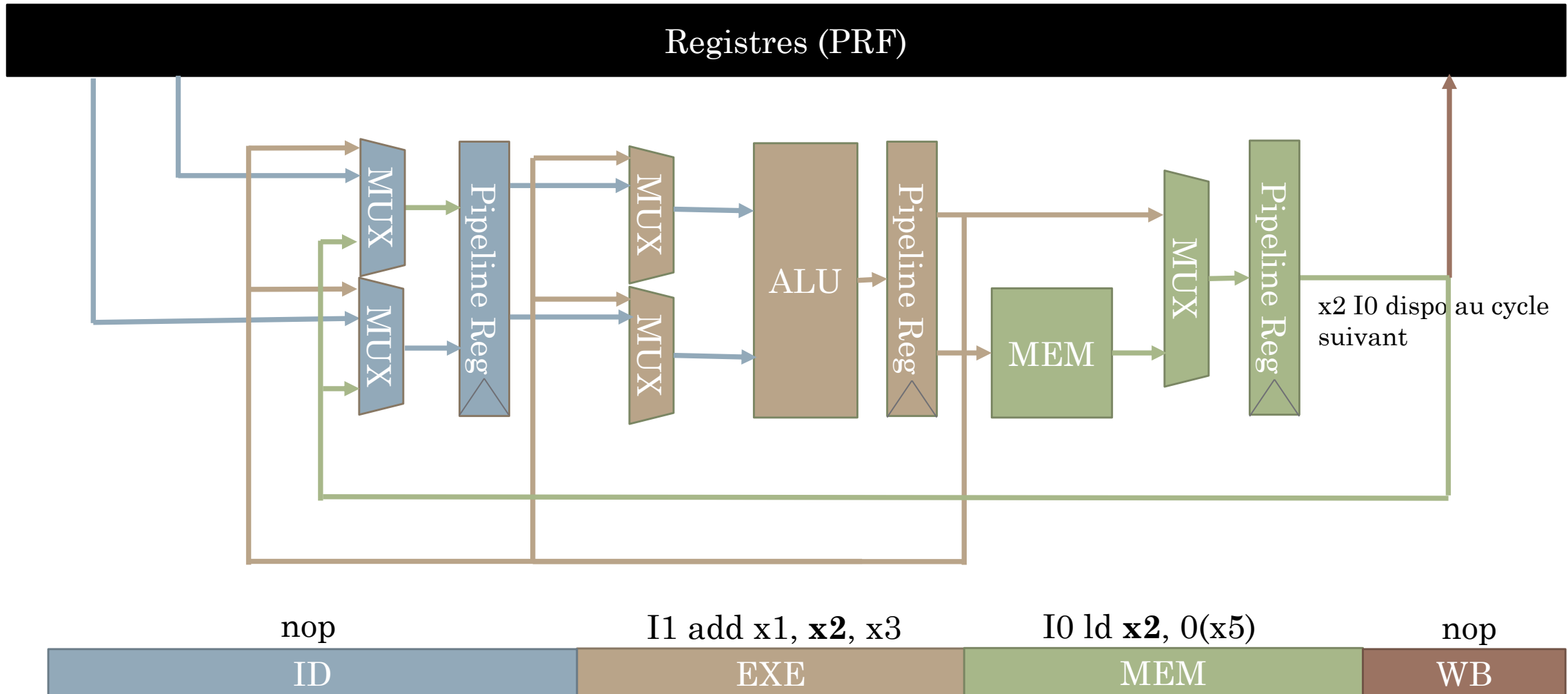
# Dépendances de données – Réseau de bypass

- On retrouve la notion de boucle microarchitecturale



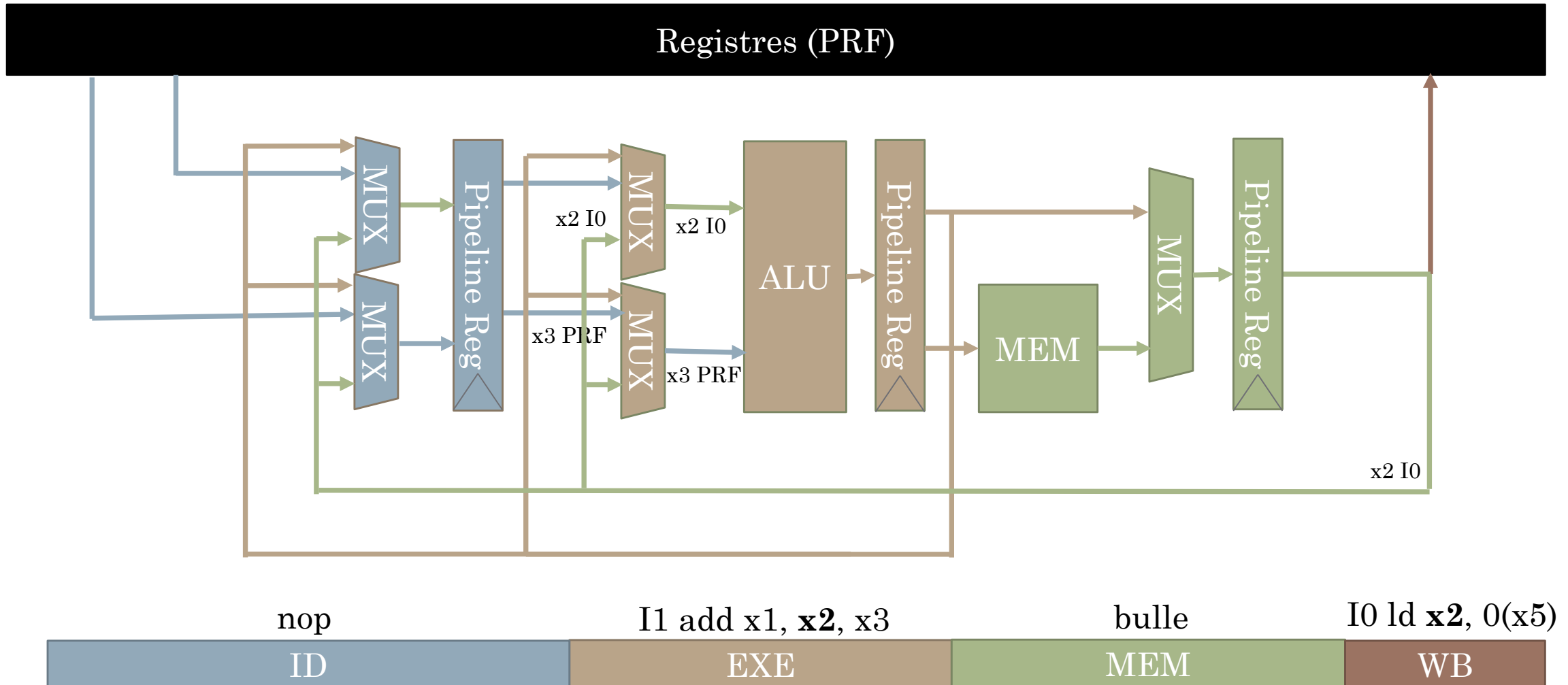
# Dépendances de données – Réseau de bypass

- Cas 4 : Chargement mémoire et consommateur dos à dos



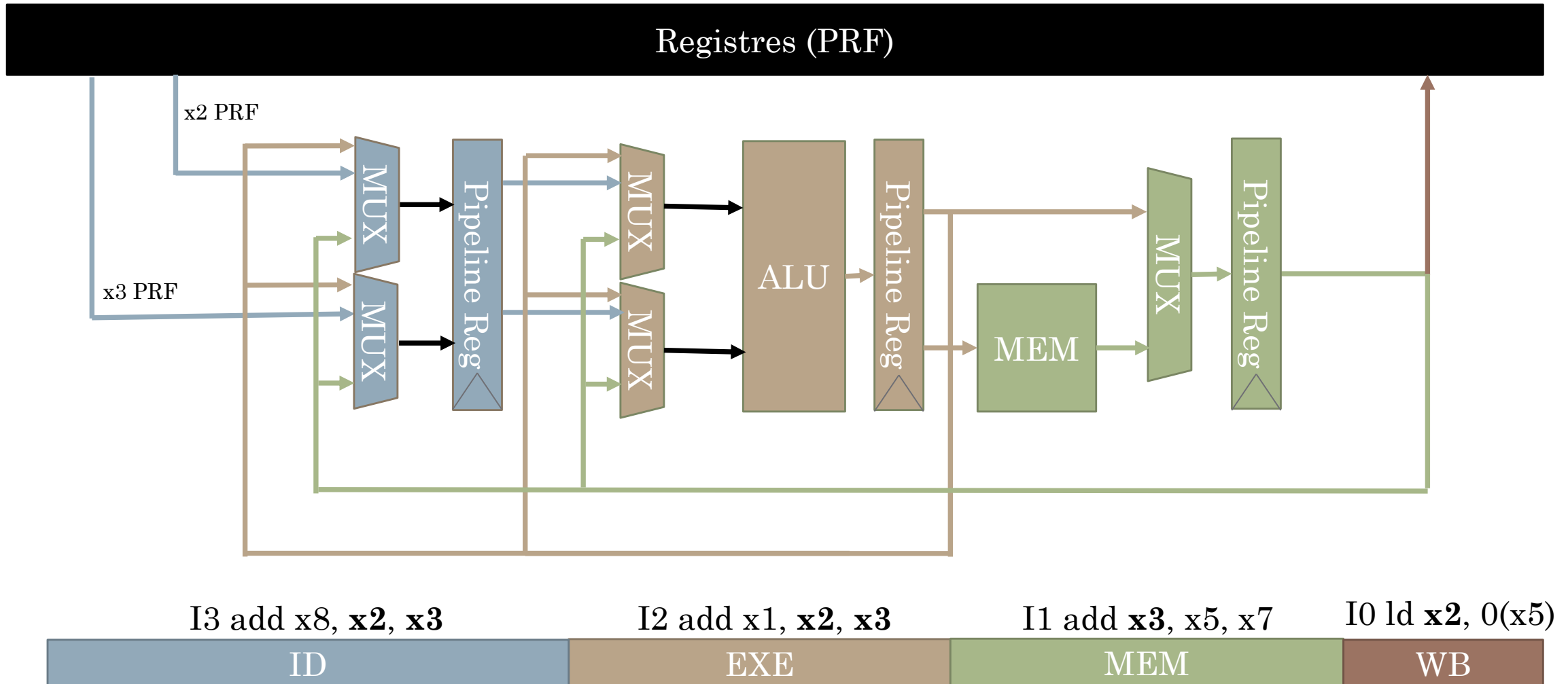
# Dépendances de données – Réseau de bypass

- Cas 4 : Chargement mémoire et consommateur dos à dos



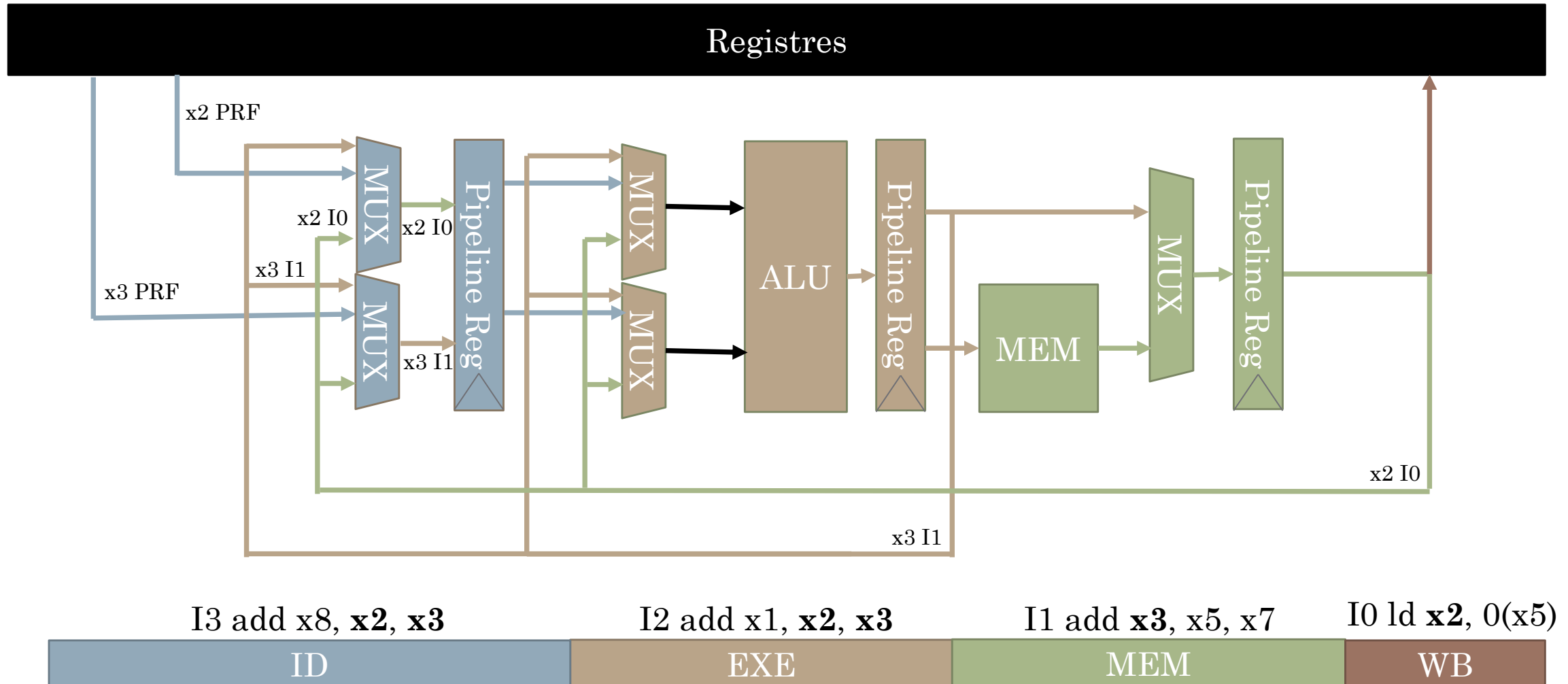
# Dépendances de données – Réseau de bypass

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



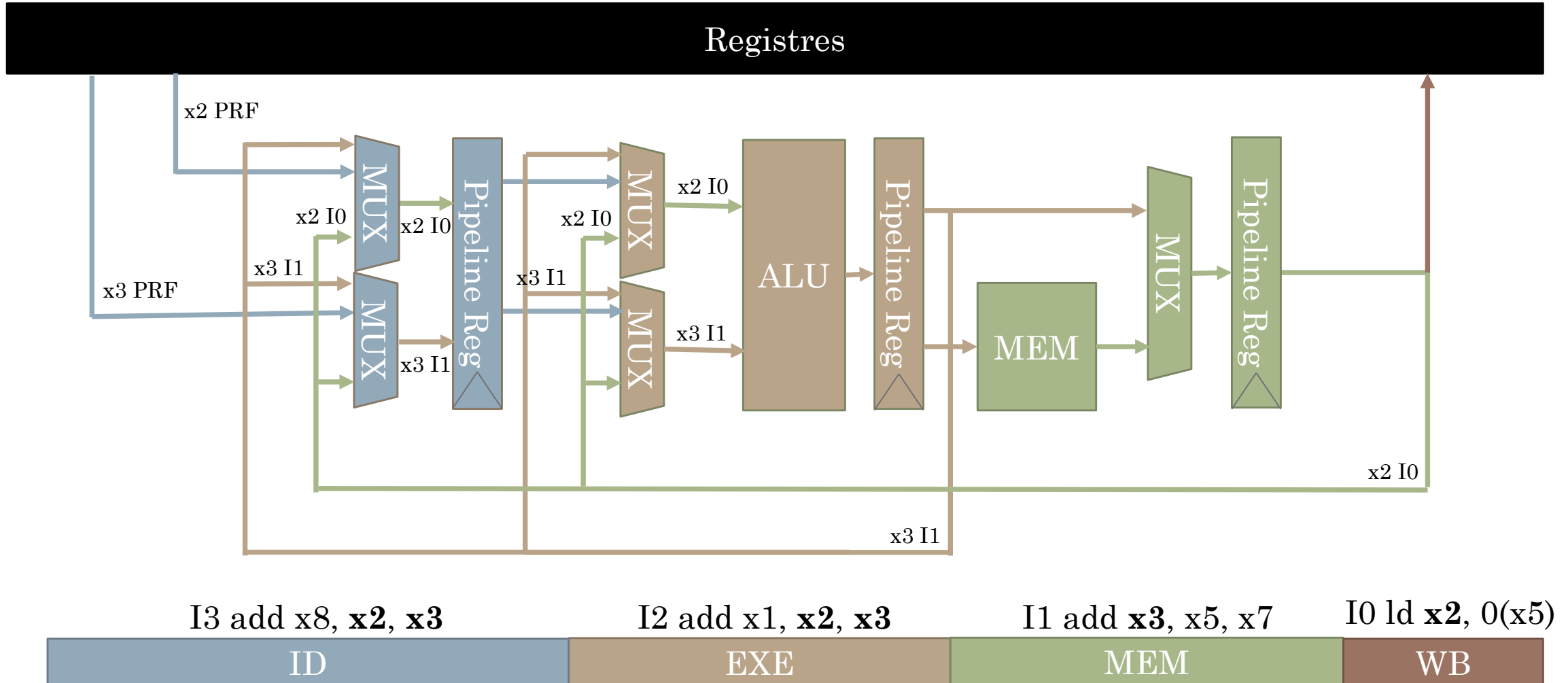
# Dépendances de données – Réseau de bypass

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



# Dépendances de données – Réseau de bypass

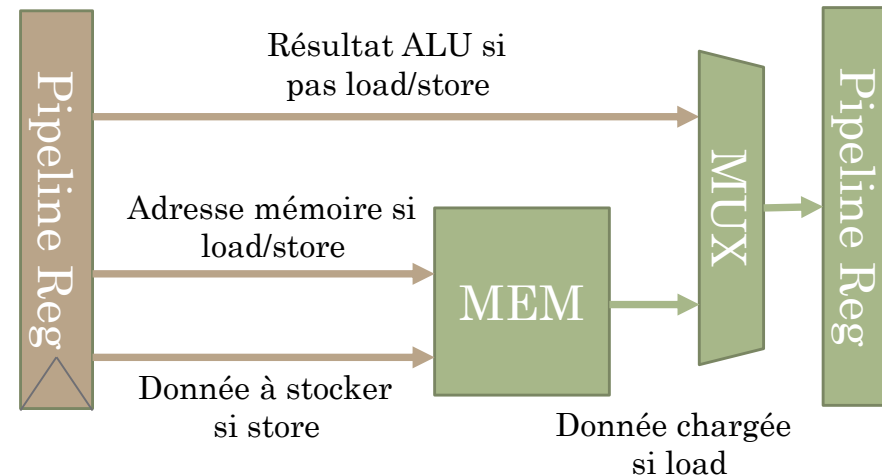
- Cas 5&6 (ld + consommateur avec un ou deux cycles d'écart) : couverts par les chemins existant





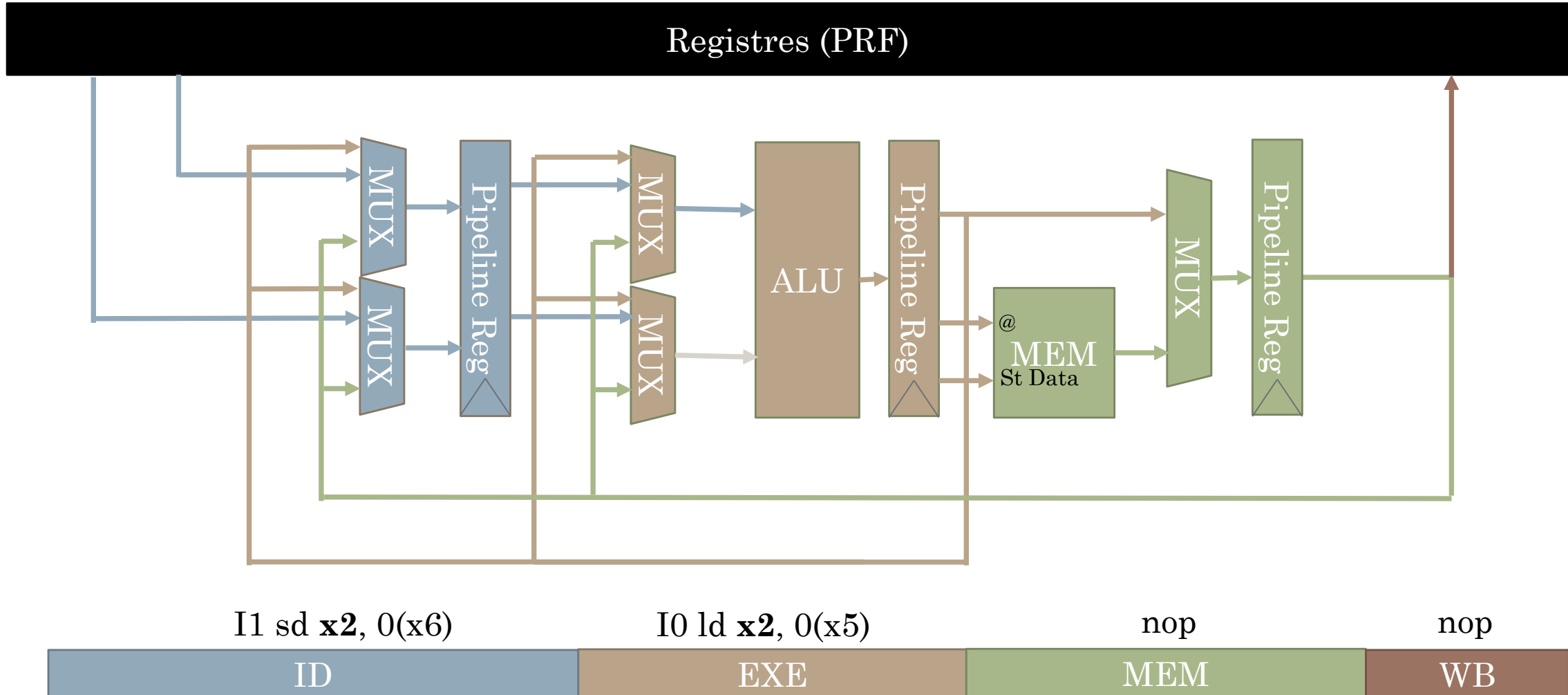
# Dépendances de données – Réseau de bypass

- 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 :



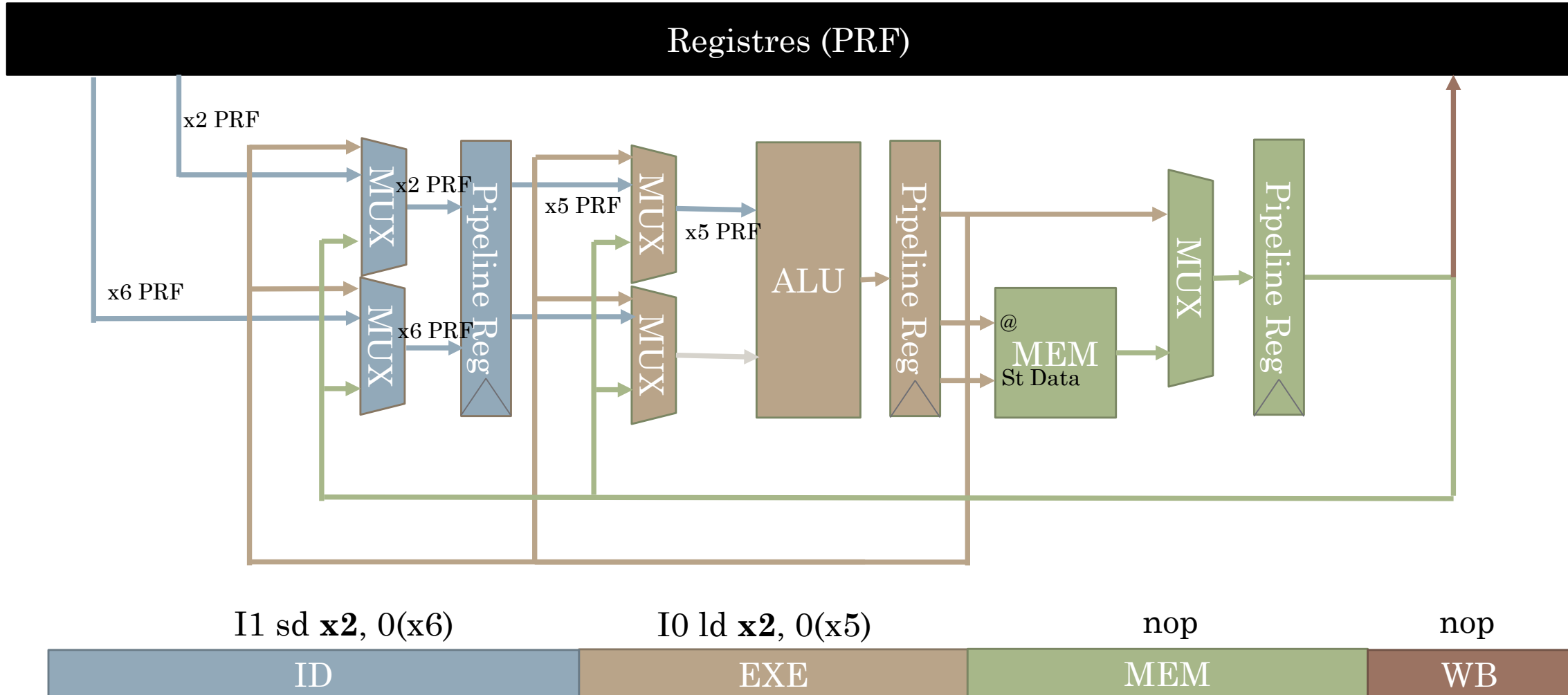
# Dépendances de données – Réseau de bypass

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



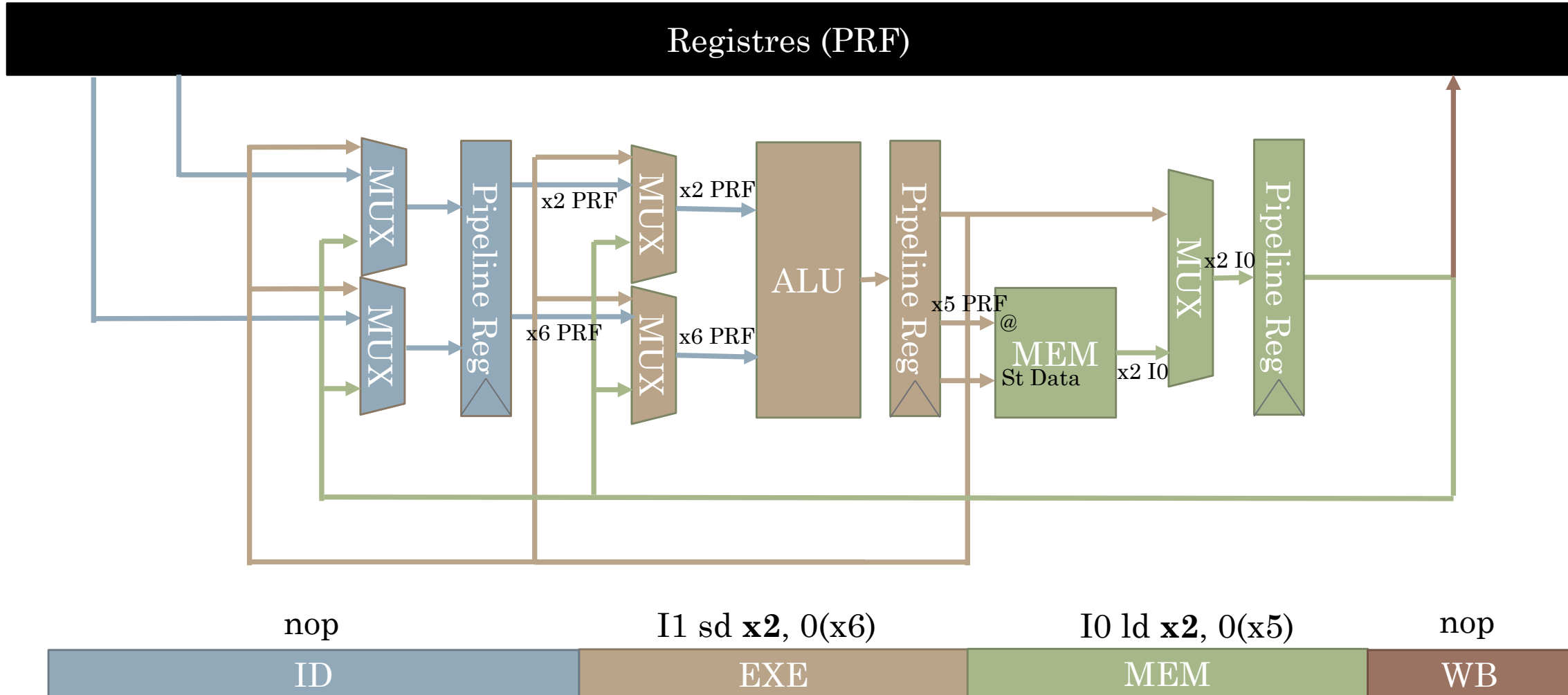
# Dépendances de données – Réseau de bypass

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



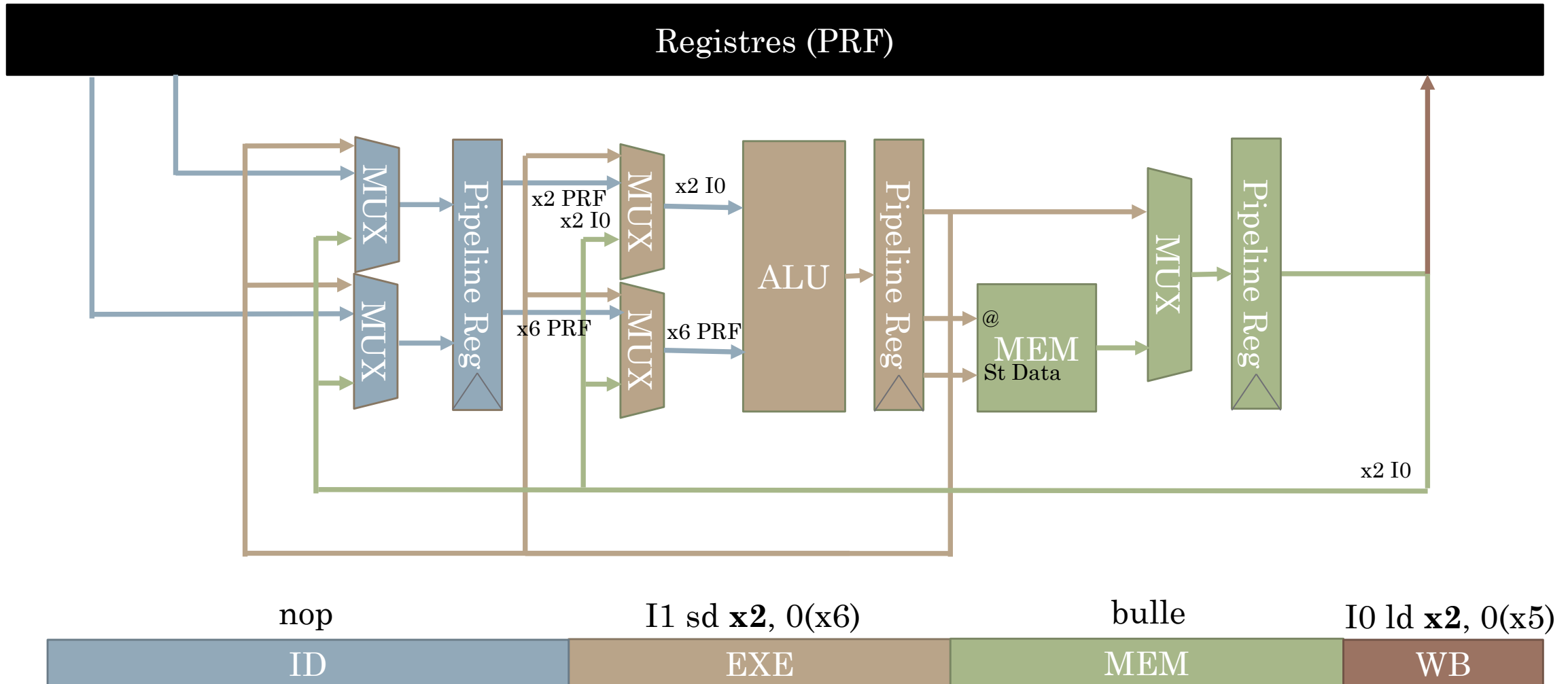
# Dépendances de données – Réseau de bypass

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



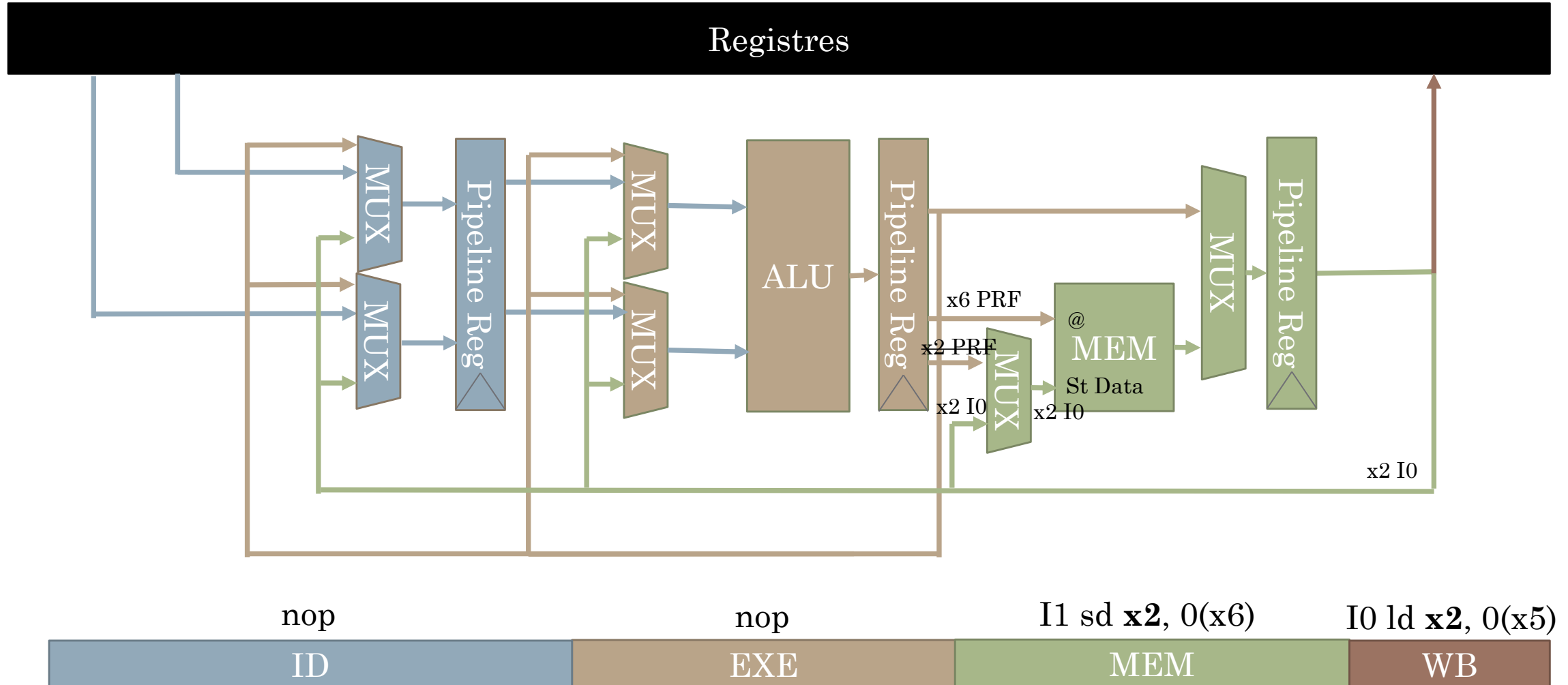
# Dépendances de données – Réseau de bypass

- 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 x2, 0(x1)
sd x2, 0(x3)
addi x1, 8
addi x3, 8
```

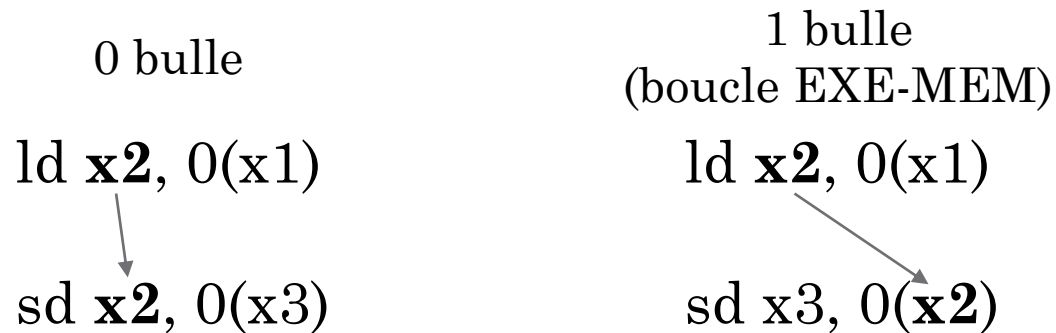
memcpy:

```
ld x2, 0(x1)
addi x1, 8
sd x2, 0(x3)
addi x3, 8
```

**Exercice:** 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 ?)



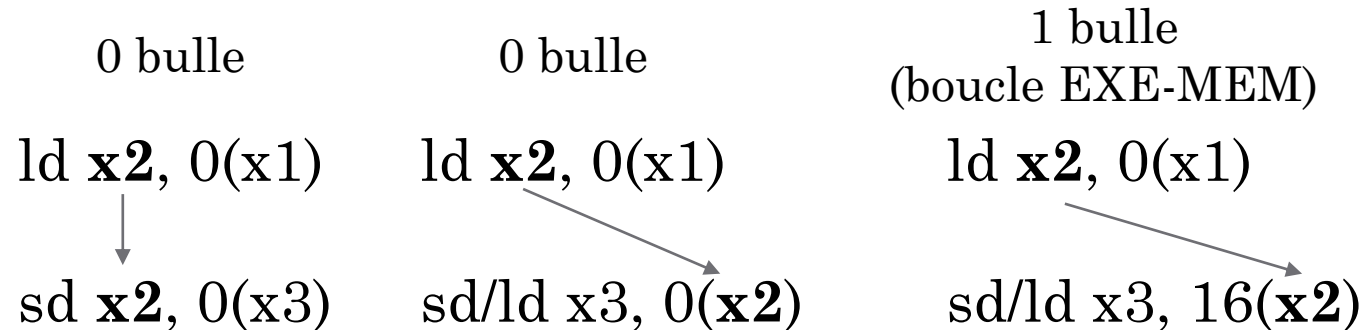


# Dépendances de données – Réseau de bypass

- 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...

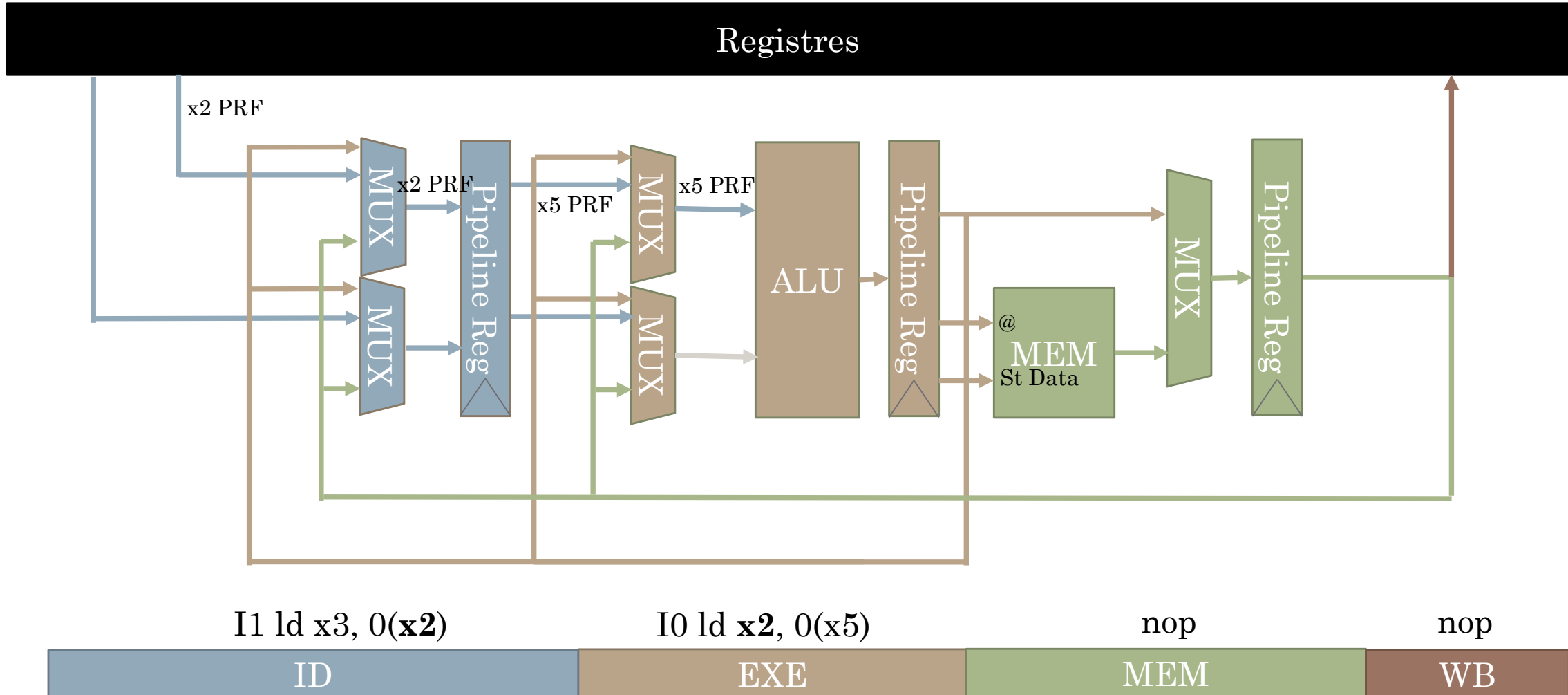
# Dépendances de données – Réseau de bypass

- 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



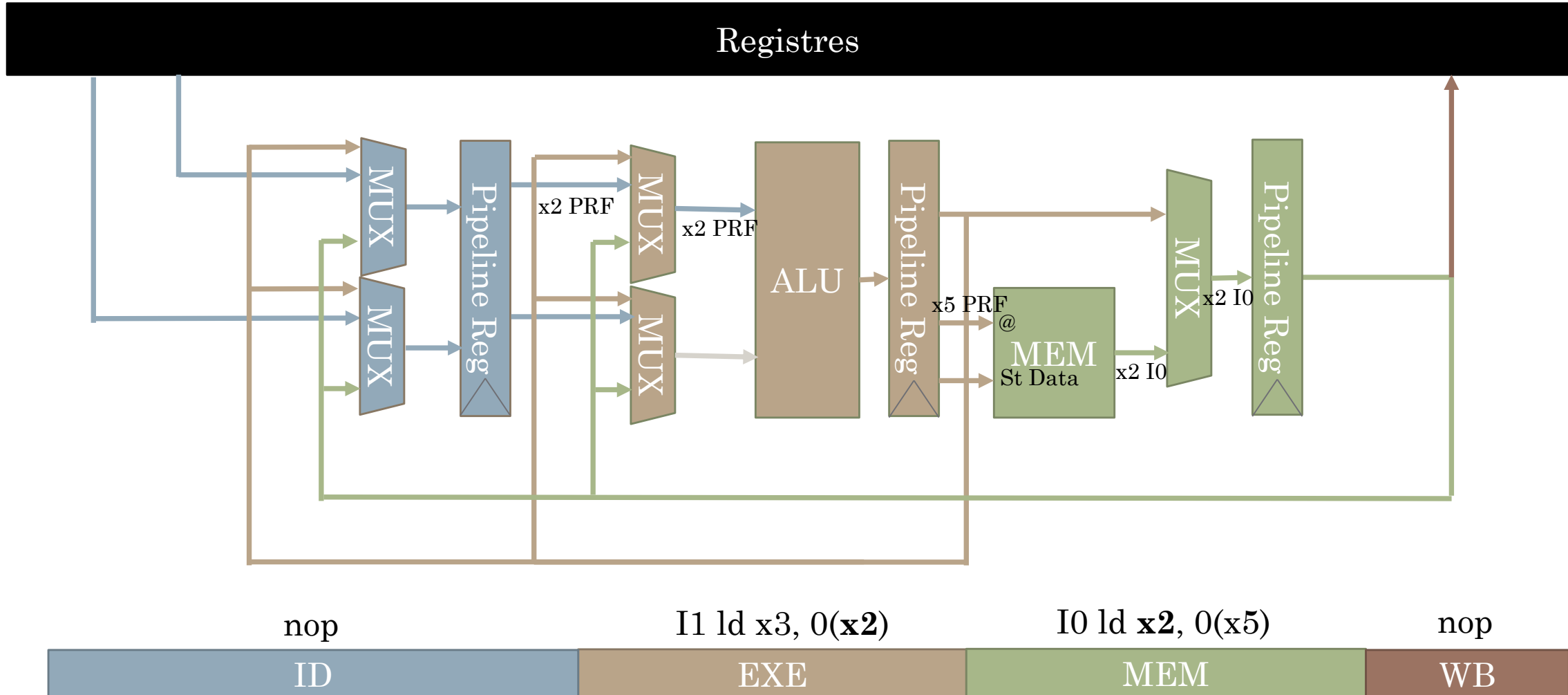
# Dépendances de données – Réseau de bypass

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



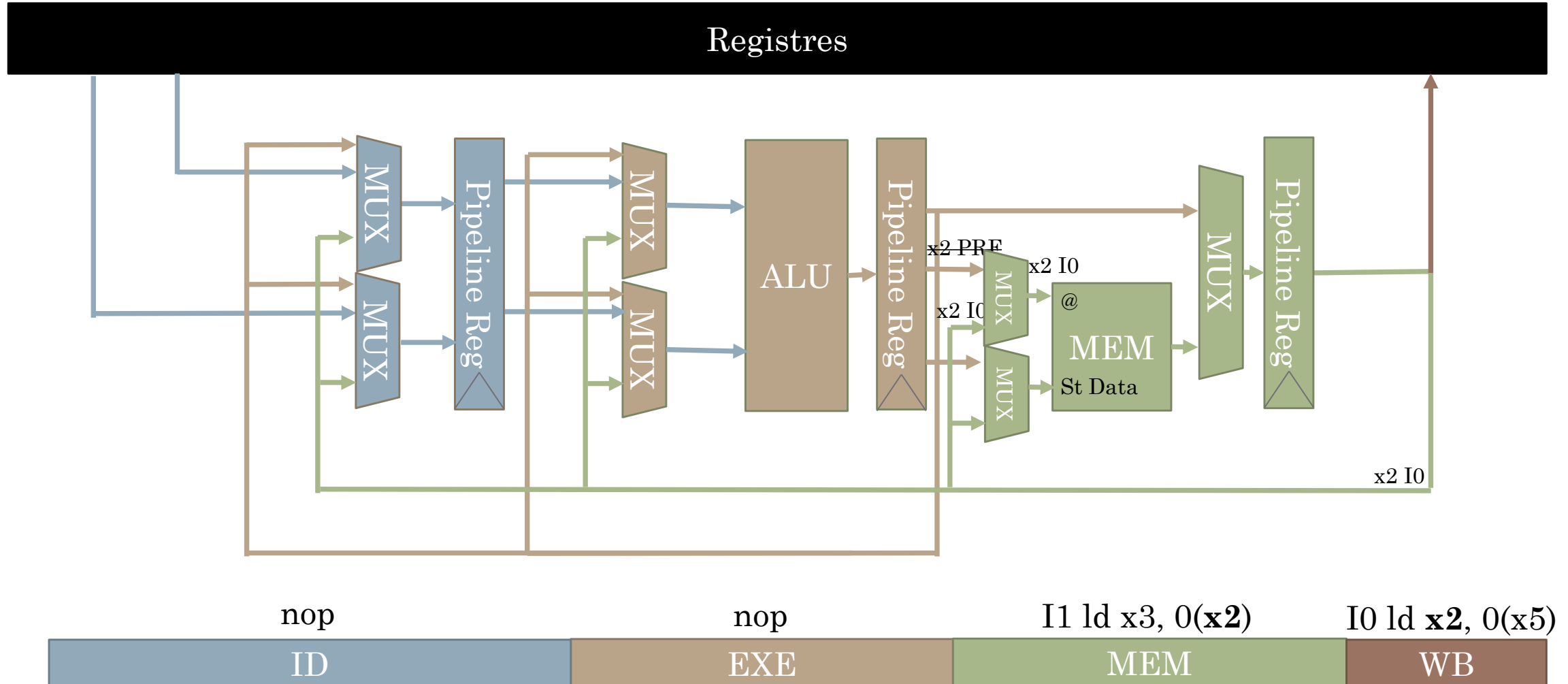
# Dépendances de données – Réseau de bypass

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



# Dépendances de données – Réseau de bypass

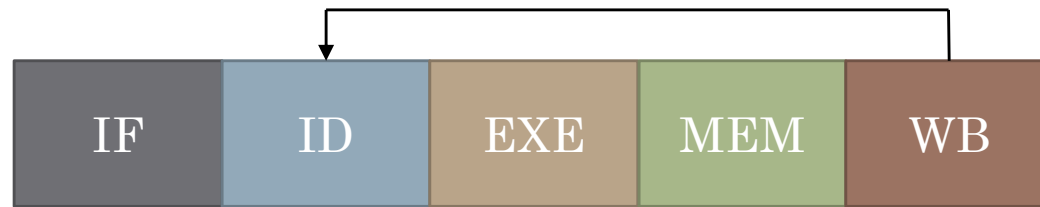
- Cas 8 : Load (producteur) puis load/store consommateur pour **adresse** avec **immédiat 0**, dos à dos



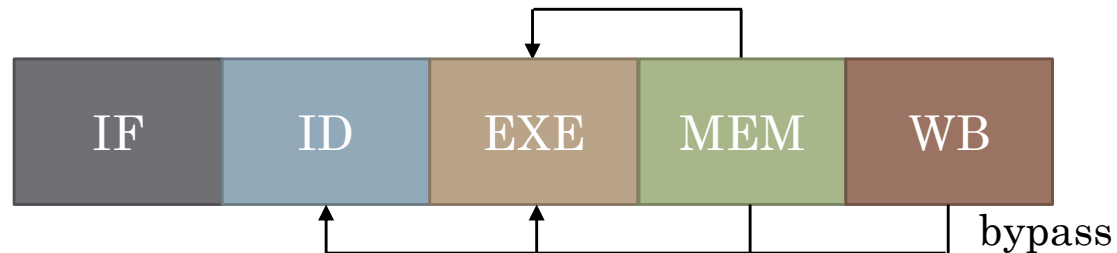
# Dépendances de données – Réseau de bypass

- Impact du respect des dépendances de données réduit 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



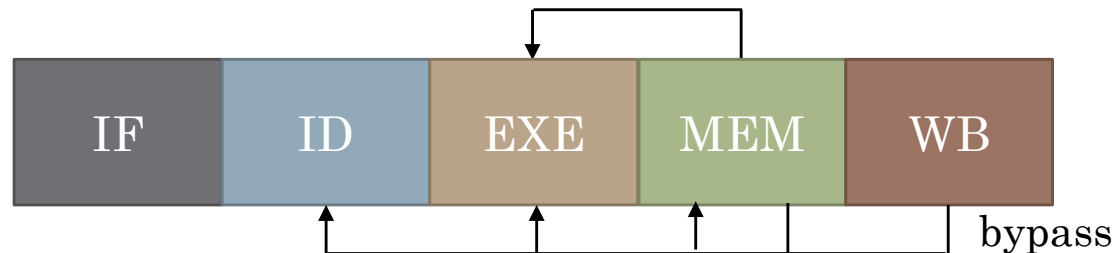
Avec bypass 2<sup>ème</sup> version : 1 bulle  
entre prod et cons si  
prod est un load

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
    - Une instruction s'exécute lorsque toutes ses dépendances sont satisfaites

# Branchements

- Rappel
  - Branchement non conditionnel : Toujours pris

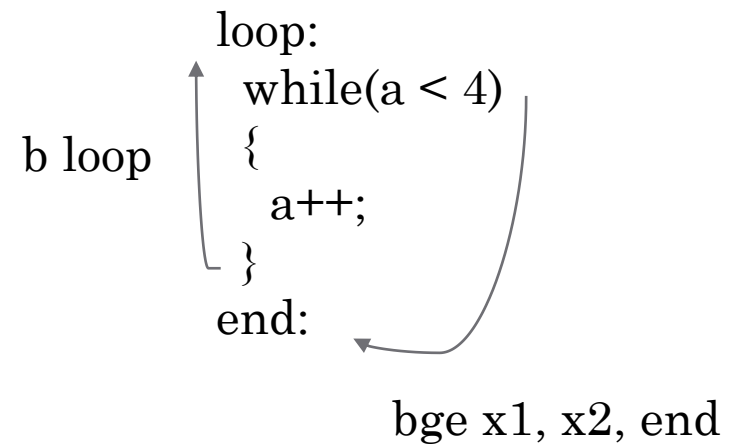
```
loop:
  while(a < 4)
  {
    a++;
  }
end:
```

b loop



# Branchements

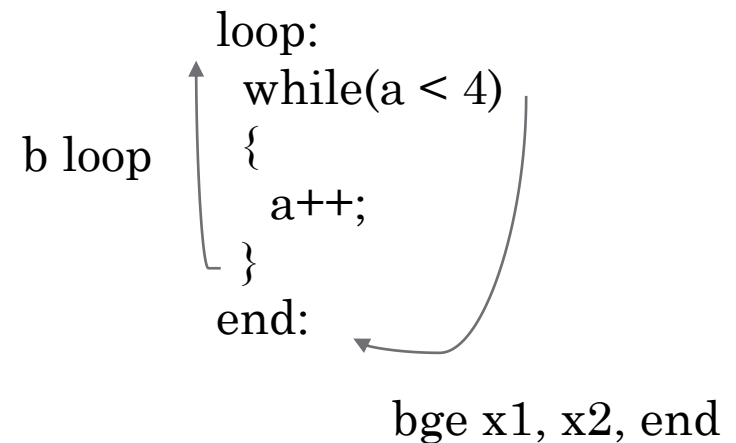
- Rappel
  - Branchement non conditionnel : Toujours pris
  - Branchement conditionnel : Pris si condition vraie



# Branchements

- Rappel

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

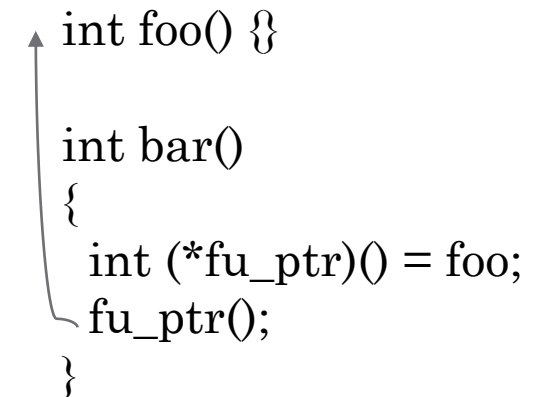


# Branchements

- 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 x1



The diagram illustrates an indirect branch instruction. On the left, the instruction 'jalr x1' is shown. A curved arrow originates from the 'x1' register and points to the 'foo' function in a code block on the right. The code block contains the following C code:

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

# Dépendances de contrôle

- Gestion de la divergence

**before\_if:**

if(a != 0)

{

  a = 4;

}

**after\_if:**

a = a + 2;



**before\_if:**

I0 : beqz x1, after\_if

I1 : li x1, 4

**after\_if:**

I2 : addi x1, x1, 2

Dépendance de contrôle au niveau *architectural* : L'exécution de I0 détermine si on exécute I1 ou I2

Déjà contenue dans la dépendance d'ordre qui dicte que I1/I2 est exécutée après I0 **même en l'absence** d'une dépendance de contrôle

# Dépendances de contrôle

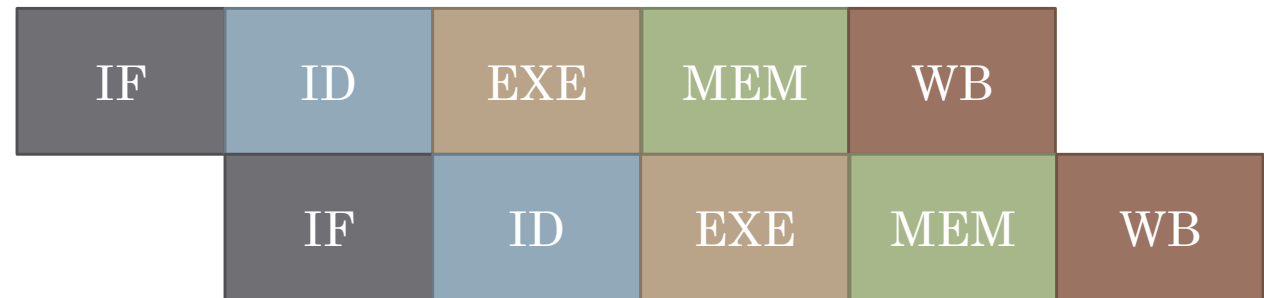
- Respecte-t-on la dépendance de contrôle en exécutant dans l'ordre ?

I0 : beqz x1, after\_if

↓

I1 : li x1, 4

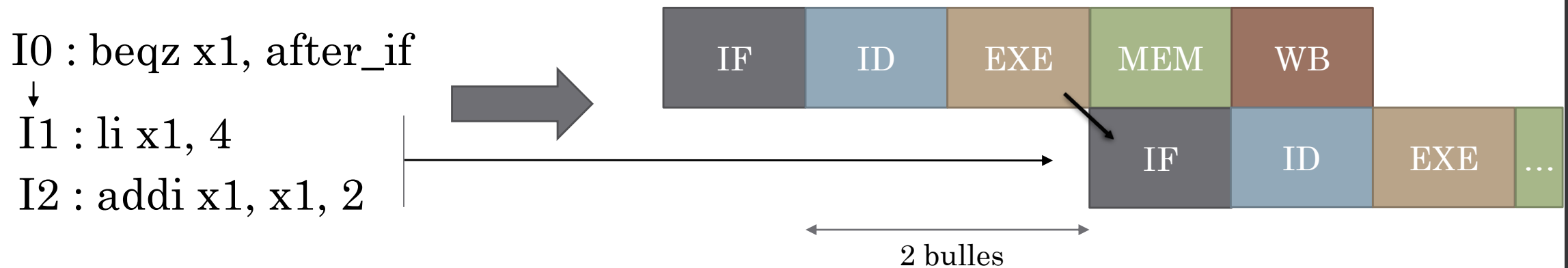
I2 : addi x1, x1, 2



- Non ! On détermine si on doit exécuter I1 ou I2 lorsque I0 sort de EXE, mais on fait rentrer une nouvelle instruction (laquelle ?) lorsque I0 est dans ID

# Dépendances de contrôle

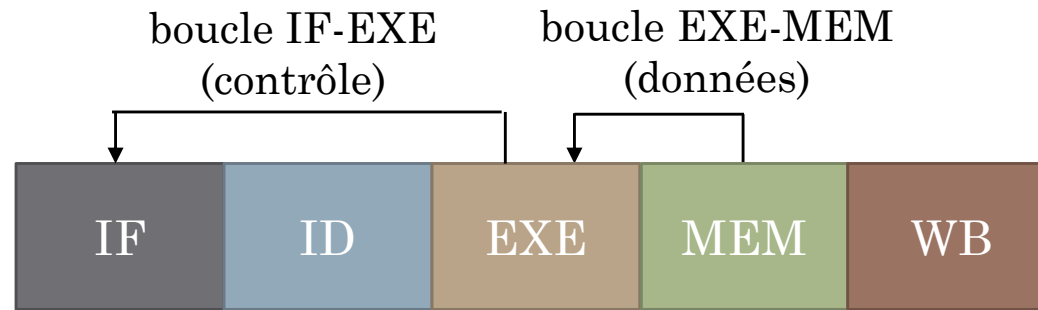
- Respecte-t-on la dépendance de contrôle en exécutant dans l'ordre ?



- Comme pour les dépendances de données, on doit explicitement bloquer le pipeline et attendre que la dépendance de contrôle soit résolue
  - Deux bulles après chaque branchement conditionnel

# 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 ?

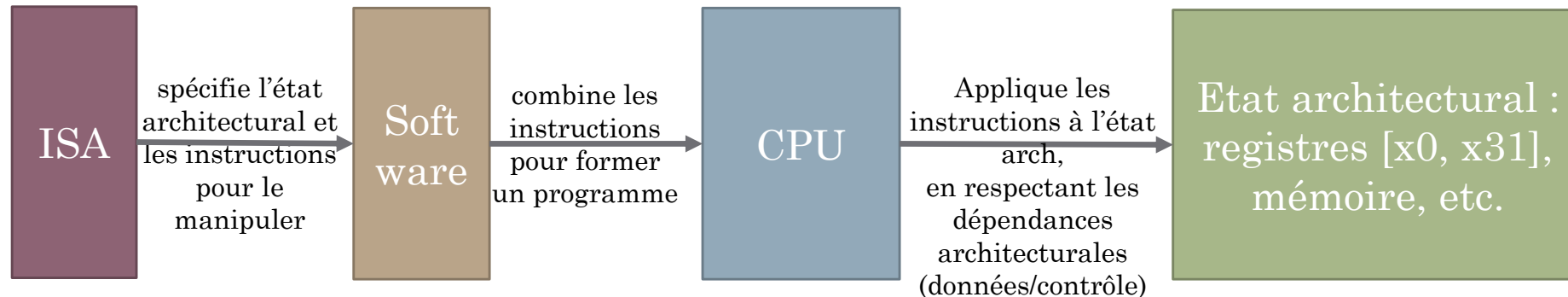
# Dépendances de contrôle – Exécution spéculative

- 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
    - On répare le pipeline si on se trompe
  - Très efficace car les programmes sont réguliers



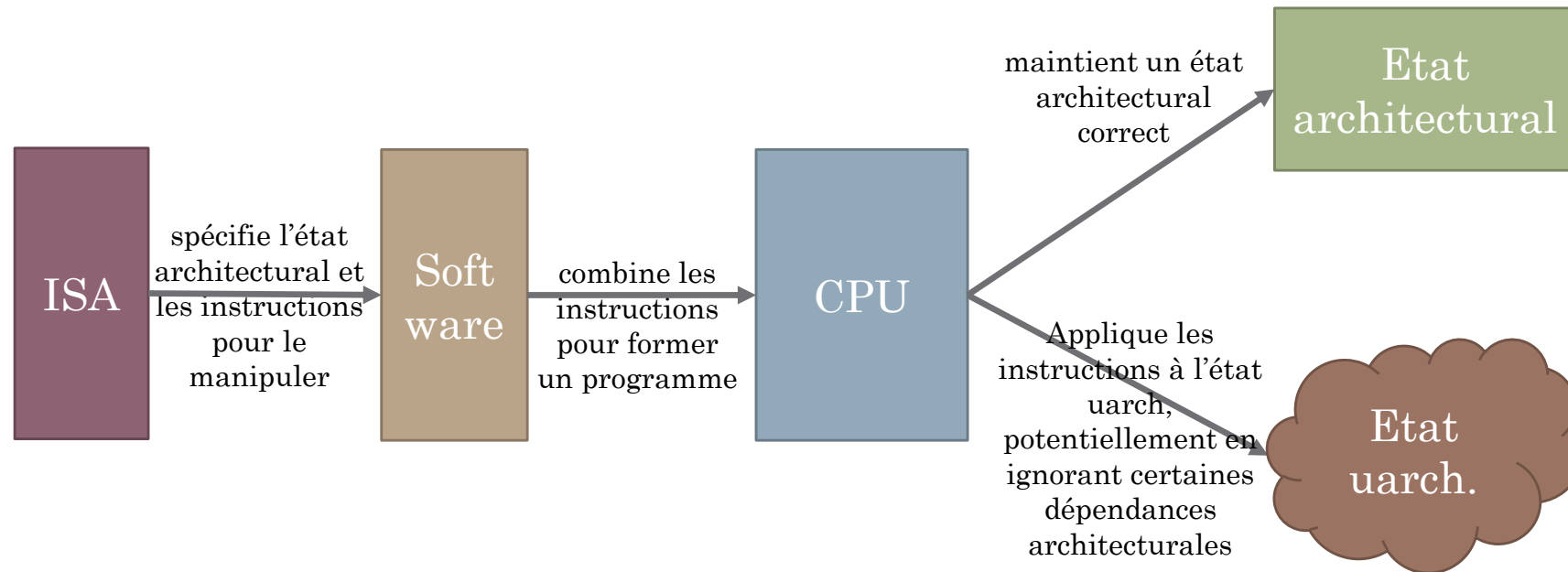
# Etat architectural vs. microarchitectural

- Etat architectural observable et manipulable par le logiciel
  - Manipulations contraintes par la spécification (notamment flot de contrôle correct)



# Etat architectural vs. microarchitectural

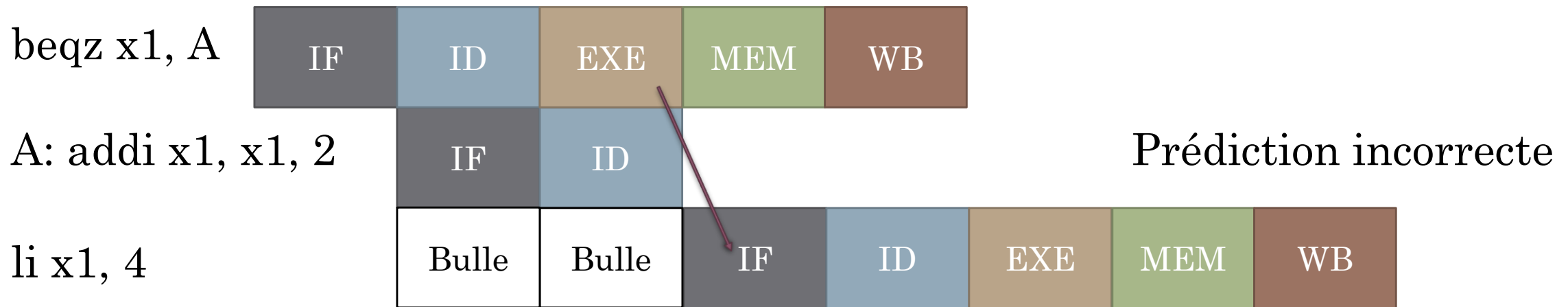
- Etat architectural observable et manipulable par le logiciel
  - Manipulations contraintes par la spécification (notamment flot de contrôle correct)
- Etat microarchitectural **non-observable** par le logiciel
  - Par exemple, les registres de pipeline
  - Donc non tenu de respecter la spécification...
  - ...tant qu'un état architectural **correct** peut-être extrait quand nécessaire



# Etat architectural vs. microarchitectural

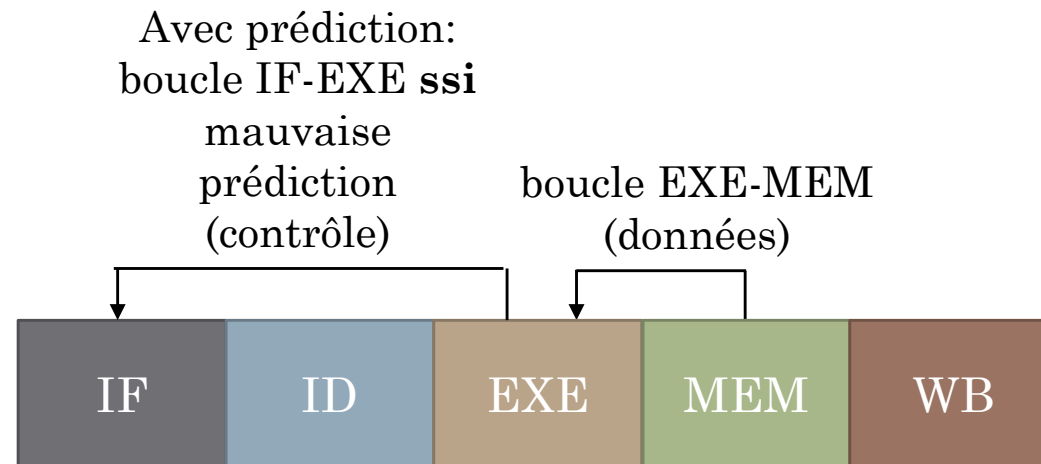
- Etat architectural observable et manipulable par le logiciel
  - Manipulations contraintes par la spécification (notamment flot de contrôle correct)
- Etat microarchitectural **non-observable** par le logiciel
  - Par exemple, les registres de pipeline
  - Donc non tenu de respecter la spécification...
  - ...tant qu'un état architectural **correct** peut-être extrait quand nécessaire
- Prédiction de branchement
  - Etat microarchitectural : Instructions potentiellement incorrectes dans le pipeline
  - Etat architectural : Instructions correctes
    - Le pipeline répare l'état microarchitectural si la prédiction est incorrecte en enlevant les instructions sur le mauvais chemin => Aucune instruction incorrecte ne se retrouve dans l'état architectural

# 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)

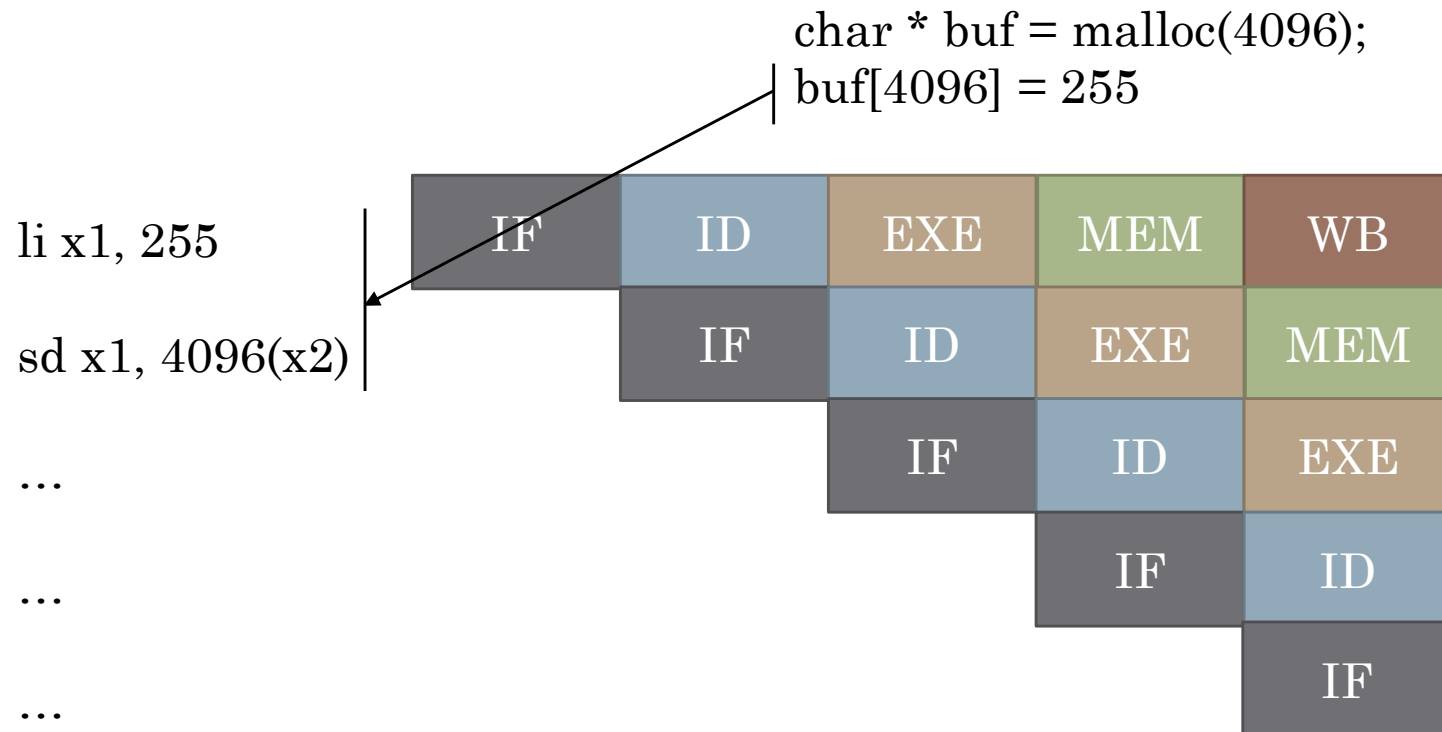
# Dépendances de contrôle – Exécution spéculative

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



# Dépendances de contrôle – Exécution spéculative

- On vide le début du pipeline lorsqu'on découvre la faute, et on saute vers le gestionnaire d'exception de l'OS
  - 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

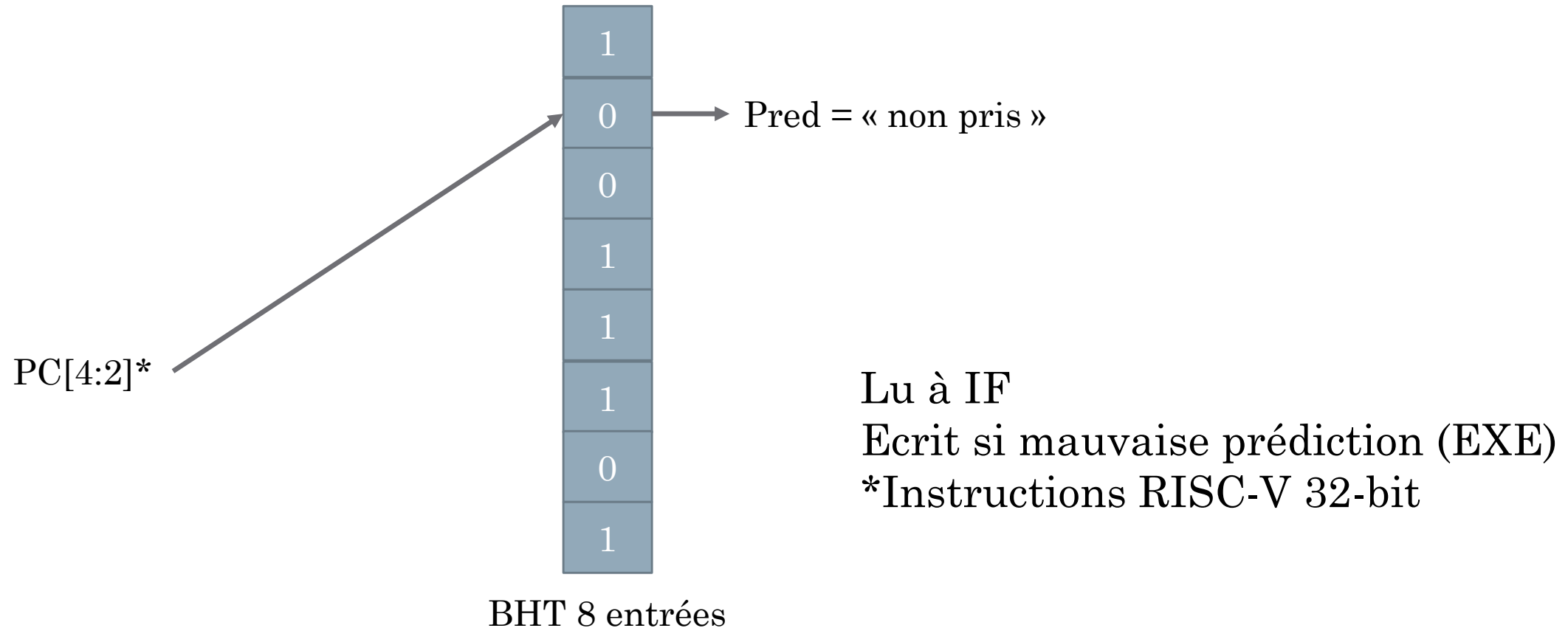
- Structure matérielle implémentant un algorithme spécifique

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

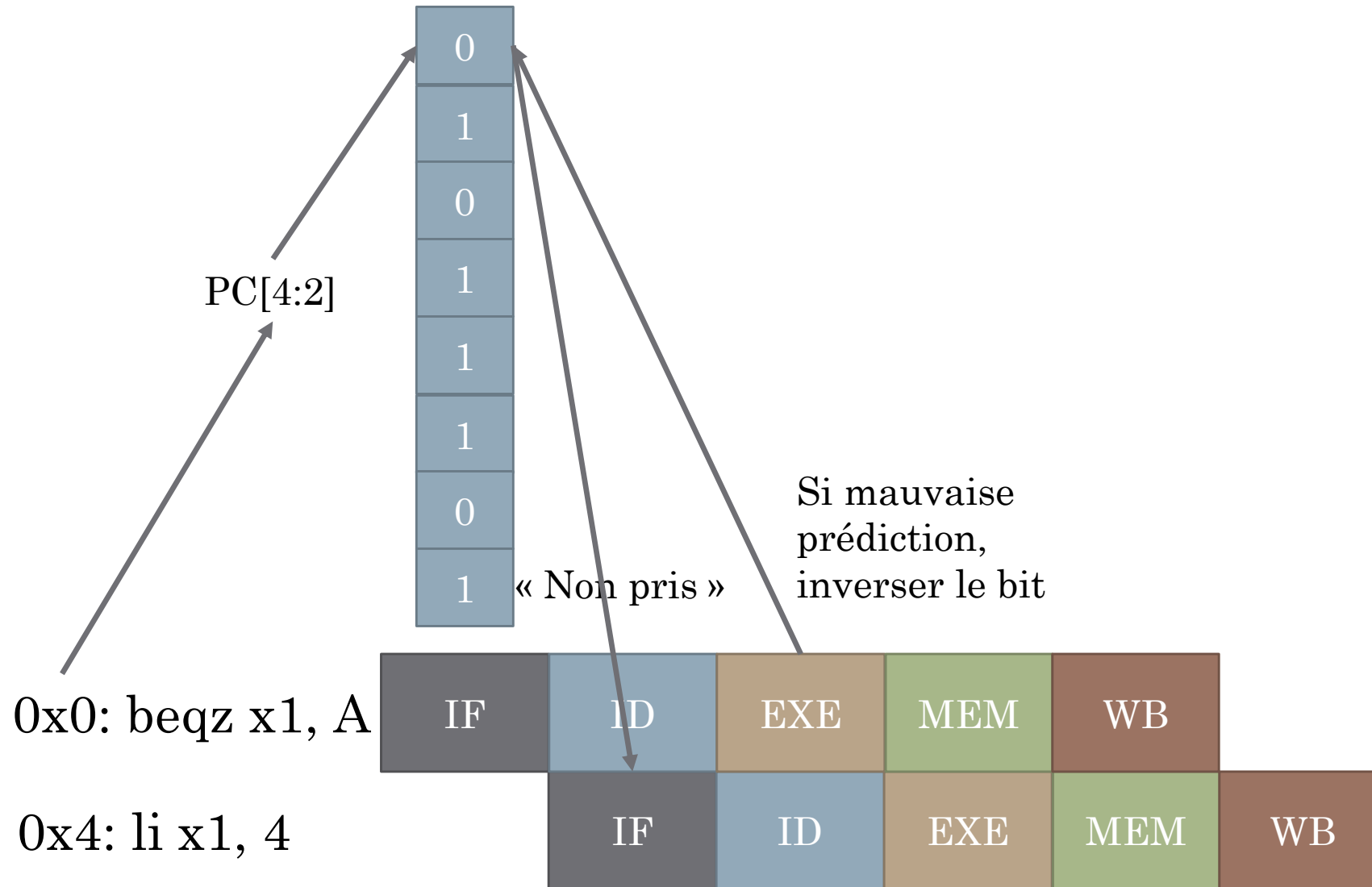
- 80-99% de bonnes prédictions
- « Règle du pouce » 10-90 (ou 20-80) : 10% des branchements du programme responsables de 90% des mauvaises prédictions de branchements

# 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**

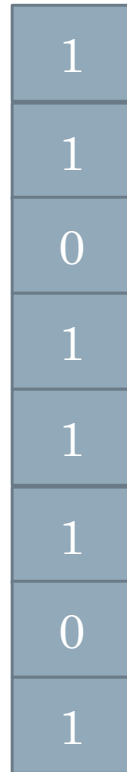
**0% correct**

- Peut mieux faire.

# Prédicteur simple : Aliasing

- En arm, une instruction fait 4B, et est toujours alignée
  - PC[1:0] vaut toujours 00b
  - Prédiction = BHT[(PC >> 2)[2:0]]

A:  
PC 0x0 mul x0, x1, x2  
PC 0x4 jnz x0, C  
B:  
PC 0x8 add x3, x4, x5  
PC 0xC jnz x3, A  
...  
C:  
PC 0x84 jnz x8, D



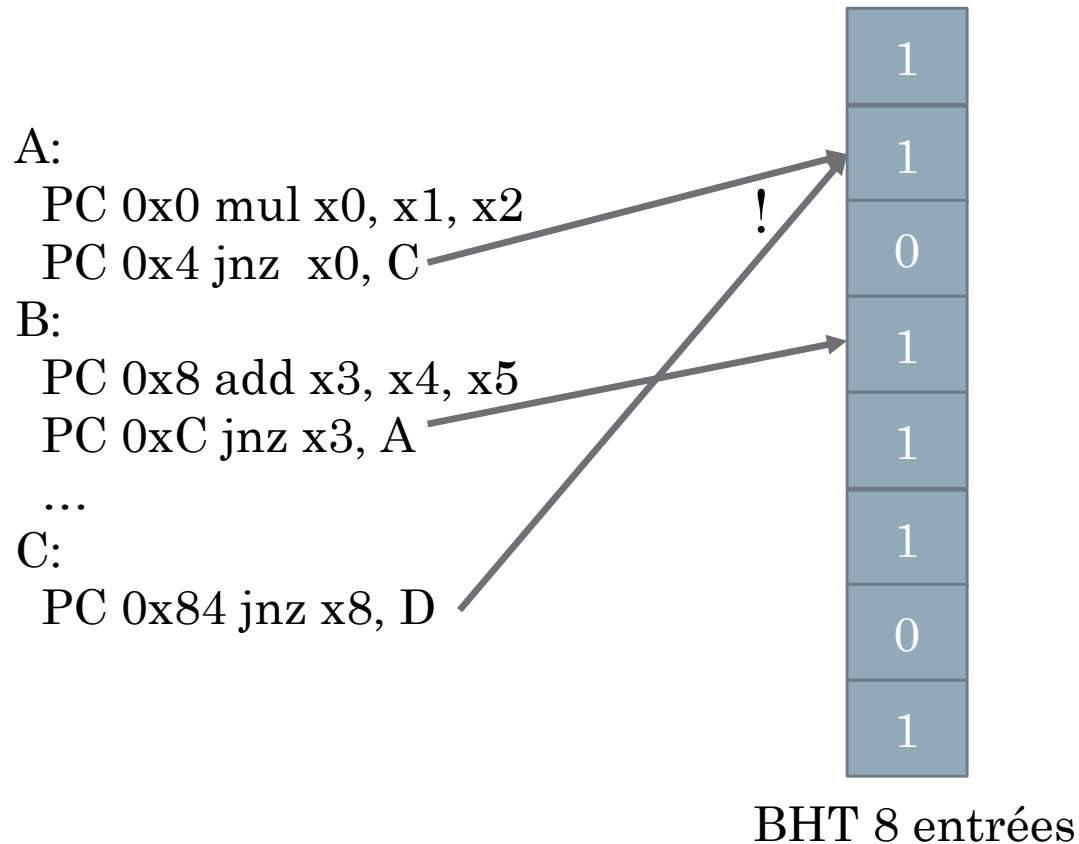
BHT 8 entrées

3 branchements dans le programme  
Quelles prédictions ?



# Prédicteur simple : Aliasing

- En arm, une instruction fait 4B, et est toujours alignée
  - PC[1:0] vaut toujours 00b
  - Prediction = BHT[(PC >> 2)[2:0]]



Prédicteur de taille finie : Plusieurs instructions peuvent partager une prédiction (**aliasing**)

# Prédicteur simple : Interférences

- L'aliasing cause des interférences
- *Destructives* : Un branchement toujours pris et un branchement jamais pris utilisent la même entrée et s'exécutent tour à tour
  - Branchement toujours pris verra prédiction « non pris », mettra à jour avec « pris »
  - Branchement jamais pris verra prédiction « pris », mettra à jour avec « non pris »
  - Si pas d'aliasing, 100% correct, si aliasing, 0% correct

# Prédicteur simple : Interférences

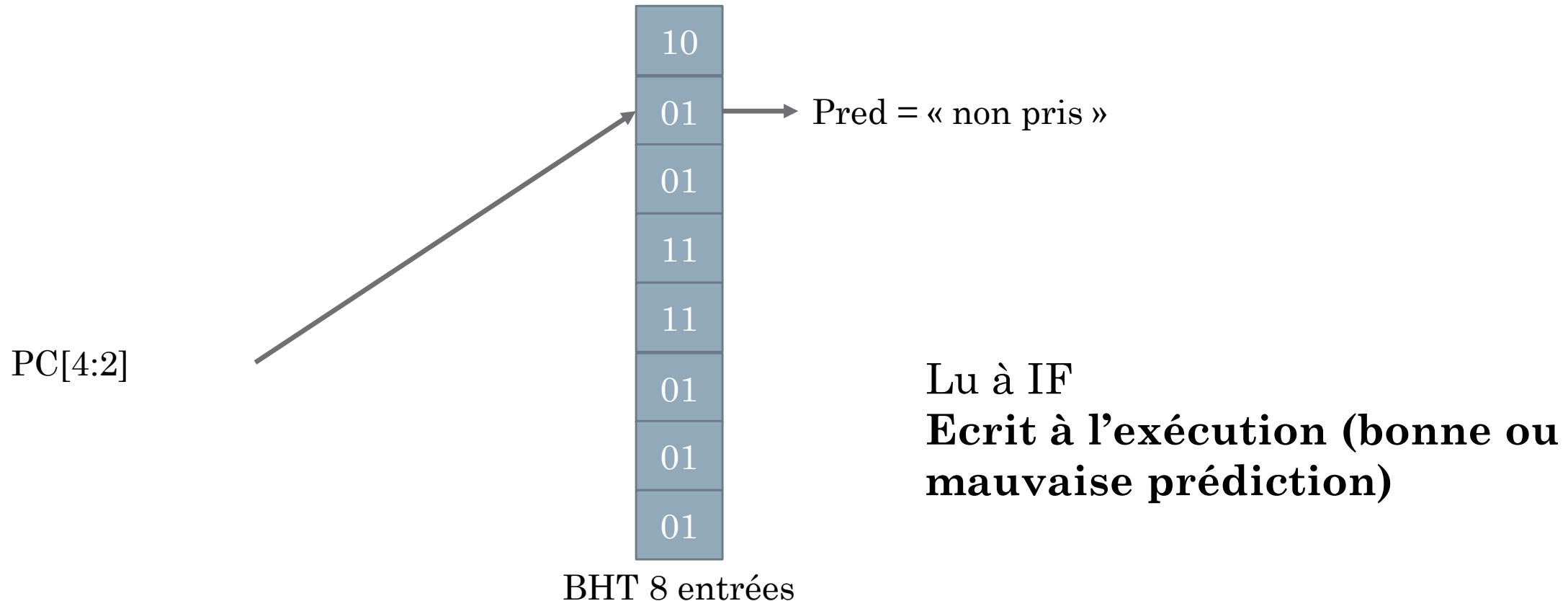
- L'aliasing cause des interférences
- *Constructives* : Un branchement pris  $N$  fois, puis non pris  $N$  fois, et un second branchement pris  $N$  fois et non pris  $N$  fois utilisent la même entrée et s'exécutent tour à tour
  - Instance  $N + 1$  du premier branchement verra « pris » et mettra à jour avec « non pris »
  - Instance  $N + 1$  du second branchement verra « non pris »
  - L'aliasing permet au deuxième branchement d'éviter une mauvaise prédiction à chaque changement de direction

# Prédicteur simple : Interférences

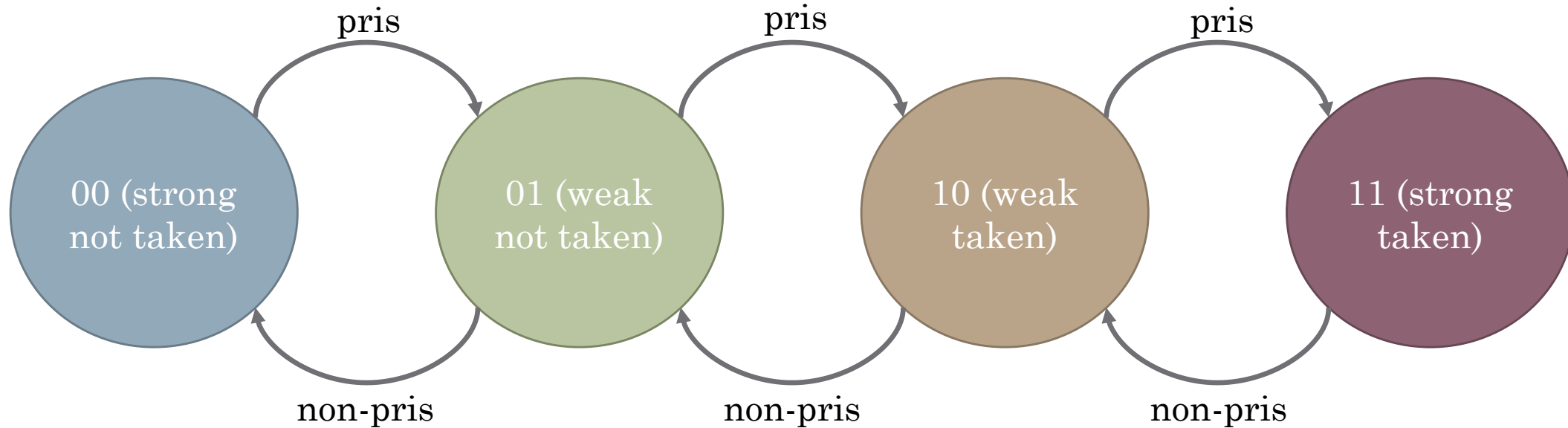
- L'aliasing cause des interférences : Comment les éviter ?
- Prédicteur plus grand : moins d'aliasing car plus d'entrées
- Prédicteur associatif : moins d'interférences à nombre d'entrées égal à mesure que l'associativité augmente
- Généralement impossible de garantir qu'on a pas d'interférence, sauf si on a un prédicteur dont l'associativité est supérieure au nombre de branchement dans le programme (en général pas le cas)

# 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 »)



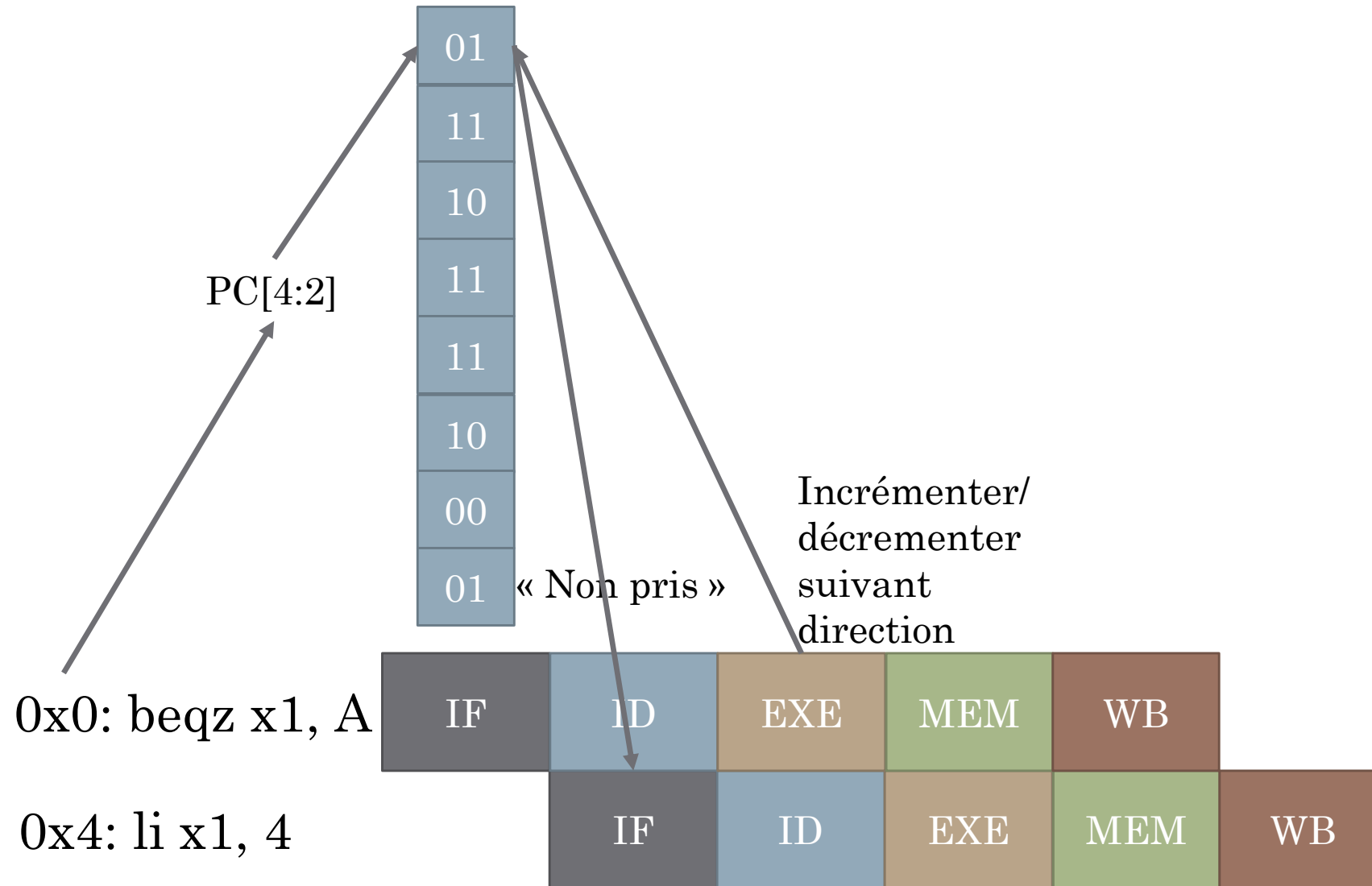
# Prédicteur simple : 2-bit bimodal



- **Question:** Qu'y gagne-t-on ici ?

```
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.

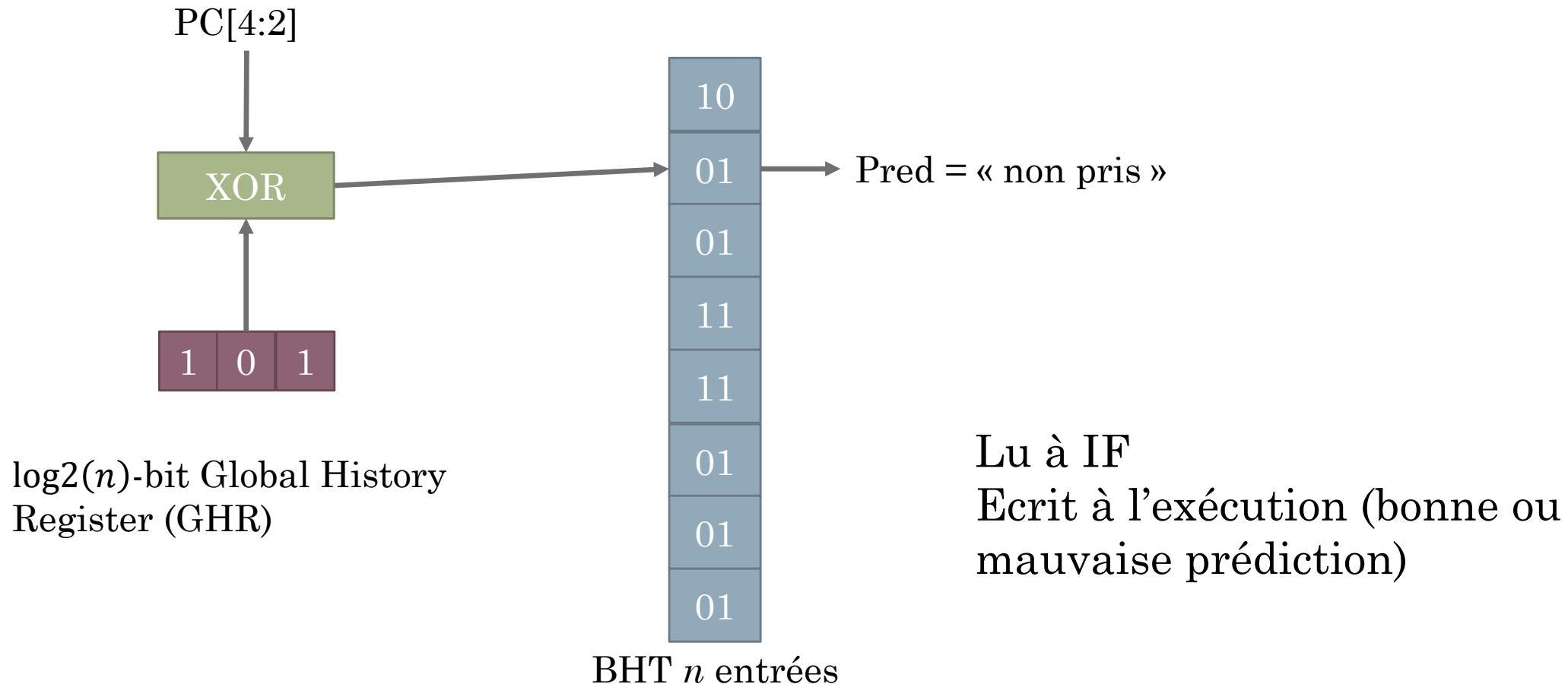
# Reconnaissance de motifs

- 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

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

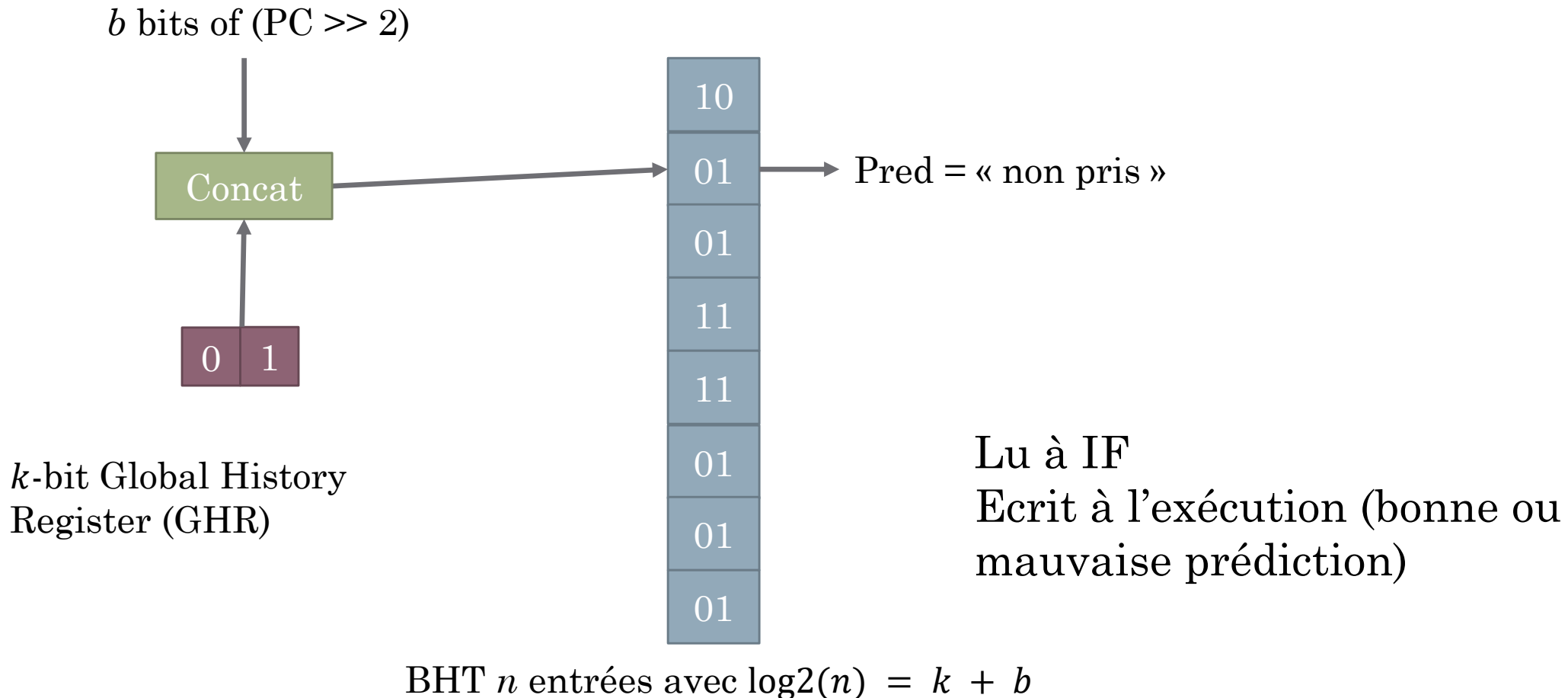
# Corrélation globale : *gshare*

- 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

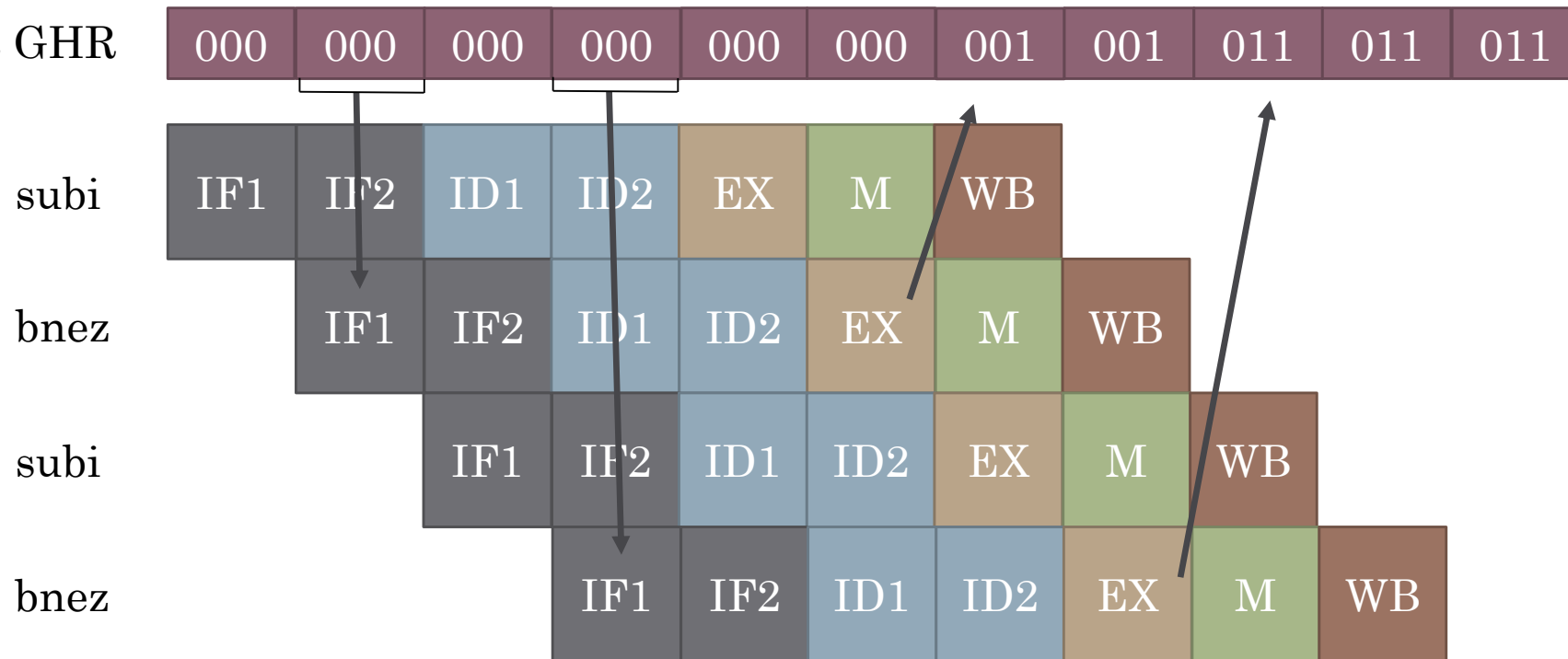
4-bit GHR :  
corrélation entre A  
et B identifiable

# Gestion de l'historique global

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

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

3-bit GHR

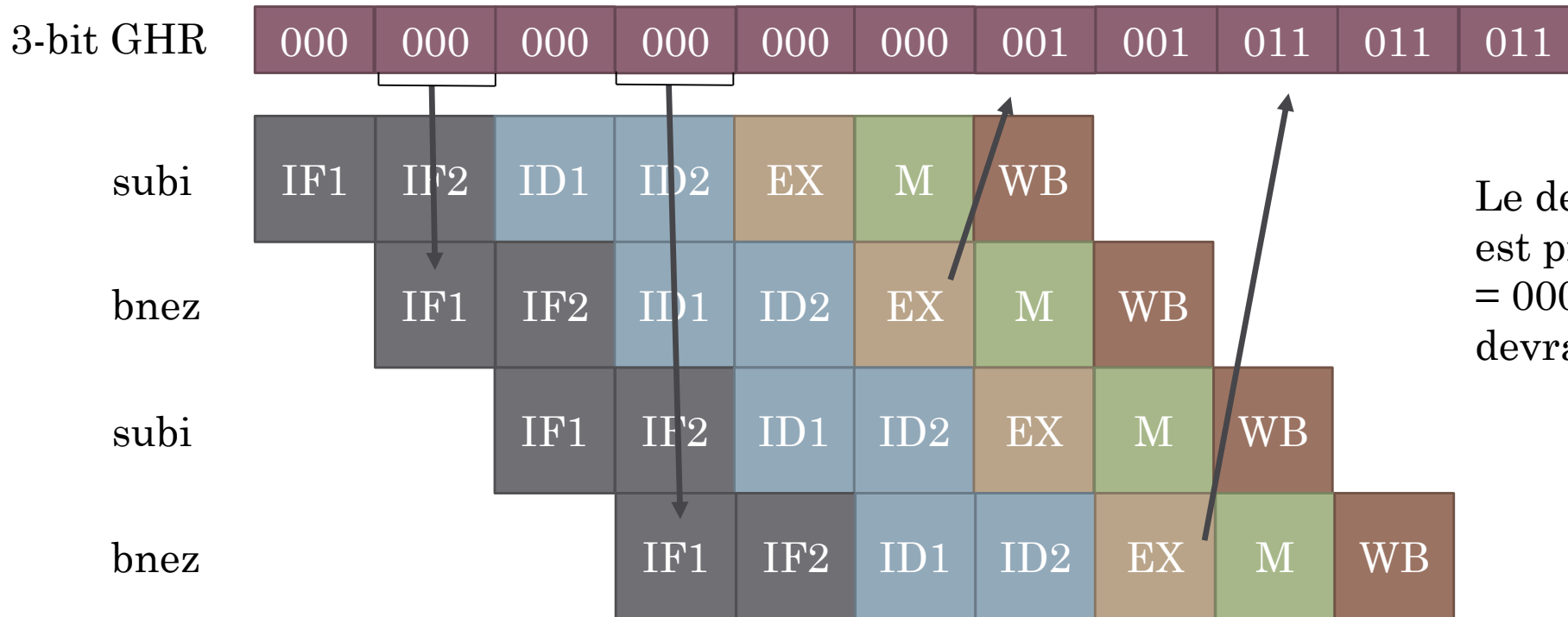




# Gestion de l'historique global

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

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

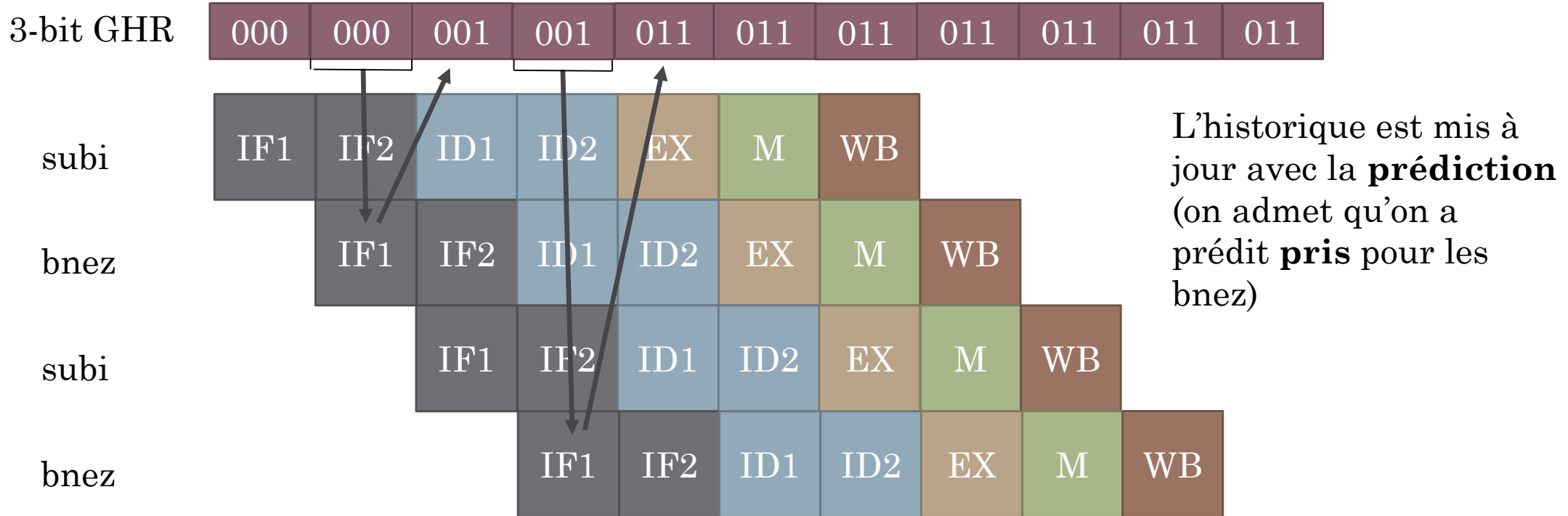


Le deuxième bnez  
est prédit avec GHR  
= 000b alors que cela  
devrait être 001b !

# Gestion de l'historique global

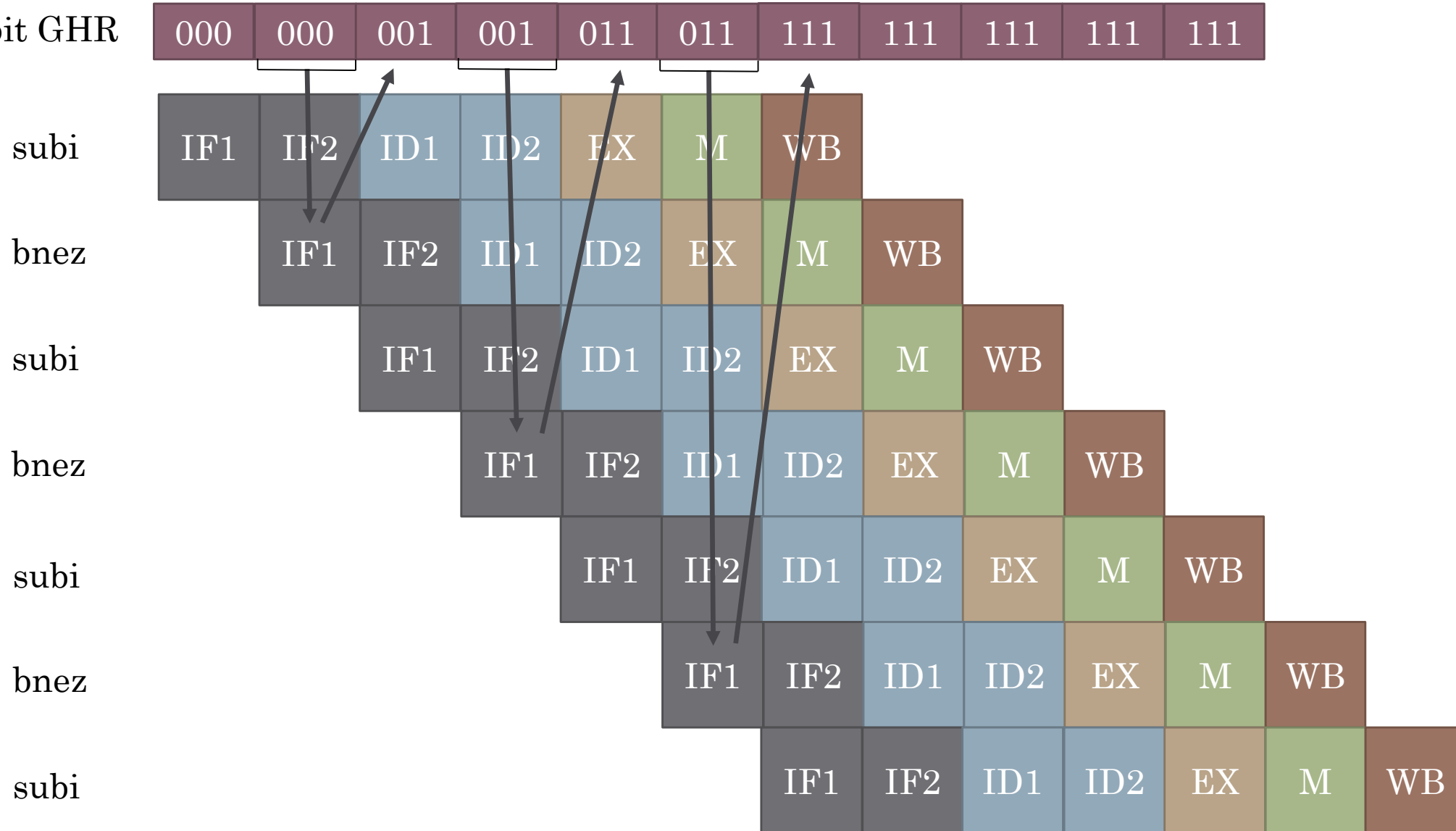
- Grave ? Oui !
- Historique et flot d'instructions désynchronisés
  - 30% de performance en moins d'après Skadron et al., « Speculative updates of local and global branch history: A quantitative analysis », JILP, 2000
- Solution : Ne pas attendre d'avoir calculé la direction du branchement avant de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

# Mise à jour spéculative



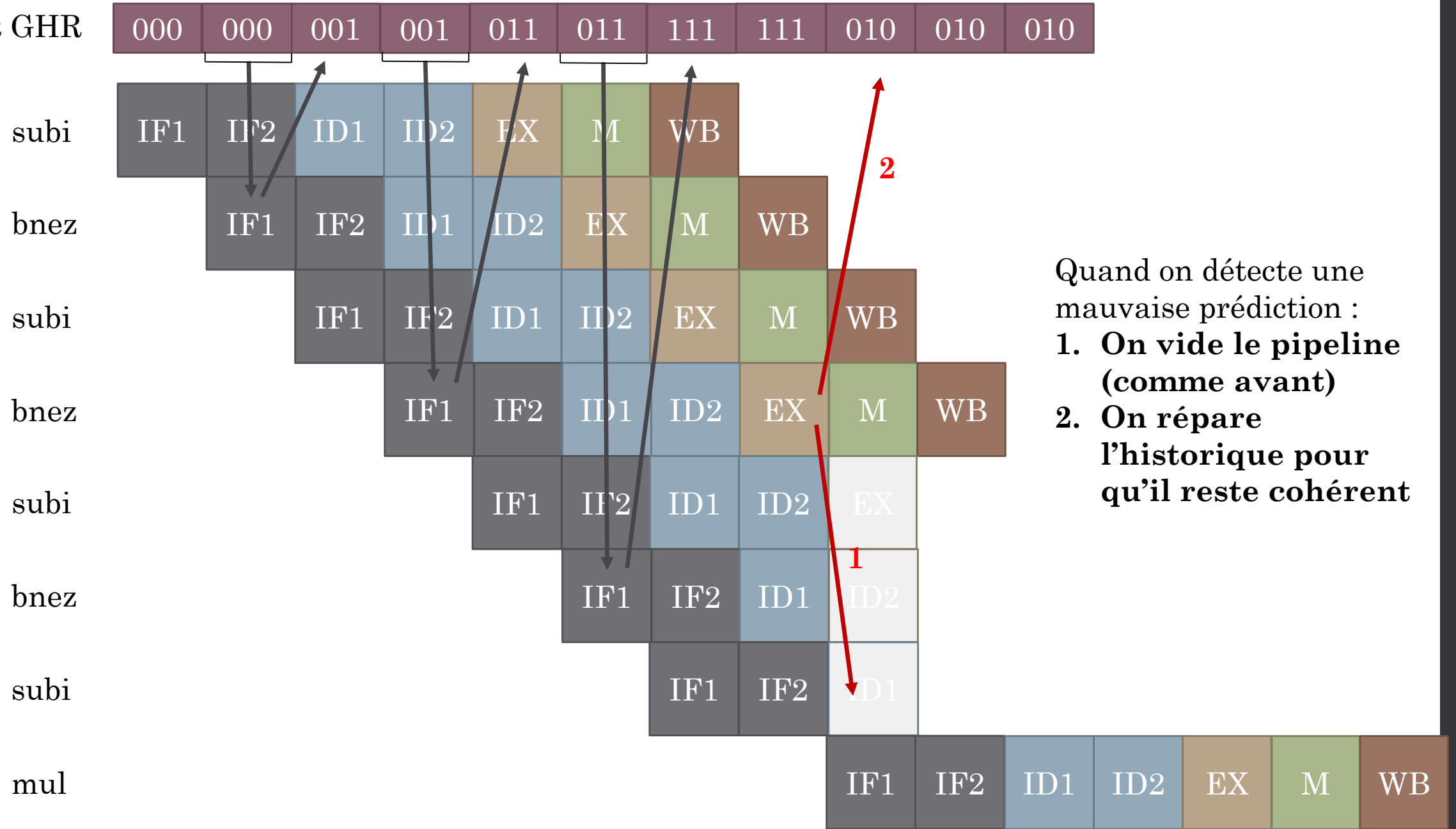
# Mauvaise prédiction

3-bit GHR



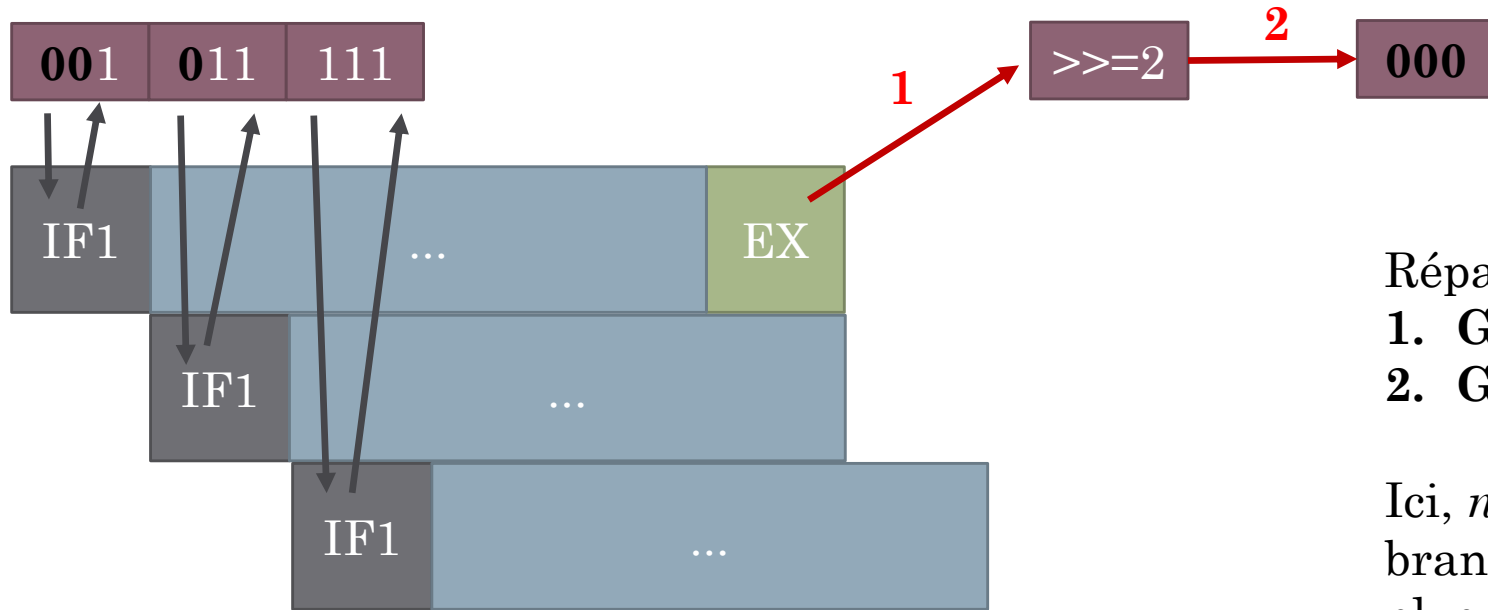
# Mauvaise prédiction

3-bit GHR



# Réparer l'historique global – Solution naïve

GHR initial = 000b (en noir)



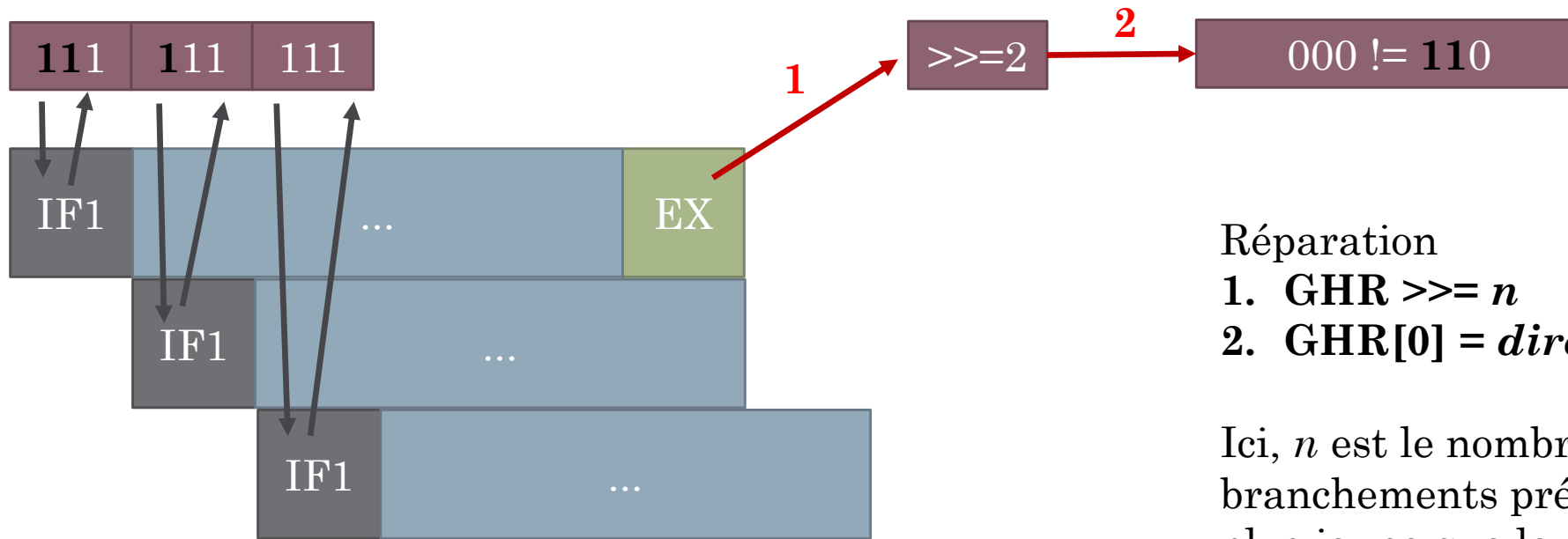
Réparation

1.  $\text{GHR} \gg= n$
2.  $\text{GHR}[0] = \textit{direction}$

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.  $\text{GHR} \gg= n$
2.  $\text{GHR}[0] = \textit{direction}$

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

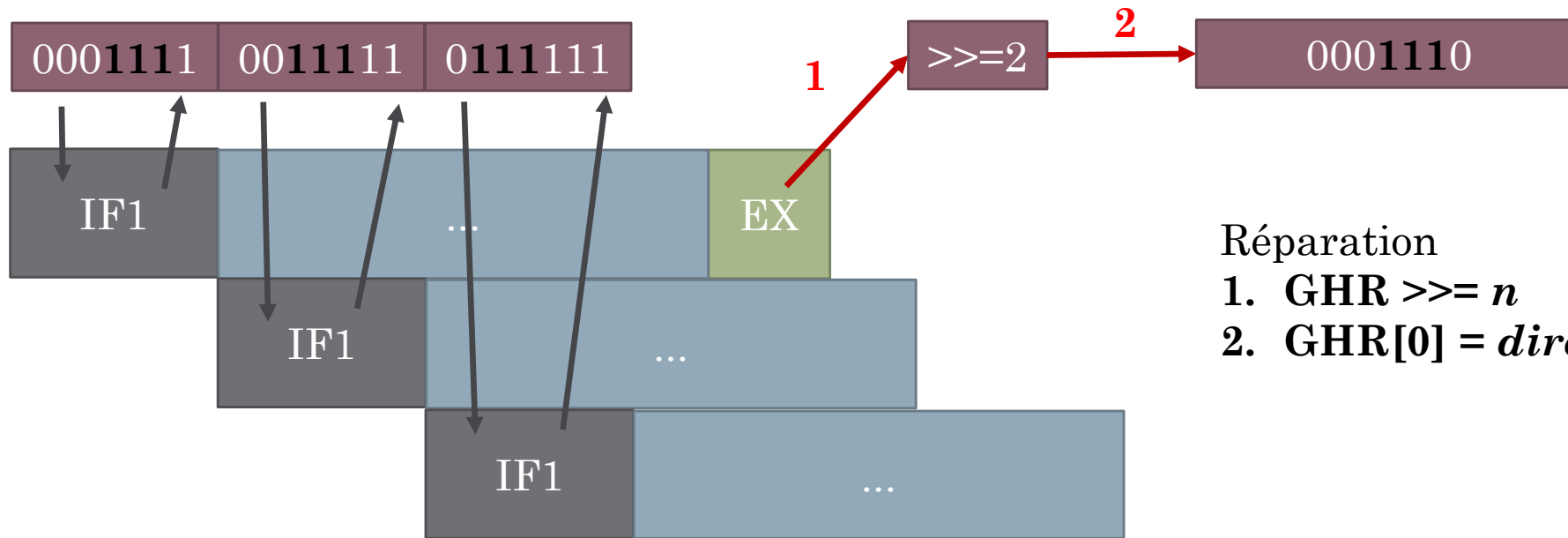
- Avoir un GHR assez grand pour pouvoir décaler à droite et modifier un seul bit lors d'une mauvaise prédiction **sans perdre d'information**
  - Assez grand = ?
  - Registre matériel avec  $n + s$  bits
    - $n$  les bits utilisés pour la fonction de hash (3 dans l'exemple), c'est le GHR logique
 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



Réparation

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

On a bien **GHR** = 110b

# Réparer l'historique global – Solutions

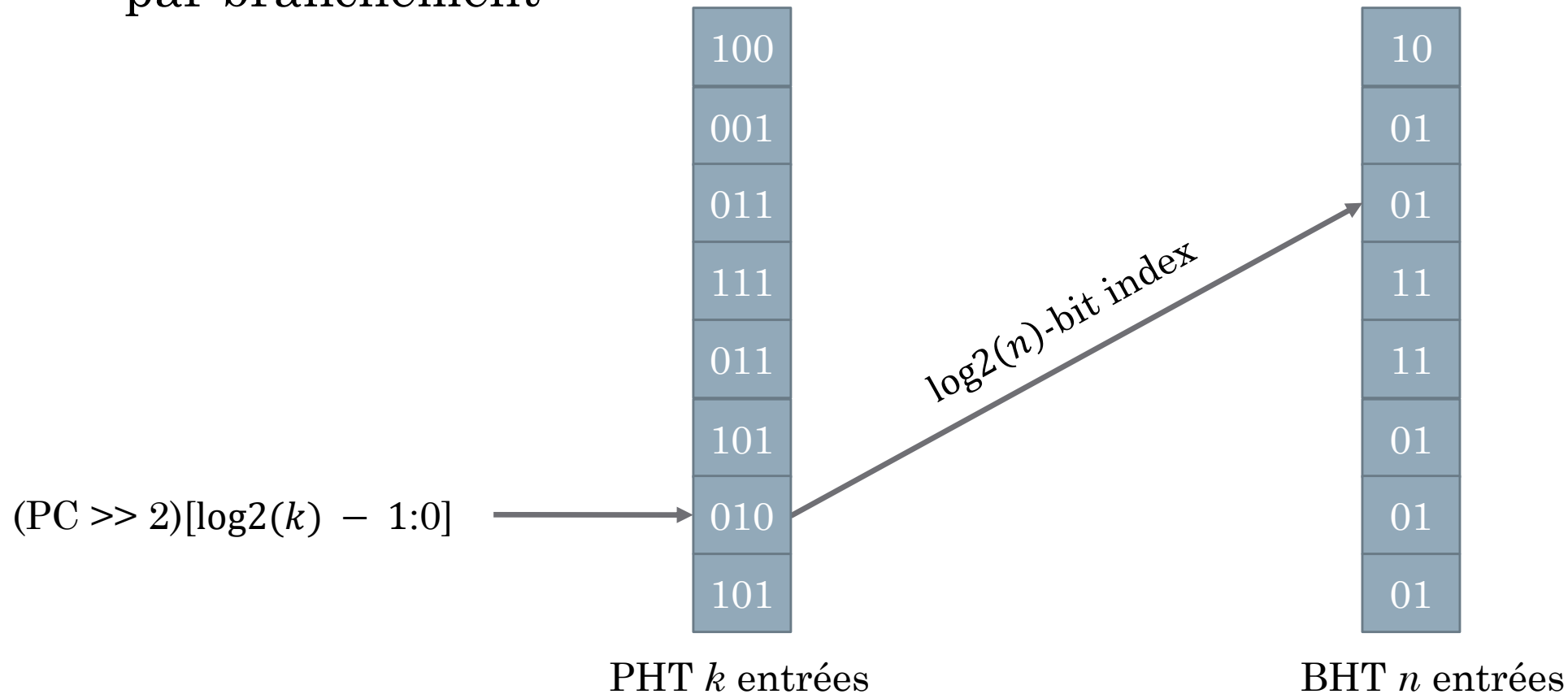
- Dans les processeurs modernes,  $s$  (nombre de mises à jours spéculatives avant la résolution d'un branchement) est grand (plusieurs dizaines), donc il faut un GHR physique significativement plus grand que le GHR logique
- Une autre solution est de transporter une copie du GHR avec chaque branchement 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 (aliasing)** que bimodal pour le même nombre d'entrées
- Plus complexe (fonction d'index, gestion de l'historique)
- Mais bien meilleur que bimodal (voir TP)

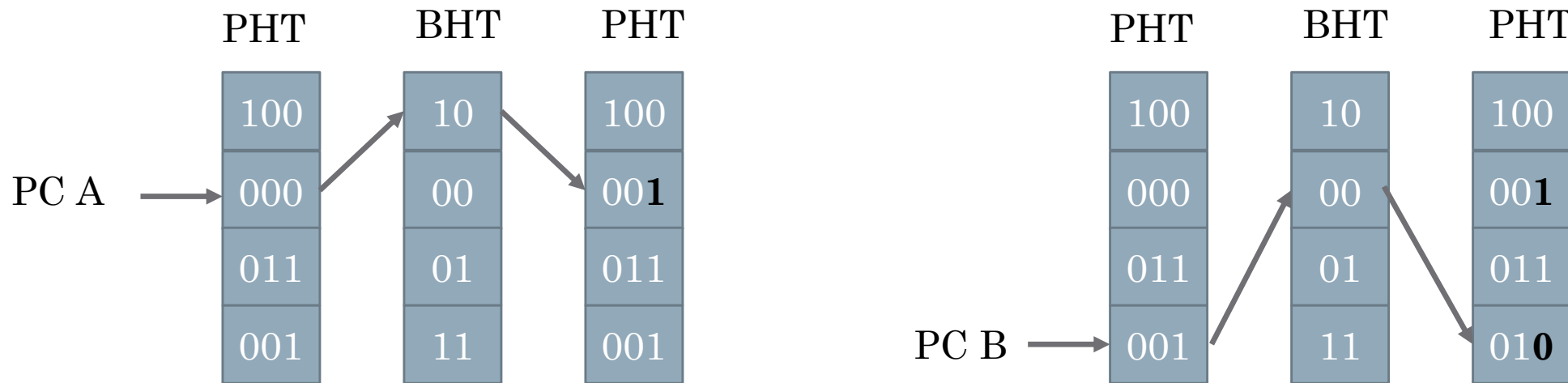
# Corrélation locale : 2-level adaptive

- 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)

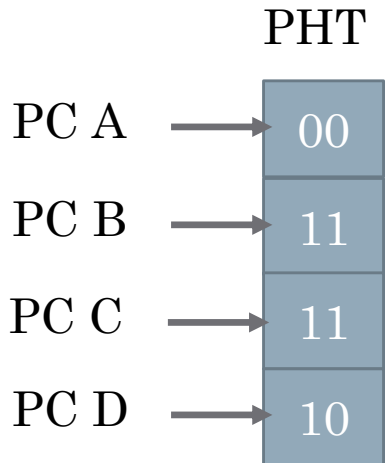
# 2-level adaptive - Inconvénients

- 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 »

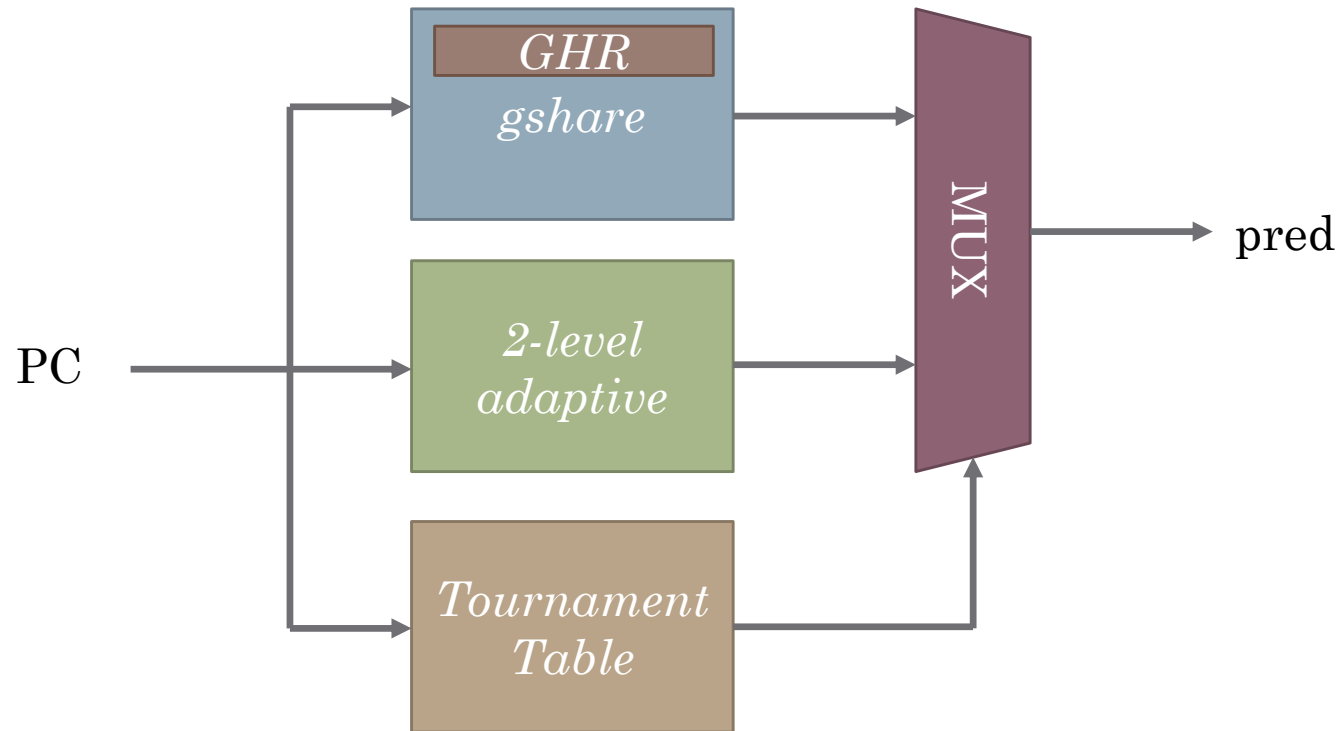
# 2-level adaptive - Inconvénients

- 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

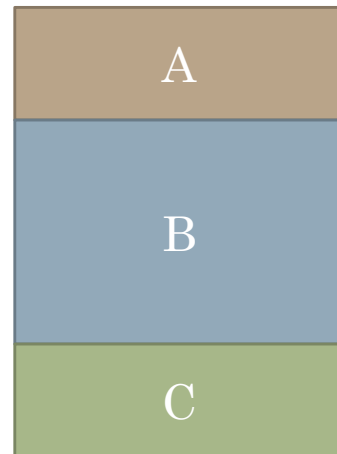
- Tournaement Table contient des compteurs  $n$ -bit
  - Par ex., 2-bit: bit1 donne le prédicteur « gagnant » (0 = 2-level, 1 = gshare), bit0 donne de l'hysteresis
  - Si gshare a bien prédit et 2-level non, incrémenter le compteur
  - Si 2-level a bien prédit et gshare non, décrémenter le compteur
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

# Alternative architecturale : Prédication

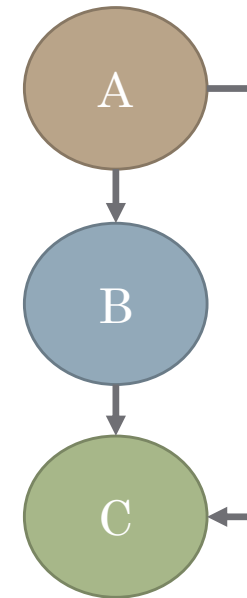
- Idée : transformer le flot de contrôle en flot de données via l'exécution conditionnelle

```
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

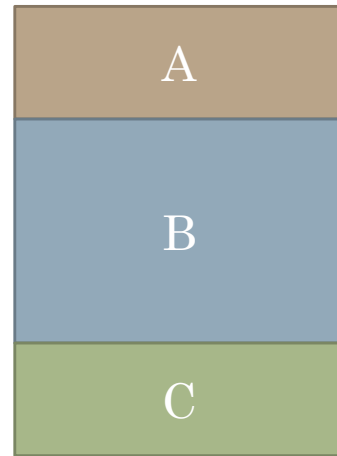
```
int x = 1;  
cond = ...
```

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

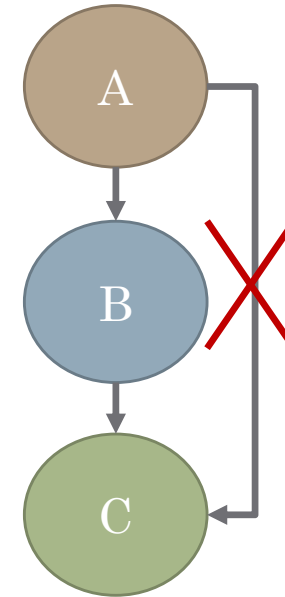
```
y = x;
```

```
...
```

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 x0, 0x1 // x = 1  
ld x1, [@] // Load cond
```

```
mov x2, x0 // temporaire  
mul x2, 0x3 // x_tmp = x * 3  
add x2, 0x2 // x_tmp += 2  
cmp x1, 0x0 // Set flags  
cmovnz x0, x2 // si !Zero Flag  
                // (cond != 0x0)  
                // x = x * 3 sinon nop
```

```
mov x3, x0 // 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 x0, 0x1 // x = 1  
ld x1, [@] // Load cond
```

```
cmp x1, 0x0 // Set flags  
mulne x0, x0, 0x3 // x = x * 3 si  
// cond != 0x0  
addne x0, x0, 0x2 // x += 2 si cond != 0
```

```
mov x2, x0 // 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)

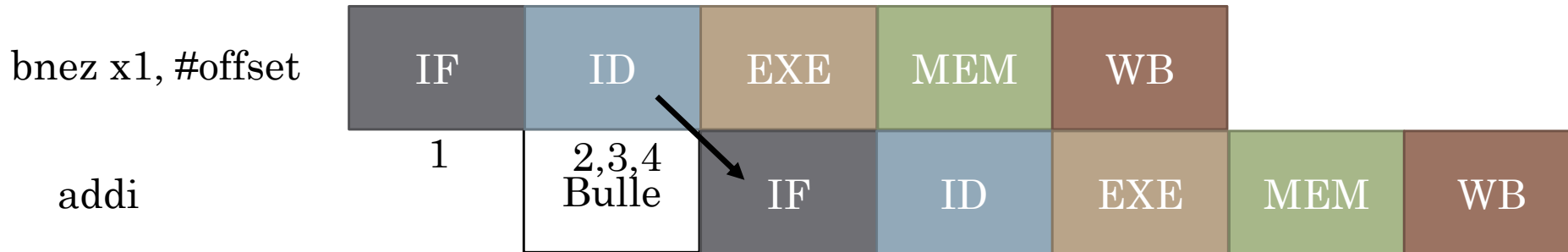
# Prédication : Inconvénients

- 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

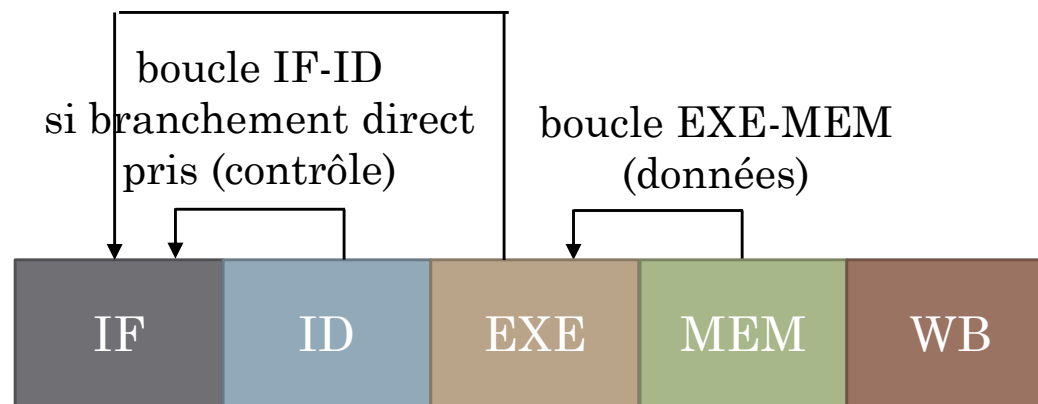
- Prédiction de branchement = prédire la direction ?
- Calcul du prochain PC pour un branchement direct requiert :
  1. D'avoir lu l'instruction dans la mémoire d'instructions
  2. D'avoir décodé l'instruction pour savoir qu'il s'agit d'un branchement
  3. D'avoir étendu le signe du déplacement encodé 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)



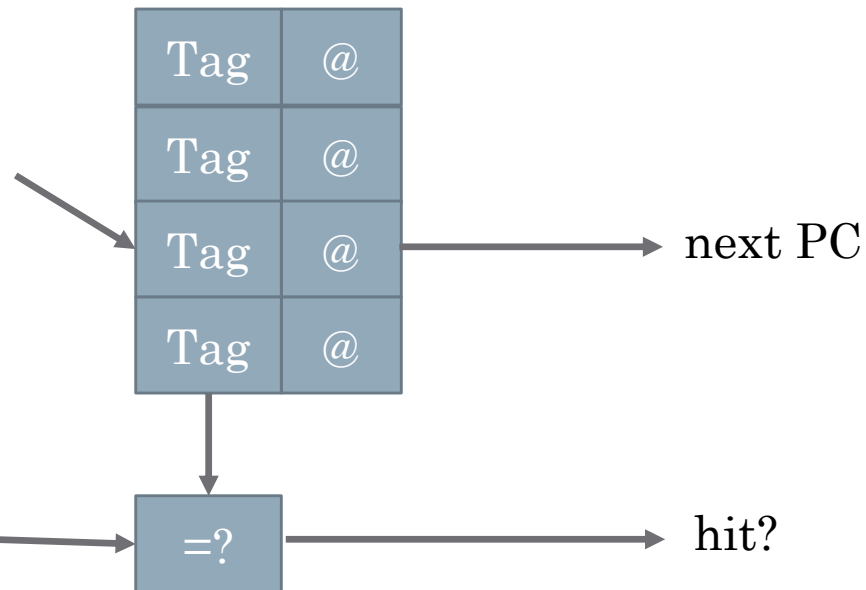
# Branch Target Buffer (BTB)

- 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

$(PC \gg 2)[61:\log_2(n)]$



BTB  $n$  entrées

# 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

pris/non pris

Mauvaise prédiction d'adresse  
branchement direct

MUX

+4

Branch  
Target Buffer

Prédicteur de  
branchements

Cache  
Instructions

Décodeur

dépl.

SEXT

ADD

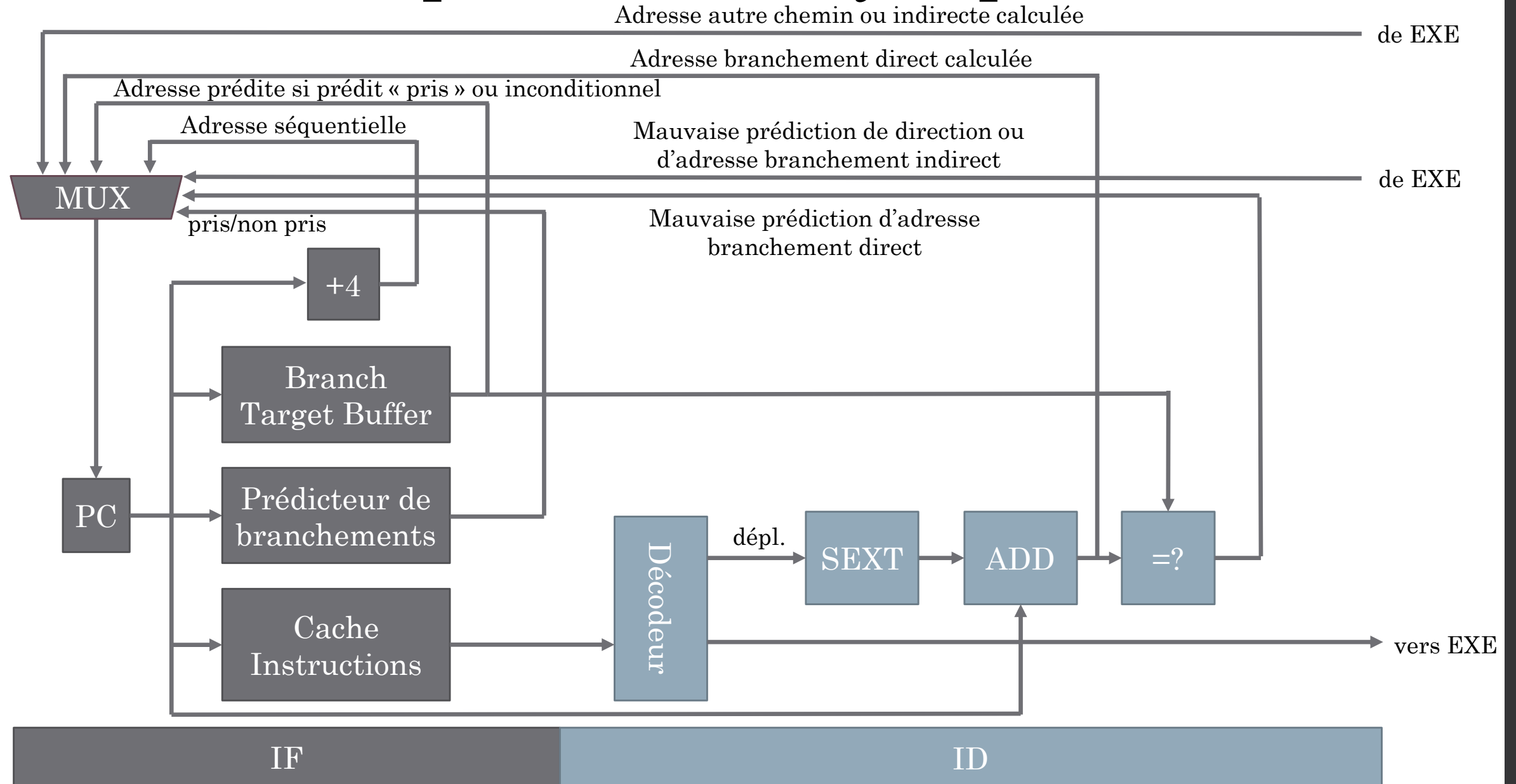
=?

vers EXE

PC

IF

ID



# 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

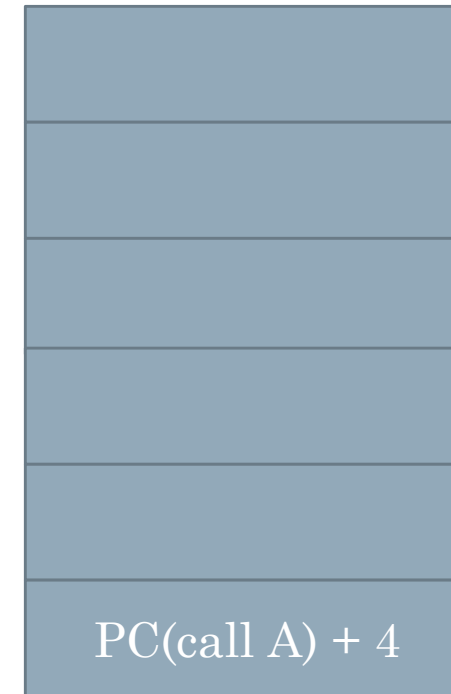
...

return

B:

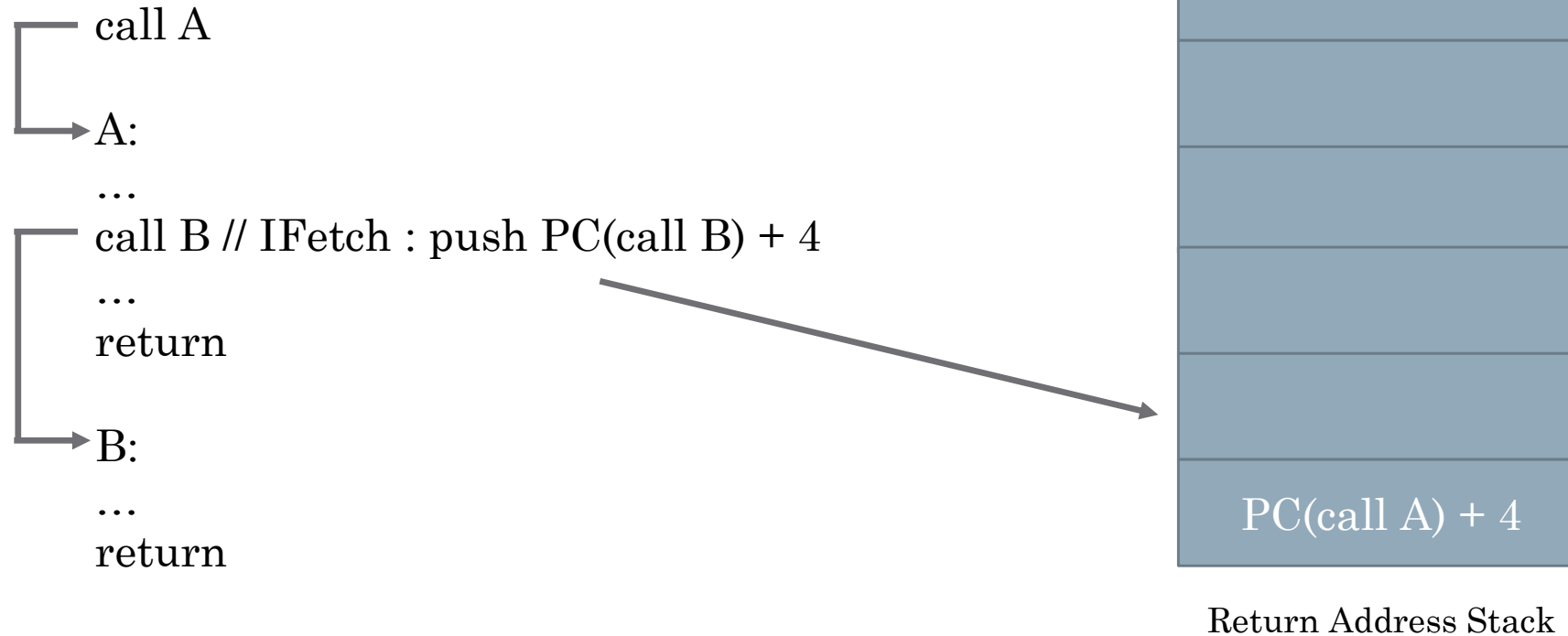
...

return

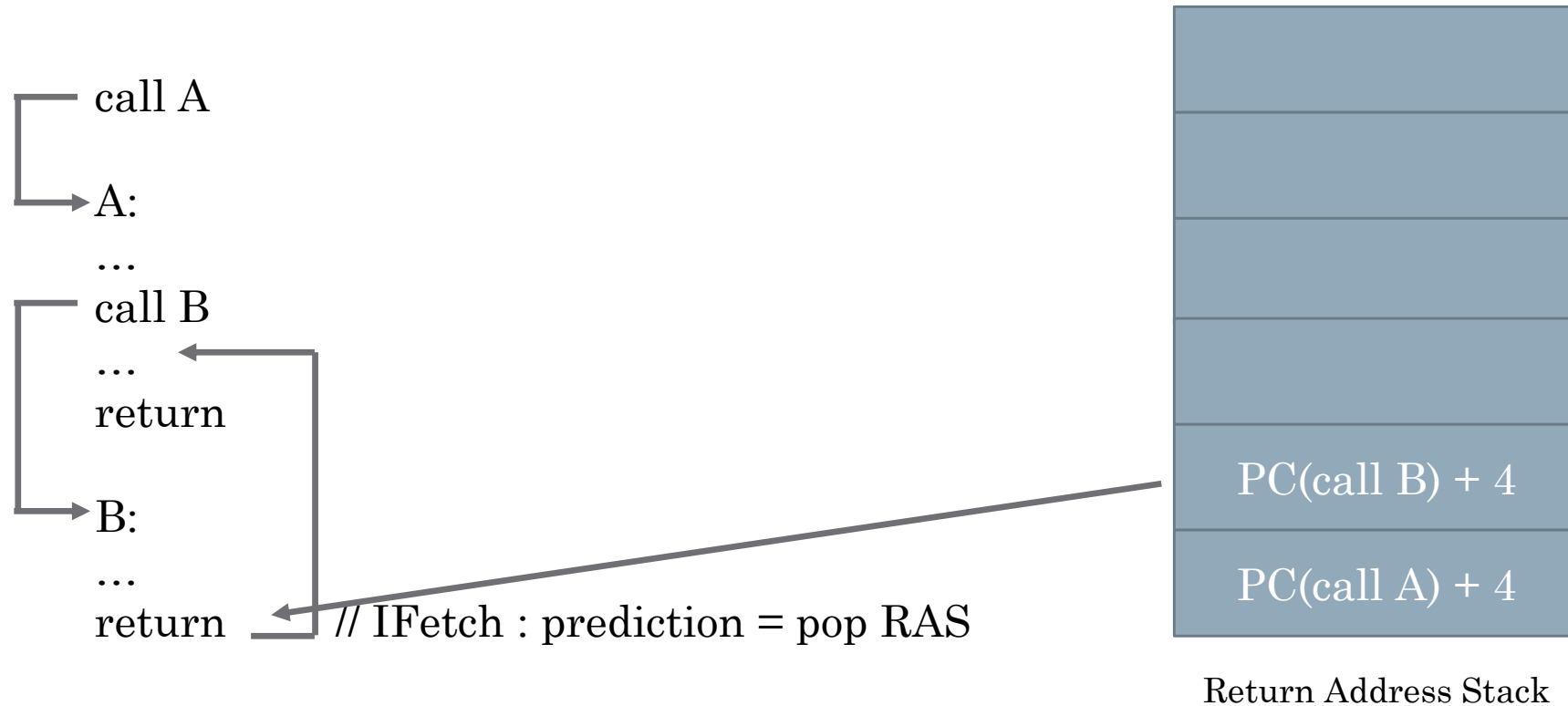


Return Address Stack

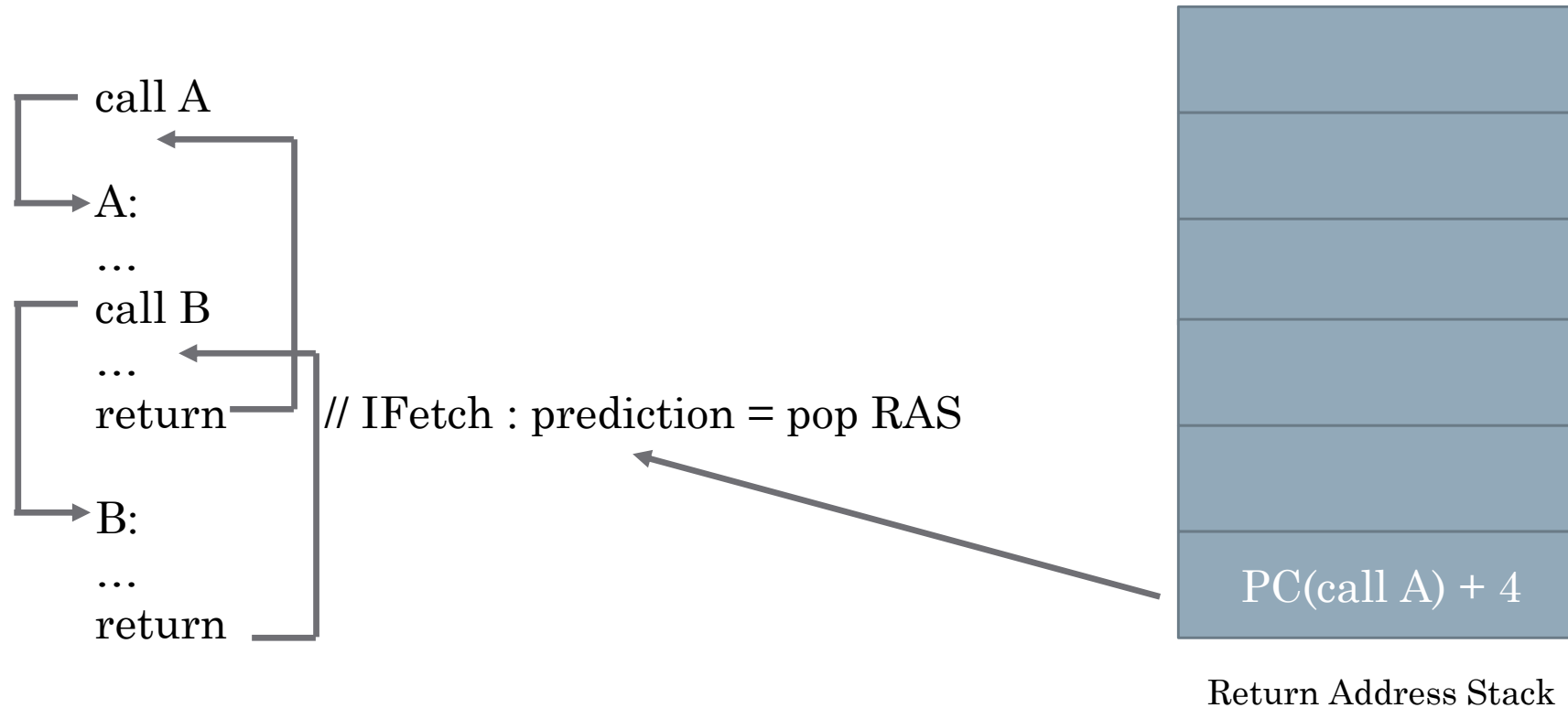
# Return Address Stack (RAS)



# Return Address Stack (RAS)



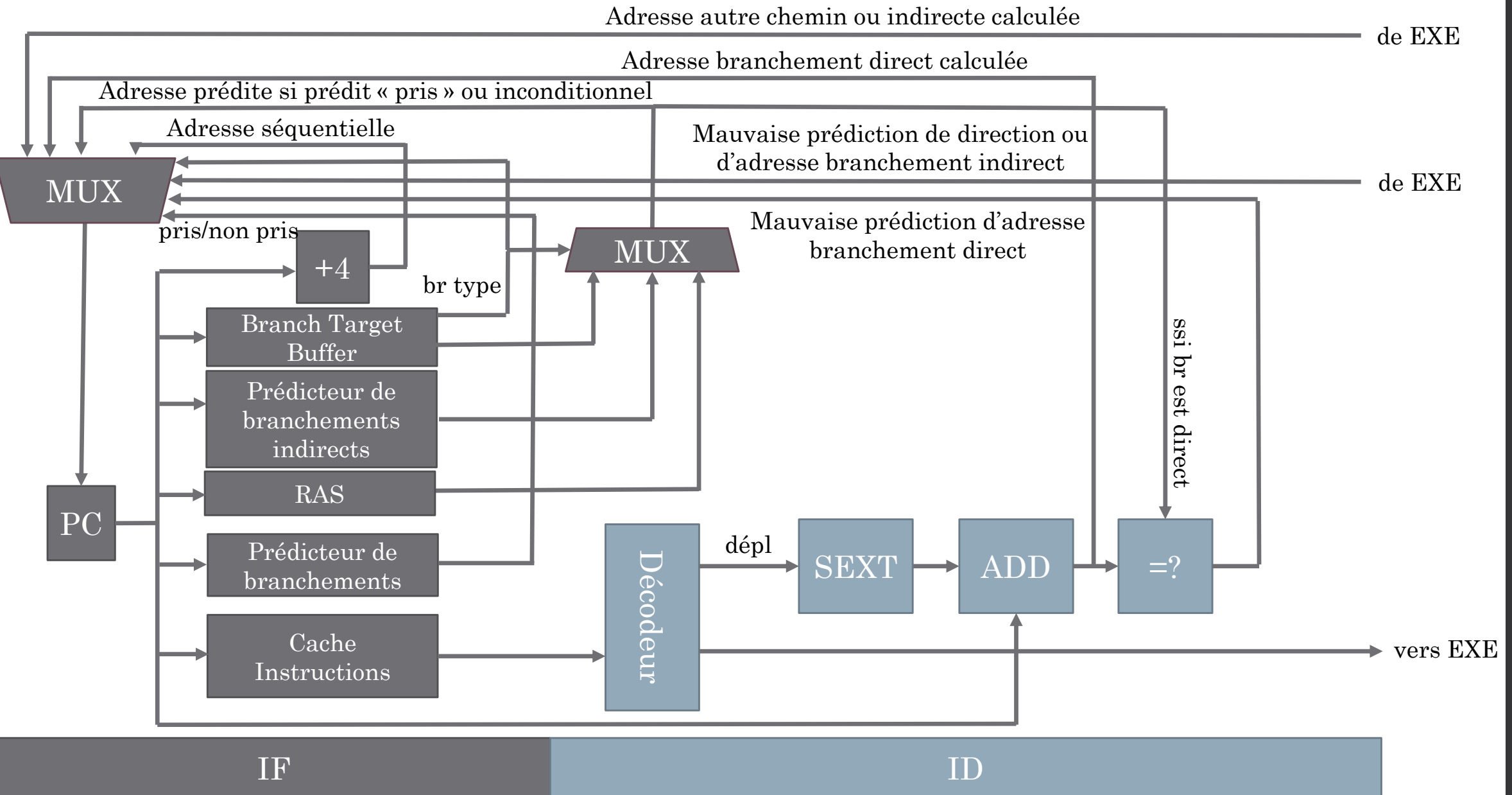
# Return Address Stack (RAS)



# 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

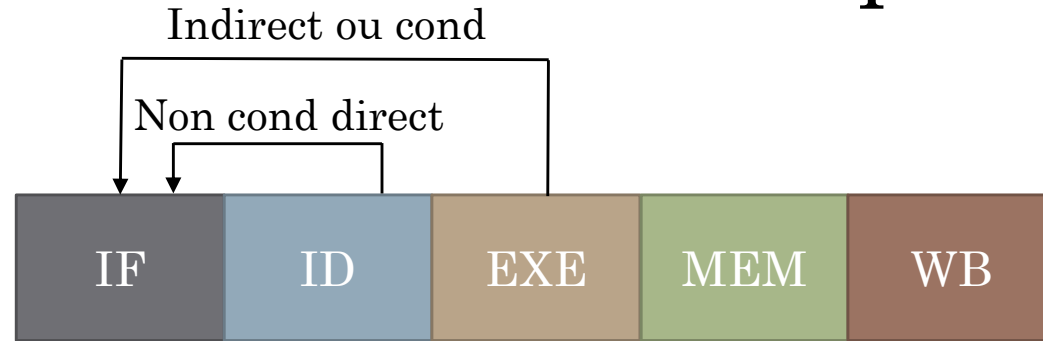


# Pour résumer

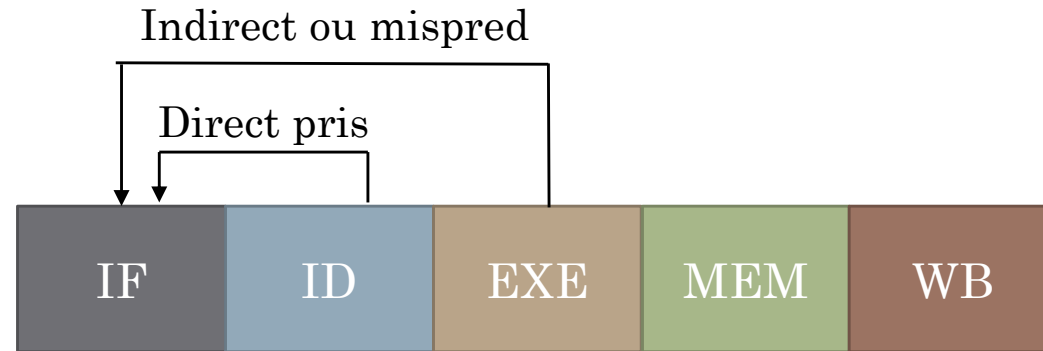
- L'étage Fetch doit minimiser les bulles dues aux branchements = calcul du prochain PC à chaque cycle, idéalement
- Implique de multiples structures et prédictions
  - Prédicteur de branchement pour les branchements conditionnels
  - BTB pour les adresses des branchements directs (cond et non cond)
  - 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

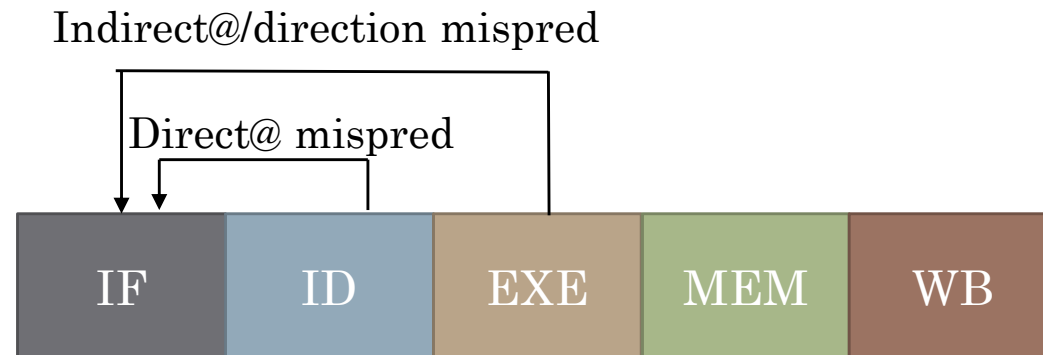
Sans prédiction: 2 bulles  
à chaque branchement  
conditionnel/indirect  
1 bulle à chaque branchement non  
cond direct



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

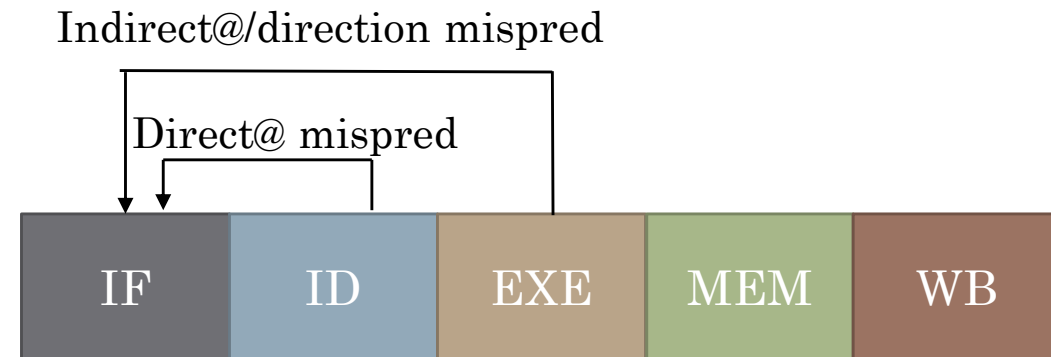




# Dépendances de contrôle – Exécution spéculative

- Impact des dépendances de contrôle réduit 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 (délais) 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 ?

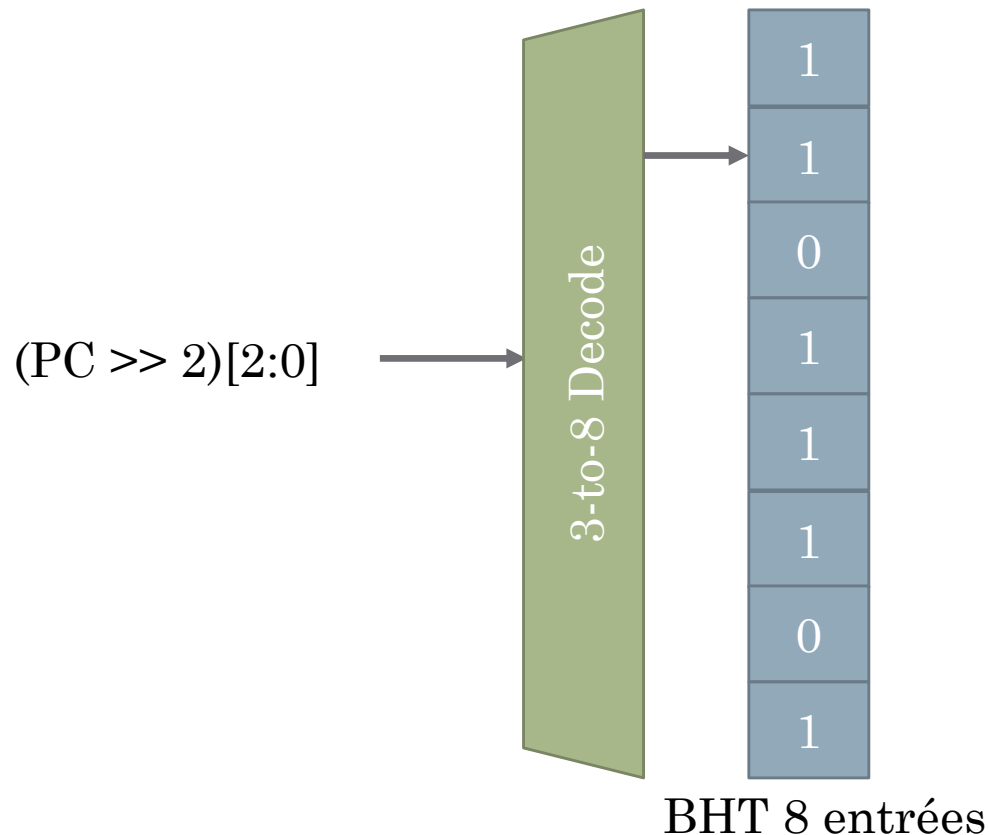
## Pipelining et prediction de branchement

SEOC3A – CEAMC

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

# Rappel : Structures Associatives

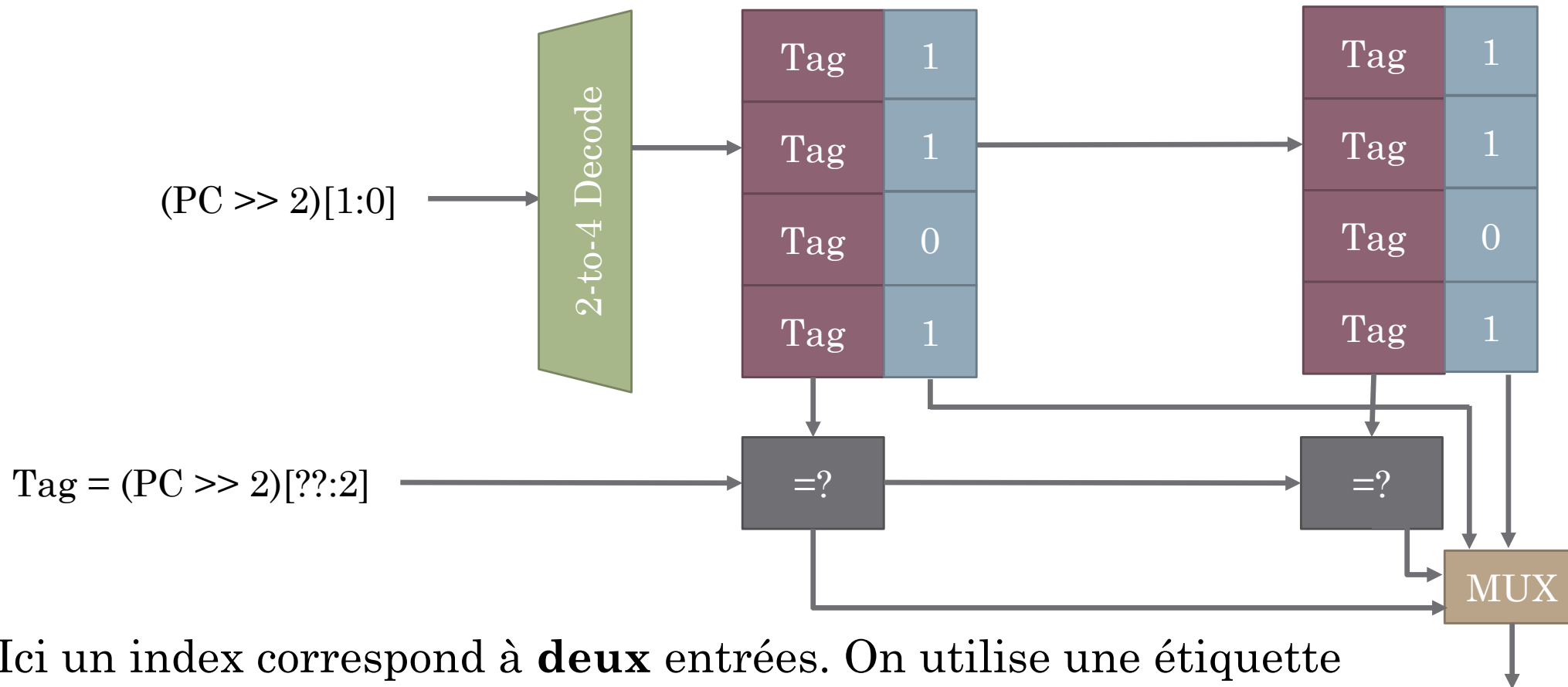
- Structure avec un ensemble d'entrées. On accède à une entrée via un décodeur, en présentant le numéro d'entrée (**index**) que l'on souhaite :



Ici un index correspond à une seule entrée. Structure **direct mapped**.

# Rappel : Structures Associatives

- On peut aussi organiser le BHT en 4 ensembles (**sets**) de deux voies (**ways**) chacun :



Ici un index correspond à **deux** entrées. On utilise une étiquette (**tag**) pour déterminer quelle entrée est pertinente.

# Rappel : Structures Associatives

- On peut aussi organiser le BHT en 1 ensemble (**set**) de 8 voies (**ways**)
- Un index correspond à **huit** entrées, donc on utilise des étiquette, comme au cas précédent
- **Fully associative** car on a autant d'entrées que de voies, une instruction peut donc récupérer sa prédiction depuis n'importe quelle entrée (en fonction des tags présents)

# Rappel : Structures Associatives

- La notion de tag introduit la notion de **hit** et de **miss** :
  - Miss : Si aucun tag ne correspond, il n'y a pas de prédiction pertinente, on peut en faire une par défaut ou ne pas prédire
  - Hit : Si un tag correspond, on sait que la prédiction est pour cette instruction en particulier
- Si le tag est **partiel** (e.g., si on utilise pas tous les bits du PC qu'on a pas utilisé pour former l'index), on peut avoir :
  - Plusieurs hits
  - Interférence constructive/destructive avec un autre branchement
- Si le tag est **complet**, on peut avoir
  - Au maximum un seul hit
  - Interférence destructive avec une autre branche
- A noter qu'on peut aussi tagger une structure de prédicteur direct-mapped, mais ce n'est pas obligatoire

# Rappel : Structures Associatives

- Remplacement d'un tag : Si on a un miss
- Lorsque le branchement s'exécute, on remplace un tag existant dans le set ou le branchement réside par le tag du branchement, et on met à jour la prédiction
- Plus on a de ways dans un set, plus on a le choix dans la **victime** qu'on va choisir
  - Beaucoup de politiques possibles, par exemple **Least Recently Used**
- Toute cette discussion s'applique aussi aux mémoires caches, sauf que le tag ne peut pas être partiel car la mémoire cache contient des données **exactes** (pas des prédictions)

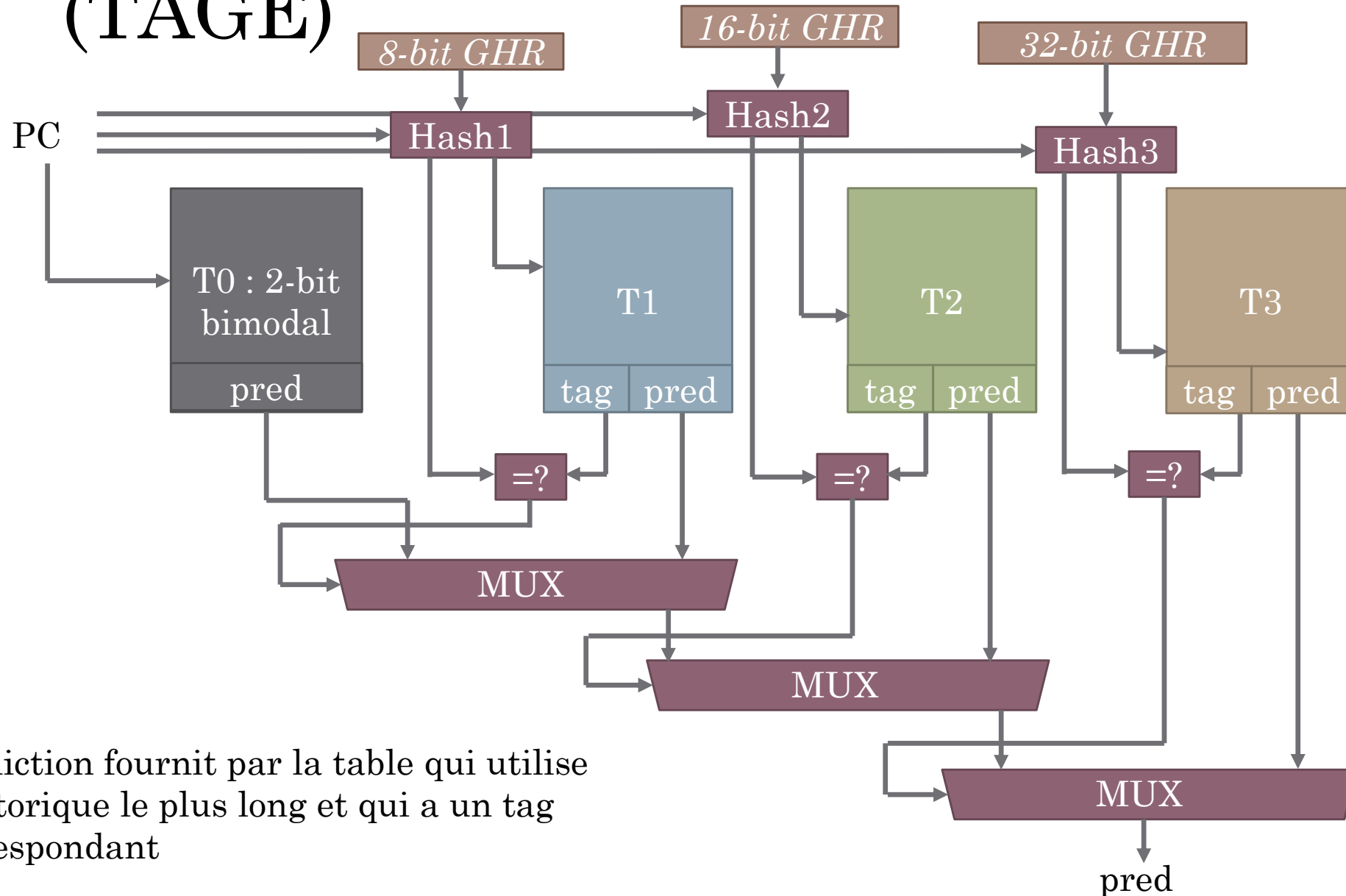
# Prédicteur moderne : TAgged GEometric (TAGE)

- Observation : Beaucoup de corrélation entre branchements proches, mais parfois des corrélations entre branchements très lointains
  - Idée : Plusieurs historiques global de tailles différentes (taille suit une série géométrique). Chaque historique correspond à un BHT
- Majorité du matériel dédié aux corrélations proches

Seznec and Michaud, « A case for (partially)-tagged geometric history length predictors », JILP, **2006**

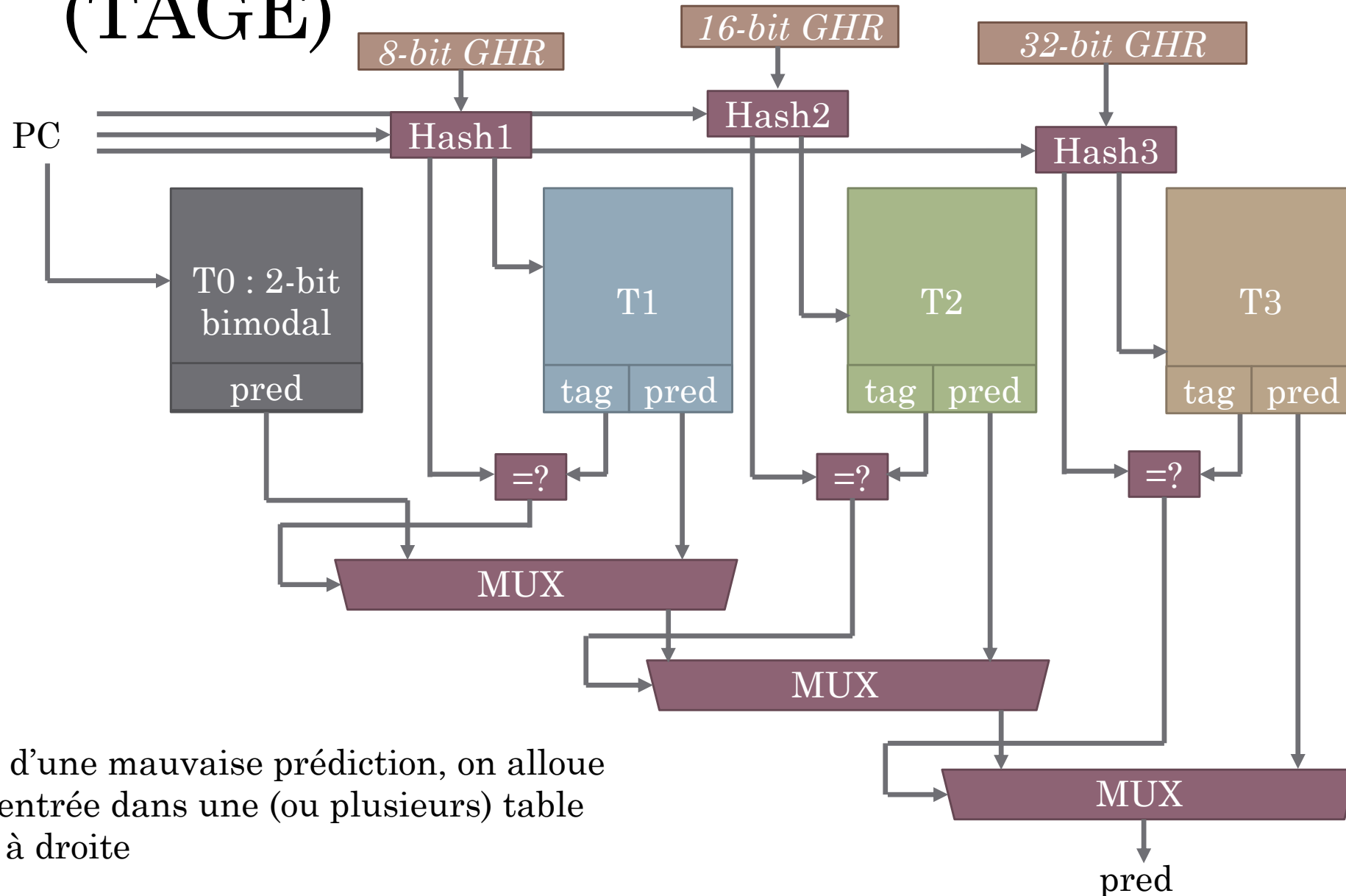


# Prédicteur moderne : TAgged GEometric (TAGE)



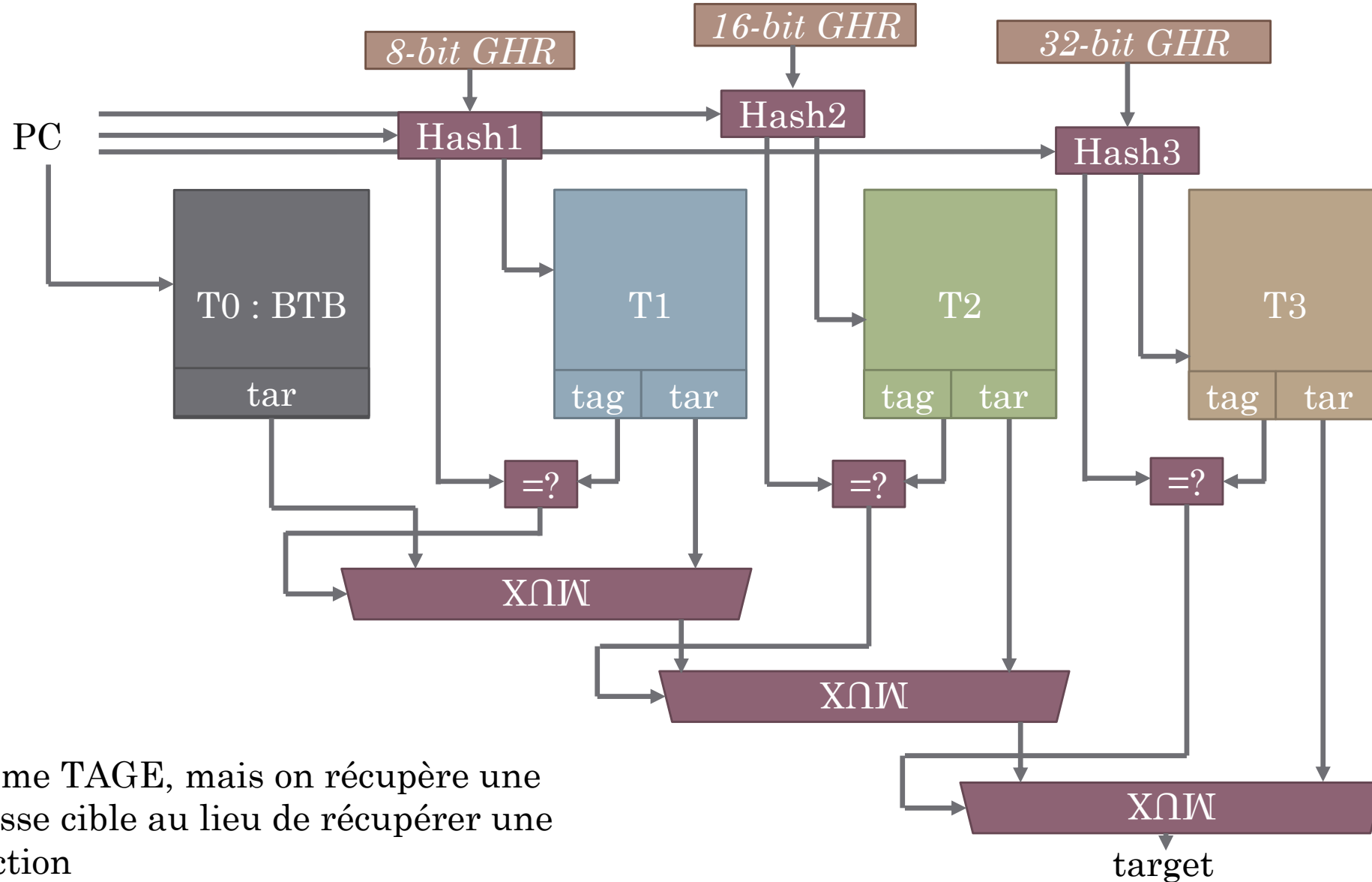
Prédiction fournit par la table qui utilise l'historique le plus long et qui a un tag correspondant

# Prédicteur moderne : TAgged GEometric (TAGE)



Lors d'une mauvaise prédiction, on alloue une entrée dans une (ou plusieurs) table plus à droite

# Prédicteur indirect moderne : Indirect Tagged GEometric (ITTAGE)



Comme TAGE, mais on récupère une adresse cible au lieu de récupérer une direction

# Prédicteur moderne : Perceptron

- Prédicteur « neural » à base de perceptron (neurone simple)
- Chaque PC est associé à un vecteur de poids
- Pour prédire, on multiplie chaque poids avec le bit correspondant de l'historique global de branchement, puis on somme le tout.
  - Pris = (somme  $\geq$  seuil)
  - Non-pris = (somme  $<$  seuil)
- Plus de détails dans le papier de Jiménez

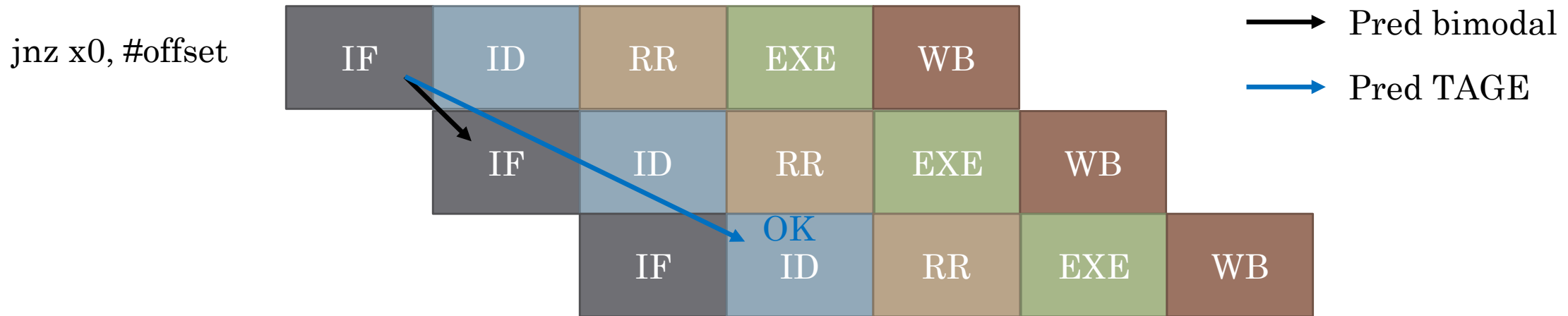
Jiménez and Lin, “Dynamic branch prediction with perceptrons”, HPCA **2001**

# Prédicteurs « Overriding »

- Il n'est pas possible de fournir une prédiction de bonne qualité avec un prédicteur petit et simple (ex. 2-bit bimodal avec 256 entrées)
- Un prédicteur grand et complexe (ex. TAGE avec plusieurs milliers d'entrées) a un temps d'accès élevé, donc pas possible de faire une prédiction par cycle
- Solution : Utiliser deux (ou plus) prédicteurs de qualité et de latence variable

# Prédicteurs « Overriding »

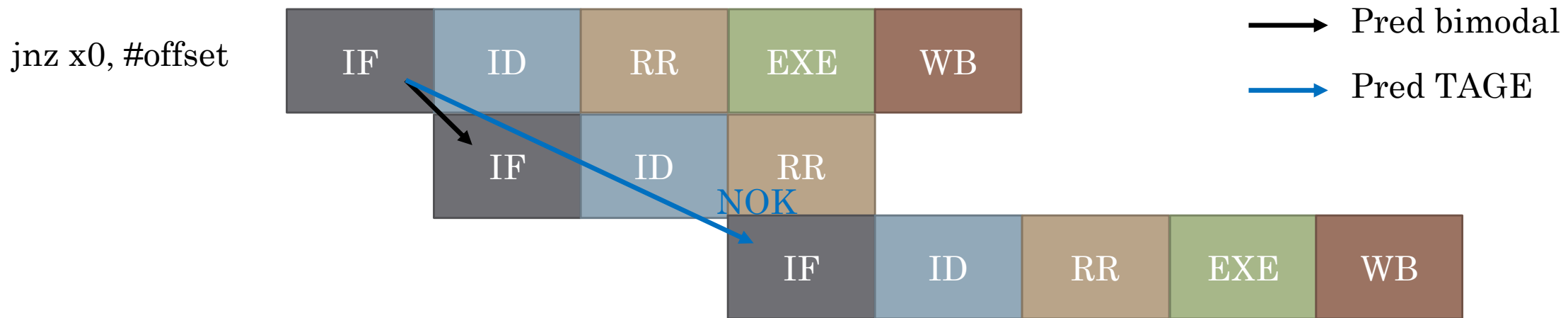
- Hypothèses :
  - 2-bit bimodal : 0 cycle de pénalité (sortie au cycle  $n$  peut contrôler l'entrée au cycle  $n + 1$ )
  - TAGE : 2 cycles de pénalité (sortie au cycle  $n$  peut contrôler l'entrée au cycle  $n + 3$ )



- Si TAGE et bimodal sont d'accord, pas de pénalité de branchement pris au niveau d'IF

# Prédicteurs « Overriding »

- Hypothèses :
  - 2-bit bimodal : 0 cycle de pénalité (sortie au cycle  $n$  peut contrôler l'entrée au cycle  $n + 1$ )
  - TAGE : 2 cycles de pénalité (sortie au cycle  $n$  peut contrôler l'entrée au cycle  $n + 3$ )



- Si TAGE et bimodal ne sont pas d'accord, TAGE « outrepassa » bimodal et vide le pipeline. Ici, on gagne un cycle par rapport à attendre EXE (bien plus dans pipeline moderne)

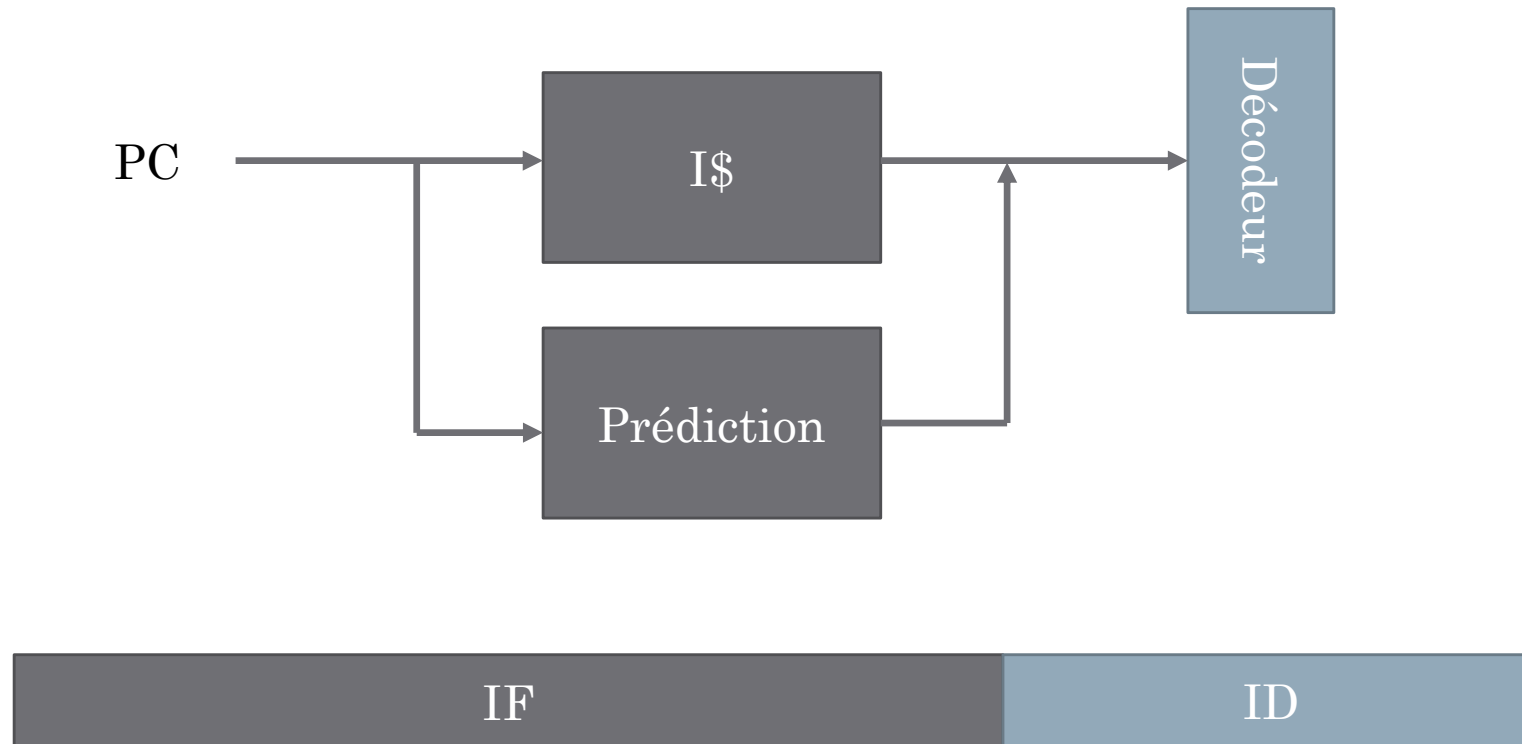
# Prédicteurs « Overriding »

- Avec cette organisation, on ne paie pas la pénalité de branchement pris lorsque le prédicteur rapide est correct
- On la paie uniquement lorsqu'il est incorrect et que le prédicteur lent est correct
- Il est possible que le prédicteur rapide soit correct et le lent incorrect, auquel cas on paiera la latence du prédicteur lent plus une mauvais prédiction de branchement, mais ce cas est très rare (si on a bien choisi ses prédicteurs)
- On peut aussi avoir plusieurs niveaux de BTB
  - L0BTB avec une pénalité de 0 cycle mais quelques dizaines d'entrées
  - L1BTB avec une pénalité de 1 cycle mais quelques centaines d'entrées
  - L2BTB avec une pénalité de 2/3/4+ cycles mais quelques milliers d'entrées
  - Pas « overriding », plutôt comme une hiérarchie de cache



# Decoupled Fetching

- Jusqu'ici, on accède au cache d'instructions et aux prédicteurs en parallèle : un miss dans le I\$ stoppe l'étage

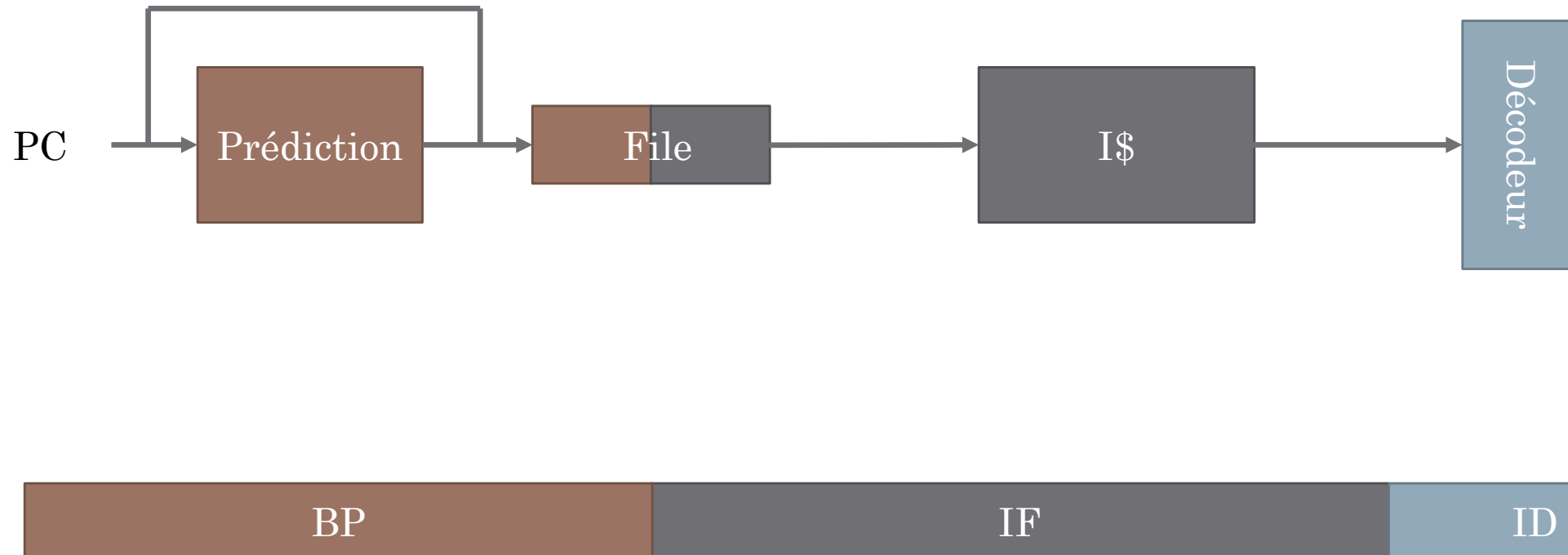


# Decoupled Fetching

- Jusqu'ici, on accède au cache d'instructions et aux prédicteurs en parallèle : un miss dans le I\$ stoppe l'étage
- Pourtant les deux structures ont des rôles différents
  - I\$ : Récupération des octets de l'instruction pour pouvoir décoder
  - Prédicteurs : Générer des adresses
  - I\$ utilise les adresses générées, mais les prédicteurs n'ont pas besoin des instructions
- Idée : Pipeliner prédiction et récupération

# Decoupled Fetching

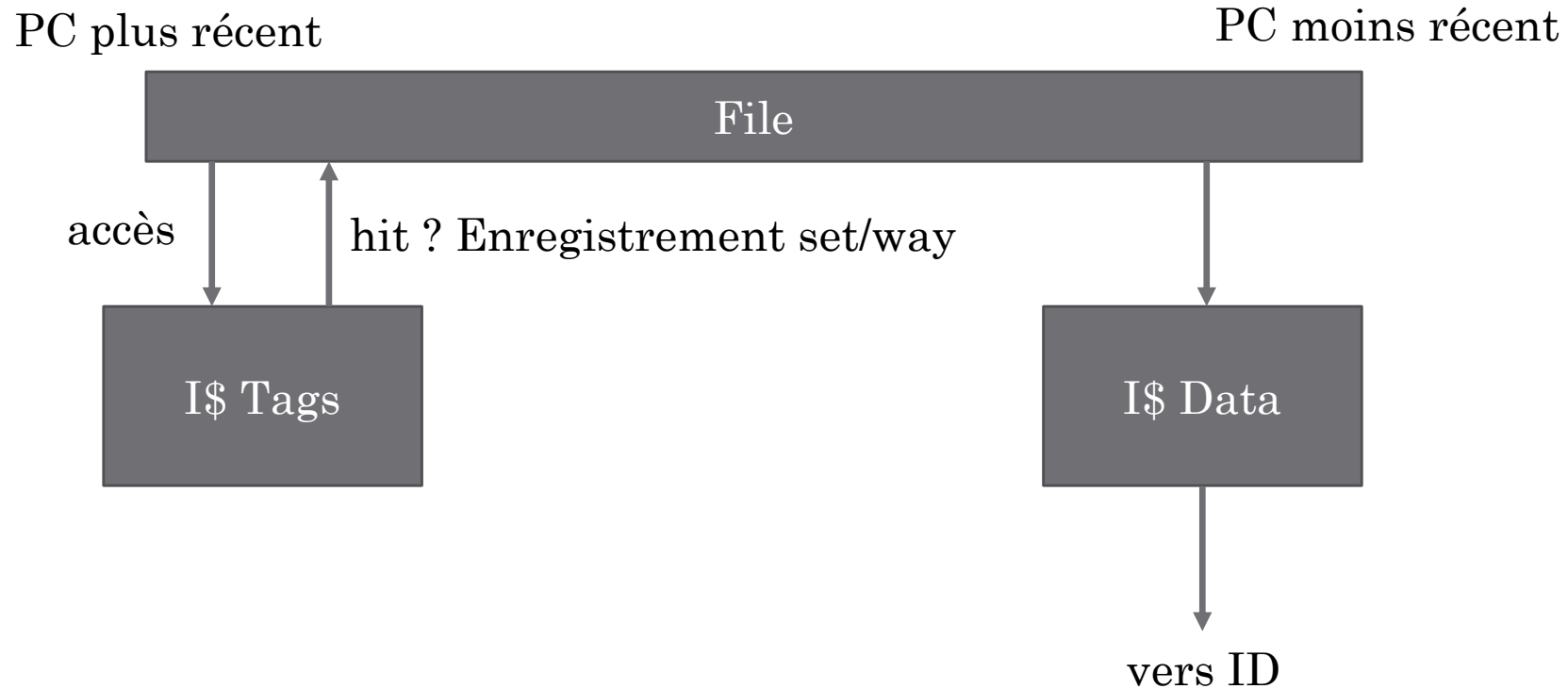
- BP : Prédicteurs + BTB
  - Assez pour générer des adresses sans l'intervention du I\$
  - Insère des adresses dans la file, IF consomme ces adresses



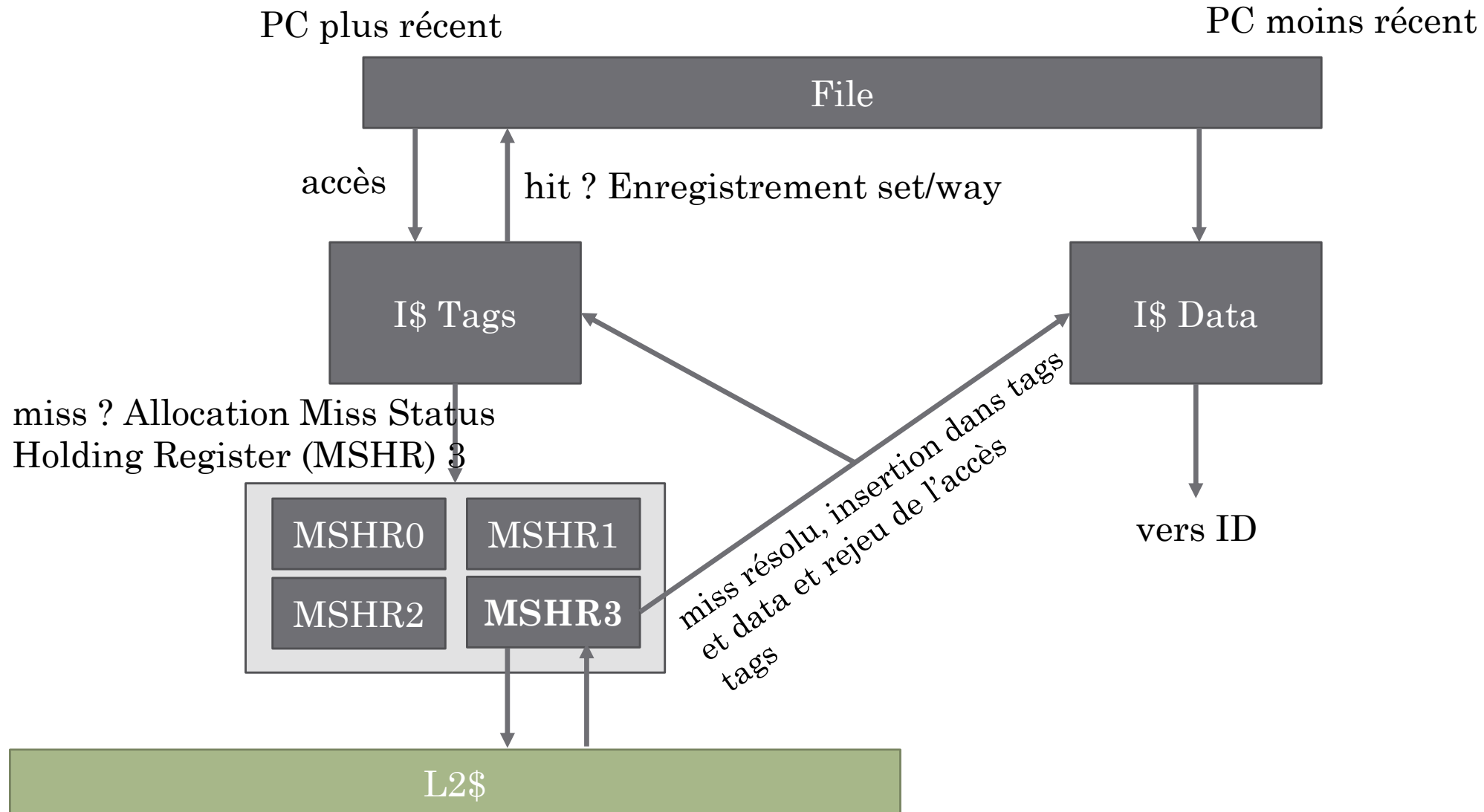
# Decoupled Fetching

- Quel intérêt ?
  - On augmente le nombre d'étages avant EXE : pénalité de mauvaise prédiction de branchement augmente
- Mais en fait plusieurs d'avantages
  1. Précharger les instructions et le TLB

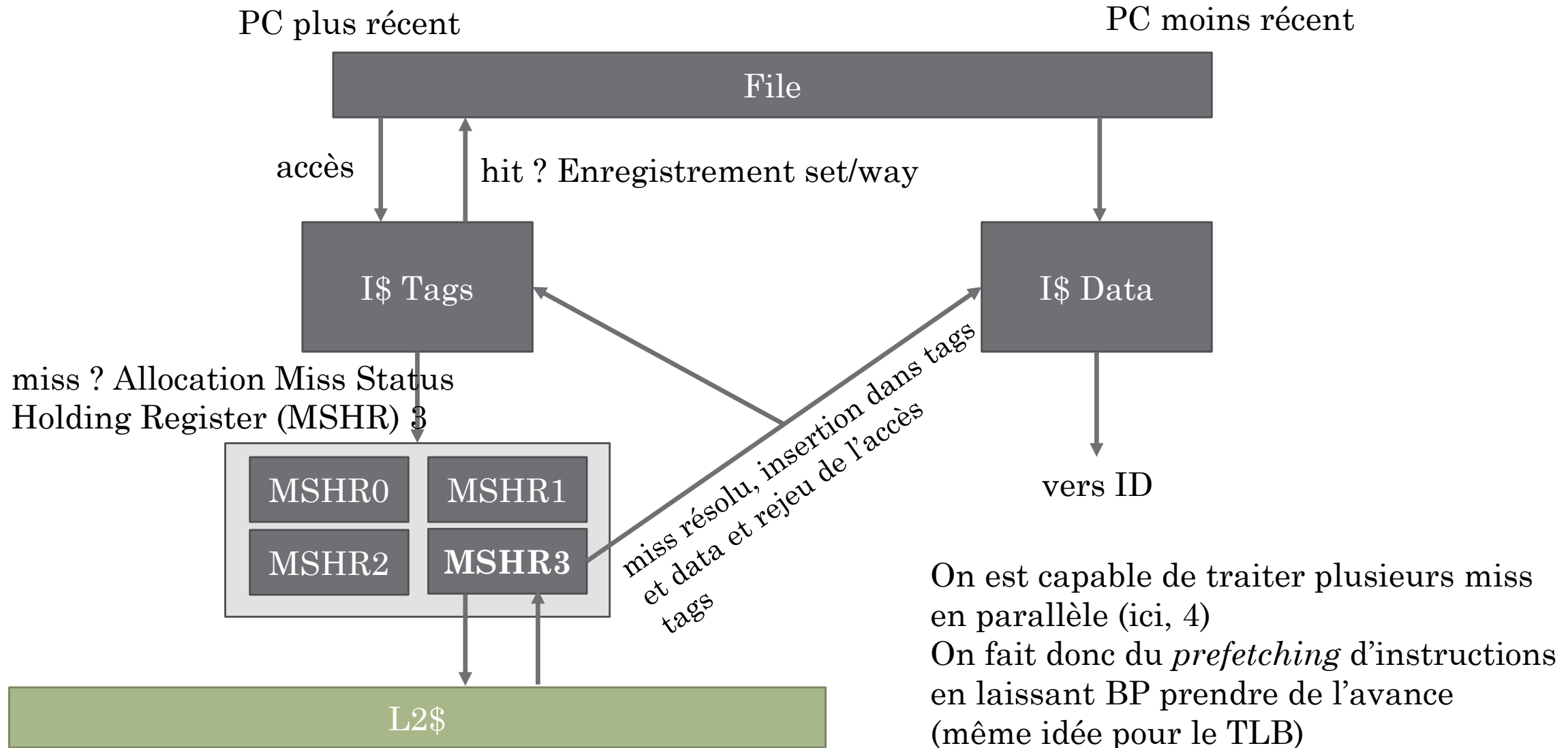
# Decoupled Fetching (DCF)



# Decoupled Fetching (DCF)



# Decoupled Fetching (DCF)



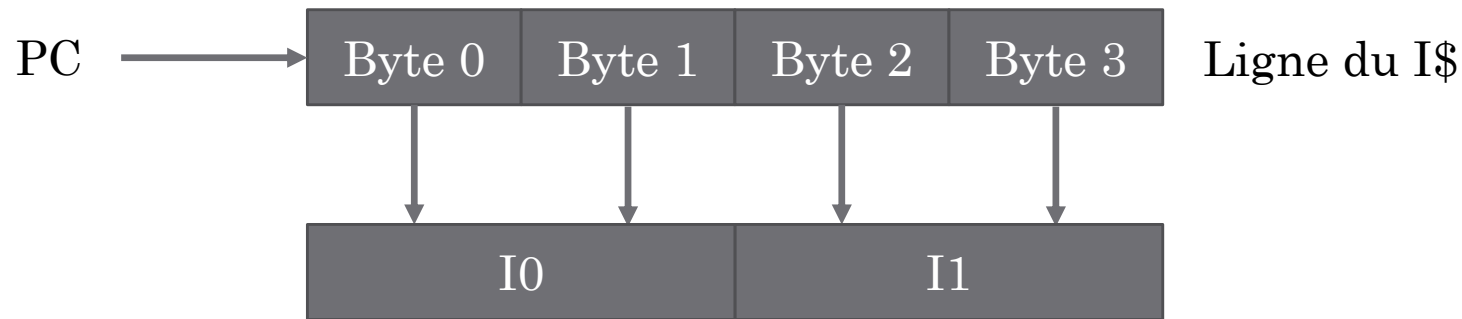
# Decoupled Fetching (DCF)

- Quel intérêt ?
  - On augmente le nombre d'étages avant EXE : pénalité de mauvaise prédiction de branchement augmente
- Mais en fait plusieurs d'avantages
  1. Précharger les instructions et le TLB
  2. Absorber certaines pénalités lorsque IF bloque mais que la file n'est pas pleine (L1BTB miss, prédicteur « overriding »)
  3. Eviter de récupérer des instructions non nécessaires



# Decoupled Fetching (DCF)

- Hypothèses : IF « classique », 2 bytes par instruction, on récupère jusqu'à deux instructions par cycle



- Imaginons que I0 soit un branchement prédit « pris »
  - On a payé de l'énergie à sortir I1 du I\$, et on va devoir jeter les bytes
- Avec DCF, on peut ajouter l'information « nombres d'instructions » dans le BTB, de telle sorte qu'on ne récupère que les bytes dont on a besoin dans le I\$