

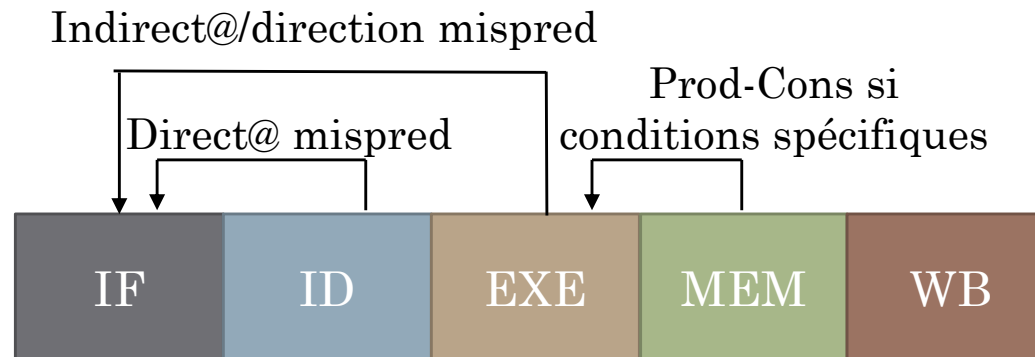
# Exécution superscalaire Exécution dans le désordre

SEOC3A – CEAMC

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

# Rappel Pipelining

- Parallélisme de pipeline en découpant l'exécution d'une instruction en plusieurs étapes
- Implémentation pipelinée introduit des dépendances *structurelles*
  - Sur les données -> Limitées via le réseau de bypass
  - Sur le contrôle -> Limitées via la prédiction de branchement



# Rappel Pipelining

- Parallélisme de pipeline en découpant l'exécution d'une instruction en plusieurs étapes
- Implémentation pipelinée introduit des dépendances *structurelles*
  - Sur les données -> Limitées via le réseau de bypass
  - Sur le contrôle -> Limitées via la prédiction de branchement
- Autre(s) limitation(s) structurelle(s) du pipeline ?

# Rappel Pipelining

- Parallélisme de pipeline en découpant l'exécution d'une instruction en plusieurs étapes
- Implémentation pipelinée introduit des dépendances *structurelles*
  - Sur les données -> Limitées via le réseau de bypass
  - Sur le contrôle -> Limitées via la prédiction de branchement
- Autre(s) limitation(s) structurelle(s) du pipeline ?
  - Chaque étage traite une seule instruction par cycle : pipeline *scalaire*
    - Performance crête : 1 IPC

# Pipeline Superscalaire

# Pipeline superscalaire

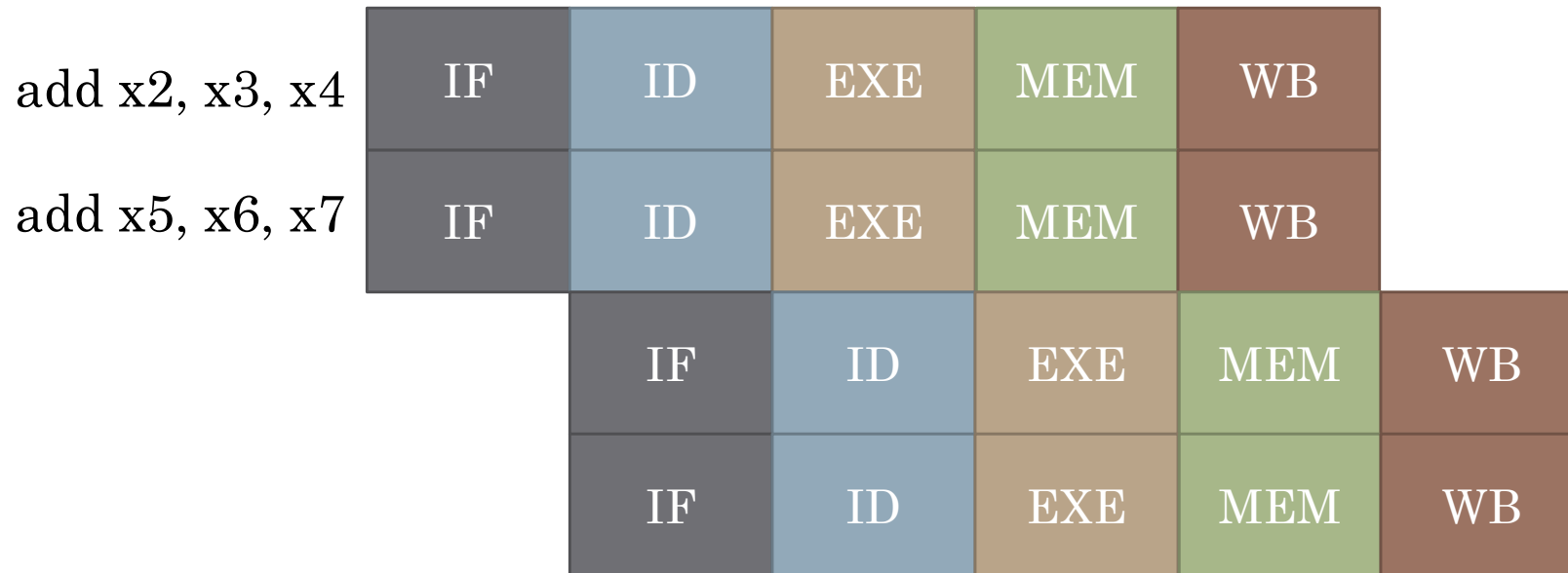
- Dupliquer les ressources pour tirer parti du parallélisme d'instruction (ILP)

```
add x2, x3, x4  
add x5, x6, x7
```

- Pas de dépendances de données
  - On peut les exécuter en même temps tout en respectant l'exécution dans l'ordre du programme

# Pipeline superscalaire

- Dupliquer les ressources pour tirer parti du parallélisme d'instruction (ILP)



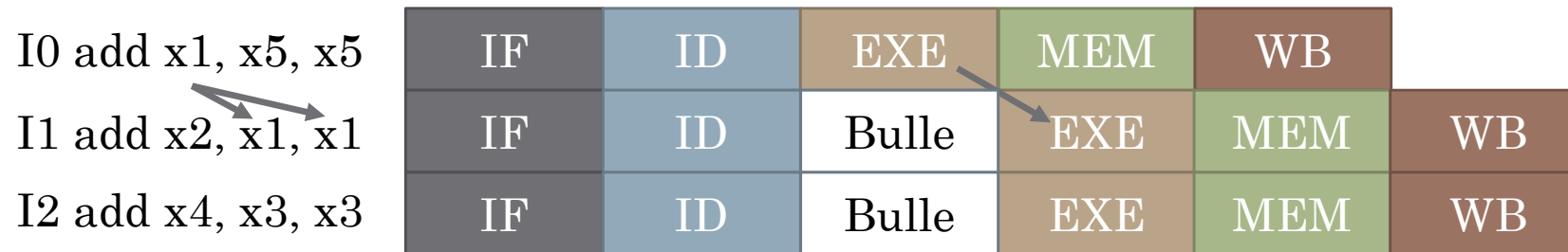
« superscalaire degré 2 »

Latence : 5 CPI

Débit : **2 IPC**

# Pipeline superscalaire

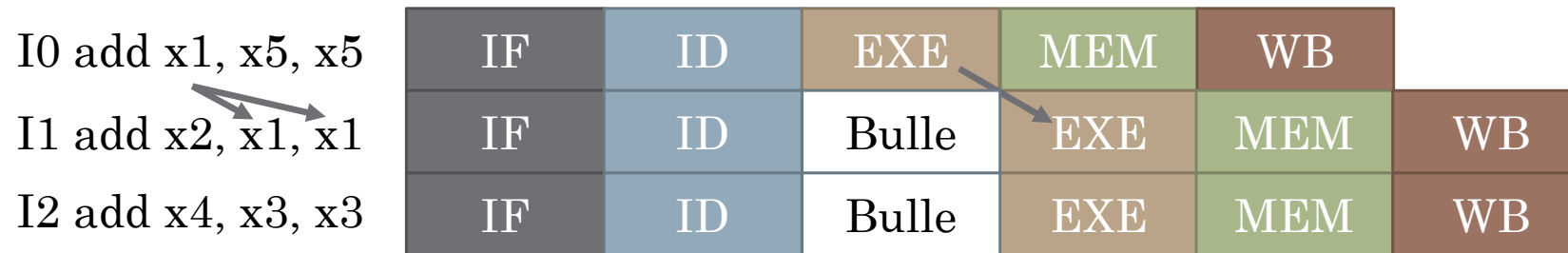
- Degré 4, 8, 16 mais...
  - Dès qu'une instruction bloque, tout le monde attend





# Pipeline superscalaire

- Degré 4, 8, 16 mais...
  - Dès qu'une instruction bloque, tout le monde attend



- Dépendance de donnée architecturale entre I0 et I1 sur x1
  - 1 cycle entre I0 et I1 : OK
- Pas de dépendance de donnée architecturale entre I2 et I1
  - 0 cycle entre I2 et I1 : OK
- Pas de dépendance de donnée architecturale entre I2 et I0
  - Pourtant, 1 cycle entre I2 et I0
  - **Dépendance de contrôle (exécution dans l'ordre du programme)**

# Pipeline superscalaire

## Question :

- Quelle exécution en pipeline superscalaire de degré 3 pour le code suivant ?
- Dépendance(s) structurelle(s) vs. architecturale(s) ?

I0 add x1, x5, x5

I1 add x2, x1, x1

I2 add x4, x3, x3

I3 ld x8, 0(x10)

I4 add x11, x8, x8

I5 sub x15, x14, x14

I6 sd x1, 0(x17)

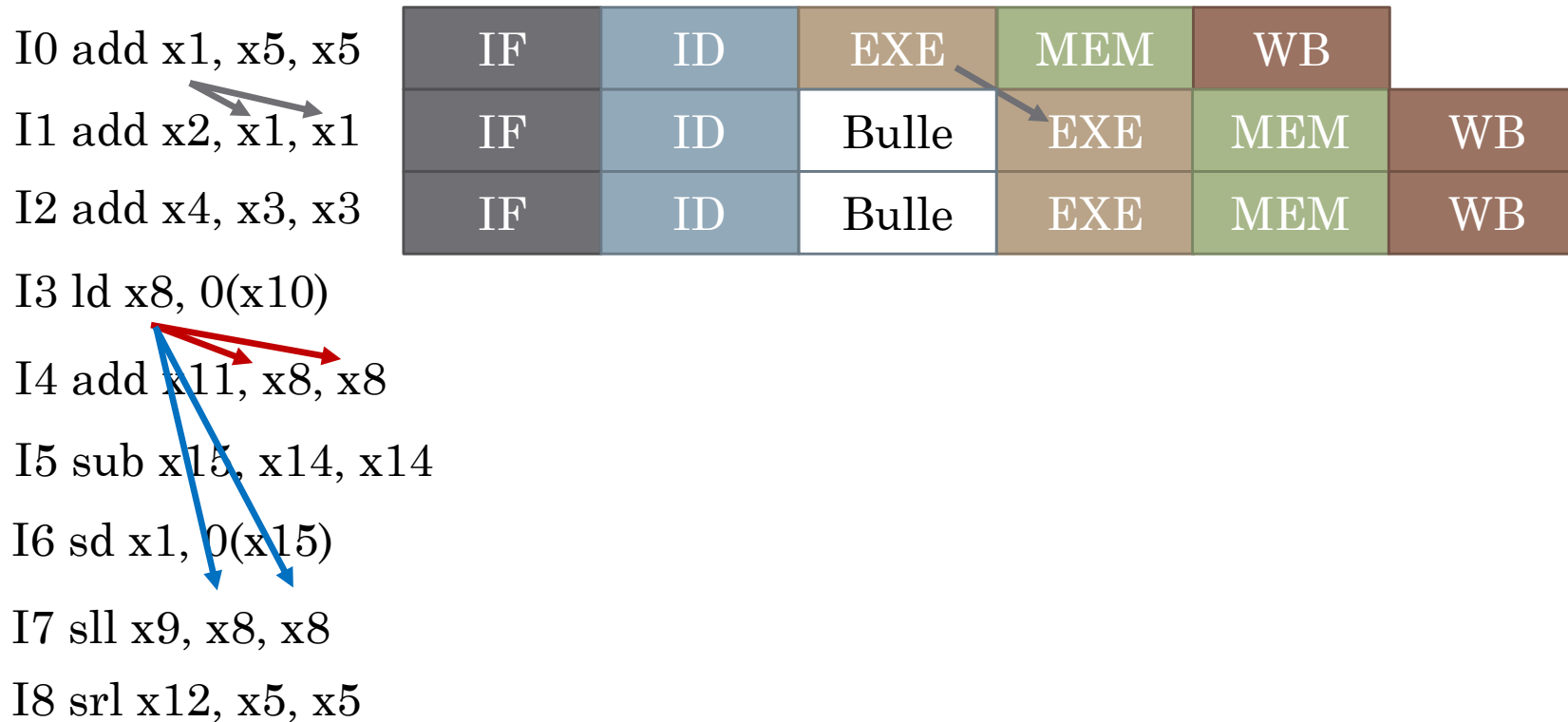
I7 sll x9, x8, x8

I8 srl x12, x5, x5

# Pipeline superscalaire

## Question :

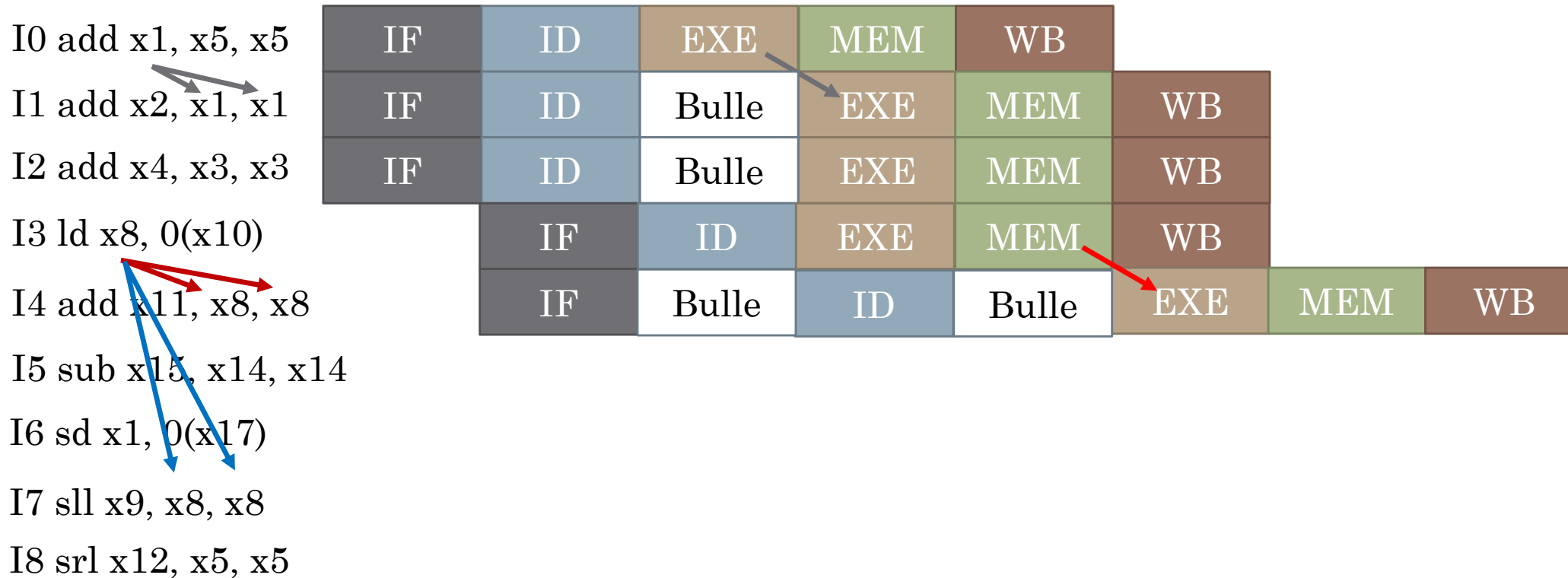
- Quelle exécution en pipeline pour le code suivant ?
- Dépendance(s) structurelle(s) vs. architecturale(s) ?



# Pipeline superscalaire

## Question :

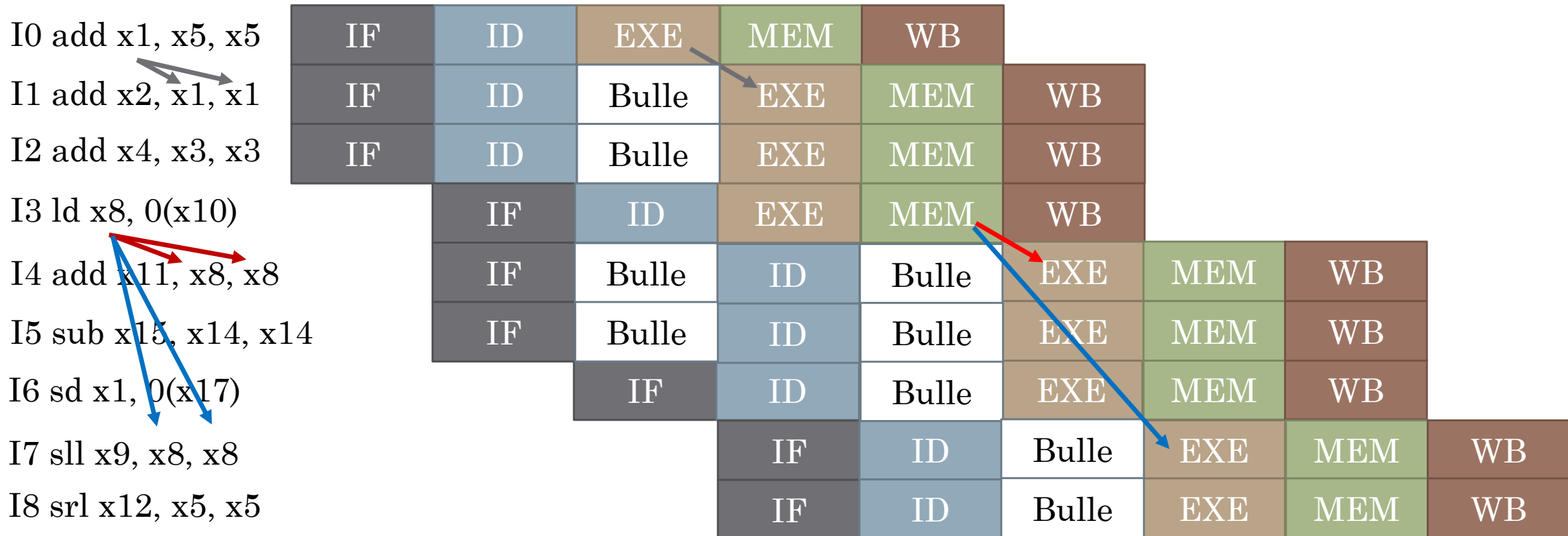
- Quelle exécution en pipeline pour le code suivant ?
- Dépendance(s) structurelle(s) vs. architecturale(s) ?



# Pipeline superscalaire

## Question :

- Quelle exécution en pipeline pour le code suivant ?
- Dépendance(s) structurelle(s) vs. architecturale(s) ?

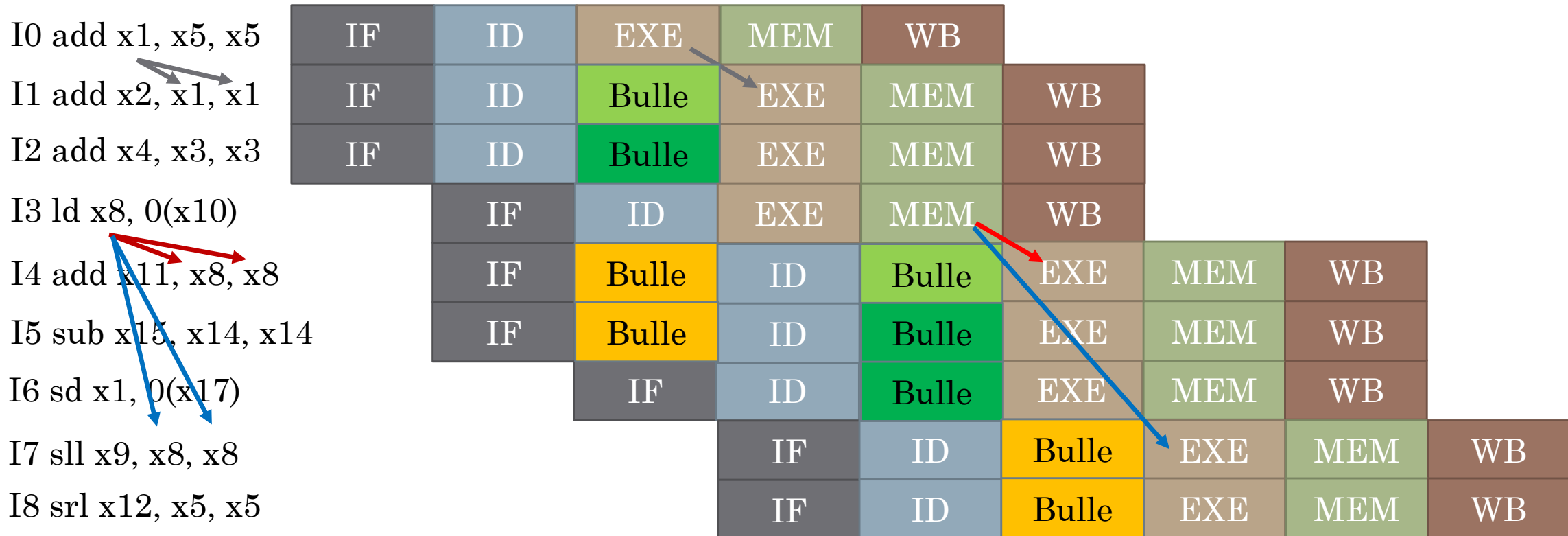


# Pipeline superscalaire

Bulle	Arch. contrôle
Bulle	Arch. donnée
Bulle	µarch. ressource

## Question :

- Quelle exécution en pipeline pour le code suivant ?
- **Dépendance(s) structurelle(s) vs. architecturale(s) ?**



# Pipeline superscalaire

- On a augmenté la performance crête via le superscalaire
  - Degré 3 => max 3 IPC
- Cependant, on met en lumière des dépendances qui empêchent de l'atteindre

## Bulle

- Architecturale (contrôle) : Si Ix apparaît **avant** Iy dans le programme, alors Iy s'exécute au plus tôt **en même temps** que Ix

I0 add x1, x5, x5

I1 add x2, x1, x1

I2 add x4, x3, x3

IF	ID	EXE	MEM	WB	
IF	ID	Bulle	EXE	MEM	WB
IF	ID	Bulle	EXE	MEM	WB

# Pipeline superscalaire

- On a augmenté la performance crête via le superscalaire
  - Degré 3 => max 3 IPC
- Cependant, on met en lumière des dépendances qui empêchent de l'atteindre

Bulle

- Architecturale (contrôle) : Si Ix apparaît **avant** Iy dans le programme, alors Iy s'exécute au plus tôt **en même temps** que Ix

Bulle

- Microarchitecturale (ressource) : Au plus *degré superscalaire* instructions peuvent être traitées par un étage lors d'un cycle donnée

I1 add x2, x1, x1

I2 add x4, x3, x3

I3 ld x8, 0(x10)

I4 add x11, x8, x8

I5 sub x15, x14, x14

IF	ID	Bulle
IF	ID	Bulle
	IF	ID
	IF	Bulle
	IF	Bulle

I4, I5 bloquées dans IF  
car I1, I2, I3 dans ID à  
cause de dépendances  
architecturales



# Pipeline superscalaire

- On a augmenté la performance crête via le superscalaire
  - Degré 3 => max 3 IPC
- On peut limiter les dépendances structurelles en améliorant la microarchitecture
  - Augmenter le degré superscalaire pour minimiser les dépendances structurelles sur les ressources. Exemple, degré 6

I0 add x1, x5, x5

I1 add x2, x1, x1

I2 add x4, x3, x3

I3 ld x8, 0(x10)

I4 add x11, x8, x8

I5 sub x15, x14, x14

IF	ID	EXE	MEM	WB			
IF	ID	Bulle	EXE	MEM	WB		
IF	ID	Bulle	EXE	MEM	WB		
IF	ID	Bulle	EXE	MEM	WB		
IF	ID	Bulle	Bulle	Bulle	EXE	MEM	WB
IF	ID	Bulle	Bulle	Bulle	EXE	MEM	WB

# Pipeline superscalaire

- On a augmenté la performance crête via le superscalaire
  - Degré 3 => max 3 IPC
- On a bien enlevé les dépendances structurelles sur les ressources (ici dans ID)
  - Mais une utilisation des ressources limitée, e.g., EXE, MEM, WB à cause des dépendances de contrôle et de données

I0 add x1, x5, x5

I1 add x2, x1, x1

I2 add x4, x3, x3

I3 ld x8, 0(x10)

I4 add x11, x8, x8

I5 sub x15, x14, x14

IF	ID	EXE	MEM	WB			
IF	ID	Bulle	EXE	MEM	WB		
IF	ID	Bulle	EXE	MEM	WB		
IF	ID	Bulle	EXE	MEM	WB		
IF	ID	Bulle	Bulle	Bulle	EXE	MEM	WB
IF	ID	Bulle	Bulle	Bulle	EXE	MEM	WB

# Pipeline superscalaire

- On a augmenté la performance crête via le superscalaire
  - Degré 3 => max 3 IPC
- Autre dépendance de contrôle : branchements
  - Prédiction de branchement permet de récupérer la cible **au prochain cycle**, pas **pendant le même cycle**
  - Ici, même exécution que superscalaire degré 3 !

I0 add x1, r0, r0

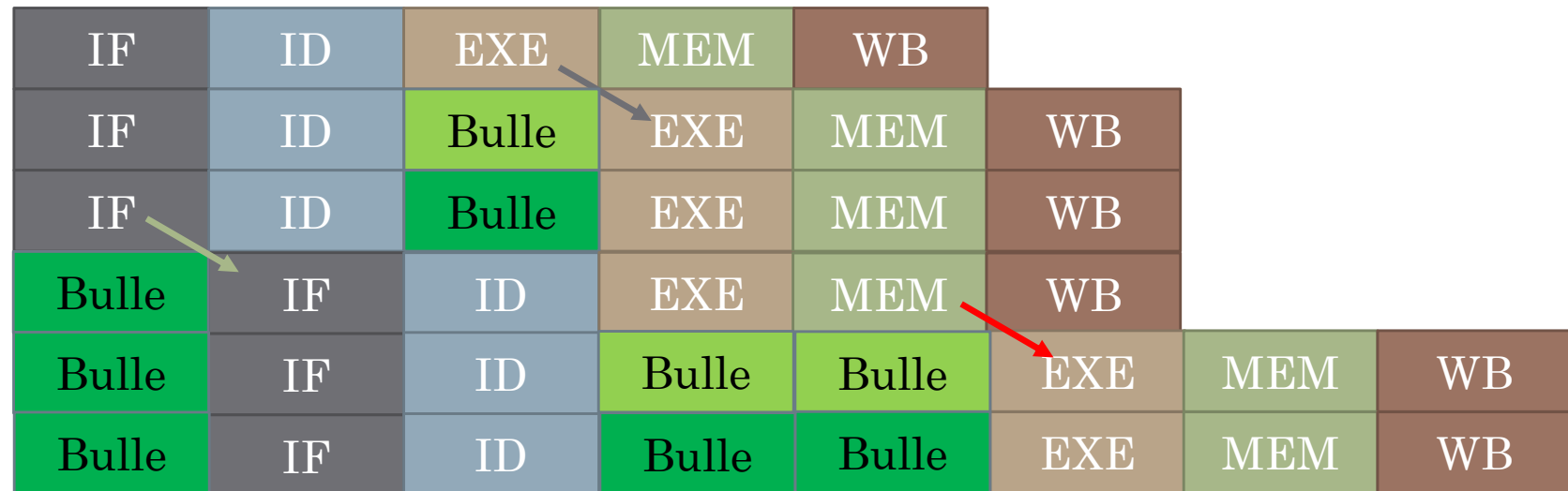
I1 add x2, x1, x1

I2 bnz x4, label

I3 ld x8, 0(x10)

I4 add x11, x8, x8

I5 sub x15, x14, x14



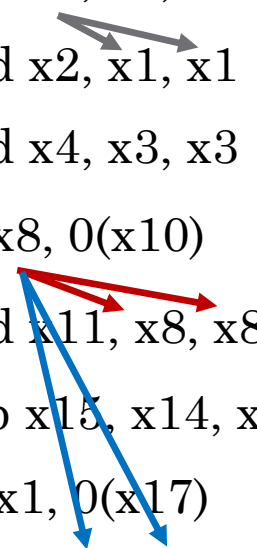
# Pipeline superscalaire

- On a augmenté la performance crête via le superscalaire
  - Degré 3 => max 3 IPC
- On peut limiter les dépendances structurelles en améliorant la microarchitecture
  - Augmenter le degré superscalaire minimise les dépendances structurelles sur les ressources.
  - Au final peu rentable à cause des dépendances de données/contrôle
- Retour à la case départ : Que spécifient vraiment les dépendances de contrôle ?
  - Les instructions sont exécutées dans l'ordre du programme

# Dépendances de contrôle

- « Les instructions sont exécutées dans l'ordre du programme »

I0 add x1, x5, x5  
I1 add x2, x1, x1  
I2 add x4, x3, x3  
I3 ld x8, 0(x10)  
I4 add x11, x8, x8  
I5 sub x15, x14, x14  
I6 sd x1, 0(x17)  
I7 sll x9, x8, x8  
I8 srl x12, x6, x6



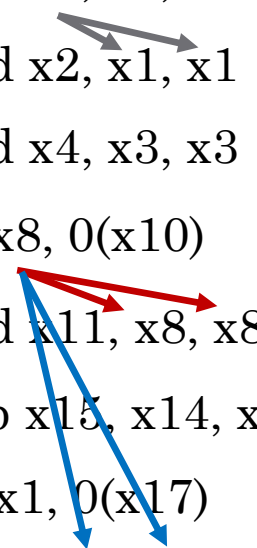
Si on observe l'état architectural après l'exécution d'une instruction Ix, alors :

- Les effets de toutes les instructions précédant Ix ont été appliqués à l'état architectural
- Les effets des instructions suivant Ix n'ont pas été appliqués à l'état architectural

# Dépendances de contrôle

- « Les instructions sont exécutées dans l'ordre du programme »

I0 add x1, x5, x5  
I1 add x2, x1, x1  
I2 add x4, x3, x3  
I3 ld x8, 0(x10)  
I4 add x11, x8, x8  
I5 sub x15, x14, x14  
I6 sd x1, 0(x17)  
I7 sll x9, x8, x8  
I8 srl x12, x6, x6



Si on observe l'état architectural après l'exécution d'une instruction Ix, alors :

- Les effets de toutes les instructions précédant Ix ont été appliqués à l'état architectural
- Les effets des instructions suivant Ix n'ont pas été appliqués à l'état architectural

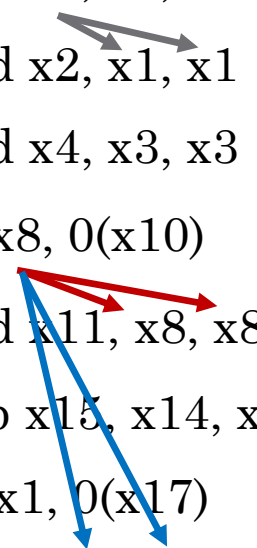
Exemple si on observe l'état arch. après I3 :

- x1 contient  $x5 + x5$  (I0), x2 contient  $x1 + x1$  (I1), x4 contient  $x3 + x3$  (I2), x8 contient  $[x10 + 0]$  (I3)
- x11 ne contient pas encore  $x8 + x8$  (I4), etc.

# Dépendances de contrôle

- « Les instructions sont exécutées dans l'ordre du programme »

I0 add x1, x5, x5  
I1 add x2, x1, x1  
I2 add x4, x3, x3  
I3 ld x8, 0(x10)  
I4 add x11, x8, x8  
I5 sub x15, x14, x14  
I6 sd x1, 0(x17)  
I7 sll x9, x8, x8  
I8 srl x12, x6, x6

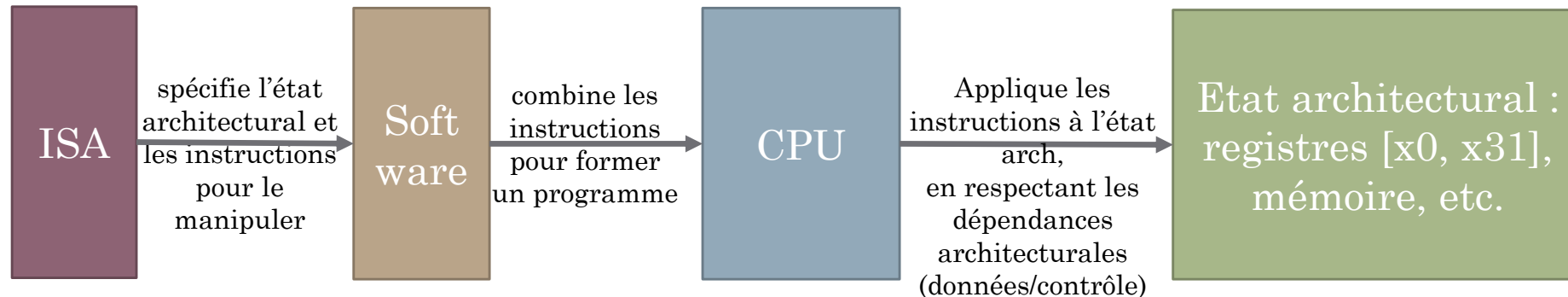


Si on observe l'état architectural après l'exécution d'une instruction Ix, alors :

- Les effets de toutes les instructions précédant Ix ont été appliqués à l'état architectural
  - Les effets des instructions suivant Ix n'ont pas été appliqués à l'état architectural
- La clé est « observe »
- La microarchitecture peut garder un état microarchitectural qui ne respecte pas la contrainte d'ordre, tant que cet état n'est pas **observable** par le logiciel

# Etat architectural vs. microarchitectural

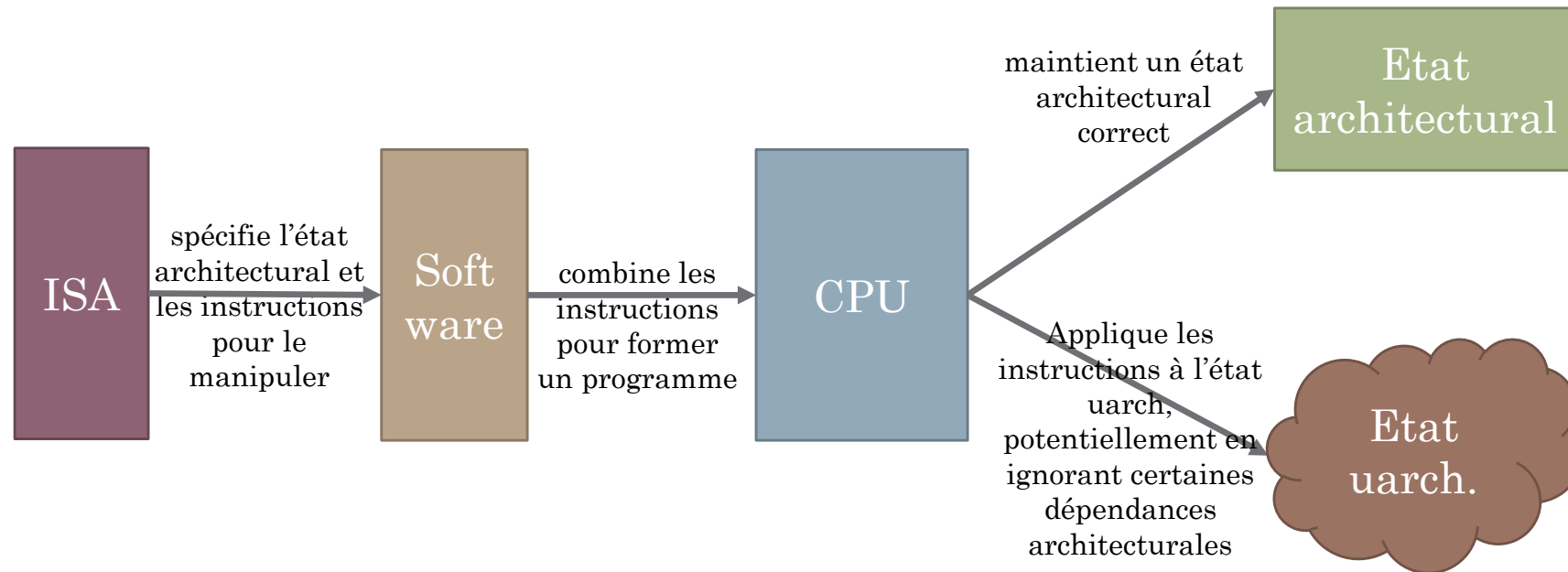
- Etat architectural observable et manipulable par le logiciel
  - Manipulations contraintes par la spécification (notamment ordre du programme)





# Etat architectural vs. microarchitectural

- Etat architectural observable et manipulable par le logiciel
  - Manipulations contraintes par la spécification (notamment ordre du programme)
- 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 ordre du programme)
  - 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
  - Utiliser un état microarchitectural pour exécuter les instructions sans respecter l'ordre du programme
    - « Migrer » les modifications de l'état microarchitectural vers l'état architectural dans l'ordre du programme pour toujours avoir un état architectural correct à « montrer » au logiciel
- On le faisait déjà avec la prédiction de branchement, en ne respectant pas la dépendance de contrôle sur les branchements

# Exécution dans le désordre

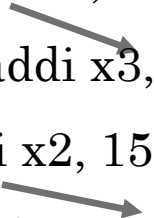
Concept

# Exécution dans le désordre

- 1<sup>er</sup> essai, on tente de minimiser les modifications du pipeline
  - Exécution dès que possible => EXE dans le désordre
  - Résultats rendus visibles au logiciel dans l'ordre => WB dans l'ordre

**Question :** Superscalaire degré 4. Quelle exécution ? Exécution correcte ?

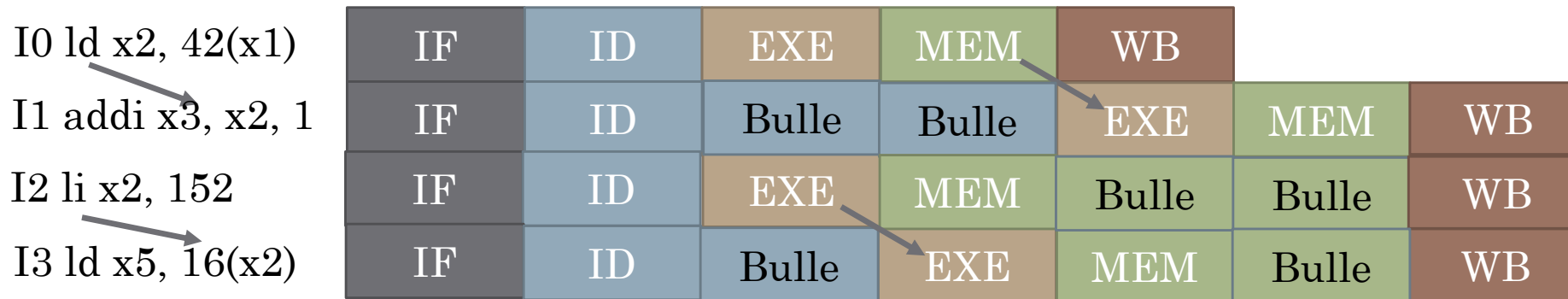
I0 ld x2, 42(x1)  
I1 addi x3, x2, 1  
I2 li x2, 152  
I3 ld x5, 16(x2)



# Exécution dans le désordre

- 1<sup>er</sup> essai, on tente de minimiser les modifications du pipeline
  - Exécution dès que possible => EXE dans le désordre
  - Résultats rendus visibles au logiciel dans l'ordre => WB dans l'ordre

**Question :** Superscalaire degré 4. Quelle exécution ? Exécution correcte ?



Etat arch.

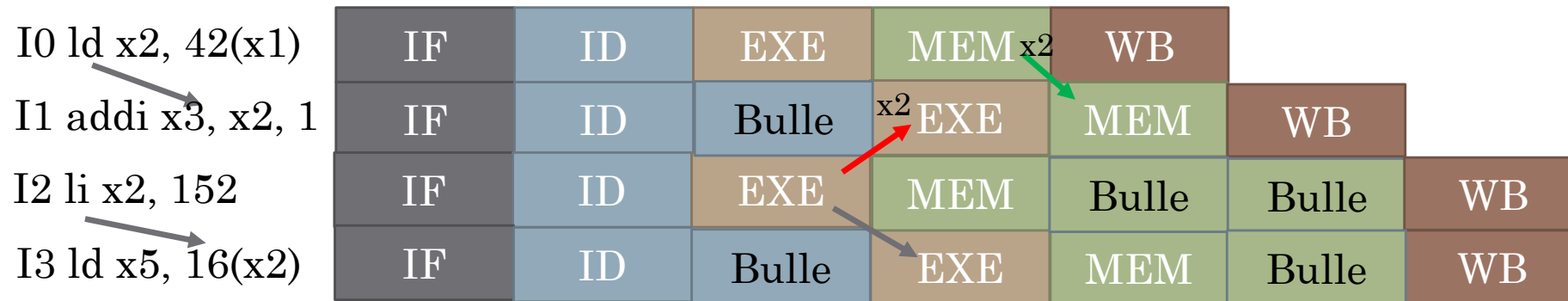
- x2 contient  $[x1 + 42]$

Etat uarch.

- Reg pipeline MEM[I2] contient  $x2 = 152$
- Reg pipeline MEM[I3] contient  $x5 = [152 + 16]$

# Exécution dans le désordre

- Problème :  $x2$  produit par  $li\ x2, 152$  présent sur le bypass avant  $x2$  de  $ld\ x2, 42(x1)$ 
  - $add\ x3, x2, 1$  s'exécute avec la mauvaise valeur de  $x2$  (152 et non  $[x1 + 42]$ )
  - Pourtant, WB dans l'ordre



# Exécution dans le désordre

- Problème : De nouvelles dépendances de données architecturales sont révélées par l'exécution dans le désordre
  - On connaît producteur-consommateur (*Read-after-Write, RaW*) →

ld x2, 42(x1)

addi x3, x2, 1

li x2, 152

ld x5, 16(x2)

# Exécution dans le désordre

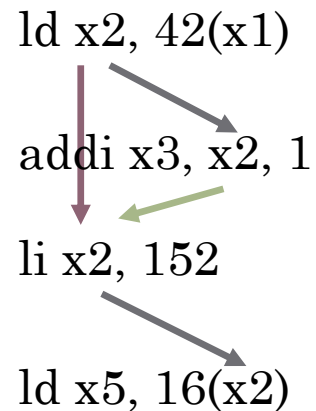
- Problème : De nouvelles dépendances de données architecturales sont révélées par l'exécution dans le désordre
  - On connaît producteur-consommateur (*Read-after-Write, RaW*)  $\longrightarrow$
- Avec exécution dans le désordre, autres dépendances à respecter
  - consommateur-producteur (*Write-after-Read, WaR*)  $\longrightarrow$

```
ld x2, 42(x1)
addi x3, x2, 1
li x2, 152
ld x5, 16(x2)
```



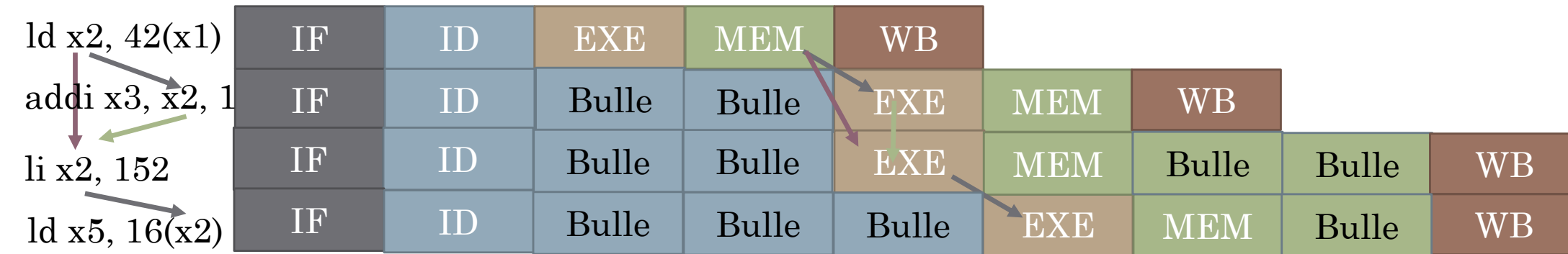
# Exécution dans le désordre

- Problème : De nouvelles dépendances de données architecturales sont révélées par l'exécution dans le désordre
  - On connaît producteur-consommateur (*Read-after-Write, RaW*)  $\longrightarrow$
- Avec exécution dans le désordre, autres dépendances à respecter
  - consommateur-producteur (*Write-after-Read, WaR*)  $\longrightarrow$
  - producteur-producteur (*Write-after-Write, WaW*)  $\longrightarrow$



# Exécution dans le désordre

- 1<sup>er</sup> essai, on tente de minimiser les modifications au pipeline
  - Exécution dès que possible => EXE dans le désordre
  - Résultats rendus visibles au logiciel dans l'ordre => WB dans l'ordre
  - En respectant WaR/WaW en plus de RaW



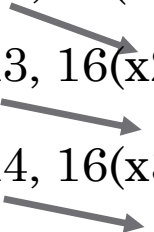
- Aucune exécution dans le désordre...

# Exécution dans le désordre

- Admettons que le compilateur ne génère pas de WaW/WaR
  - Exécution dès que possible => EXE dans le désordre
  - Résultats rendus visibles au logiciel dans l'ordre => WB dans l'ordre

**Question :** Superscalaire degré 4. Quelle exécution ? Exécution correcte ?

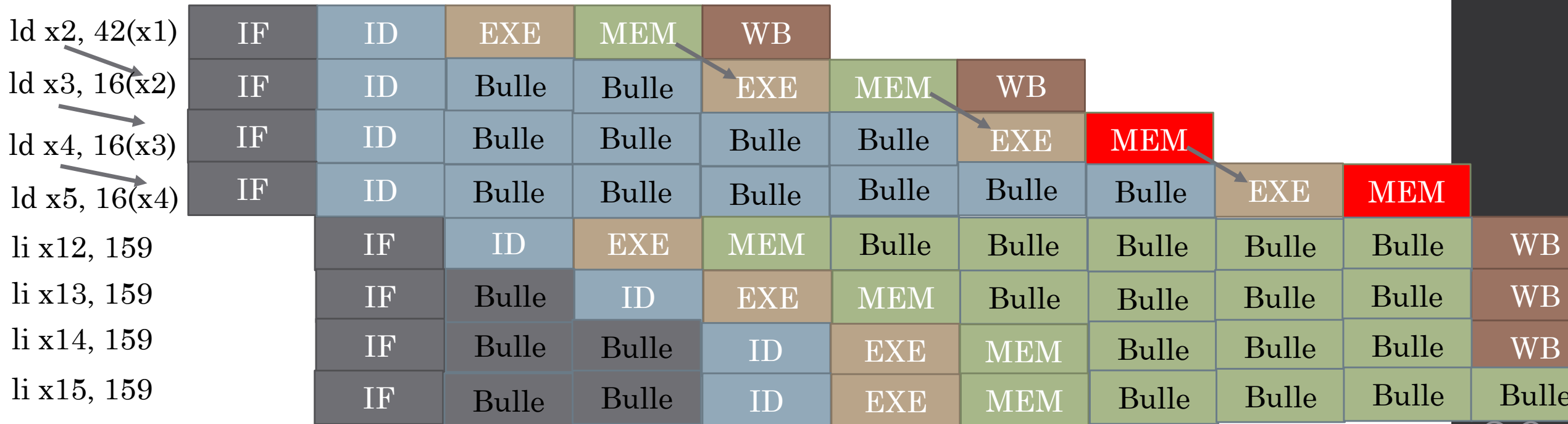
```
ld x2, 42(x1)
ld x3, 16(x2)
ld x4, 16(x3)
ld x5, 16(x4)
li x12, 159
li x13, 159
li x14, 159
li x15, 159
```



# Exécution dans le désordre

- **Deadlock matériel :**

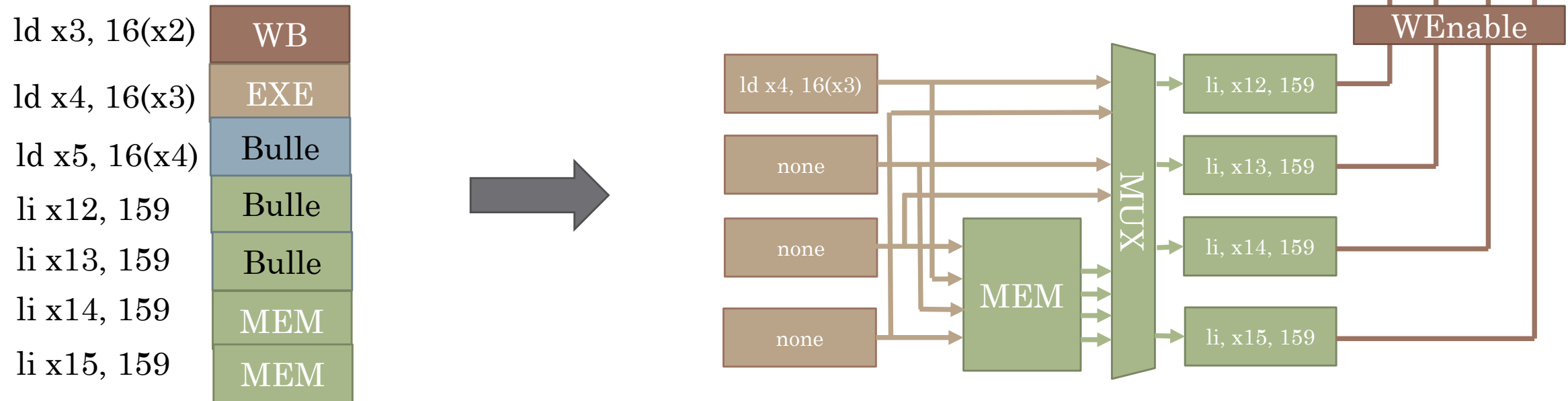
- Instructions plus jeunes attendent que ld x4, 16(x3) quitte WB pour avancer dans WB
- ld x4, 16(x3) ne peut pas entrer dans MEM. **Pourquoi ?**



# Exécution dans le désordre

- **Deadlock matériel :**

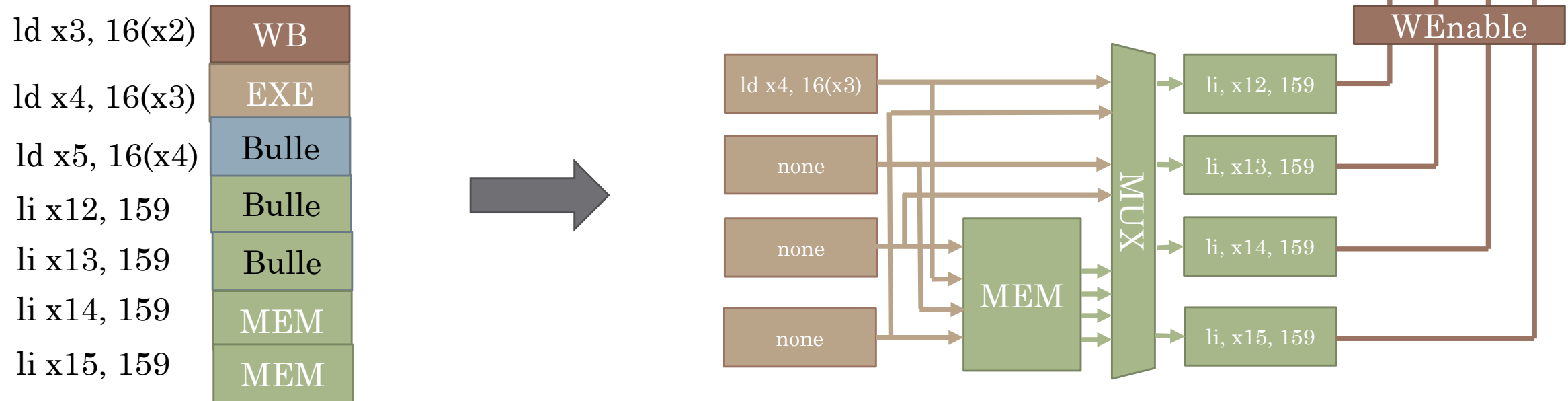
- Car on conserve les résultats calculés dans les registres de pipeline de EXE/MEM en attendant WB



# Exécution dans le désordre

- **Deadlock matériel :**

- Car on conserve les résultats calculés dans les registres de pipeline de EXE/MEM en attendant WB
- Si *ld x4, 16(x3)* écrit par-dessus *li x12, 159*, la nouvelle valeur de x12 est perdue
- *li x12, 159* ne peut pas écrire x12 car *ld x4, 16(x3)* n'a pas encore écrit x4

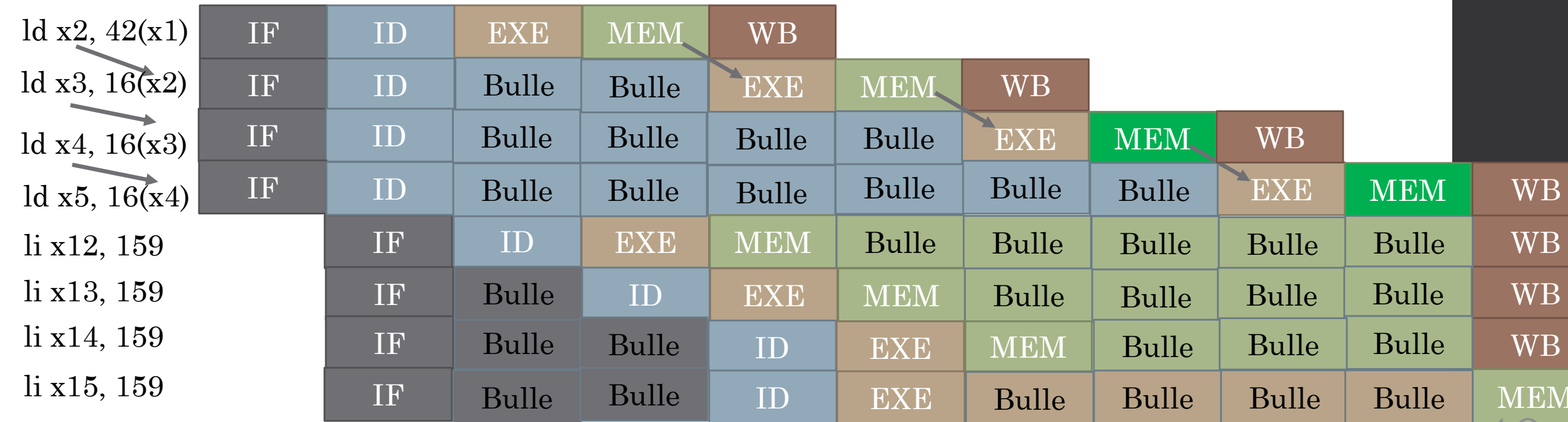


# Exécution dans le désordre

- Exécuter dans le désordre mais écriture des registres dans l'ordre
  - Performance fortement limitée par les dépendances de données WaR/WaW quand présentes
  - On doit garder les résultats dans le bypass (EXE et MEM) en attendant WB
  - **Deadlock matériel** : ld ne peut pas entrer dans MEM car les 4 emplacements sont occupés par des instructions plus récentes
- Une solution serait d'empêcher une instruction plus jeune d'entrer dans EXE (resp. MEM) si elle prend le dernier emplacement ET qu'une instruction plus vieille n'est pas encore passée par EXE (resp. MEM)

# Exécution dans le désordre

- Si on empêche une instruction plus jeune de prendre le dernier emplacement dans les registres de pipeline





# Exécution dans le désordre

- Exécuter dans le désordre mais écriture des registres dans l'ordre
  - Performance fortement limitée par les dépendances de données WaR/WaW quand présentes
  - On doit garder les résultats dans le bypass (EXE et MEM) en attendant WB
  - **Deadlock matériel** : ld ne peut pas entrer dans MEM car les 4 emplacements sont occupés par des instructions plus récentes
- Une solution serait d'empêcher une instruction plus jeune d'entrer dans EXE si elle prend le dernier emplacement ET qu'une instruction plus vieille n'est pas encore passée par EXE
  - Limite encore l'exécution dans le désordre
  - Ne résout pas le problèmes des dépendances WaW/WaR

# Exécution dans le désordre

- Concrètement :
  - WaW/WaR : Il peut exister plusieurs version d'un même registre architectural dans le pipeline, mais aucune façon des les différencier, donc on bloque si WaW/WaR
  - Deadlock : il peut y avoir plus de résultats en vol attendant d'être écrit dans WB que de registres de pipelines pour un étage donné

# Exécution dans le désordre

- Concrètement :
  - WaW/WaR : Il peut exister plusieurs version d'un même registre architectural dans le pipeline, mais aucune façon des les différencier, donc on bloque si WaW/WaR
  - Deadlock : il peut y avoir plus de résultats en vol attendant d'être écrit dans WB que de registres de pipelines pour un étage donné
- Les deux problèmes se résolvent en donnant à chaque instruction son propre espace de stockage physique où écrire son résultat, qui a un *identifiant (nom)* qui lui est propre
  - Registres arch. renommés en registres microarch. (physiques)
  - Permet d'effectuer WB dans le désordre sans deadlock
  - Permet d'ignorer WaR/WaW : Les versions différentes d'un registre arch. sont identifiables via leur *nom physique*

# Exécution dans le désordre

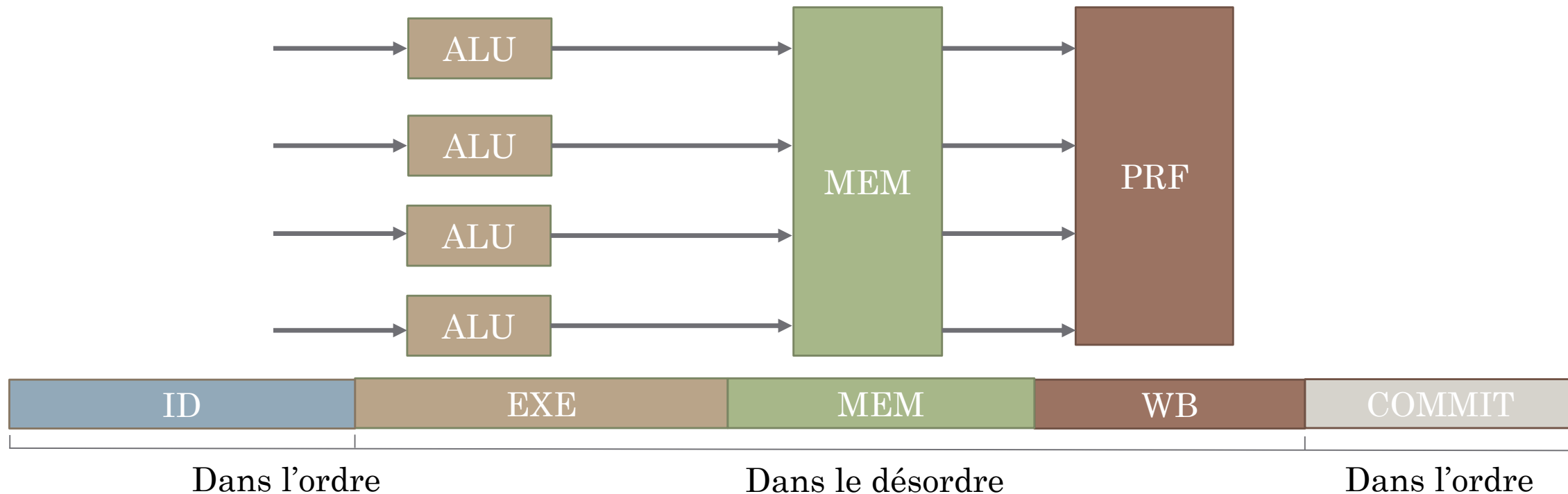
- Concrètement :
  - WaW/WaR : Il peut exister plusieurs version d'un même registre architectural dans le pipeline, mais aucune façon des les différencier, donc on bloque si WaW/WaR
  - Deadlock : il peut y avoir plus de résultats en vol attendant d'être écrit dans WB que de registres de pipelines pour un étage donné
- Les deux problèmes se résolvent en donnant à chaque instruction son propre espace de stockage physique ou écrire son résultat, qui a un *identifiant (nom)* qui lui est propre
  - Registres arch. renommés en registres microarch. (physiques)
  - Permet d'effectuer WB dans le désordre sans deadlock
  - Permet d'ignorer WaR/WaW : Les versions différentes d'un registre arch. sont identifiables via leur *nom physique*
- On doit toujours « copier » le registre temporaire vers le registre architectural dans l'ordre du programme

# Exécution dans le désordre

Renommage de registres

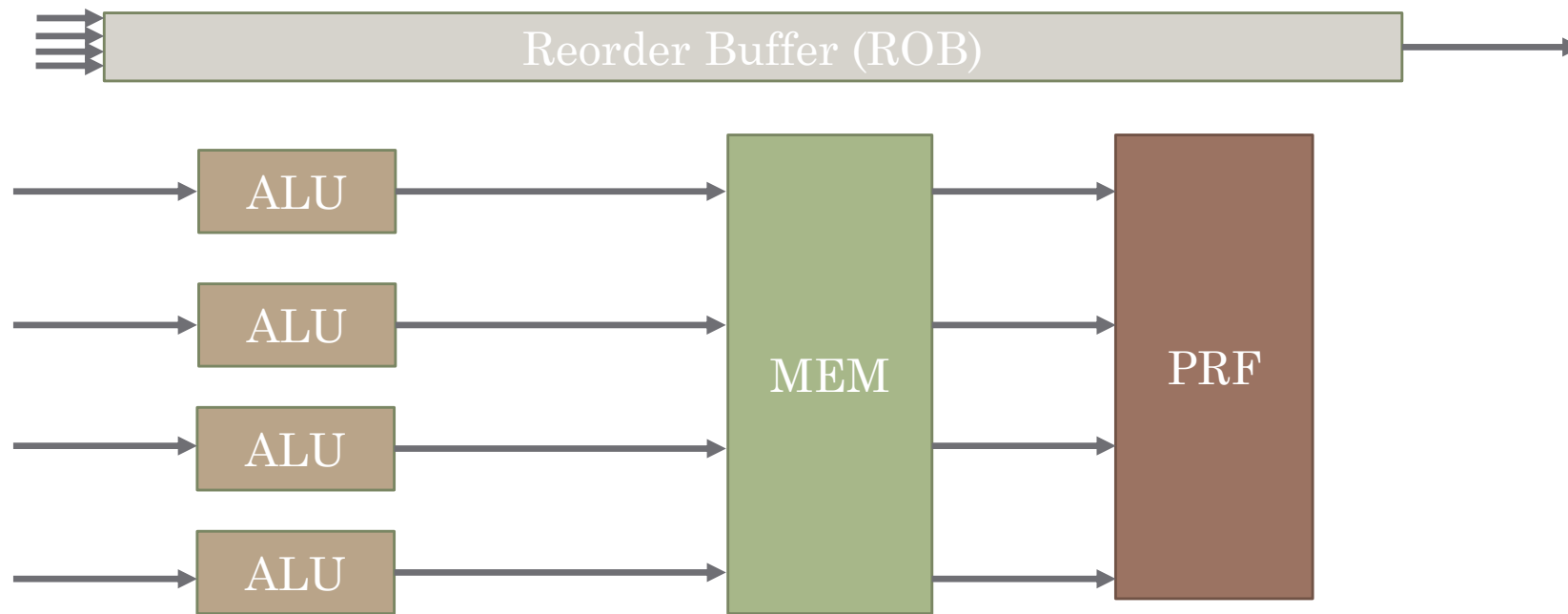
# Exécution dans le désordre – 2<sup>nd</sup>e itération

- On ajoute un fichier de registre physiques « étendu » (PRF)
  - Contient l'état architectural et l'état microarchitectural
  - Chaque instruction obtient un registre où écrire à ID



# Exécution dans le désordre – 2<sup>nd</sup>e itération

- On ajoute un fichier de registre microarchitectural
  - On doit aussi garder une trace des instructions pour pouvoir les traiter dans l'ordre => ROB (structure First In First Out) + **Commit**
    - Va notamment nous aider pour libérer les registres physiques



# Exécution dans le désordre – 2<sup>nd</sup>e itération

- On admet pour le moment que chaque instruction détermine le nom des ses registres physiques sources dans ID
  - Plus de détails sur ce mécanisme de **renommage** à venir
- Plus de WaW/WaR

ld x2, 42(x1)

ld x3, 16(x2)

li x2, 152

ld x5, 16(x2)



# Exécution dans le désordre – 2<sup>nd</sup>e itération

- On admet pour le moment que chaque instruction détermine le nom des ses registres physiques sources dans ID
  - Plus de détails sur ce mécanisme de **renommage** à venir
- Plus de WaW/WaR
  - Car ld x2, 42(x1) et li x2, 152 écrivent dans leur propre registre *physique*

ld x2, 42(x1)

ld x3, 16(x2)

li x2, 152

ld x5, 16(x2)

# Exécution dans le désordre – 2<sup>nd</sup>e itération

- On admet pour le moment que chaque instruction détermine le nom des ses registres physiques sources dans ID
  - Plus de détails sur ce mécanisme de **renommage** à venir
- Plus de WaW/WaR
  - Car ld x2, 42(x1) et li x2, 152 écrivent dans leur propre registre *physique*
  - ld x3, 16(x2) lit le registre *physique* écrit par ld x2, 42(x1)

ld x2, 42(x1)

ld x3, 16(x2)

li x2, 152

ld x5, 16(x2)

# Exécution dans le désordre – 2<sup>nd</sup>e itération

- On admet pour le moment que chaque instruction détermine le nom des ses registres physiques sources dans ID
  - Plus de détails sur ce mécanisme de **renommage** à venir
- Plus de WaW/WaR
  - Car ld x2, 42(x1) et li x2, 152 écrivent dans leur propre registre *physique*
  - ld x3, 16(x2) lit le registre *physique* écrit par ld x2, 42(x1)
  - ld x5, 16(x2) lit le registre *physique* écrit par li x2, 152

ld x2, 42(x1)

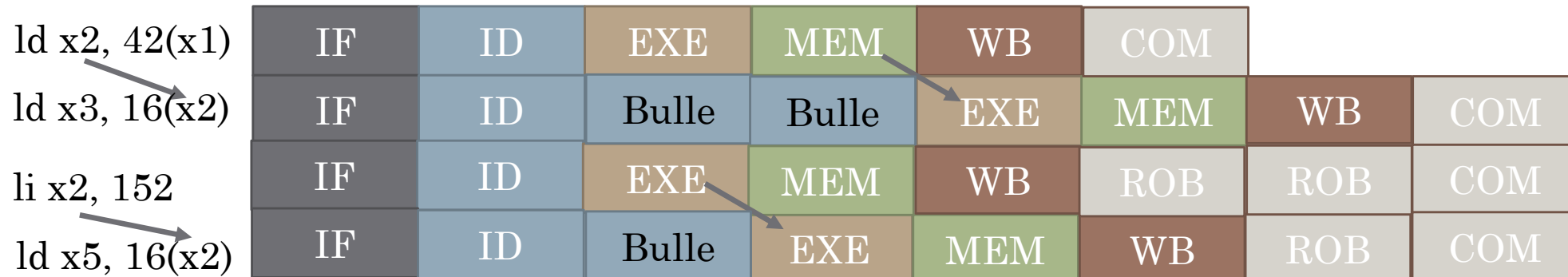
ld x3, 16(x2)

li x2, 152

ld x5, 16(x2)

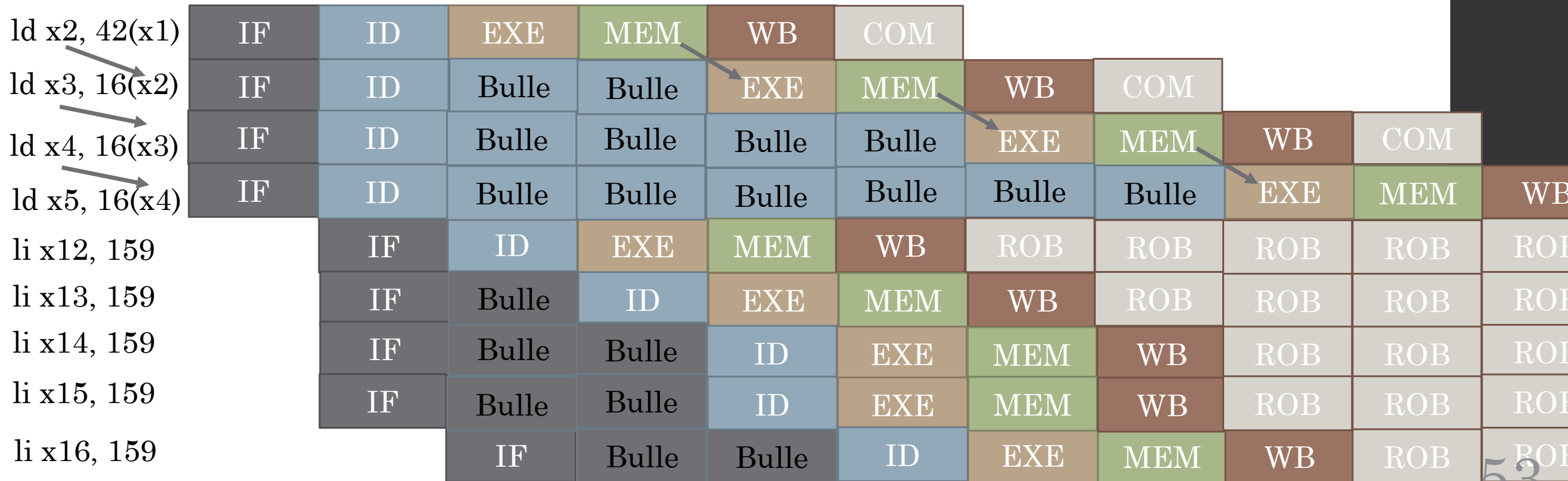
# Exécution dans le désordre – 2<sup>nd</sup>e itération

- On admet pour le moment que chaque instruction détermine le nom des ses registres physiques sources dans ID
  - Plus de détails sur ce mécanisme de **renommage** à venir
- Plus de WaW/WaR
  - Car ld x2, 42(x1) et li x2, 152 écrivent dans leur propre registre *physique*
  - ld x3, 16(x2) lit le registre *physique* écrit par ld x2, 42(x1)
  - ld x5, 16(x2) lit le registre *physique* écrit par li x2, 15



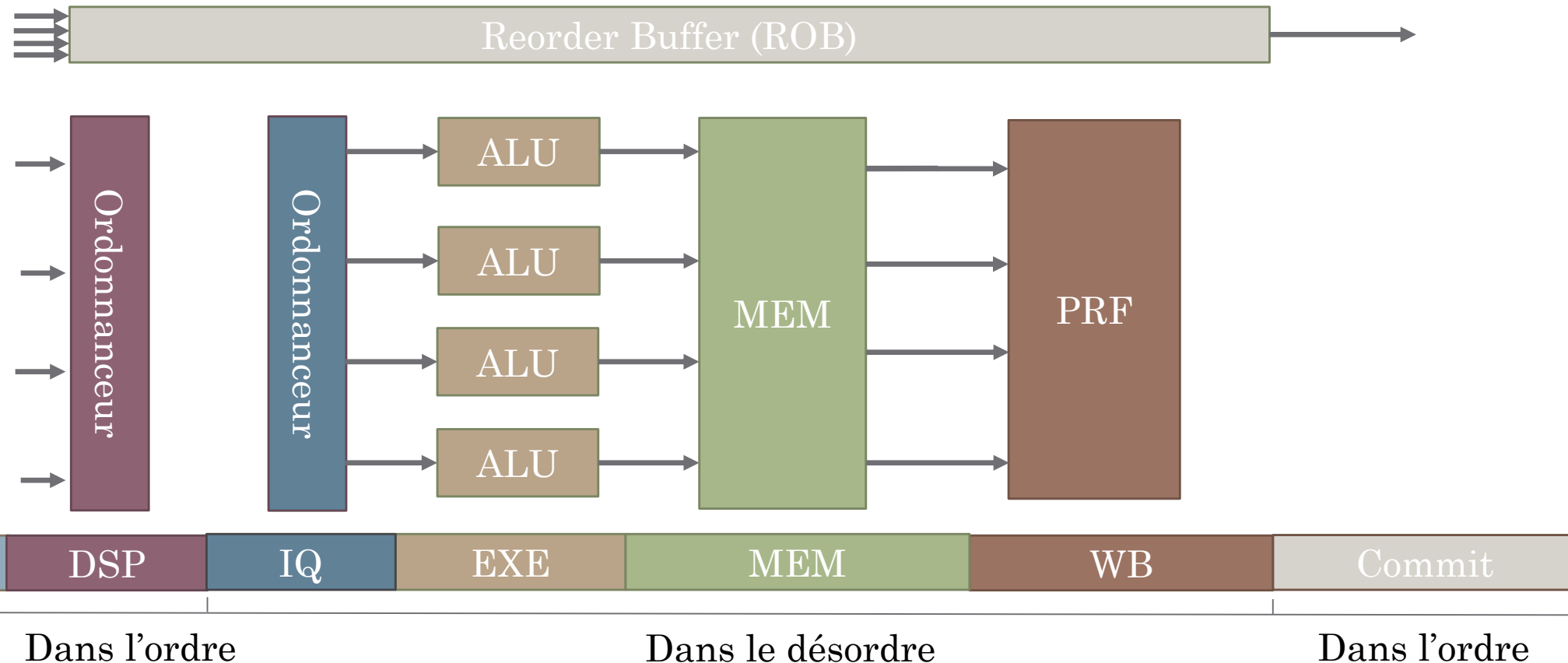
# Exécution dans le désordre – 2<sup>nd</sup>e itération

- Meilleure utilisation des ressources (étage EXE et MEM non bloqués)
- Toujours pas idéal car les instructions plus jeunes sont bloquées si 4 instructions plus vieilles attendent leurs opérandes dans ID



# Exécution dans le désordre – 3<sup>ème</sup> itération

- On veut limiter la dépendance structurelle liée à l'étage ID
  - On introduit une mémoire afin de découpler ID et EXE : **l'ordonnanceur** (« Instruction Queue »)
  - Chaque cycle, Dispatch (DSP) insère 4 instructions, et IQ sélectionne jusqu'à 4 instructions prêtes



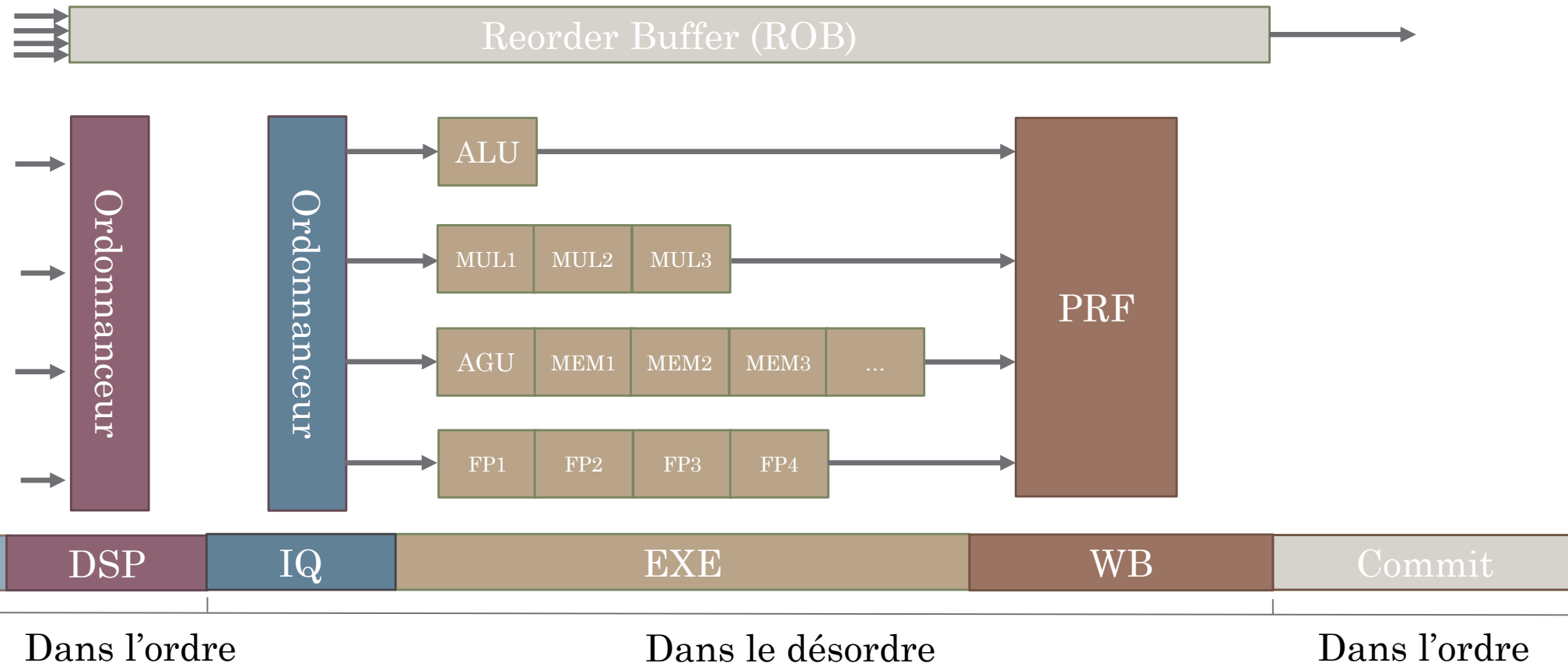
# Exécution dans le désordre – 3ème itération

- Tant que l'ordonnanceur n'est pas plein, on peut continuer à insérer des instructions dans le pipeline
- On note cependant l'élongation du pipeline : latence et pénalité de mauvaise prédiction de branchement plus élevée



# Exécution dans le désordre – Hétérogénéité

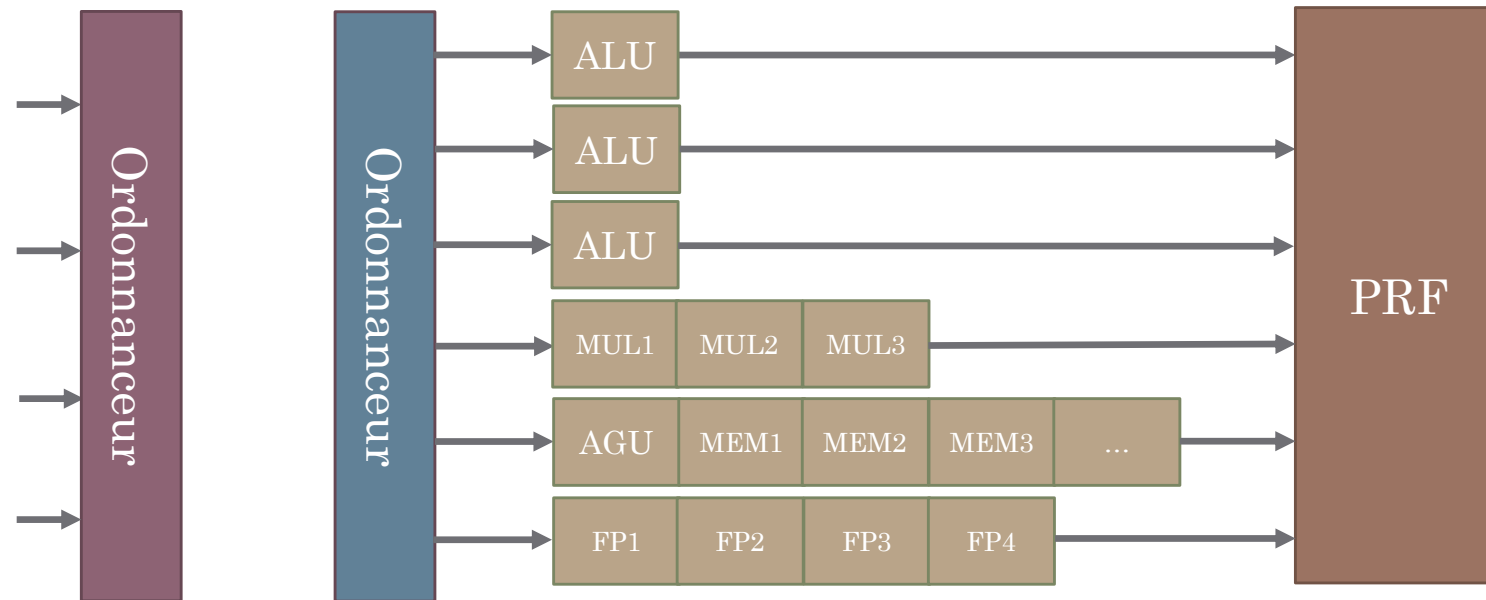
- En général, on implémente des unités fonctionnelles différentes en fonctions des besoins, avec des latences différentes (accès mémoire)
  - MEM se fond dans l'étage d'exécution





# Exécution dans le désordre – Hétérogénéité

- Certains étages plus larges que d'autres : par exemple, ID/DSP 4, IQ/EXE 6
  - Degré superscalaire : étage le moins large, dicte la performance max



Dans l'ordre

Dans le désordre

Dans l'ordre

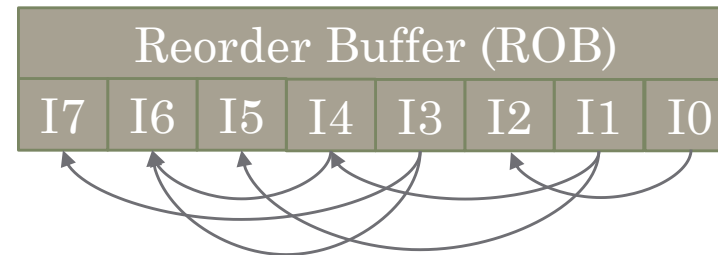
# Exécution dans le désordre - Avantages

- Efficacité du pipeline augmente (moins de bulles)
  - Bien plus performant que l'exécution dans l'ordre
- Même si aucune instruction n'est prête, on peut continuer à récupérer de nouvelles instructions, tant que l'ordonnanceur n'est pas plein
  - Si plein, on bloque les étages précédents en attendant que des places se libèrent

# Exécution dans le désordre - Renommage

- Jusqu'ici, on considère qu'une instruction obtient le « nom » de son opérande source dans ID. Comment ?

I0 ld x1, 0(x2)  
I1 addi x2, x2, 8  
I2 sd x1, 8(x3)  
I3 addi x3, x3, 8  
I4 ld x1, 0(x2)  
I5 addi x2, x2, 8  
I6 sd x1, 8(x3)  
I7 addi x3, x3, 8



Dépendances RaW (prod-cons)

# Renommage de registres

- La microarchitecture implémente  $n$  registres physiques, avec  $n > \#regs\_archs$  (PRF)

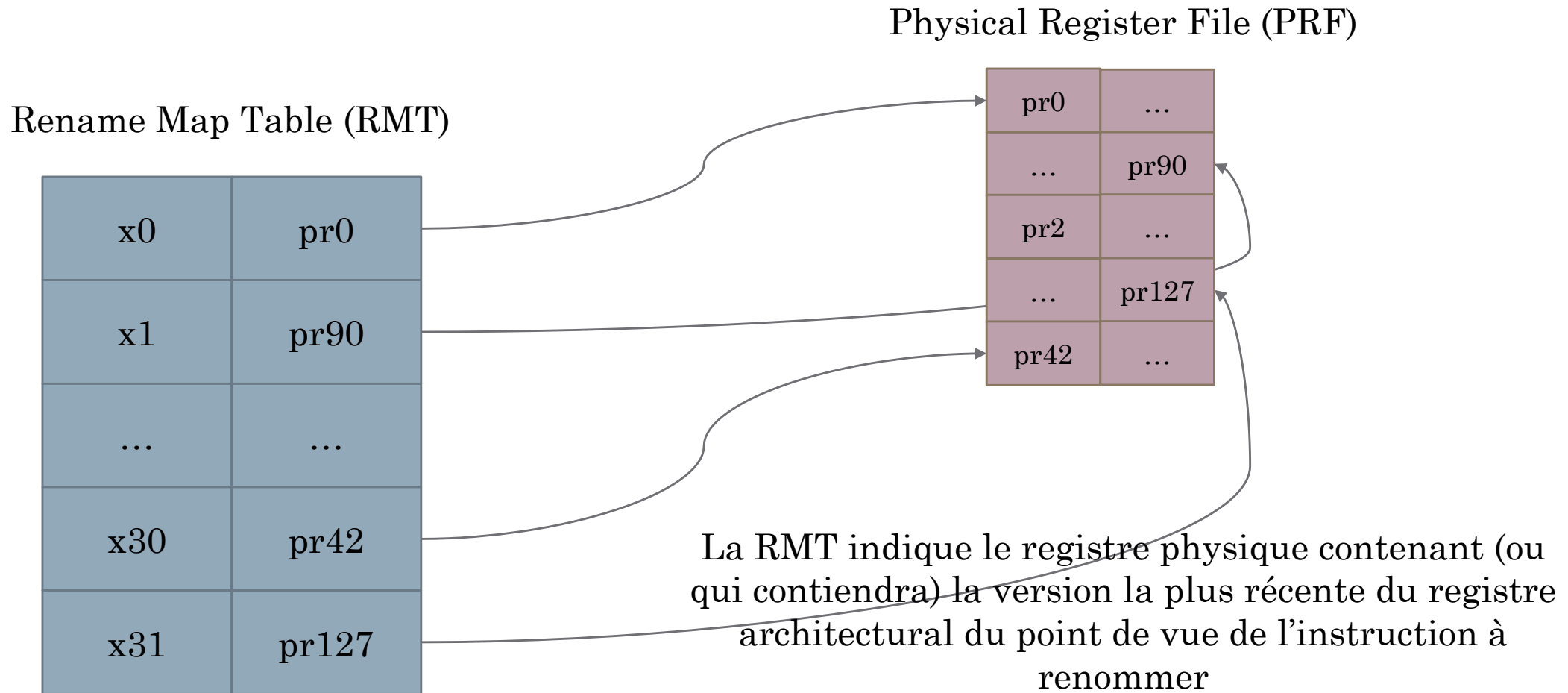
# Renommage de registres

- La microarchitecture implémente  $n$  registres physiques, avec  $n > \#regs\_archs$  (PRF)
- Chaque instruction se voit attribuer un nouveau registre physique pour écrire son résultat
  - On crée une nouvelle « version » du registre architectural
  - Supprime les fausses dépendances (WaR et WaW)

# Renommage de registres

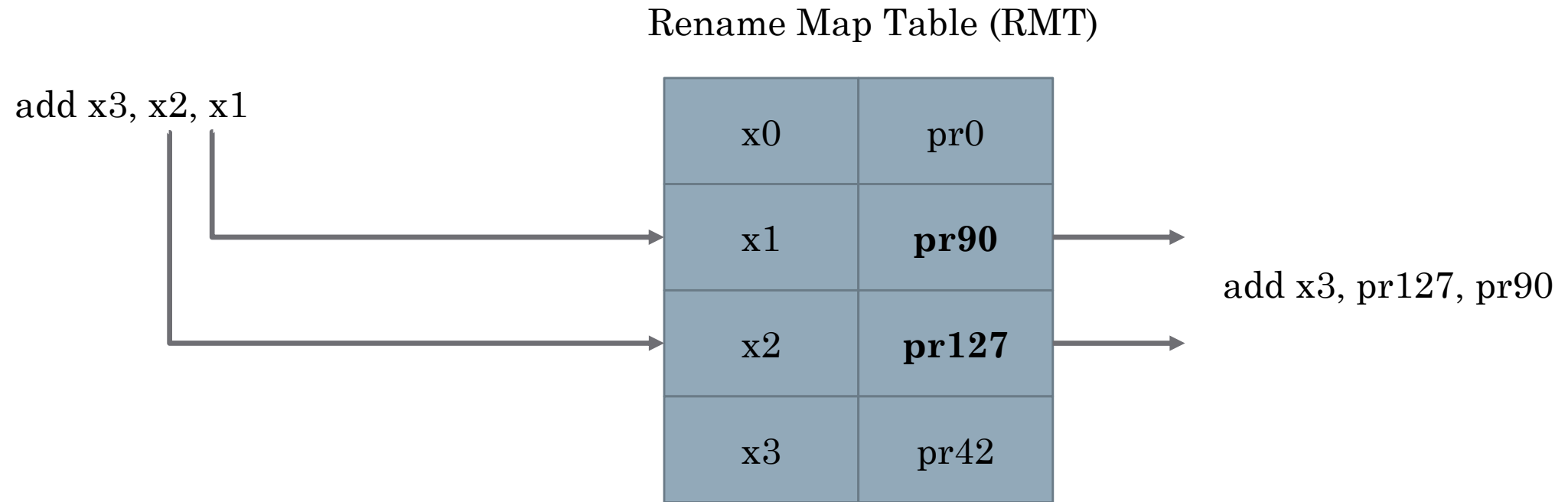
- La microarchitecture implémente  $n$  registres physiques, avec  $n > \text{\#regs\_archs}$  (PRF)
- Chaque instruction se voit attribuer un nouveau registre physique pour écrire son résultat
  - On crée une nouvelle « version » du registre architectural
  - Supprime les fausses dépendances (WaR et WaW)
- Une table associe un registre architectural (logique) à sa version la plus récente (physique)

# Renommage de registres



*Chaque instruction capture ces associations lors du renommage*

# Renommage des registres sources



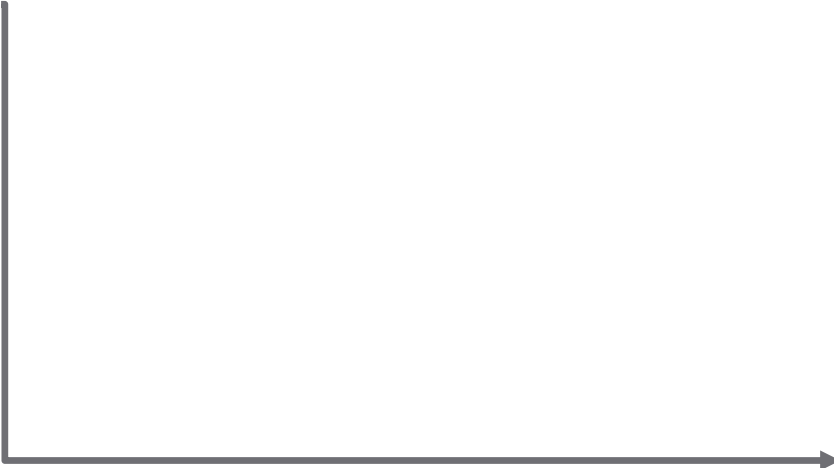


# Renommage du registre de destination

- On crée une nouvelle version de x3, et on invalide l'ancienne dans la RMT

add x3, x2, x1

Rename Map Table (RMT)

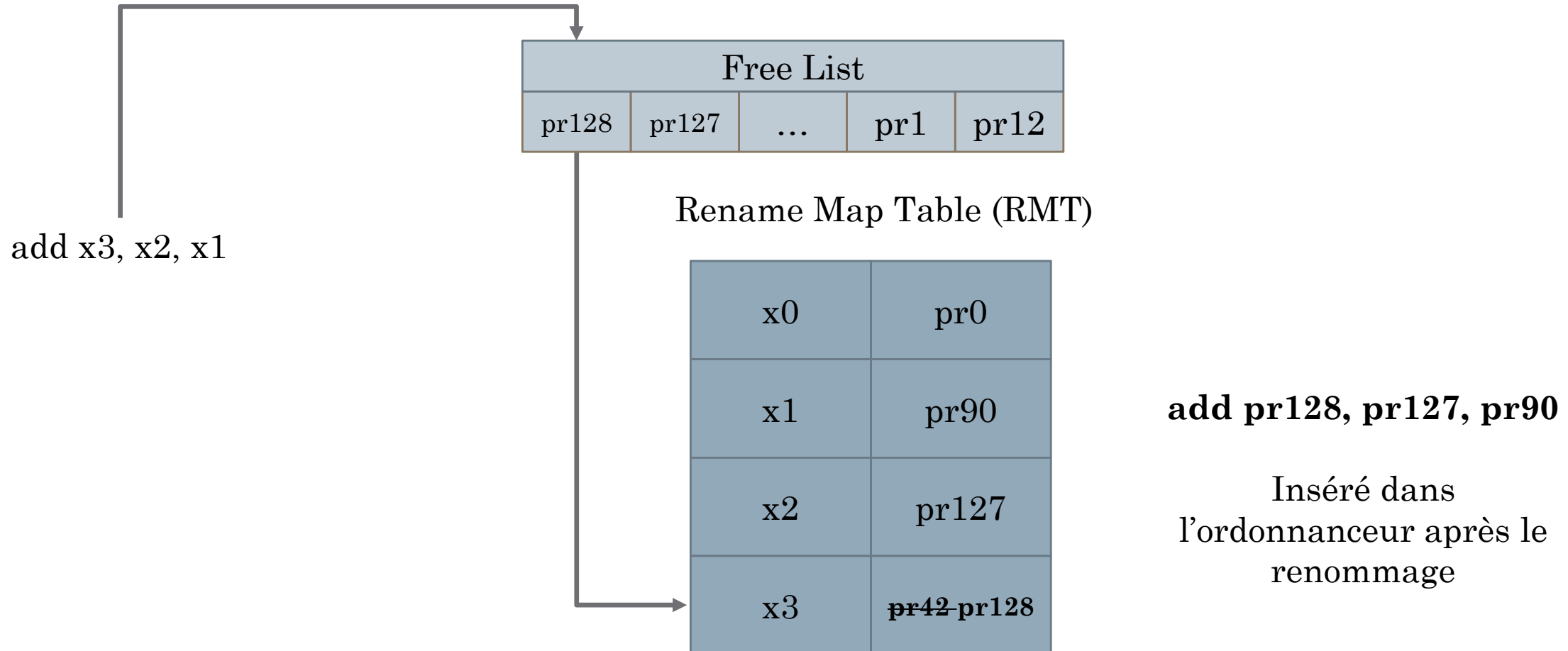


The diagram shows a vertical line from the instruction 'add x3, x2, x1' that turns into a horizontal arrow pointing to the 'x3' entry in the RMT table.

x0	pr0
x1	pr90
x2	pr127
x3	<b>pr42</b>

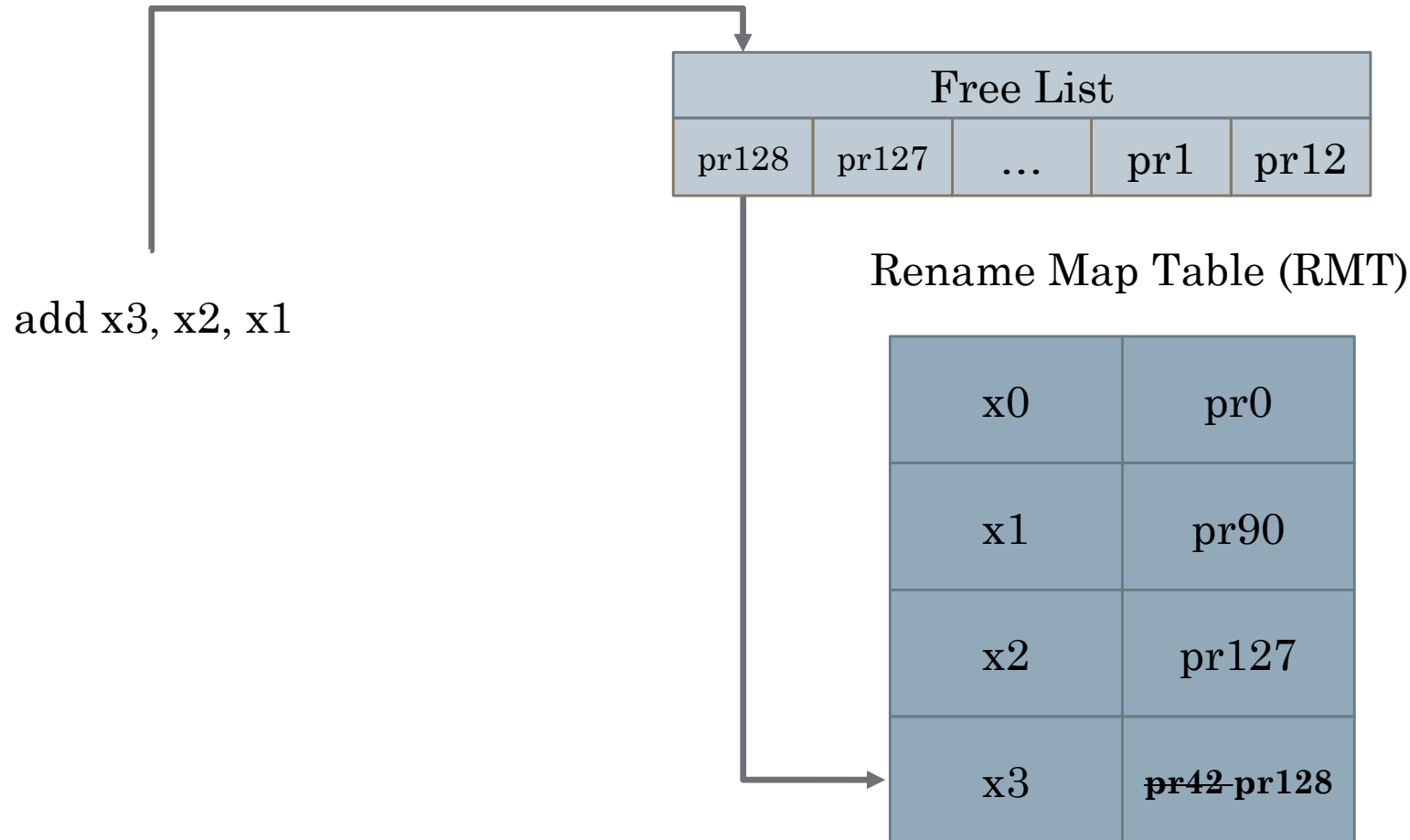
# Renommage du registre de destination

La Free List (FIFO) contient les registres physiques non alloués



# Renommage du registre de destination

La Free List (FIFO) contient les registres physiques non alloués



En substance:

- Toutes les instructions **plus vieilles que add** iront lire la valeur de x3 dans **pr42**
- Toutes les instructions **plus jeunes que add** iront lire la valeur de x3 dans **pr128**

# Renommage de registres : RaW

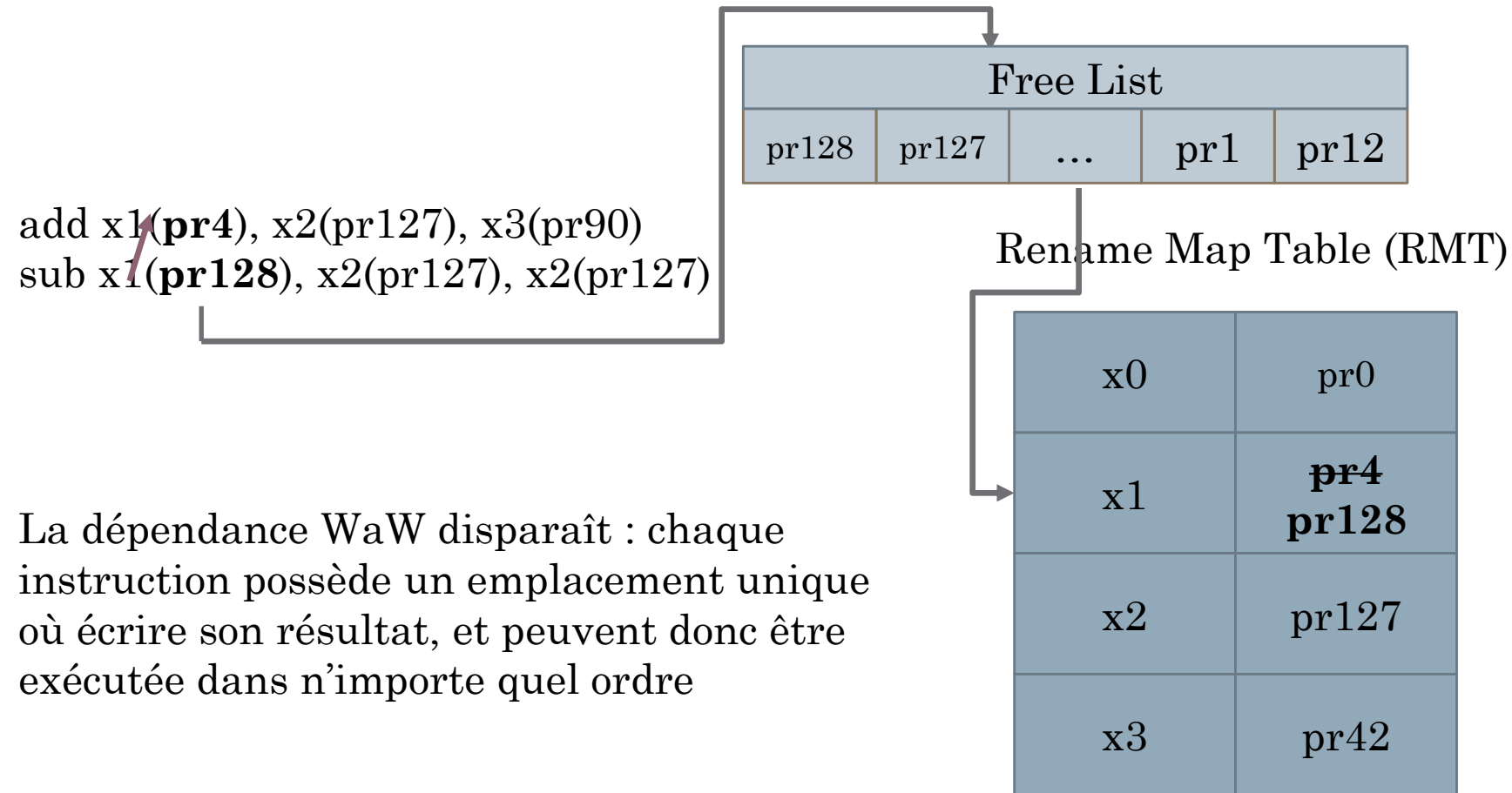
add x1(**pr127**), x2(pr4), x1(pr90)  
sub x3, x1(**pr127**), x2(**pr4**)

Rename Map Table (RMT)

x0	pr0
x1	<b>pr127</b>
x2	<b>pr4</b>
x3	pr42

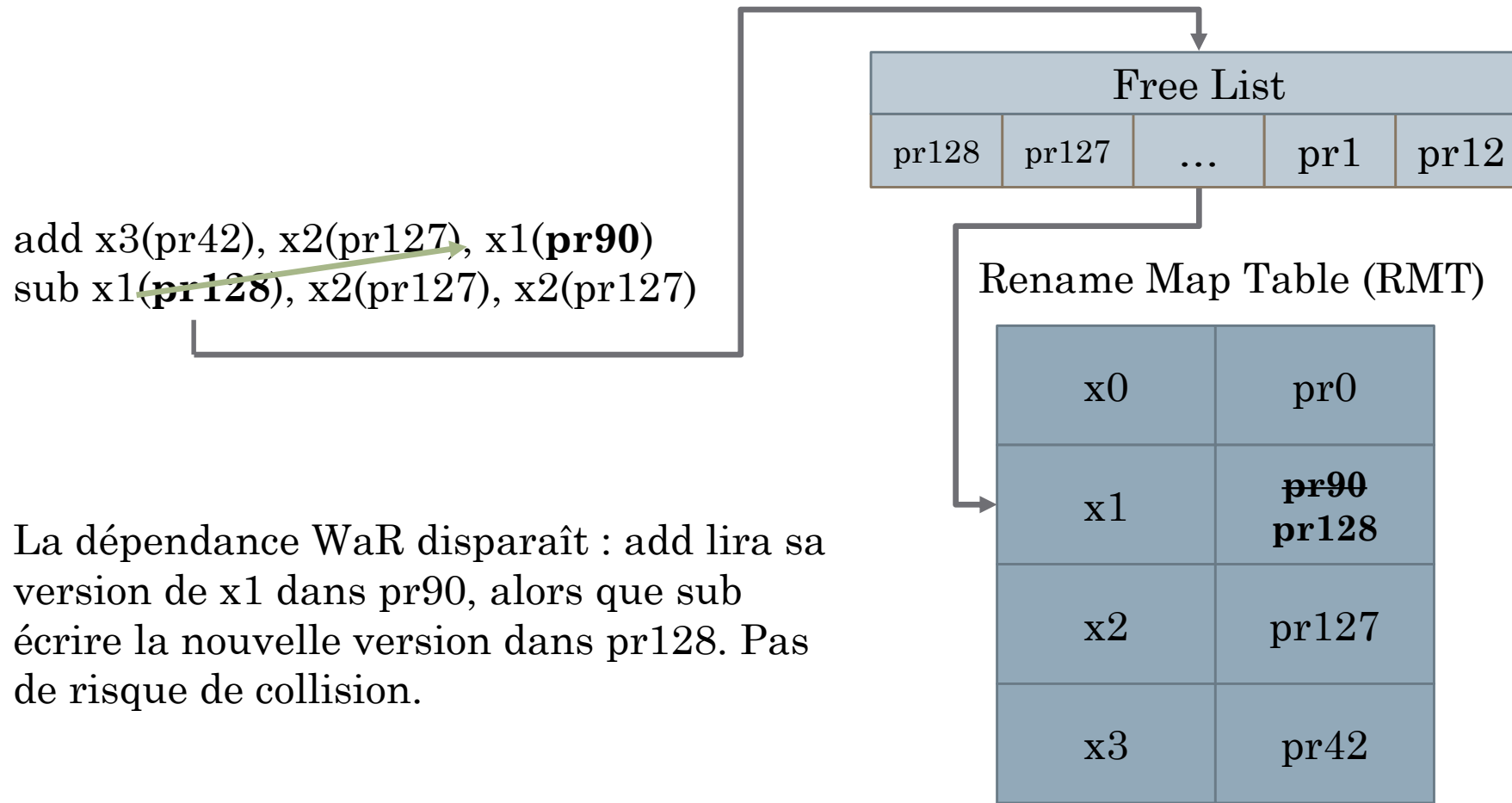
sub x3, **pr127**, **pr4**

# Renommage de registres : WaW



La dépendance WaW disparaît : chaque instruction possède un emplacement unique où écrire son résultat, et peuvent donc être exécutée dans n'importe quel ordre

# Renommage de registres : WaR



La dépendance WaR disparaît : add lira sa version de x1 dans pr90, alors que sub écrire la nouvelle version dans pr128. Pas de risque de collision.

# Exécution dans l'ordre (InO) vs. Désordre (OoO)

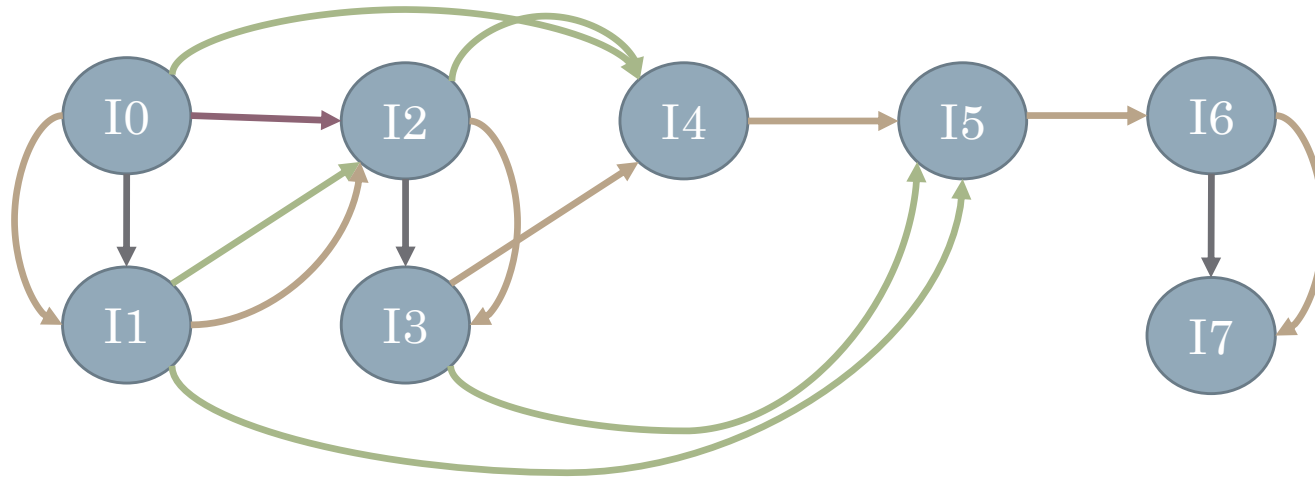
- Question :

```
memcpy_loop:  
I0 ld x1, 0(x2)  
I1 sd x1, 0(x3)  
I2 ld x1, 8(x2)  
I3 sd x1, 8(x3)  
I4 addi, x2, x2, 16  
I5 addi, x3, x3, 16  
I6 subi, x4, x4, 2  
I7 bnez x4, memcpy_loop
```

- Faire apparaître les dépendances arch. (commencer par données)
  - Calculer l'IPC maximum sur une machine InO avec des ressources illimitées et prédiction de branchement parfaite
  - Calculer l'IPC maximum sur une machine OoO avec des ressources illimitées et prédiction de branchement parfaite

# Exécution dans l'ordre (InO) vs. Désordre (OoO)

- Dépendances arch.



Données

→ RaW

→ WaR

→ WaW

Contrôle

→ Ordre

memcpy\_loop:

I0 ld x1, 0(x2)

I1 sd x1, 0(x3)

I2 ld x1, 8(x2)

I3 sd x1, 8(x3)

I4 addi, x2, x2, 16

I5 addi, x3, x3, 16

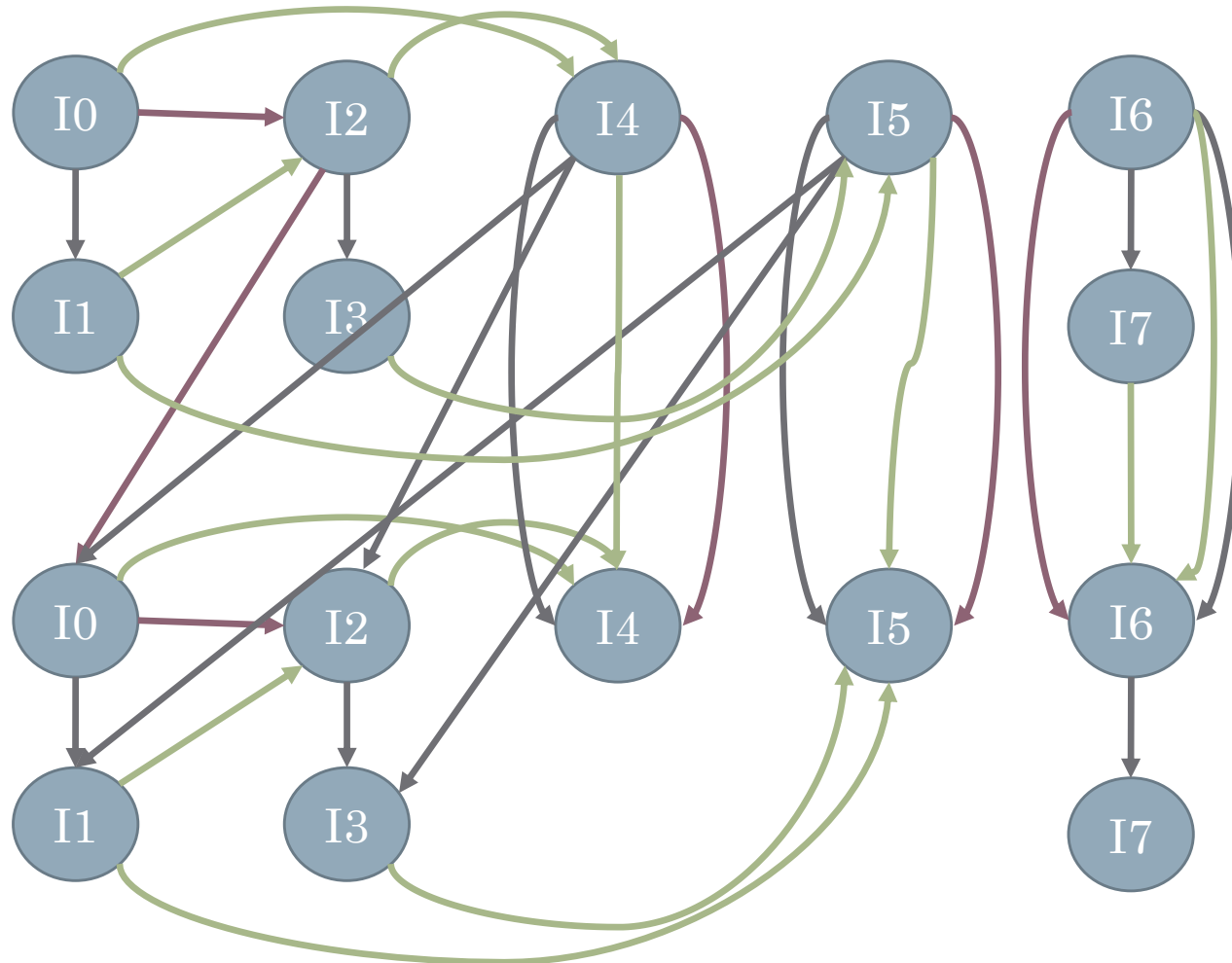
I6 subi, x4, x4, 2

I7 bnez x4, memcpy\_loop



# Exécution dans l'ordre (InO) vs. Désordre (OoO)

- Dépendances arch.



Données

→ RaW

→ WaR

→ WaW

Contrôle

→ Ordre (omis)

memcpy\_loop:

I0 ld x1, 0(x2)

I1 sd x1, 0(x3)

I2 ld x1, 8(x2)

I3 sd x1, 8(x3)

I4 addi, x2, x2, 16

I5 addi, x3, x3, 16

I6 subi, x4, x4, 2

I7 bnez x4, memcpy\_loop

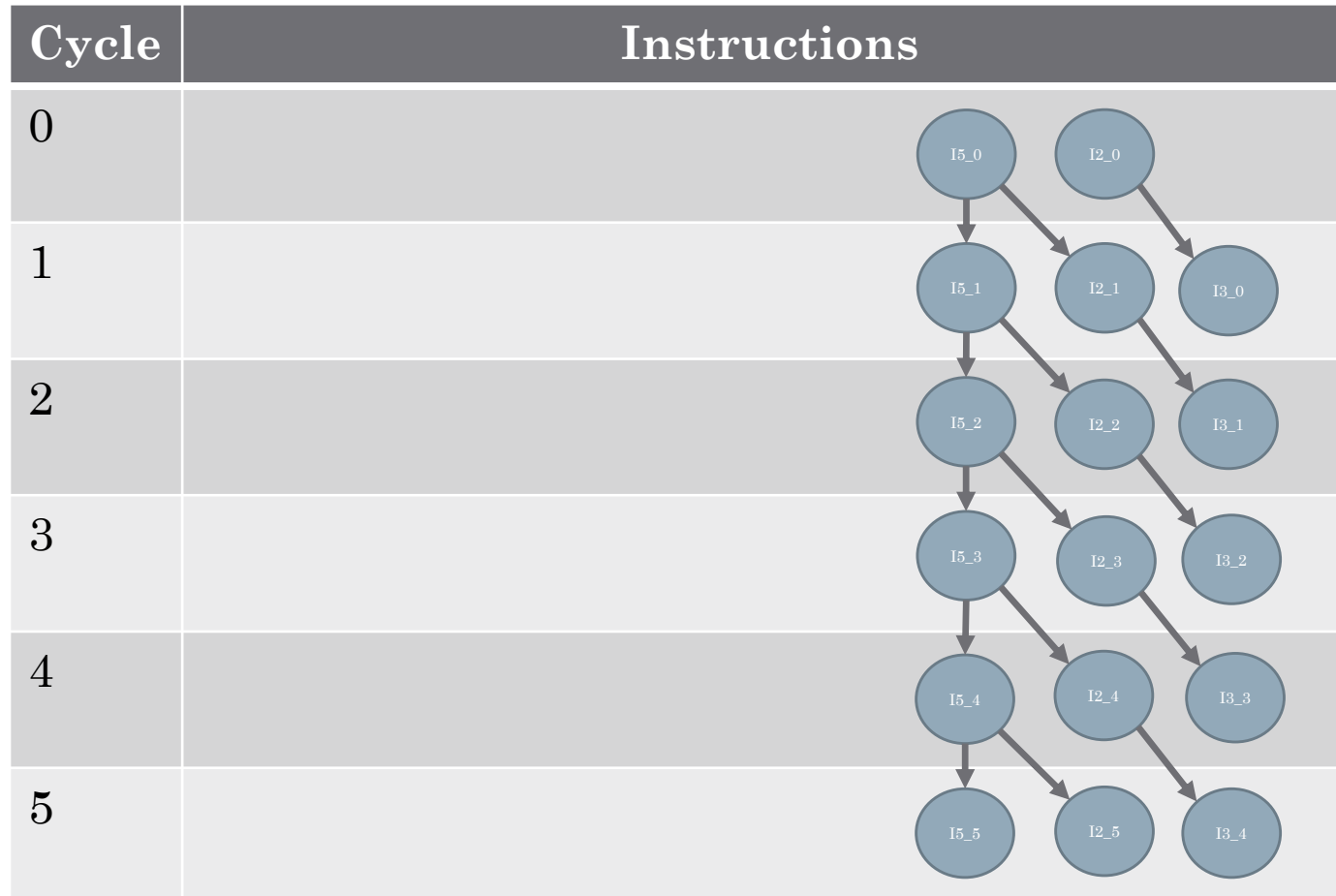
# Exécution dans l'ordre (InO) vs. Désordre (OoO)

- IPC maximum sur machine OoO illimitée (= uniquement RaW) : on ordonnance

Cycle	Instructions
0	
1	
2	
3	
4	
5	

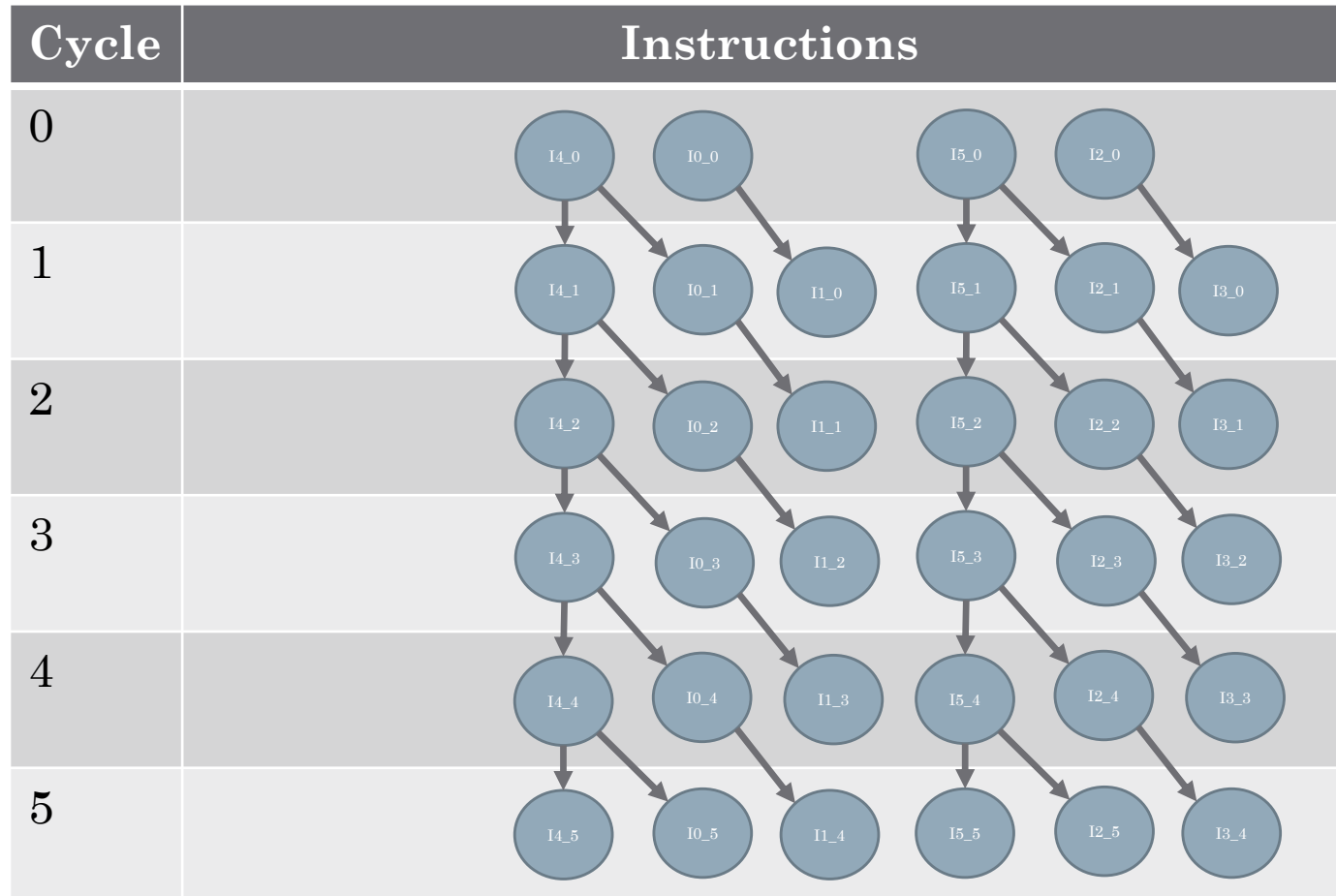
# Exécution dans l'ordre (InO) vs. Désordre (OoO)

- IPC maximum sur machine OoO illimitée (= uniquement RaW) : on ordonnance



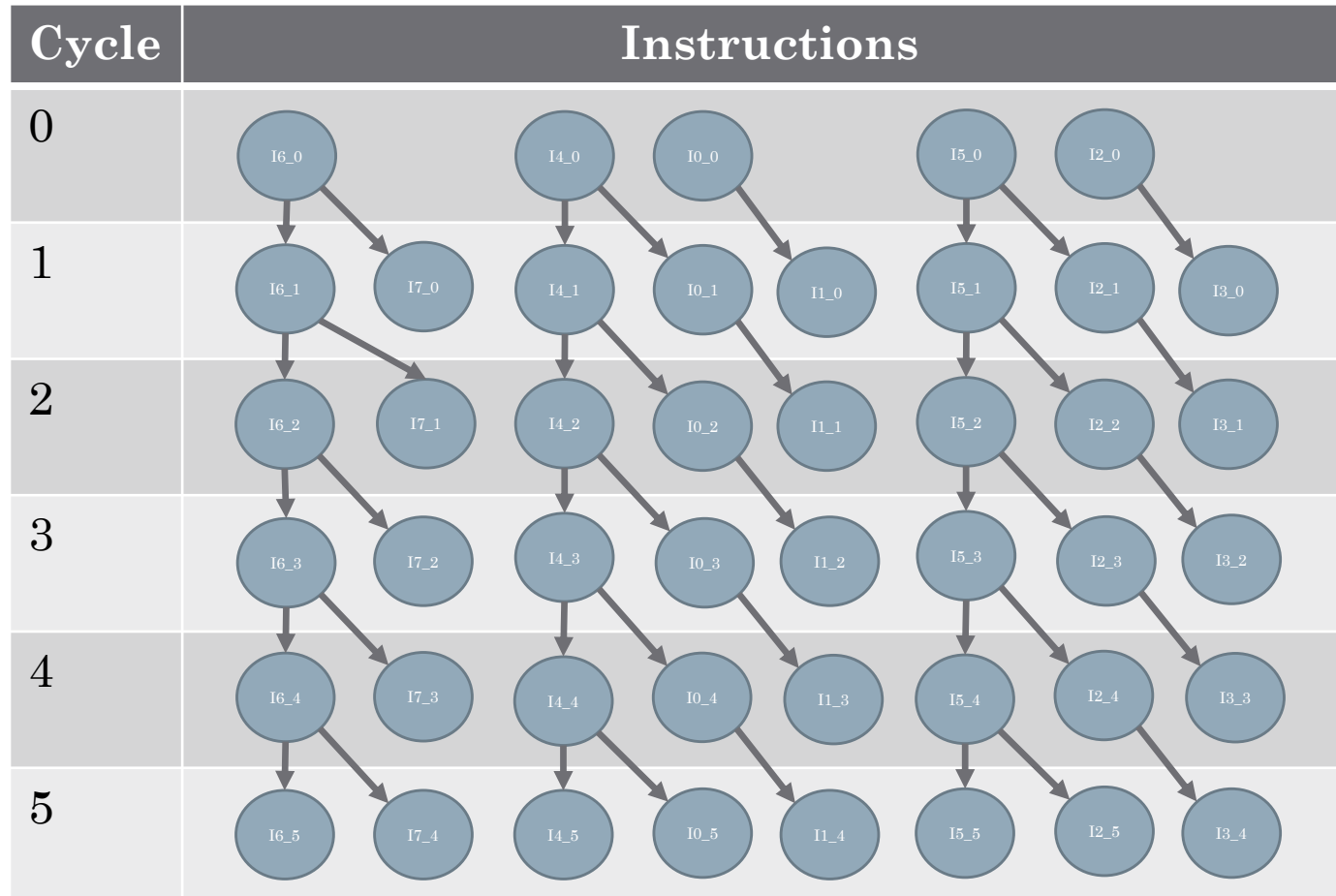
# Exécution dans l'ordre (InO) vs. Désordre (OoO)

- IPC maximum sur machine OoO illimitée (= uniquement RaW) : on ordonnance



# Exécution dans l'ordre (InO) vs. Désordre (OoO)

- IPC maximum sur machine OoO illimitée (= uniquement RaW) : on ordonnance



IPC max  
(intrinsèque à  
l'algorithme) = 8

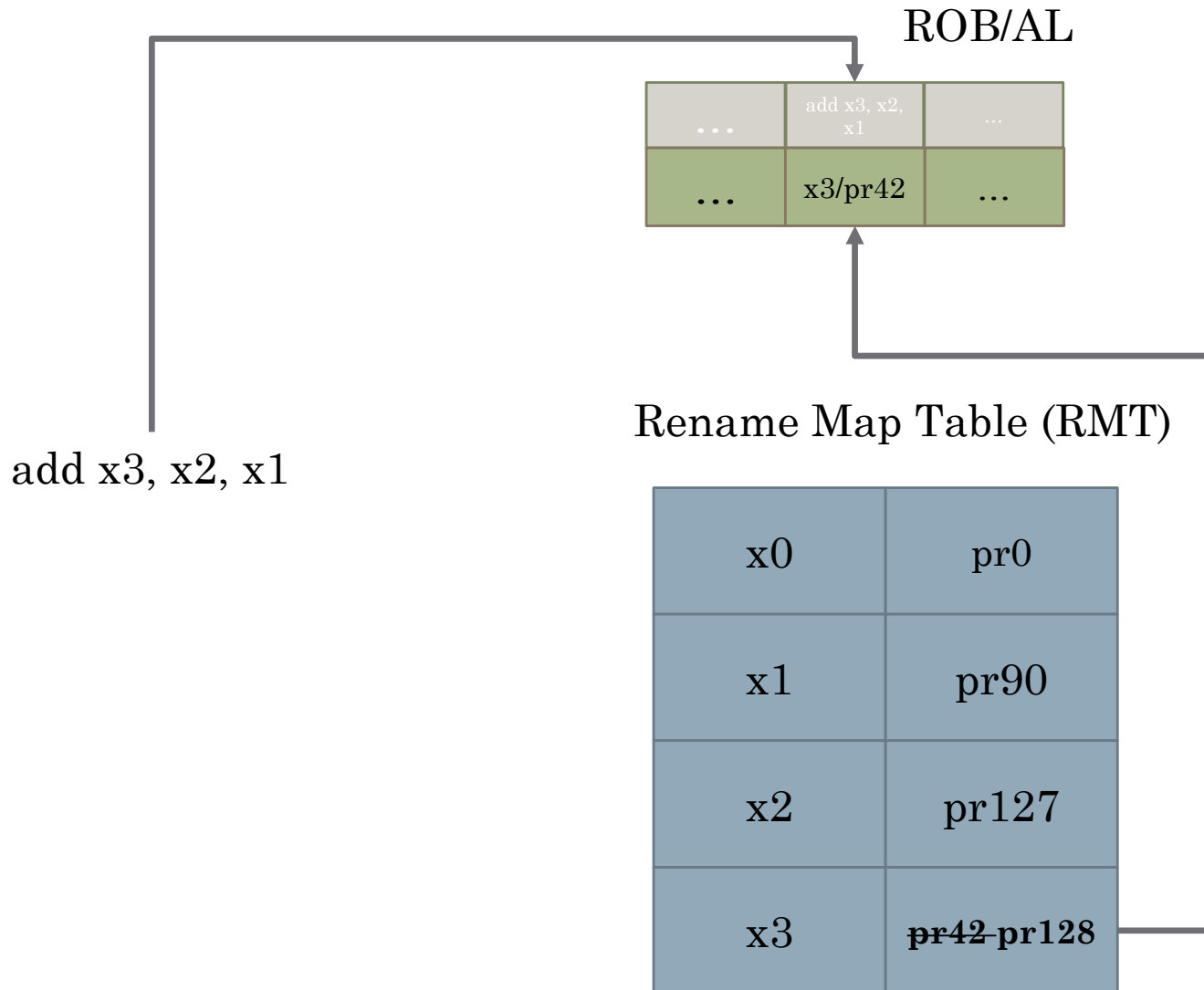
# Renommage de registres

- Jusqu'ici, on a considéré une exécution qui se passe correctement
  - Pas de mauvaise prédiction de branchement
- Exécution dans l'ordre et mauvaise prédiction :
  - On vide les étages plus jeunes (= à gauche) pour enlever les instructions sur le mauvais chemin
- Exécution dans le désordre :
  - On vide les étages plus jeunes (qui sont dans l'ordre, IF, ID, DSP)
  - Et ensuite ?

# Renommage de registres

- Jusqu'ici, on a considéré une exécution qui se passe correctement
  - Pas de mauvaise prédiction de branchement
- Exécution dans l'ordre et mauvaise prédiction :
  - On vide les étages plus jeunes (= à gauche) pour enlever les instructions sur le mauvais chemin
- Exécution dans le désordre :
  - On vide les étages plus jeunes (qui sont dans l'ordre, IF, ID, DSP)
  - Et ensuite ?
    - Réparer la RMT
    - Enlever les instructions plus jeunes dans l'Ordonnanceur, le ROB et les registres de pipeline IQ/EXE/MEM

# Renommage de registres : Active List (AL)

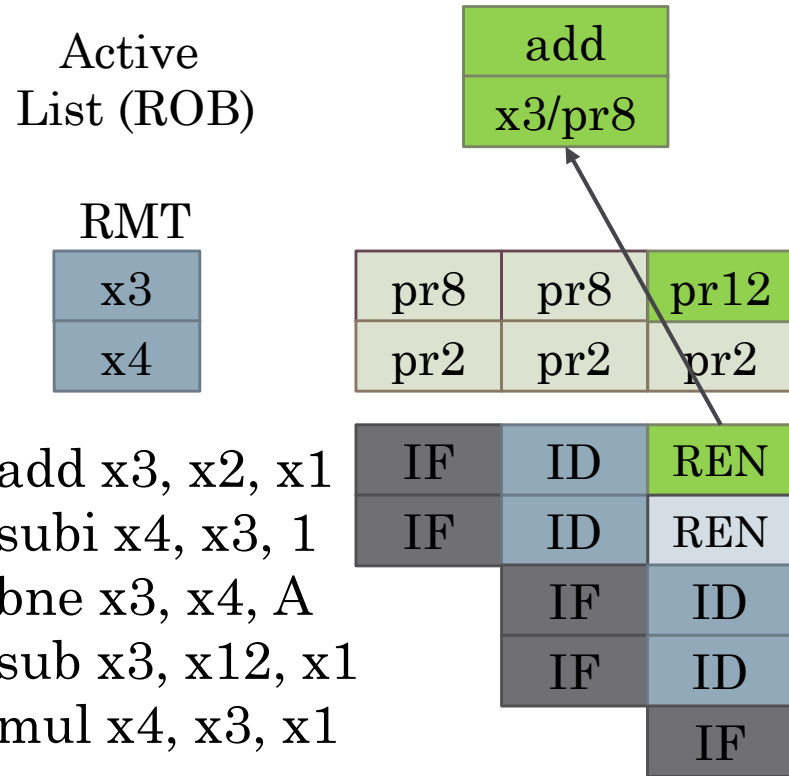


Lors du renommage de la destination, une entrée est allouée dans la Active List (ici incluse dans le ROB, en vert)

L'AL contient l'association arch/phy précédente



# Renommage de registres : Active List (AL)



A:  
add x4, x3, x1

# Renommage de registres : Active List (AL)

Active  
List (ROB)

add	subi
x3/pr8	x4/pr2

RMT

x3
x4

add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

pr8	pr8	pr12
pr2	pr2	pr14
IF	ID	REN
IF	ID	REN
	IF	ID
	IF	ID
		IF

A:  
add x4, x3, x1

# Renommage de registres : Active List (AL)

Active  
List (ROB)

add	subi	sub
x3/pr8	x4/pr2	x3/pr12

RMT

x3
x4

add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

pr8	pr8	pr12	pr19
pr2	pr2	pr14	pr14
IF	ID	REN	IQ
IF	ID	REN	IQ
	IF	ID	REN
	IF	ID	REN
		IF	ID

A:  
add x4, x3, x1

# Renommage de registres : Active List (AL)

Active  
List (ROB)

add	subi	sub	mul
x3/pr8	x4/pr2	x3/pr12	x4/pr14

RMT

x3
x4

add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

pr8	pr8	pr12	pr19	pr19
pr2	pr2	pr14	pr14	pr13
IF	ID	REN	IQ	EXE
IF	ID	REN	IQ	IQ
	IF	ID	REN	IQ
	IF	ID	REN	IQ
		IF	ID	REN

A:  
add x4, x3, x1

# Renommage de registres : Active List (AL)

Active  
List (ROB)

add	subi	sub	mul
x3/pr8	x4/pr2	x3/pr12	x4/pr14

RMT

x3
x4

pr8	pr8	pr12	pr19	pr19		pr19
pr2	pr2	pr14	pr14	pr13		pr13

add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

IF	ID	REN	IQ	EXE	WB	CO
IF	ID	REN	IQ	IQ	EXE	WB
	IF	ID	REN	IQ	IQ	EXE
	IF	ID	REN	IQ	EXE	WB
		IF	ID	REN	IQ	EXE

add passe Commit,  
devient visible de  
l'état architectural

=> Plus besoin de  
garder l'ancienne  
association x3 <> pr8

A:  
add x4, x3, x1

# Renommage de registres : Active List (AL)

Active  
List (ROB)

subi	sub	mul
x4/pr2	x3/pr12	x4/pr14

RMT

x3
x4

pr8	pr8	pr12	pr19	pr19		pr19
pr2	pr2	pr14	pr14	pr13		pr13

add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

IF	ID	REN	IQ	EXE	WB	CO
IF	ID	REN	IQ	IQ	EXE	WB
	IF	ID	REN	IQ	IQ	EXE
	IF	ID	REN	IQ	EXE	WB
		IF	ID	REN	IQ	EXE

add passe Commit,  
devient visible de  
l'état architectural

=> Plus besoin de  
garder l'ancienne  
association x3 <> pr8

A:  
add x4, x3, x1

# Renommage de registres : Active List (AL)

Active  
List (ROB)

subi	sub	mul
x4/pr2	x3/pr12	x4/pr14

Br mispred (bne x3, x4, A)

RMT

x3
x4

pr8	pr8	pr12	pr19	pr19		pr19
pr2	pr2	pr14	pr14	pr13		pr13
IF	ID	REN	IQ	EXE	WB	CO
IF	ID	REN	IQ	IQ	EXE	WB
	IF	ID	REN	IQ	IQ	EXE
	IF	ID	REN	IQ	EXE	WB
		IF	ID	REN	IQ	EXE

add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

A:  
add x4, x3, x1

sub et mul renommée  
sur le mauvais chemin

Après le branchement,  
on devrait avoir :  
x3 = pr12  
x4 = pr14

On a :  
x3 = pr19  
x4 = pr13

# Renommage de registres : Active List (AL)

Active  
List (ROB)

subi	sub	mul
x4/pr2	x3/pr12	x4/pr14

Br mispred (bne x3, x4, A)

RMT

x3
x4

pr8	pr8	pr12	pr19	pr19		pr19
pr2	pr2	pr14	pr14	pr13		pr13
IF	ID	REN	IQ	EXE	WB	CO
IF	ID	REN	IQ	IQ	EXE	WB
	IF	ID	REN	IQ	IQ	EXE
	IF	ID	REN	IQ	EXE	WB
		IF	ID	REN	IQ	EXE

add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

A:  
add x4, x3, x1

Idée : Parcourir la Active List pour réparer la RMT, en partant de l'entrée la plus jeune vers la plus vieille



# Renommage de registres : Active List (AL)

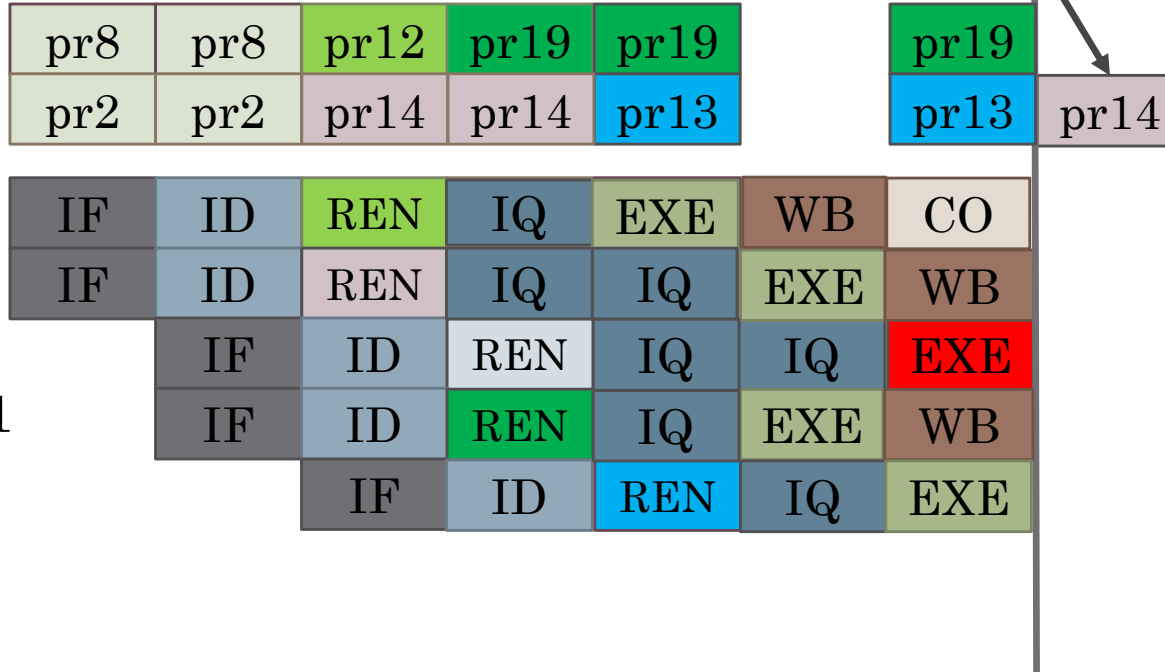
Active  
List (ROB)

subi	sub	mul
x4/pr2	x3/pr12	x4/pr14

Br mispred (bne x3, x4, A)

RMT

x3
x4



add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

A:

add x4, x3, x1

Les registres physiques sont  
« désalloués » car l'entrée du ROB est  
désallouée

# Renommage de registres : Active List (AL)

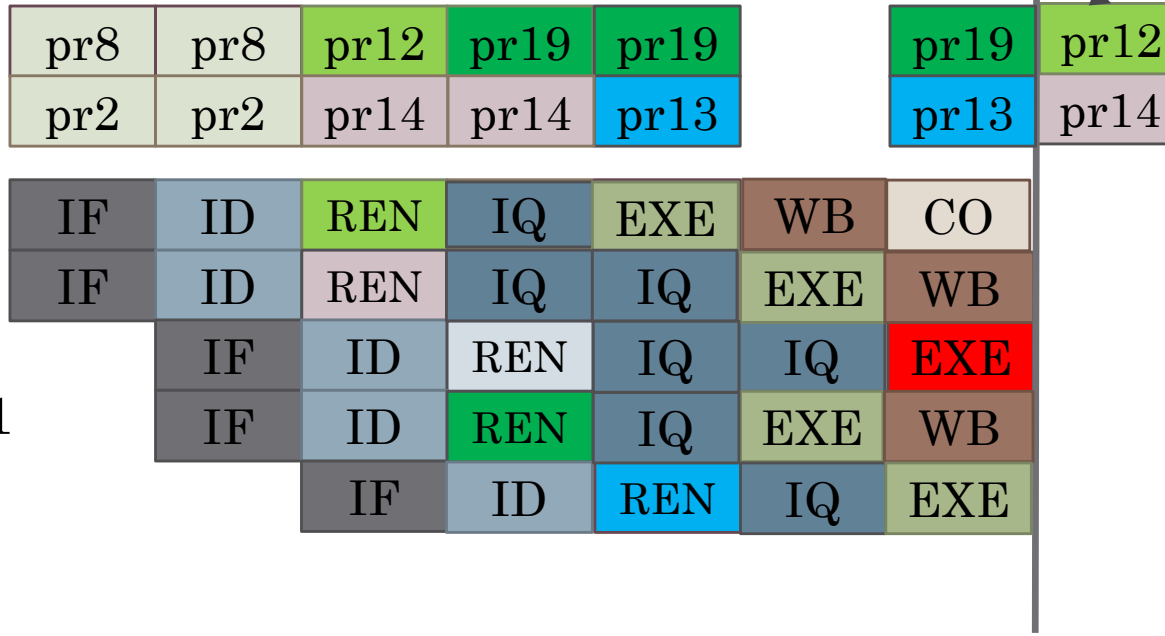
Active  
List (ROB)

subi	sub
x4/pr2	x3/pr12

Br mispred (bne x3, x4, A)

RMT

x3
x4



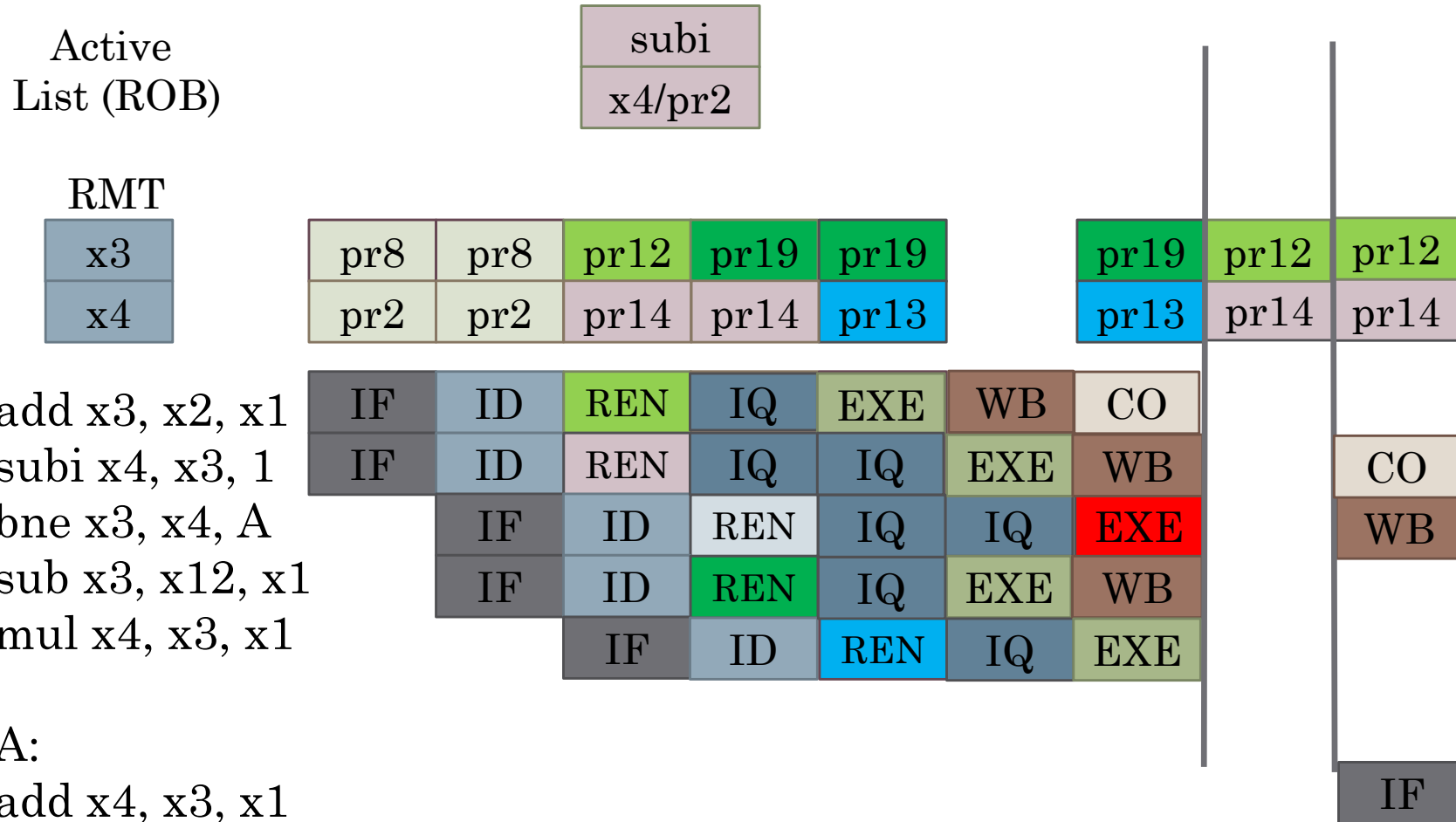
add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

A:

add x4, x3, x1

On s'arrête lorsque l'entrée de l'AL (ROB) la plus jeune est plus vieille que le branchement (ici subi est plus vieille que bne)

# Renommage de registres : Active List (AL)



On reprend ensuite l'exécution

# Renommage de registres : Active List (AL)

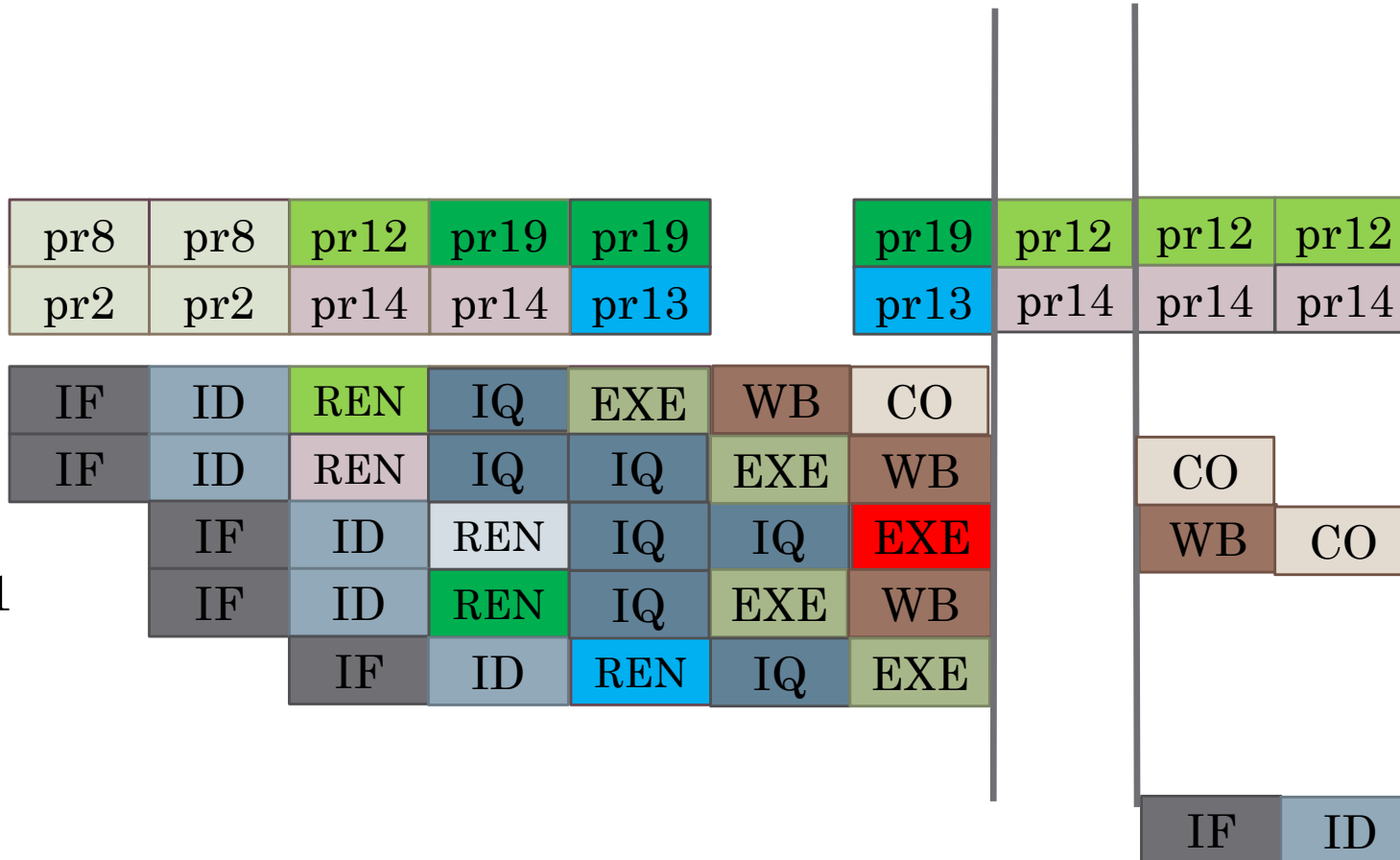
Active  
List (ROB)

RMT

x3
x4

add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

A:  
add x4, x3, x1



On reprend ensuite l'exécution

# Renommage de registres : Active List (AL)

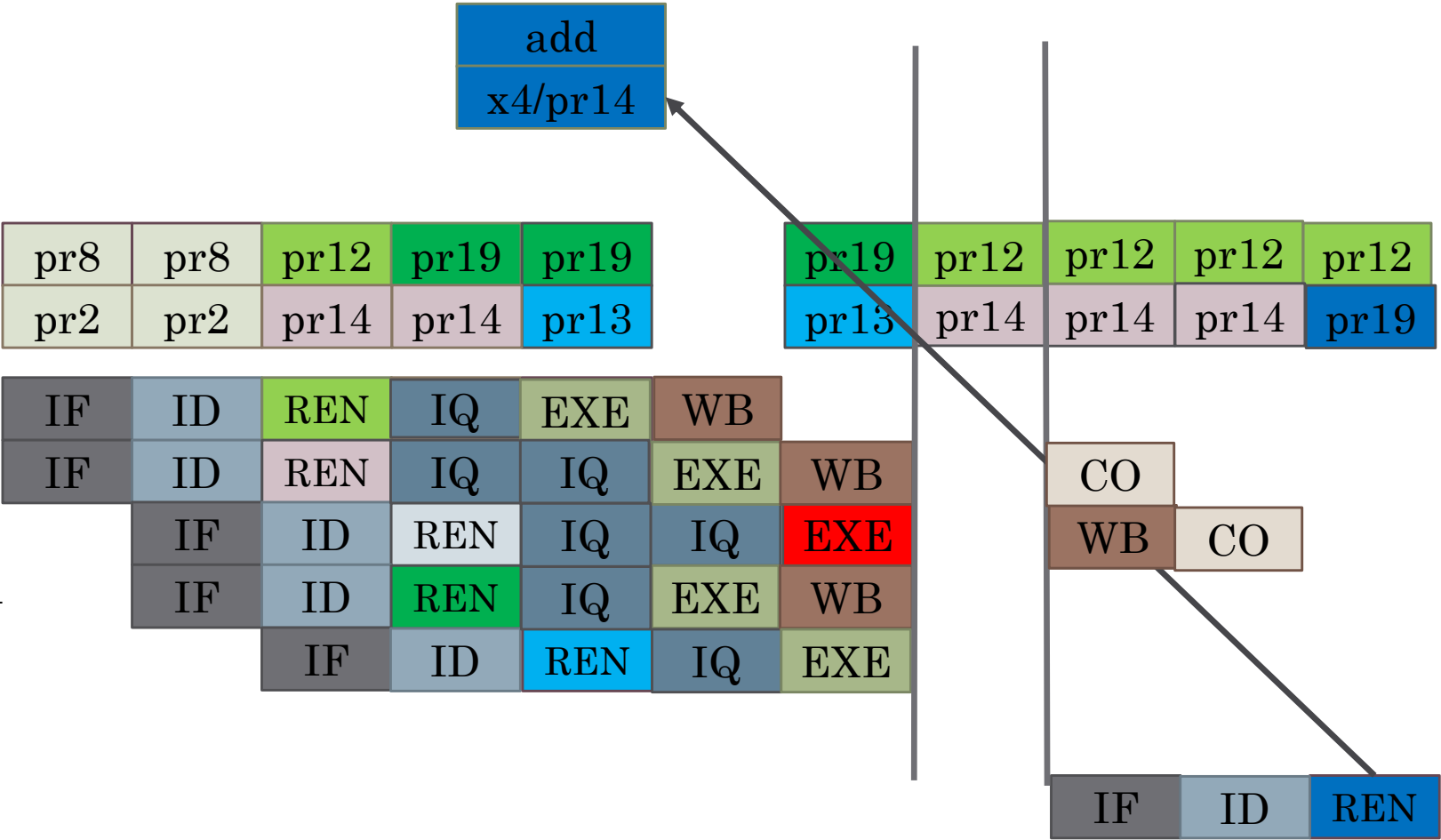
Active  
List (ROB)

RMT

x3
x4

add x3, x2, x1  
subi x4, x3, 1  
bne x3, x4, A  
sub x3, x12, x1  
mul x4, x3, x1

A:  
add x4, x3, x1



# Renommage de registres : Active List (AL)

- En parcourant le ROB, qui contient ici la Active List, on répare la RMT de manière active
  - On invalide l'entrée du ROB ( $\Rightarrow$  enlever l'instruction du pipeline) après avoir réparé la RMT
  - On cherche aussi l'instruction dans l'ordonnanceur et les registres de pipeline IQ/EXE/WB pour l'enlever
- Processus itératif : peut prendre plusieurs cycles
- Plusieurs algorithmes possibles selon les variations microarchitecturales

# Recyclage de registres physiques

- Jusqu'ici, on a parlé de l'attribution des registres physiques
  - Au renommage
- Libération du registre (retour dans la Free List)?
  - Quand la valeur n'est plus nécessaire
    - Valeur écrite dans le registre (produite)
    - Valeur lue par tous les consommateurs (consommée)
    - Nouvelle version du registre fait partie de l'état du programme

# Recyclage de registres physiques

- Jusqu'ici, on a parlé de l'attribution des registres physiques
  - Au renommage
- Libération du registre (retour dans la Free List)?
  - Quand la valeur n'est plus nécessaire
    - Valeur écrite dans le registre (produite)
    - Valeur lue par tous les consommateurs (consommée)
    - Nouvelle version du registre fait partie de l'état du programme
- Concrètement: On utilise le Commit
  - Registre **PhyY** associé au registre **ArchX** libéré quand une instruction plus jeune écrivant **PhyZ** associé à **ArchX** est Commit



# Recyclage de registres physiques

Initialement :

- lsl redéfinit x8

```
add x8(pr3), x2, x1
sub x3, x1, x8(pr3)
mul x2, x1, x8(pr3)
lsl x8(pr12), x2, x1
div x3, x8(pr12), x8(pr12)
```

# Recyclage de registres physiques

**Initialement :**

```
add x8(pr3), x2, x1
sub x3, x1, x8(pr3)
mul x2, x1, x8(pr3)
lsl x8(pr12), x2, x1
div x3, x8(pr12), x8(pr12)
```

- lsl redéfinit x8
- Si lsl passe le Commit, alors add, sub, mul ont passé le Commit

# Recyclage de registres physiques

**Initialement :**

```
add x8(pr3), x2, x1
sub x3, x1, x8(pr3)
mul x2, x1, x8(pr3)
lsl x8(pr12), x2, x1
div x3, x8(pr12), x8(pr12)
```

- lsl redéfinit x8
- Si lsl passe le Commit, alors add, sub, mul ont passé le Commit
  - Implique que add a écrit son résultat dans pr3 car on ne peut pas passer le Commit sans être exécutée

# Recyclage de registres physiques

**Initialement :**

```
add x8(pr3), x2, x1
sub x3, x1, x8(pr3)
mul x2, x1, x8(pr3)
lsl x8(pr12), x2, x1
div x3, x8(pr12), x8(pr12)
```

- lsl redéfinit x8
- Si lsl passe le Commit, alors add, sub, mul ont passé le Commit
  - Implique que add a écrit son résultat dans pr3 car on ne peut pas passer le Commit sans être exécutée
  - Implique que sub et mul ont lu pr3, pour la même raison

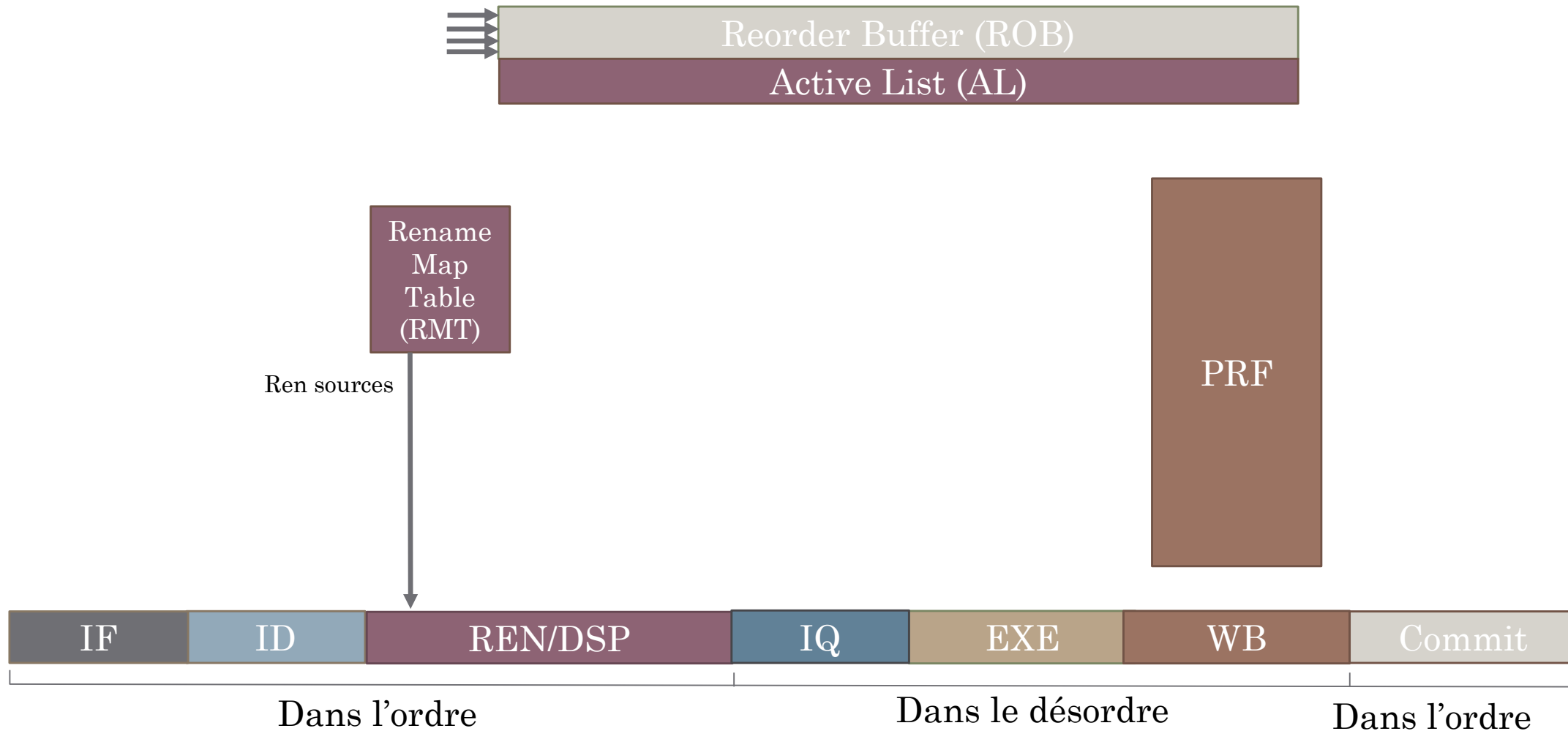
# Recyclage de registres physiques

Initialement :

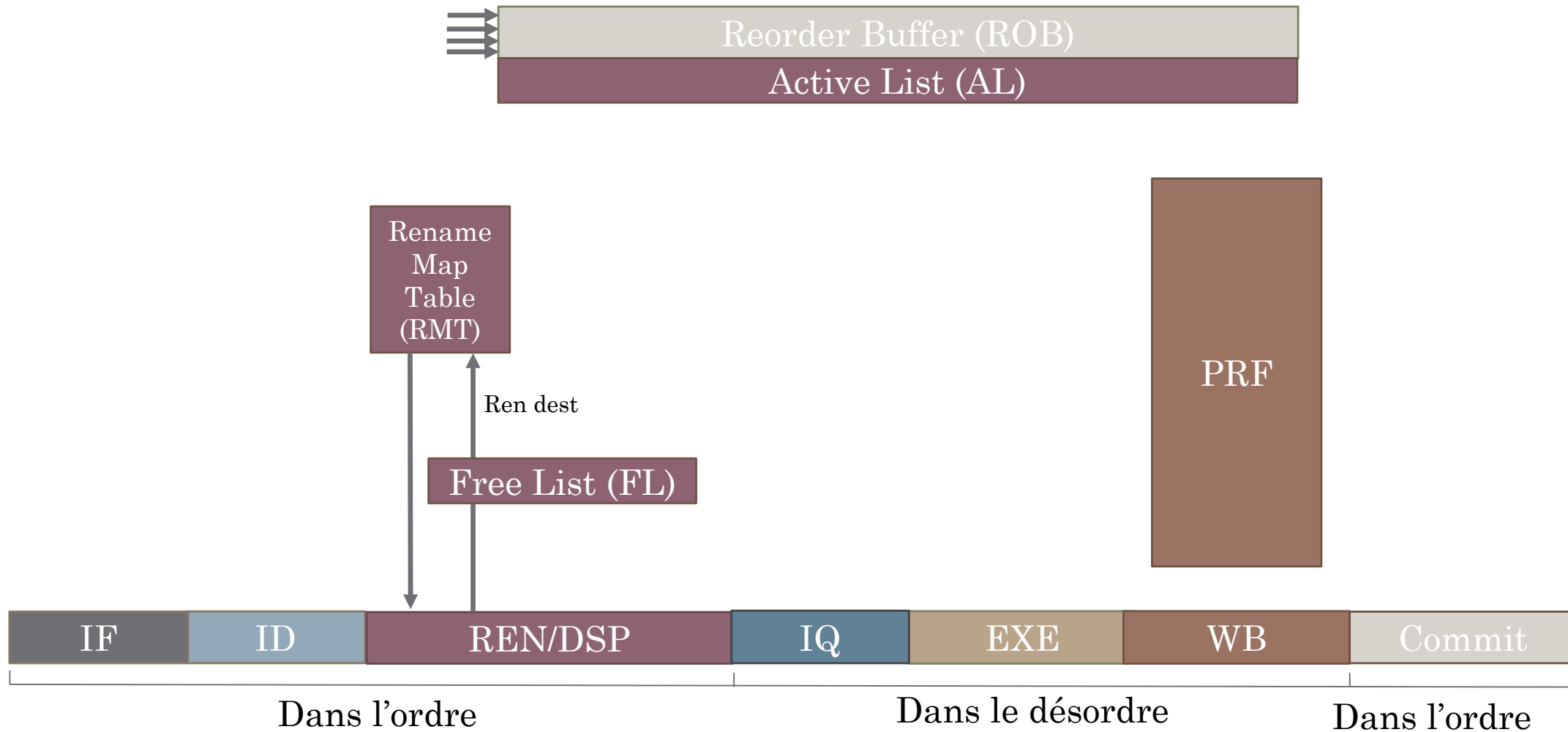
```
add x8(pr3), x2, x1
sub x3, x1, x8(pr3)
mul x2, x1, x8(pr3)
lsl x8(pr12), x2, x1
div x3, x8(pr12), x8(pr12)
```

- lsl redéfinit x8
- Si lsl passe le Commit, alors add, sub, mul ont passé le Commit
  - Implique que add a écrit son résultat dans pr3 car on ne peut pas passer le Commit sans être exécutée
  - Implique que sub et mul ont lu pr3, pour la même raison
- Toutes les instructions plus jeunes que lsl consommeront pr12 et non pr3
- lsl passe le Commit = on peut remettre pr3 sur la Free List en toute sécurité

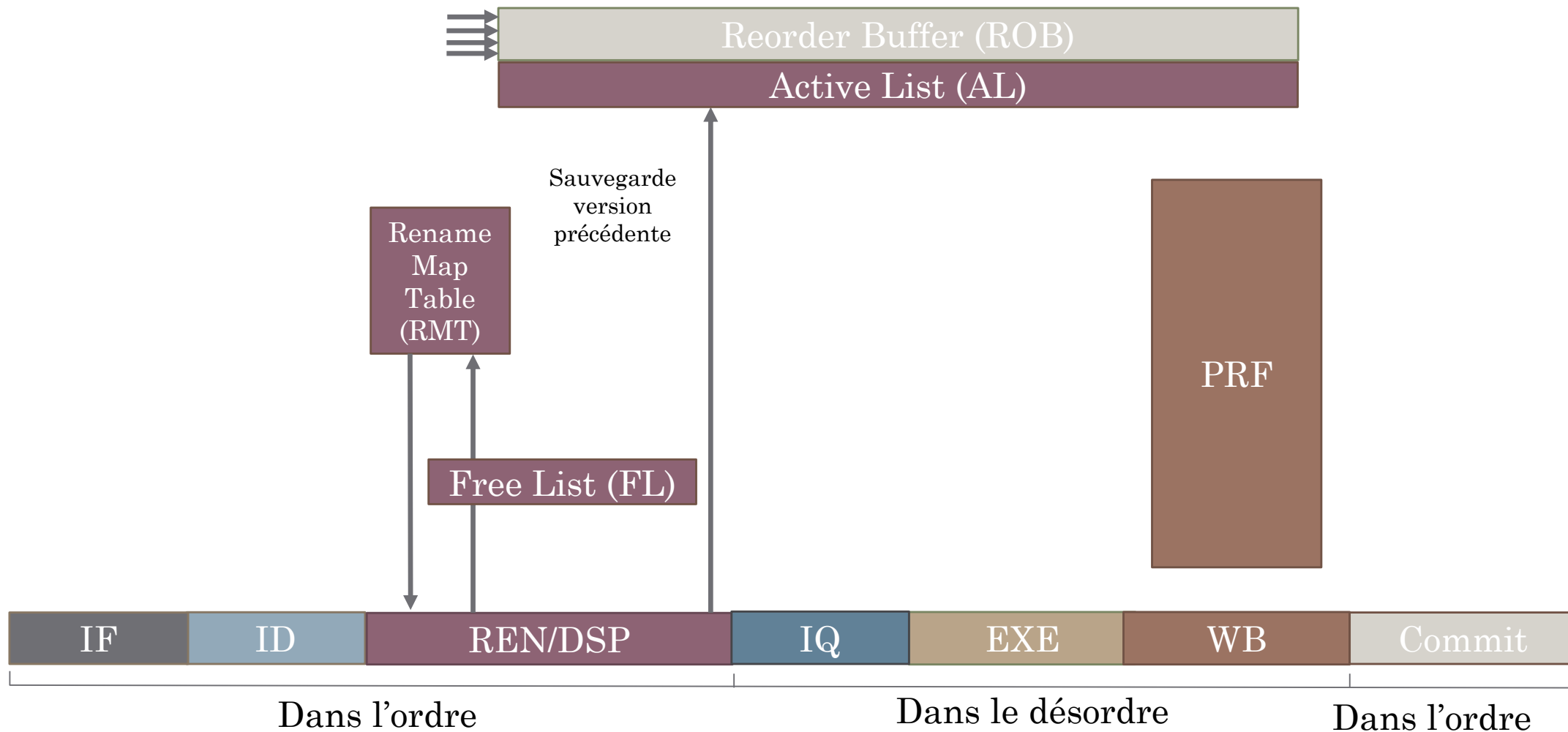
# Renommage de registres



# Renommage de registres

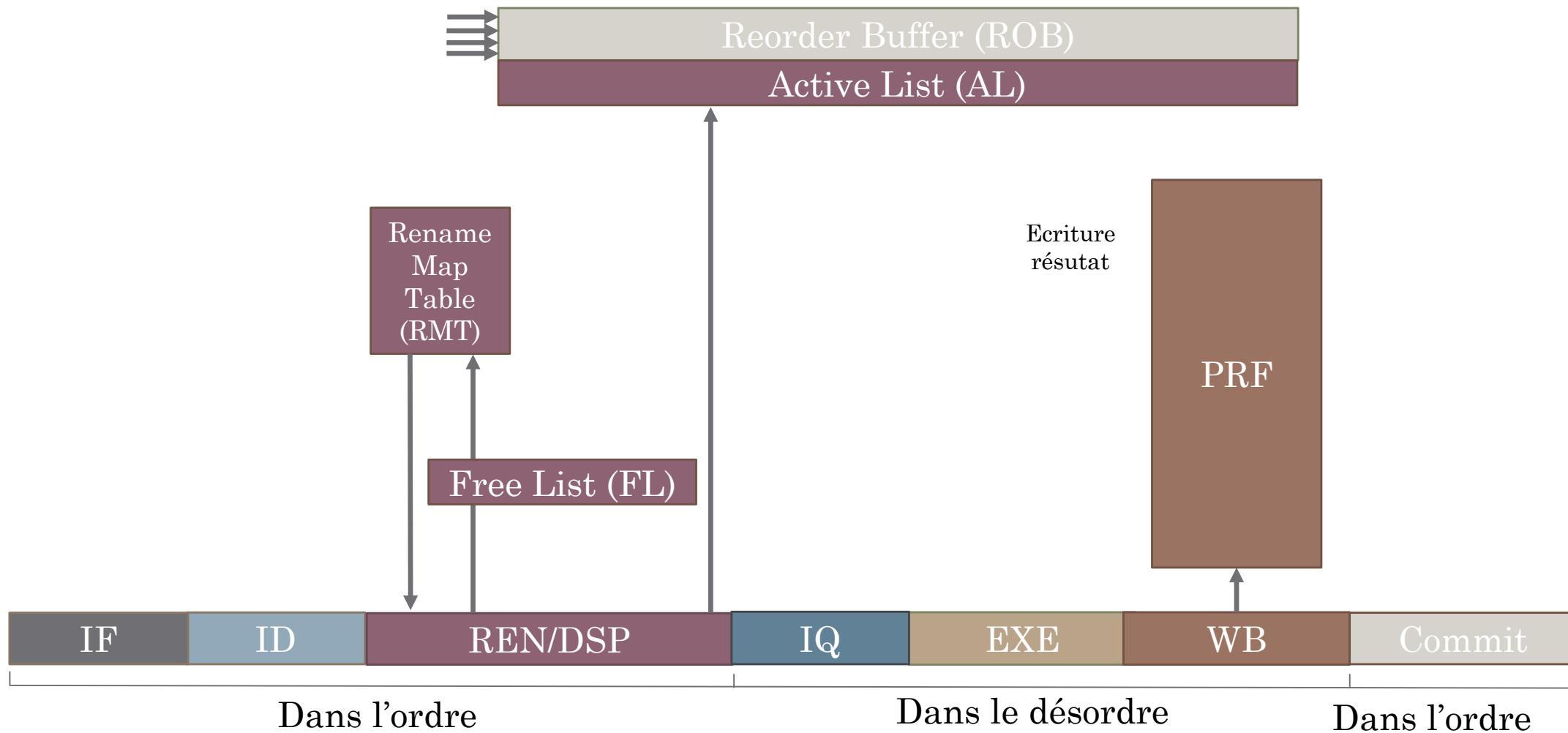


# Renommage de registres

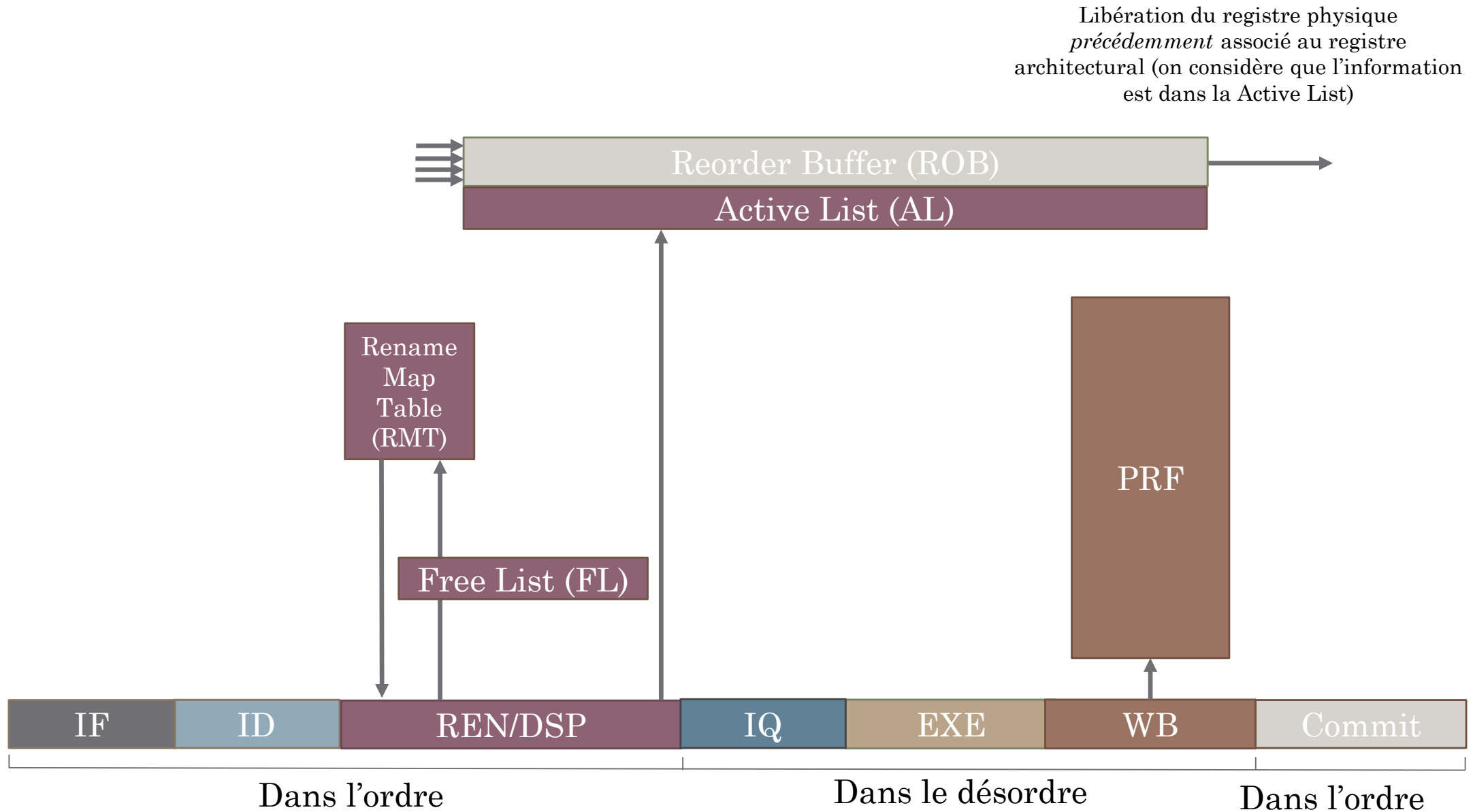




# Renommage de registres



# Renommage de registres



# Travaux pratiques

## RMT

x0	pr42
x1	pr98
x2	pr97
x3	pr96
x4	pr95
x5	pr94
x6	pr93
x7	pr92

## Code

```

I1: add x2, x1, x1
I2: add x3, x2, x2
I3: add x5, x4, x4
I4: sub x6, x7, x7
I5: add x4, x5, x3
I6: sub x2, x2, x1
I7: or x2, x1, x1
I8: sll x7, x5, x7
I9: srl x7, x7, x1
    
```

## Active List

Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
---------	---------	---------	---------	---------	---------	---------	---------

Pointeur de tête



pr7	pr6	pr5	pr4	pr3	pr2	pr1	pr0
-----	-----	-----	-----	-----	-----	-----	-----

## Free List

Pointeur de tête

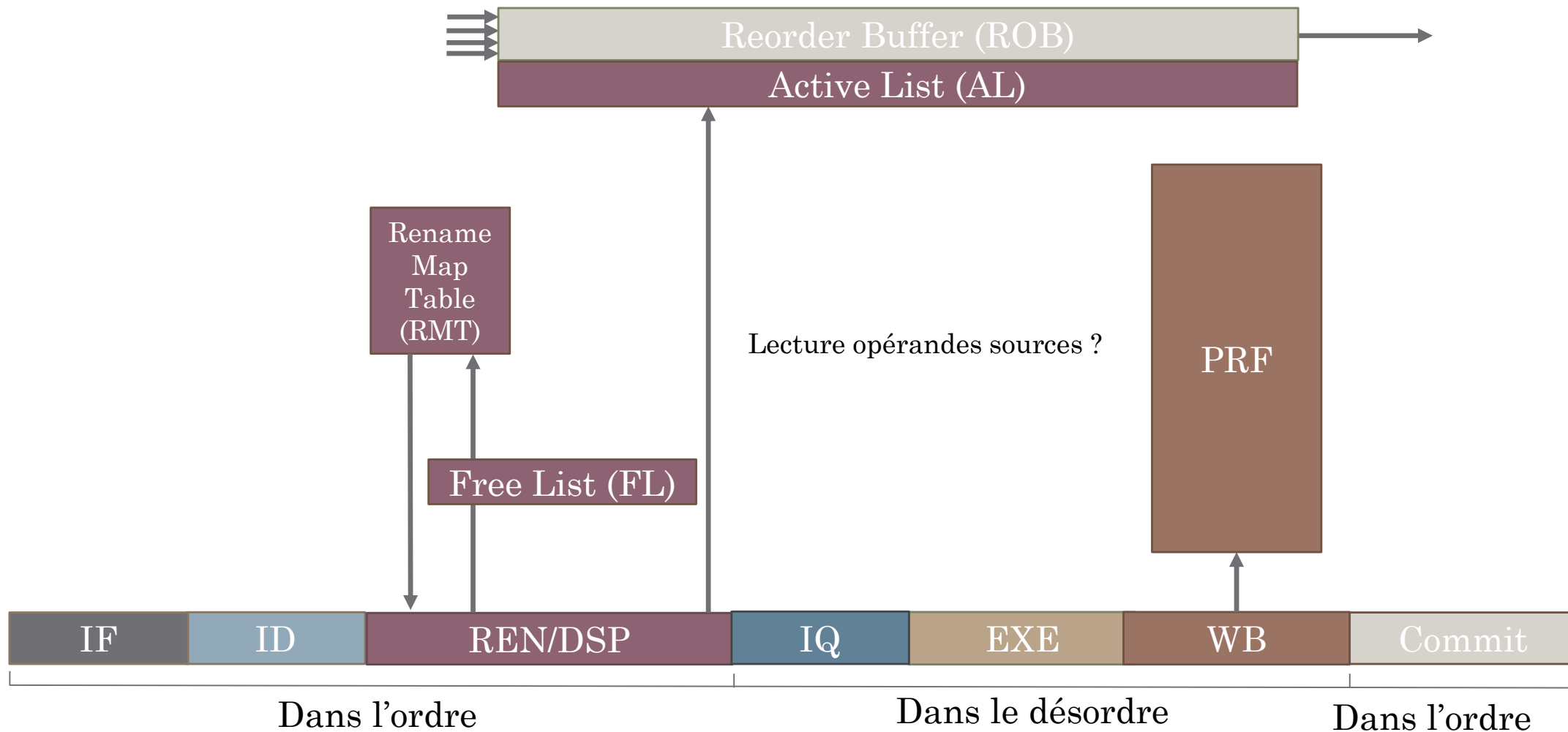


## Questions :

- Par quels états passent la RMT/FL/AL\* lors de l'exécution des instructions ?
- Quel est l'état de la RMT une fois que I9 a passé Commit ?

\*On considère que la AL est remplie avant que I1 passe Commit

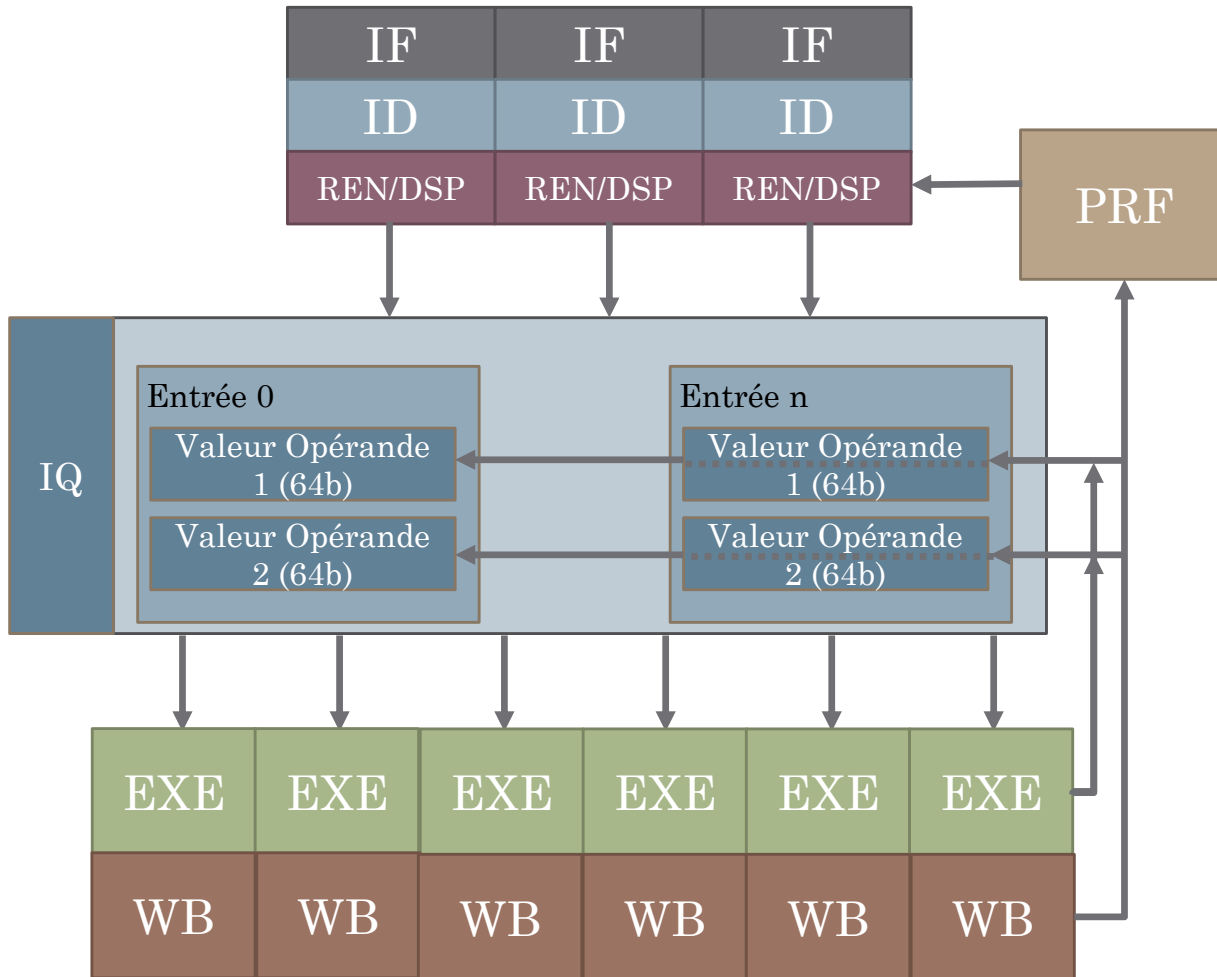
# Renommage de registres



# Lecture des sources

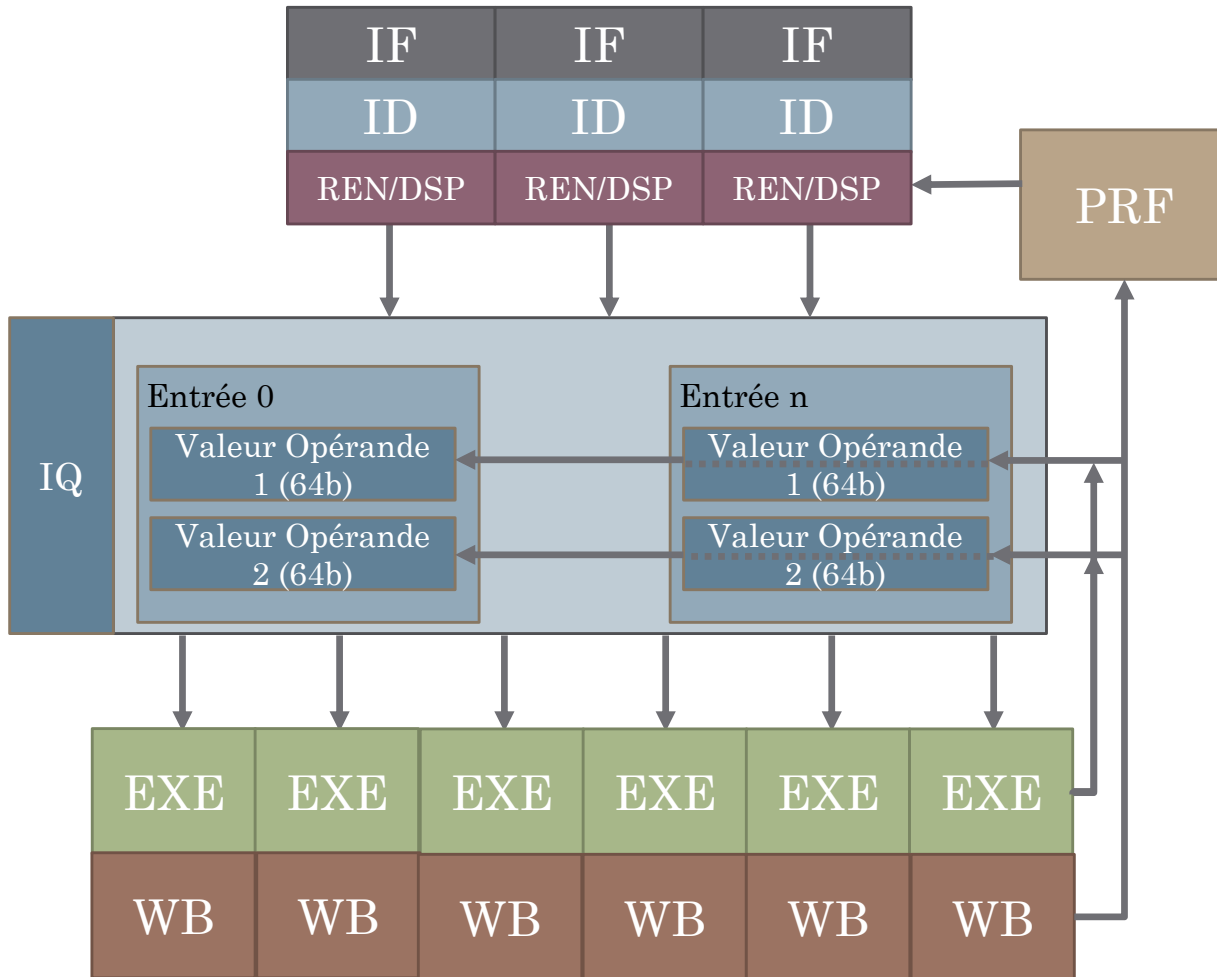
- Pipeline dans l'ordre :
  - Dans ID
  - Sur le bypass
- Dans le désordre :
  - Dans REN/DSP, après avoir renommé les sources
  - Sur le bypass
- L'ordonnanceur doit aussi capturer les résultats qui arrivent sur le réseau de bypass, sinon le résultat est perdu pour les consommateurs

# Ordonnanceur « value capture »



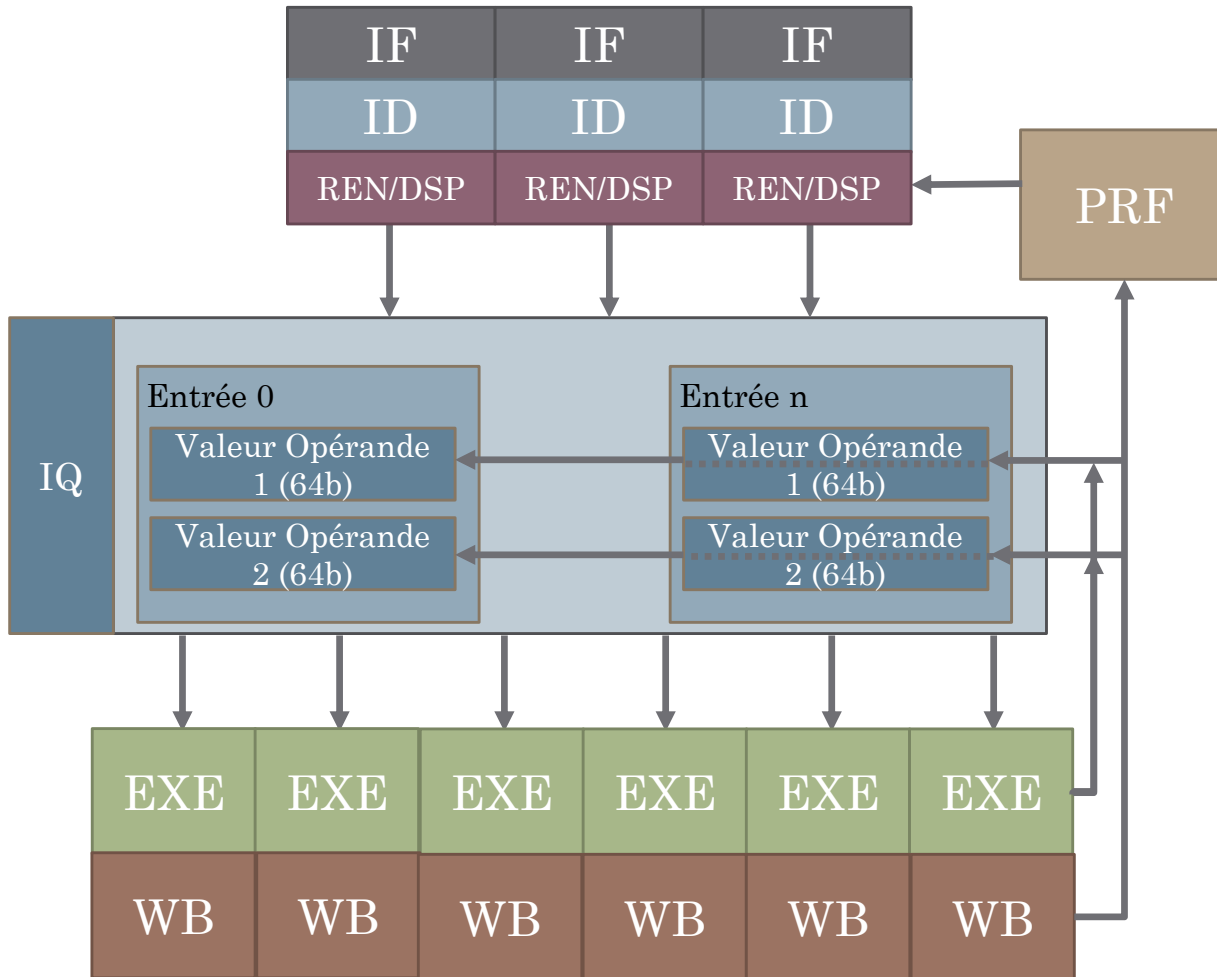
- L'opérande est lue depuis un registre dans DSP si le **registre est prêt**

# Ordonnanceur « value capture »



- L'opérande est lue depuis un registre dans DSP si le **registre est prêt**
- Sinon, le registre est produit alors que le consommateur attend dans IQ

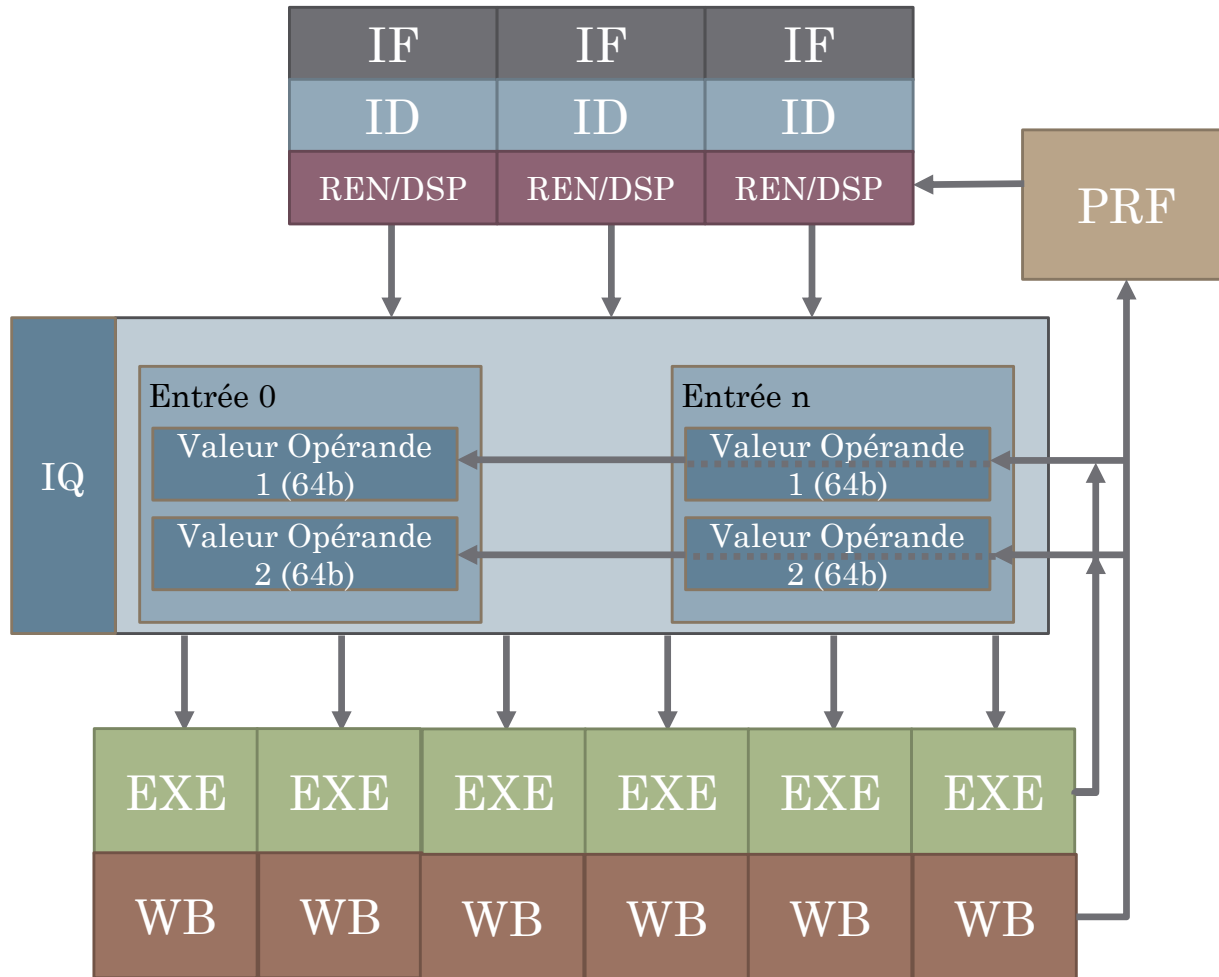
# Ordonnanceur « value capture »



- L'opérande est lue depuis un registre dans DSP si le **registre est prêt**
- Sinon, le registre est produit alors que le consommateur attend dans IQ
- Avec l'exécution OoO, une instruction prête n'est pas forcément exécutée immédiatement



# Ordonnanceur « value capture »

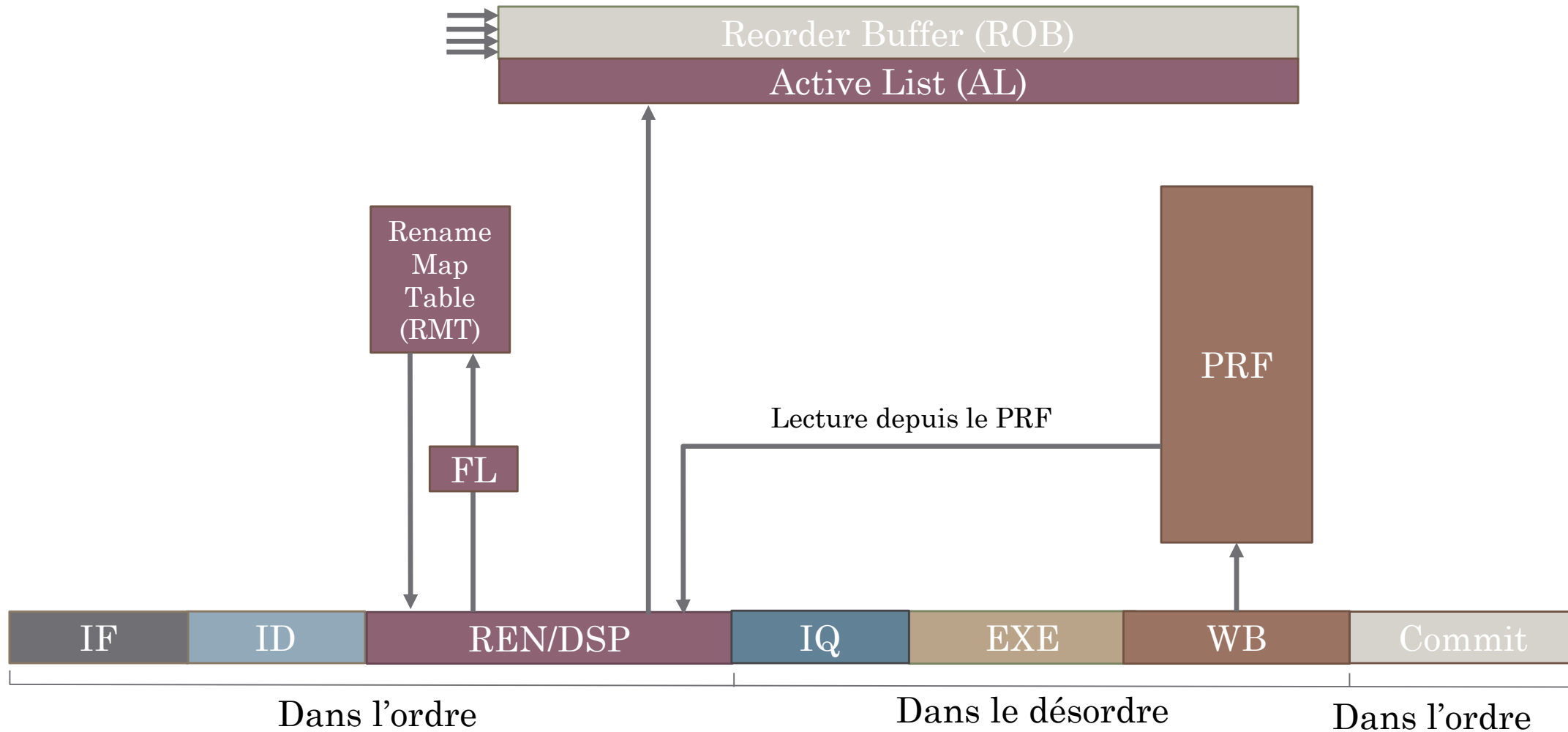


- L'opérande est lue depuis un registre dans DSP si le **registre est prêt**
- Sinon, le registre est produit alors que le consommateur attend dans IQ
- Avec l'exécution OoO, une instruction prête n'est pas forcément exécutée immédiatement
- Les entrées de l'IQ observent le réseau de bypass et « capturent » les opérandes à mesure qu'ils sont produits

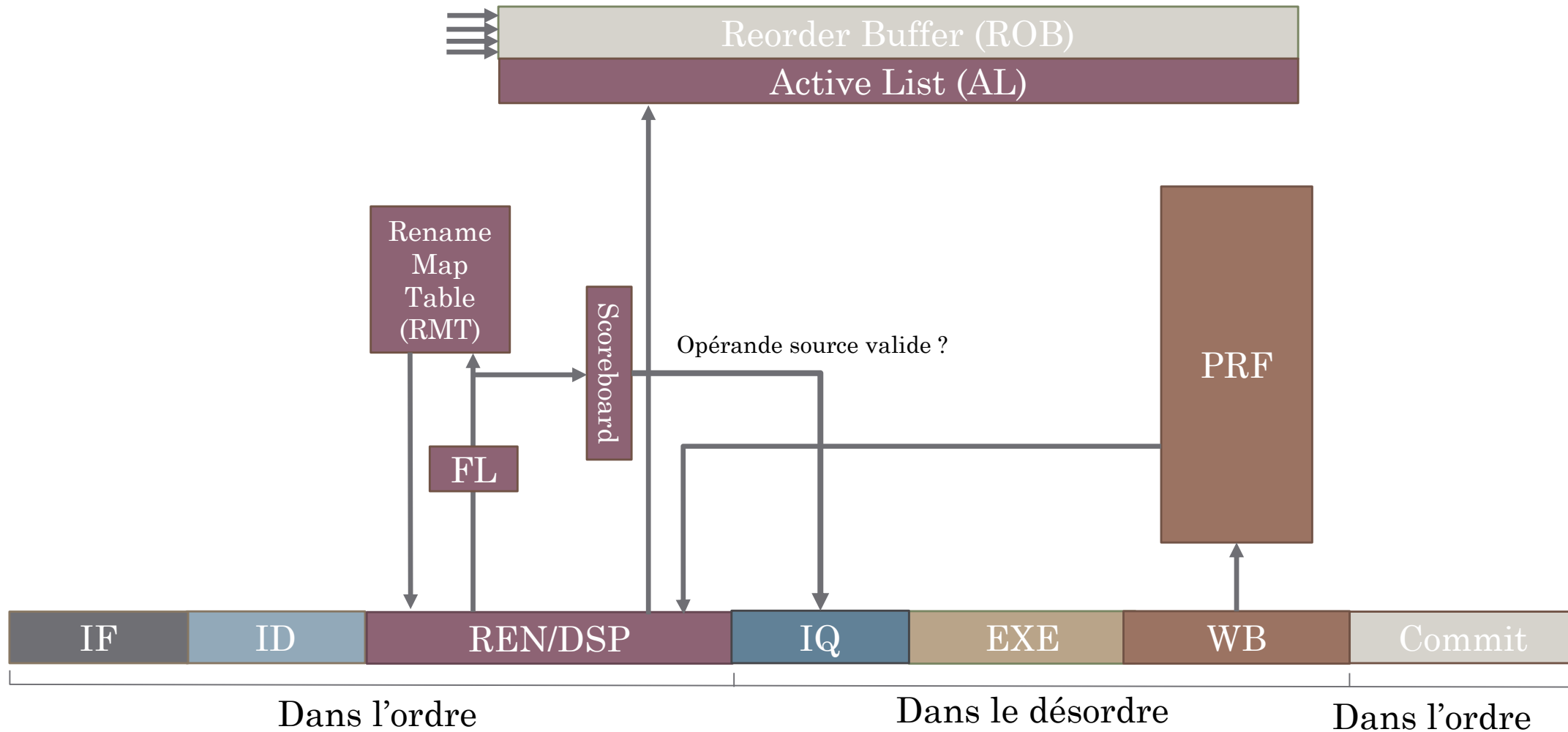
# Lecture des sources

- « L'opérande est lue depuis un registre dans DSP **si le registre est prêt** »
- Nouvelle structure lue dans REN/DSP, le *Scoreboard*
  - 1 bit par registre physique (1 = prêt, 0 = non prêt)
  - Mis à 0 lorsque le registre est alloué
  - Mis à 1 lorsque le registre est produit (instruction exécutée)
- Permet à REN/DSP de déterminer si l'opérande source
  - Doit être lu depuis les fichiers de registres
  - Sera capturé par l'entrée de l'IQ/directement lu sur le bypass

# Renommage de registres



# Renommage de registres



# Renommage : Résumé

- Casser les fausses dépendances de données (WaW, WaR)
- Ignorer les dépendances de contrôle
- Au moins cinq structures
  - **Physical Register File (PRF)** : Registres physiques
  - **Rename Map Table (RMT)** : Association arch. reg./phy. reg. la plus récente
  - **Active List (AL, FIFO)** : Association arch. reg/phy. reg. précédente du reg. arch. destination de l'instruction
  - **Free List (FL)** : Reg. phy. inutilisés
  - **Scoreboard (SCB)** : Reg. phy. prêt/Non prêt (on va y revenir, mais à ne pas confondre avec la technique de « Scoreboarding » pour faire de l'exécution dans le désordre)

# Renommage de registres - Variations

- Dans ce cours, on a présenté une version avec un banc de registre physiques (PRF) contenant l'état architectural ET l'état microarchitectural
- Des variations sont possibles mais c'est le schéma utilisé dans les processeurs modernes

# Exécution dans le désordre

Mémoire

# Exécution dans le désordre et accès mémoires

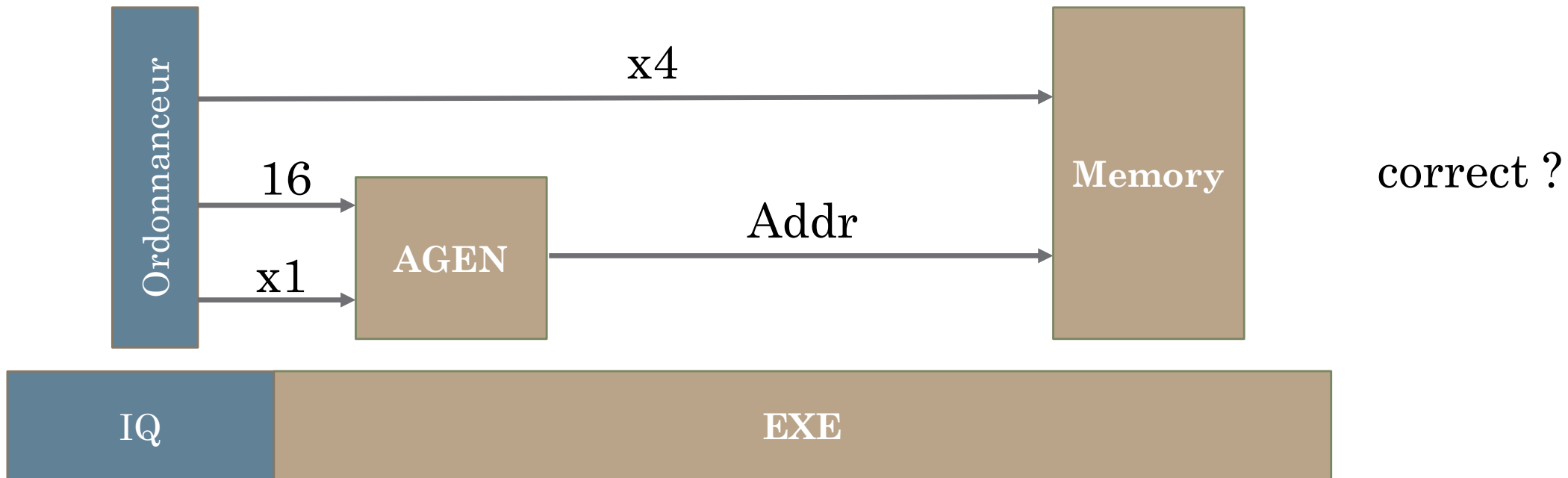
- Jusqu'ici, on s'est intéressé uniquement à la gestion des registres et aux dépendances de données via les registres
- Pas suffisant pour les accès mémoires
  - Ecritures
  - Lectures



# Exécution dans le désordre – Stores

- Quand peut-on écrire une donnée dans la mémoire ?

sd x4, 16(x1)



# Exécution dans le désordre – Stores

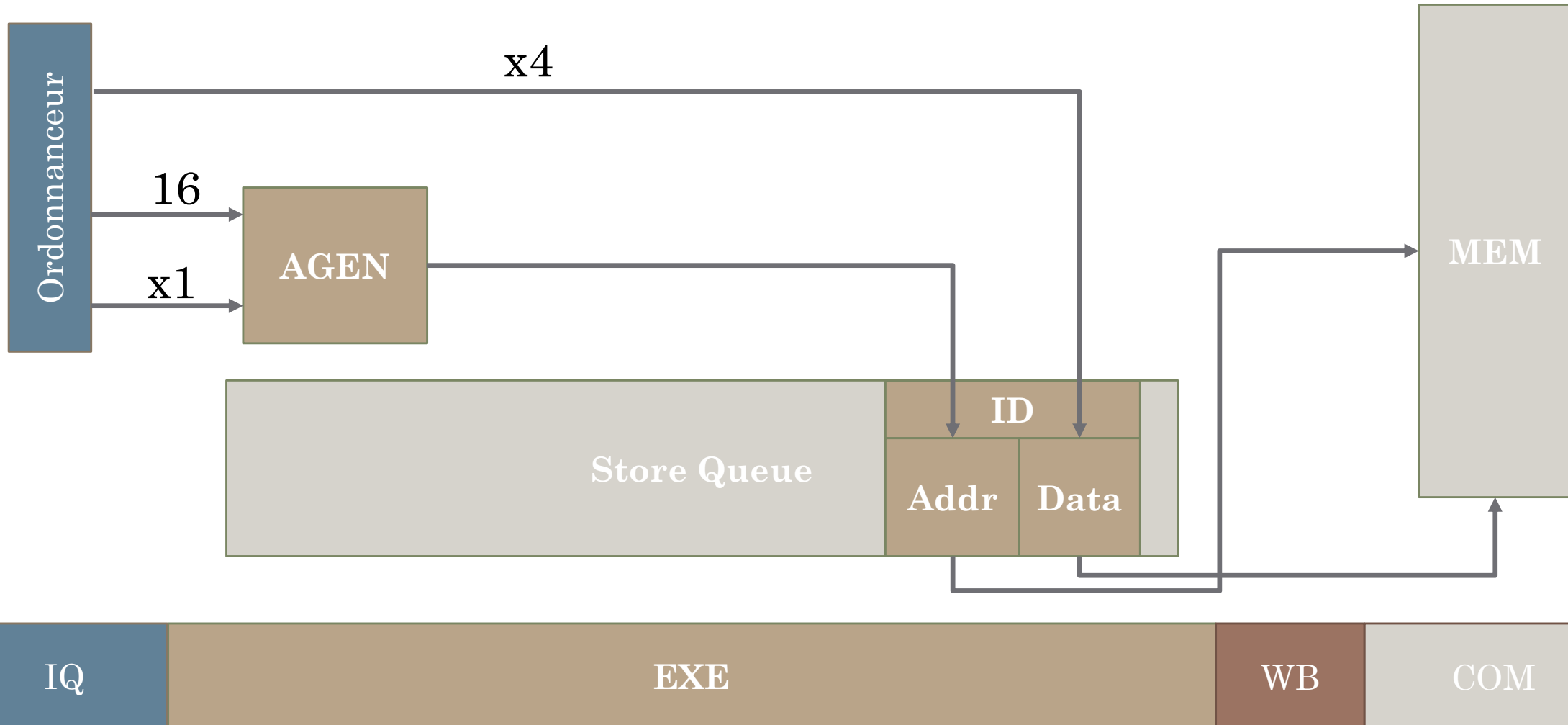
- La mémoire fait partie de l'état architectural (que le programmeur peut observer)
- Tant qu'une instruction n'a pas passé le Commit, elle est spéculative : elle sera peut-être jetée du pipeline
- Ecrire dans la mémoire spéculativement = effets du store observables architecturalement avant Commit
  - Ne respecte pas le contrat !

# Exécution dans le désordre – Stores

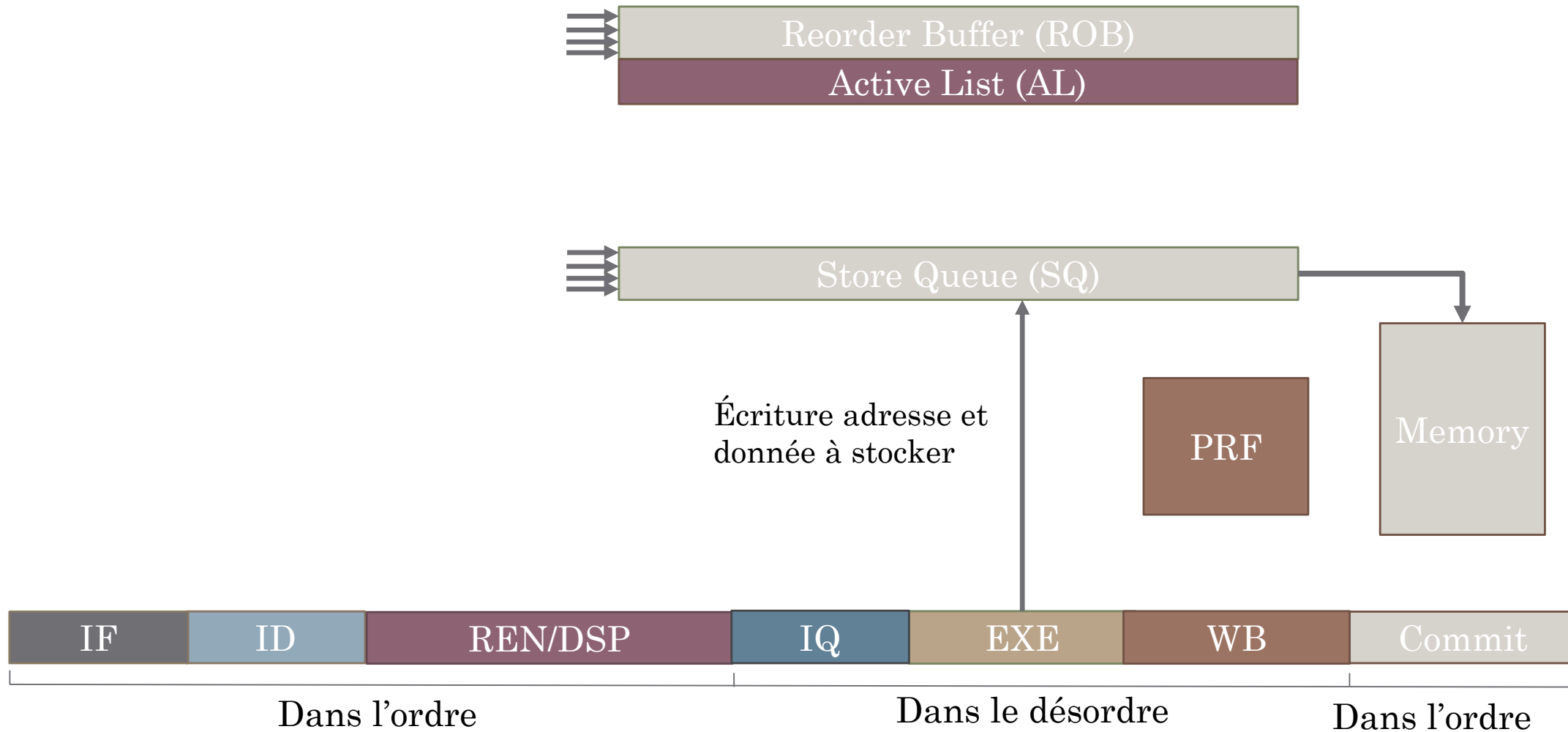
- Exécuter les stores au Commit
  - Dans l'ordre donc on ne peut pas cacher la latence d'exécution via exécution dans le désordre
- On introduit donc une nouvelle mémoire matérielle : La Store Queue (FIFO)
  - Peut être vue comme un sous-ensemble du ROB spécialisé pour les stores
  - Stocke l'adresse et les données de chaque store « en vol » (spéculatif)

# Exécution dans le désordre – Stores

sd x4, 16(x1)



# Renommage de registres



# Exécution dans le désordre – Stores

- La Store Queue fournit un endroit unique à chaque store pour écrire leur « résultat »
  - Les stores s'exécutent dans le désordre et modifient l'état *microarchitectural* (SQ)
  - Modifications de l'état architectural (mémoire) au Commit
- Similaire au renommage de registres
  - Pas de WaW : Exécution store-store dans le désordre
  - Pas de WaR : Exécution load-store dans le désordre

# Exécution dans le désordre – Mémoire

- Quand peut-on écrire une donnée dans la mémoire ?
  - **Store Queue**
- Quand peut-on lire une donnée depuis la mémoire ?
  - Spéculativement, tant que c'est la bonne donnée : Dépendance RaW potentielle avec les instructions stores plus vieilles

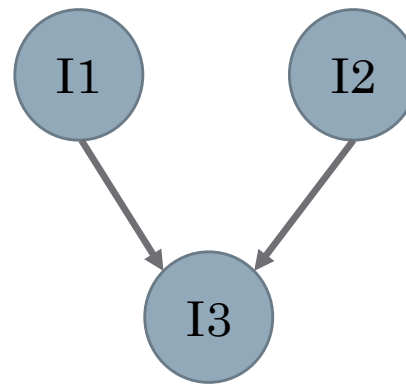
# Exécution dans le désordre – Mémoire

- On a parlé des dépendances de données entre registres, mais les données passent aussi par la mémoire

I1 : sd x3, 0(x1)

I2 : sd x4, 16(x2)

I3 : ld x5, 0(x6)



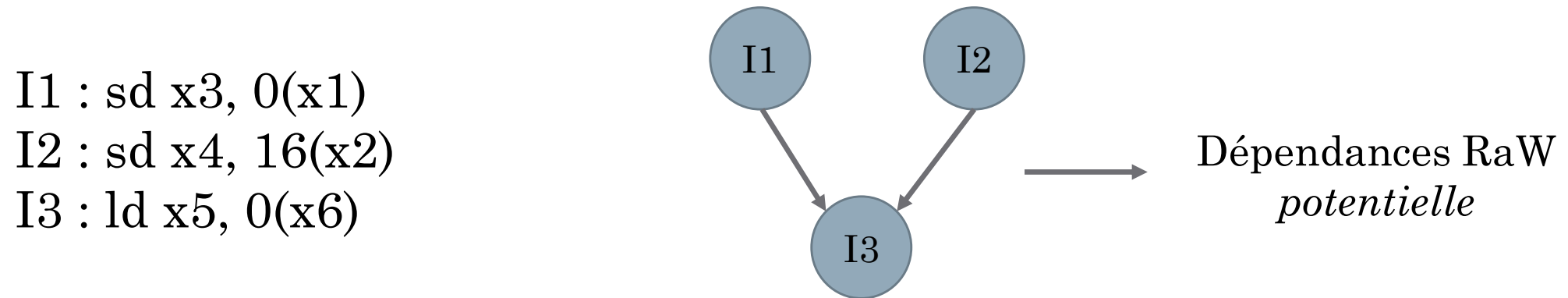
→ Dépendances RaW  
*potentielle*

- Quelle est la bonne valeur pour x5 ?



# Exécution dans le désordre – Mémoire

- On a parlé des dépendances de données entre registres, mais les données passent aussi par la mémoire



- Quelle est la bonne valeur pour x5 ?
  - Dépend de si  $(x2 + 16) == x6$  et si  $x1 == x6$ , ce qu'on ne sait pas avant d'avoir exécuté les instructions

# Exécution dans le désordre – Mémoire

- Via les registres, une dépendance RaW existe toujours, indépendamment de la valeur du registre
  - Dépendance connue lors du décodage/renommage : **avant** d'avoir choisi une instruction à exécuter
- Via la mémoire, existe seulement si les registres utilisés pour calculer les adresses ont des valeurs spécifiques
  - Dépendance découverte à l'exécution, une fois les adresses connues : **après** avoir choisi une instruction à exécuter

# Exécution dans le désordre – Mémoire

- Problème 1 : un load ne peut s'exécuter que si ses dépendances sont satisfaites, i.e. :
  - Après l'exécution de tous les stores plus vieux (RaW potentielles)

# Exécution dans le désordre – Mémoire

- Problème 1 : un load ne peut s'exécuter que si ses dépendances sont satisfaites, i.e. :
  - Après l'exécution de tous les stores plus vieux (RaW potentielles)
- Un load ne peut s'exécuter que si tout les stores plus vieux sont exécutés ?
  - Limite fortement l'exécution dans le désordre...
  - Présence de dépendance même pas garantie, on peut attendre puis découvrir qu'en fait aucun store ne stocke à la même adresse que le load

# Dépendances mémoire

- Prédiction de dépendances mémoire :
  - Gros grains : Prédire qu'un load dépend d'un ou plusieurs stores sans savoir le(s)quel(s)
  - Grains fins : Prédire le(s) producteur(s) précis d'un load
- Solution prédictive => spécule qu'une dépendance RaW *potentielle* est artificielle ou bien réelle

# Dépendances mémoire

- Prédiction de dépendances mémoire :
  - Gros grains : Prédire qu'un load dépend d'un ou plusieurs stores sans savoir le(s)quel(s)
  - Grains fins : Prédire le(s) producteur(s) précis d'un load
- Solution prédictive => spécule qu'une dépendance RaW *potentielle* est artificielle ou bien réelle
  - Artificielle : Load ne dépend pas de stores plus vieux, dépendances RaW via la mémoire immédiatement satisfaite (spéculativement)

# Dépendances mémoire

- Prédiction de dépendances mémoire :
  - Gros grains : Prédire qu'un load dépend d'un ou plusieurs stores sans savoir le(s)quel(s)
  - Grains fins : Prédire le(s) producteur(s) précis d'un load
- Solution prédictive => spécule qu'une dépendance RaW *potentielle* est artificielle ou bien réelle
  - Artificielle : Load ne dépend pas de stores plus vieux, dépendances RaW via la mémoire immédiatement satisfaite (spéculativement)
  - Réelle : Load dépend de store plus vieux, dépendance RaW via la mémoire satisfaite quand ces stores seront exécutés

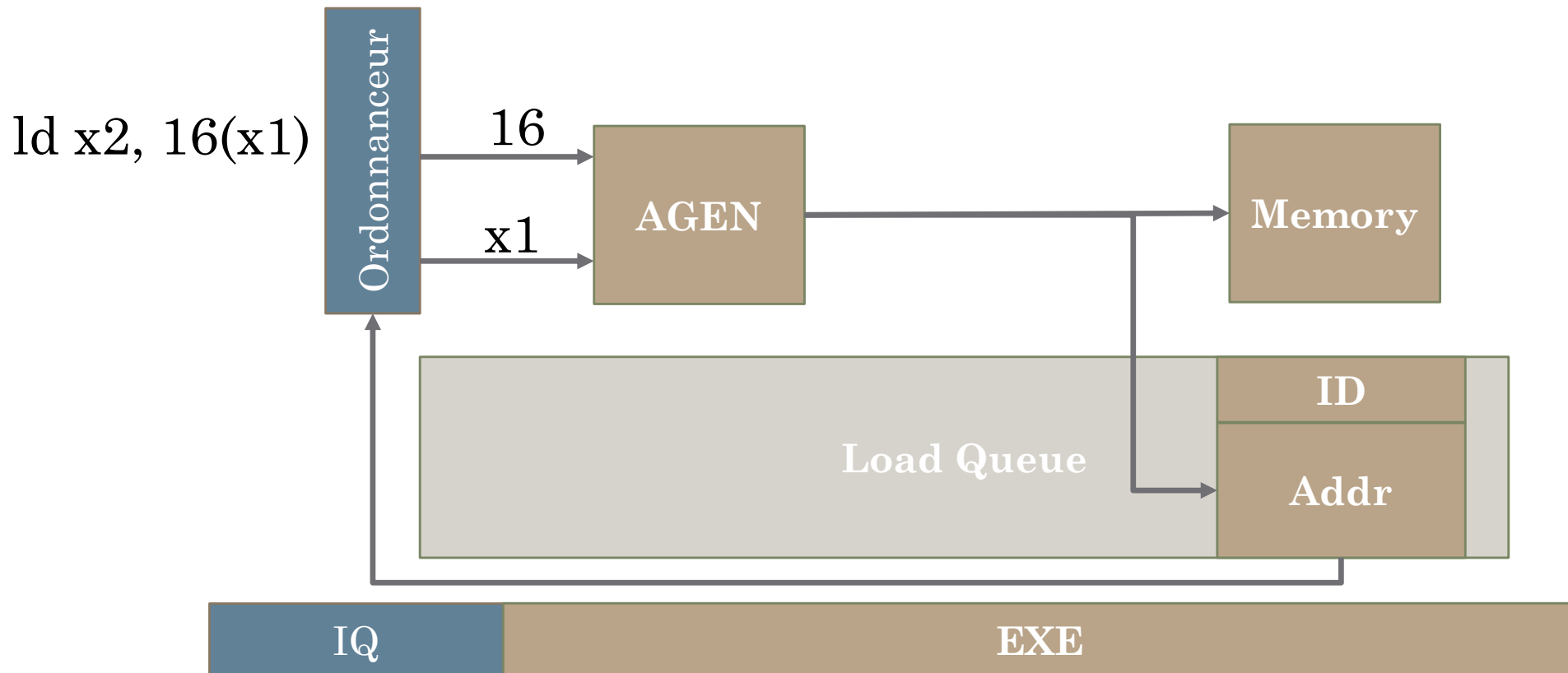
# Dépendances mémoire

- Prédiction de dépendances mémoire :
  - Gros grains : Prédire qu'un load dépend d'un ou plusieurs stores sans savoir le(s)quel(s)
  - Grains fins : Prédire le(s) producteur(s) précis d'un load
- Solution prédictive => spécule qu'une dépendance RaW *potentielle* est artificielle ou bien réelle
- **On peut se tromper :**
  - Comment *détecter* une erreur ?
  - Comment *réparer* une erreur ?

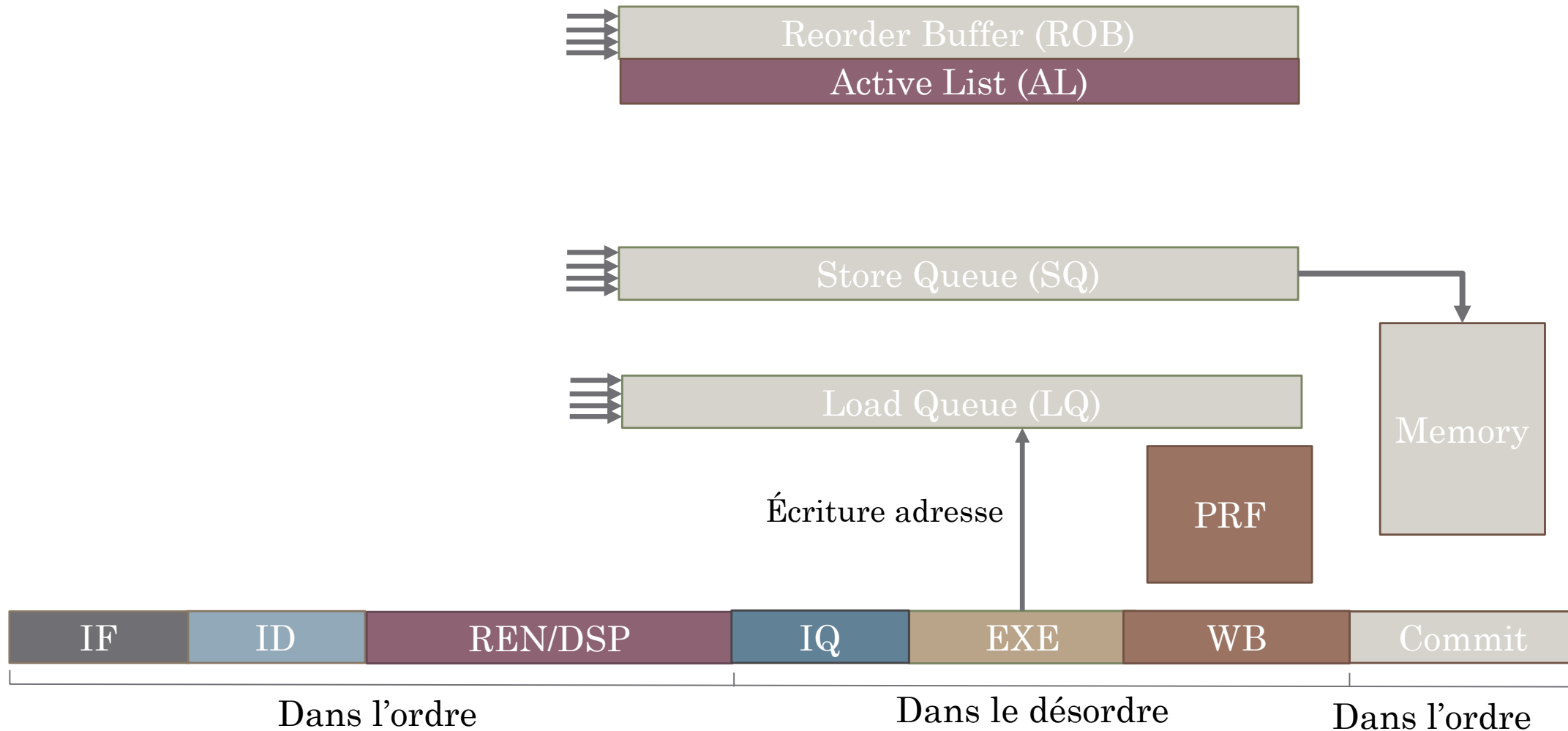


# Dépendances mémoire – Load Queue

- Comme pour les stores, on introduit une FIFO pour les loads « en vol » (spéculatifs) : La Load Queue
  - Contient notamment l'adresse accédée par le load, une fois calculée



# Dépendances mémoire - Spéculation

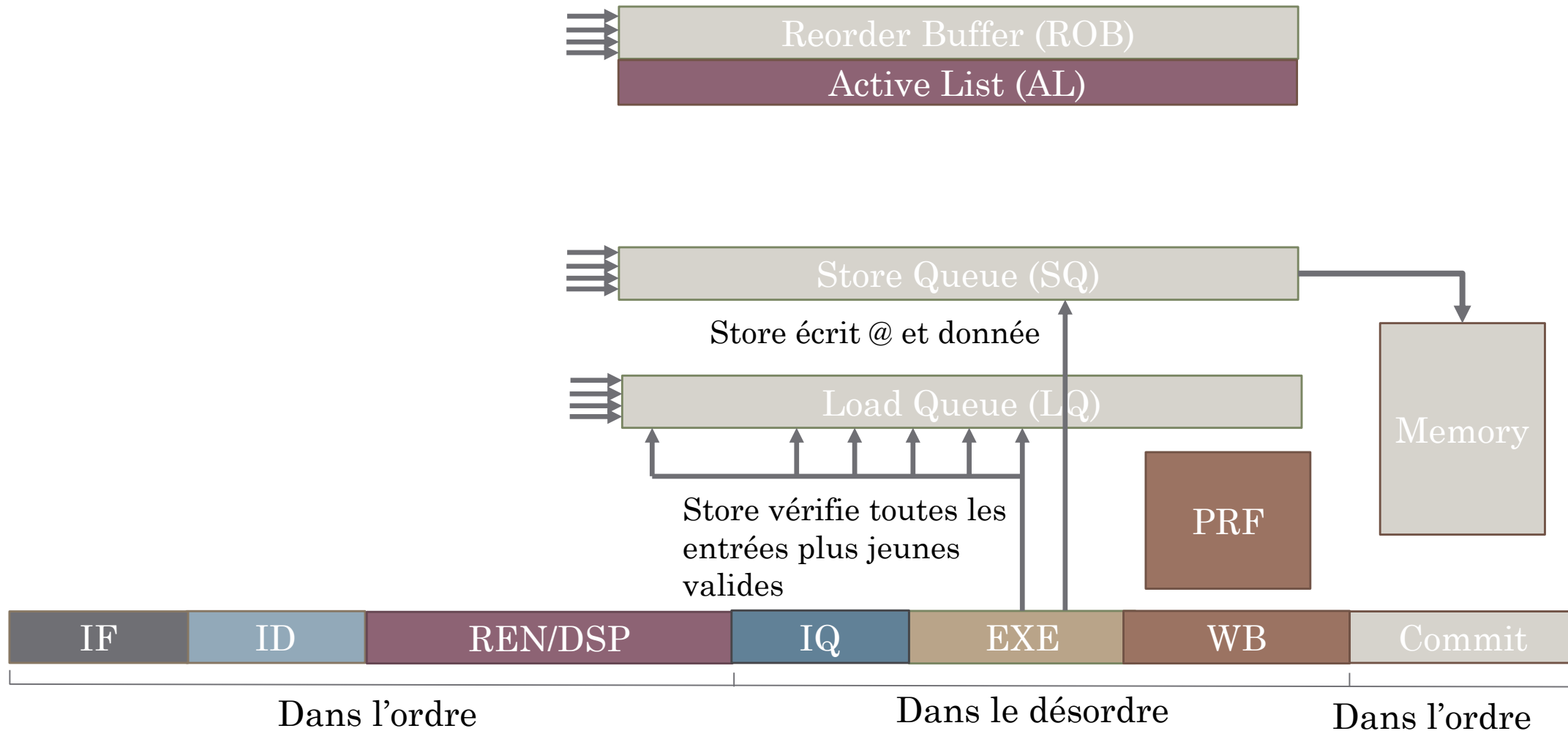


# Dépendances mémoire

- Lorsqu'un store s'exécute, il scanne toutes les entrées de la Load Queue qui :
  - Contiennent une adresse valide (load exécuté)
  - Sont plus jeunes
- Si l'adresse contenue dans l'entrée de la LQ correspond à l'adresse du store, alors la *RaW potentielle* est réelle, et n'a pas été respectée
  - On vide le pipeline depuis le load (inclus), comme pour une mauvaise prédiction de branchement\*
- Si aucune correspondance, la *RaW potentielle* entre ce store et les loads plus jeunes déjà exécutés était artificielle

\*On doit aussi enlever les instructions plus jeunes de la LQ/SQ

# Dépendances mémoire - Spéculation



# Exécution dans le désordre – Mémoire

- ~~Problème 1 : une instruction ne peut s'exécuter que si ses dépendances sont satisfaites~~
  - ~~Un load a une dépendance RaW potentielle sur tous les stores plus vieux en vol dans le pipeline, ce qui limite fortement l'exécution dans le désordre~~
  - Spéculation + store accède LQ pour détecter les mauvaises prédictions
- Problème 2 : Exécution dans l'ordre mais dépendance RaW avec un store encore en vol (spéculatif)
  - Comment identifier la dépendance ?

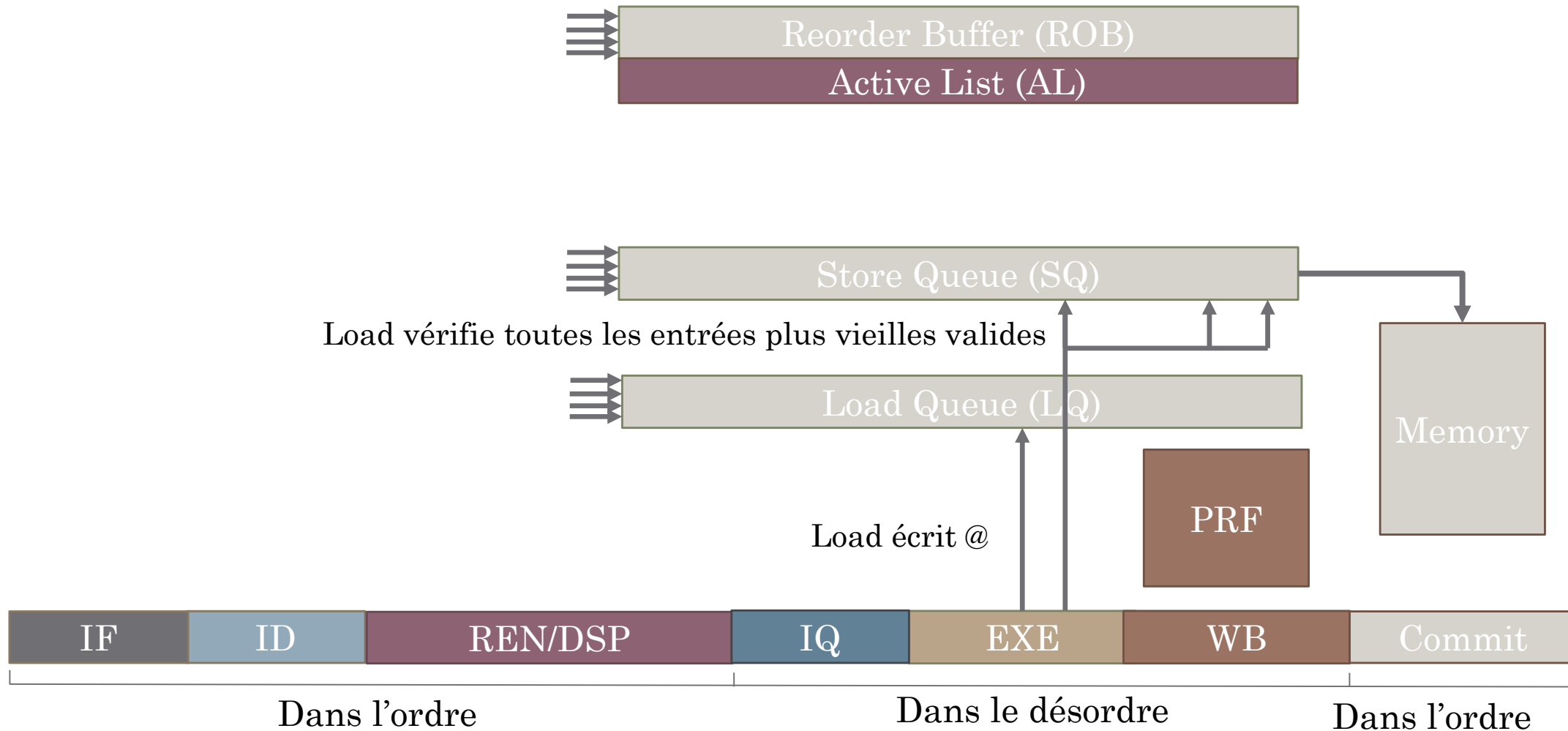
# Dépendances mémoire - Spéculation

- Même si un load s'exécute après un store à la même adresse, il faut quand même détecter l'existence de la dépendance RaW
  - On le fait via la RMT pour les registres, mais pour la mémoire ?

# Dépendances mémoire - Spéculation

- Même si un load s'exécute après un store à la même adresse, il faut quand même détecter l'existence de la dépendance RaW
  - On le fait via la RMT pour les registres, mais pour la mémoire ?
- Dual de la technique précédente
  - Chaque load qui s'exécute vérifie les entrées de la SQ qui
    - Contiennent une adresse valide (store exécuté)
    - Sont plus vieilles

# Dépendances mémoire - Spéculation





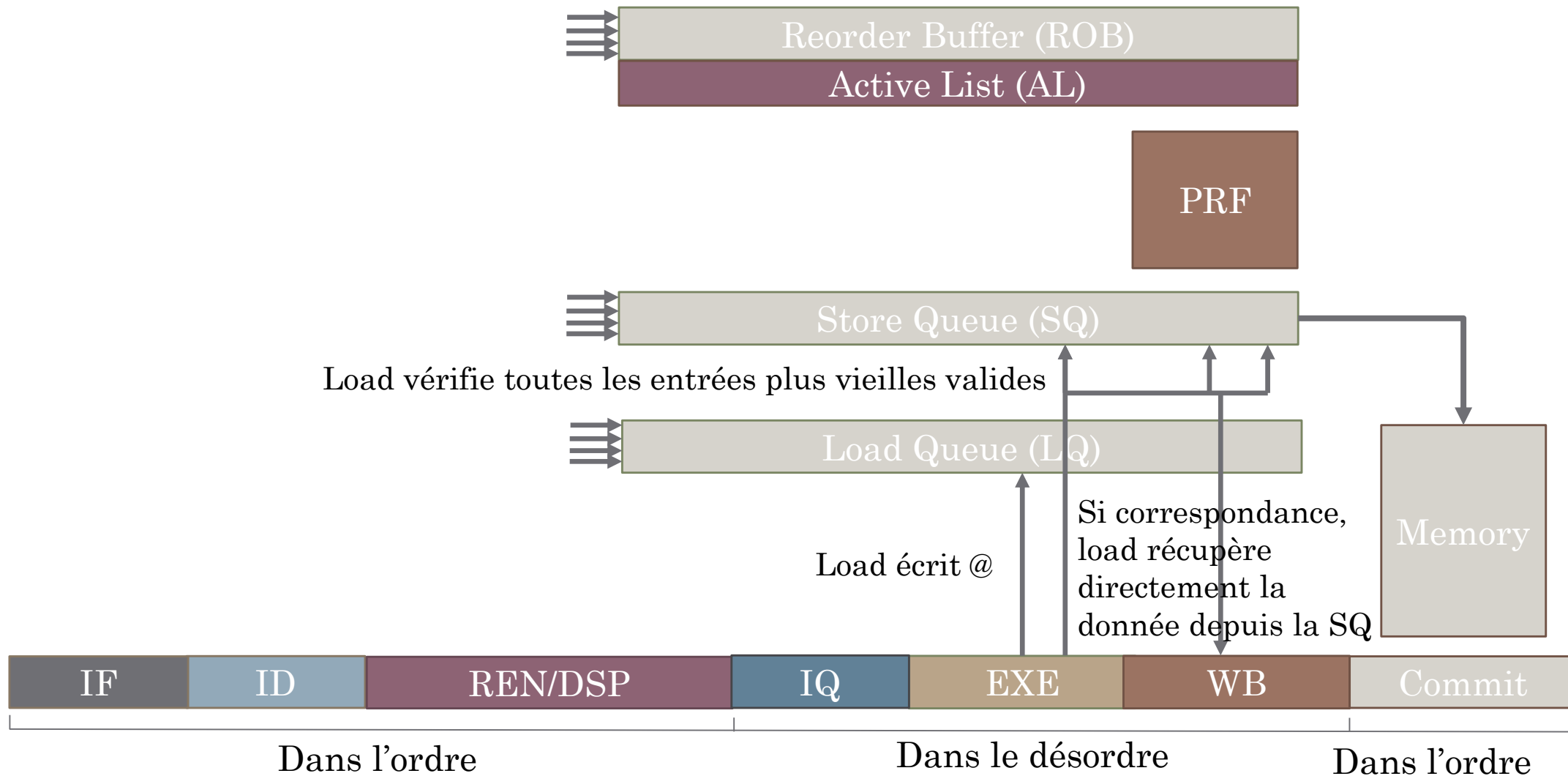
# Dépendances mémoire - Spéculation

- Même si un load s'exécute après un store à la même adresse, il faut quand même détecter l'existence de la dépendance RaW
  - On le fait via la RMT pour les registres, mais pour la mémoire ?
- Dual de la technique précédente
  - Chaque load qui s'exécute scanne les entrées de la SQ qui
    - Contiennent une adresse valide (store exécuté)
    - Sont plus vieilles
- Si correspondance, alors RaW *potentielle* est réelle
  - Load doit attendre que le store écrive dans la mémoire ?

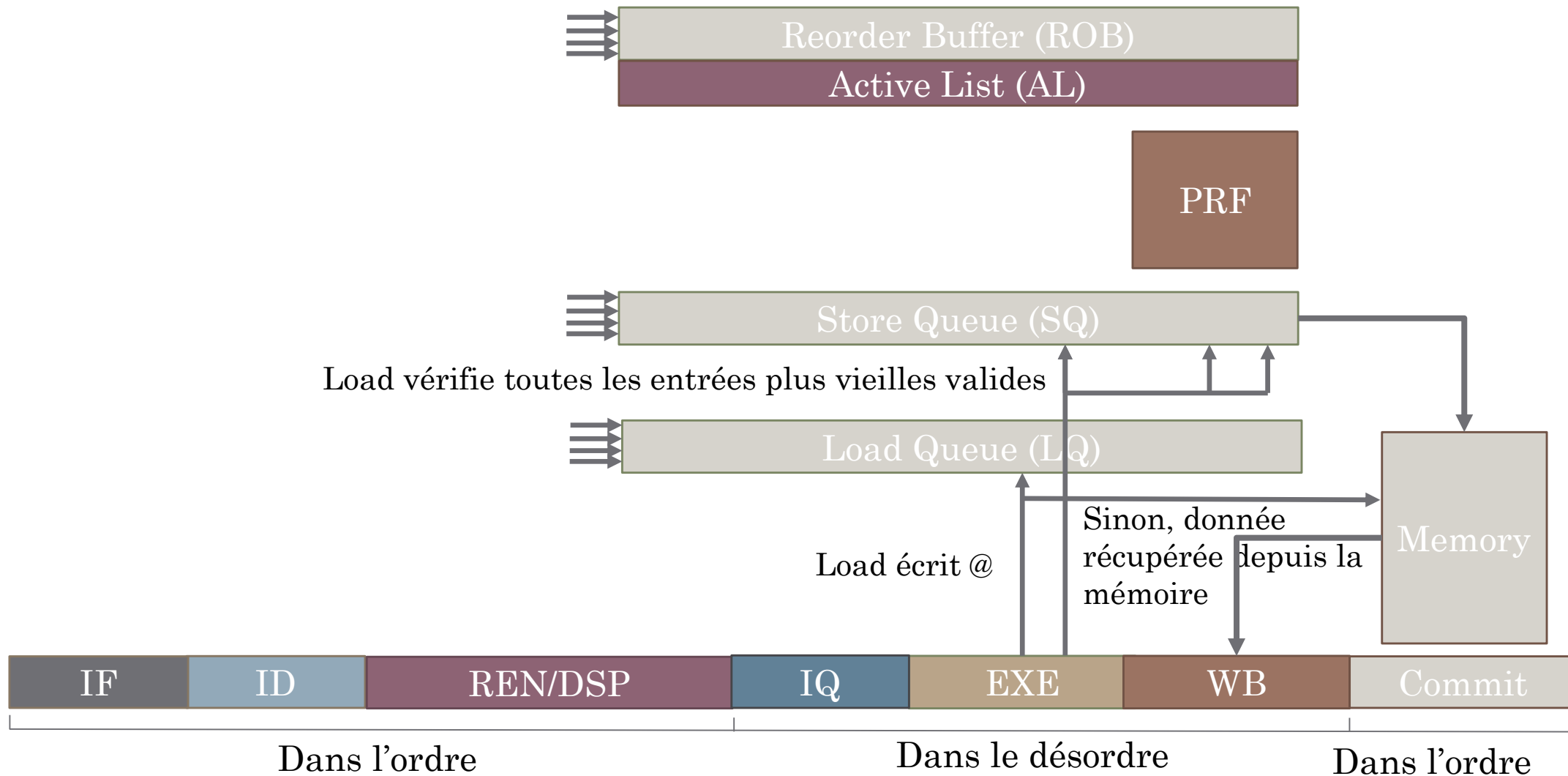
# Dépendances mémoire - Spéculation

- Même si un load s'exécute après un store à la même adresse, il faut quand même détecter l'existence de la dépendance RaW
  - On le fait via la RMT pour les registres, mais pour la mémoire ?
- Dual de la technique précédente
  - Chaque load qui s'exécute scanne les entrées de la SQ qui
    - Contiennent une adresse valide (store exécuté)
    - Sont plus vieilles
- Si correspondance, alors RaW *potentielle* est avérée
  - Load doit attendre que le store écrive dans la mémoire ?
  - Autant récupérer la donnée depuis la SQ directement
    - Store-to-Load Forwarding (STLDF)

# Dépendances mémoire - Spéculation



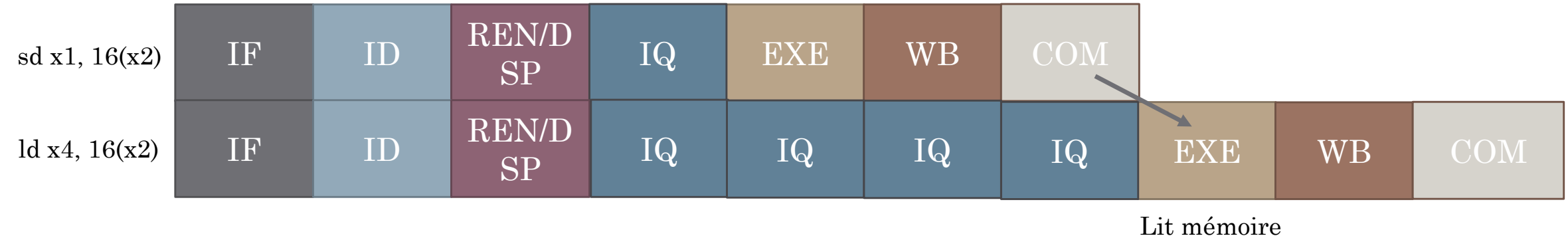
# Dépendances mémoire - Spéculation



# STLDF = Bypass mais pour la mémoire

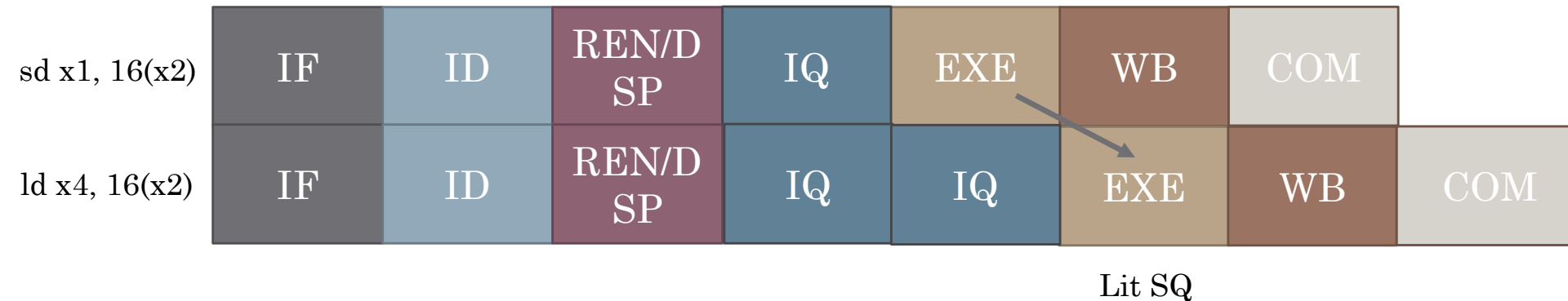
Sans STLDF

Écrit mémoire



Avec STLDF

Écrit SQ



# Exécution dans le désordre – Mémoire

- ~~Problème 1 : une instruction ne peut s'exécuter que si ses dépendances sont satisfaites~~
  - ~~Un load a une dépendance RaW potentielle sur tous les stores plus vieux en vol dans le pipeline, ce qui limite fortement l'exécution dans le désordre~~
  - ~~Spéculation + store accède LQ pour identifier les RaW mémoire non respectées (mauvaises prédictions de dépendance)~~
- ~~Problème 2 : Exécution dans l'ordre mais dépendance RaW avec un store encore en vol (spéculatif)~~
  - ~~Le load doit de toute façon attendre que le store écrive la mémoire (= Commit) avant de lire la mémoire~~
  - ~~Load accède SQ pour identifier les RaW mémoire réelles~~
  - ~~Si RaW réelle et respectée (prédiction de dépendance correcte) récupère la donnée depuis la SQ (STLDF)~~

# Exécution dans le désordre

Un mot sur l'ordonnanceur

# Ordonnanceur

- Jusqu'ici, on a considéré que les instructions rentraient dans l'ordonnanceur dans l'ordre, puis attendaient que leur opérandes sources soient disponibles pour s'exécuter au plus tôt
- A quoi ressemble l'ordonnanceur ?



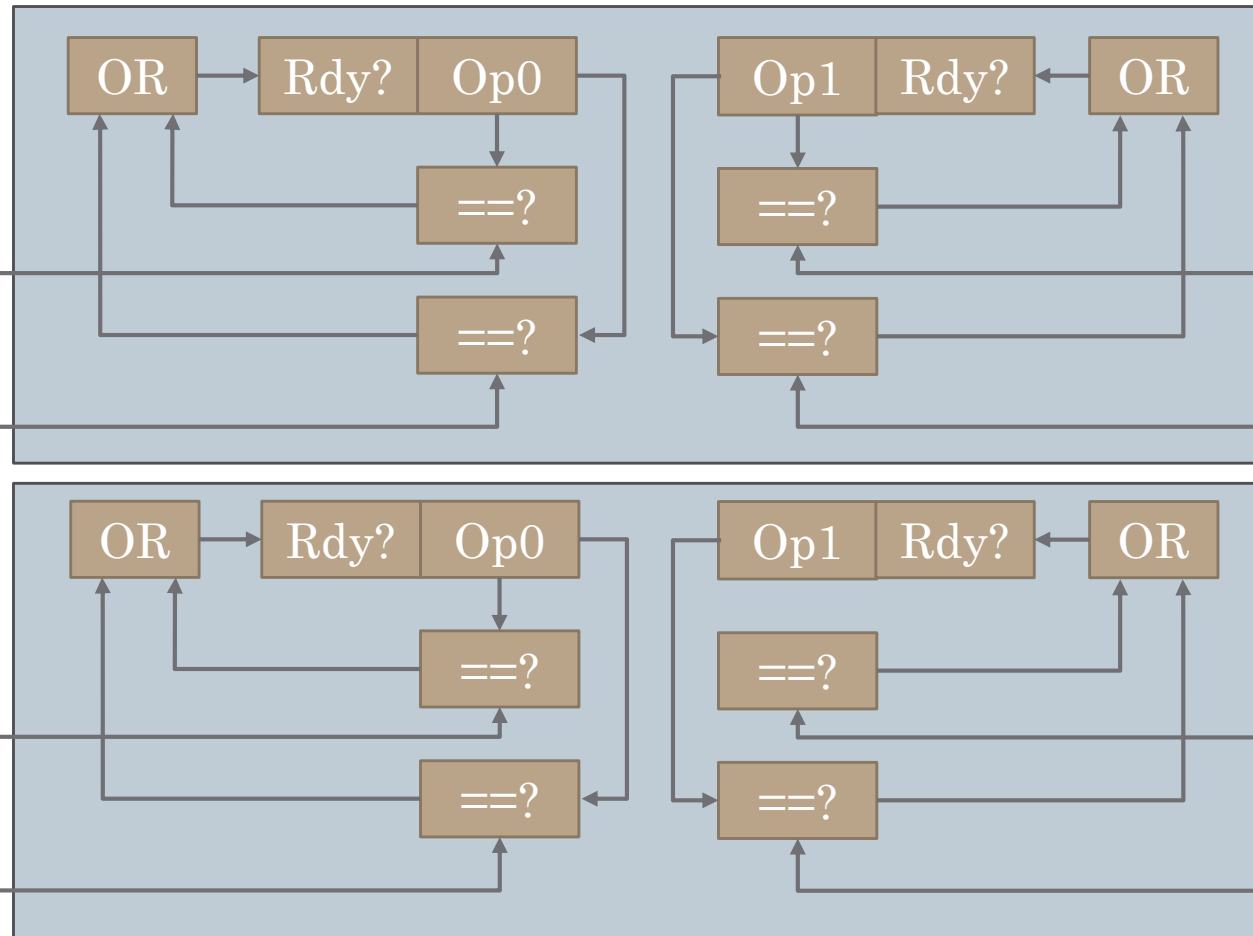
# Ordonnanceur

- Jusqu'ici, on a considéré que les instructions rentraient dans l'ordonnanceur dans l'ordre, puis attendaient que leur opérandes sources soient disponibles pour s'exécuter au plus tôt
- A quoi ressemble l'ordonnanceur ?
- Deux responsabilités, chaque cycle
  - *Wakeup* : Déterminer si une instruction est prête à être exécutée
  - *Select* : Choisir  $n$  instructions prêtes et les envoyer aux unités fonctionnelles

# Ordonnanceur – Wakeup

pregx pregx

pregx pregx



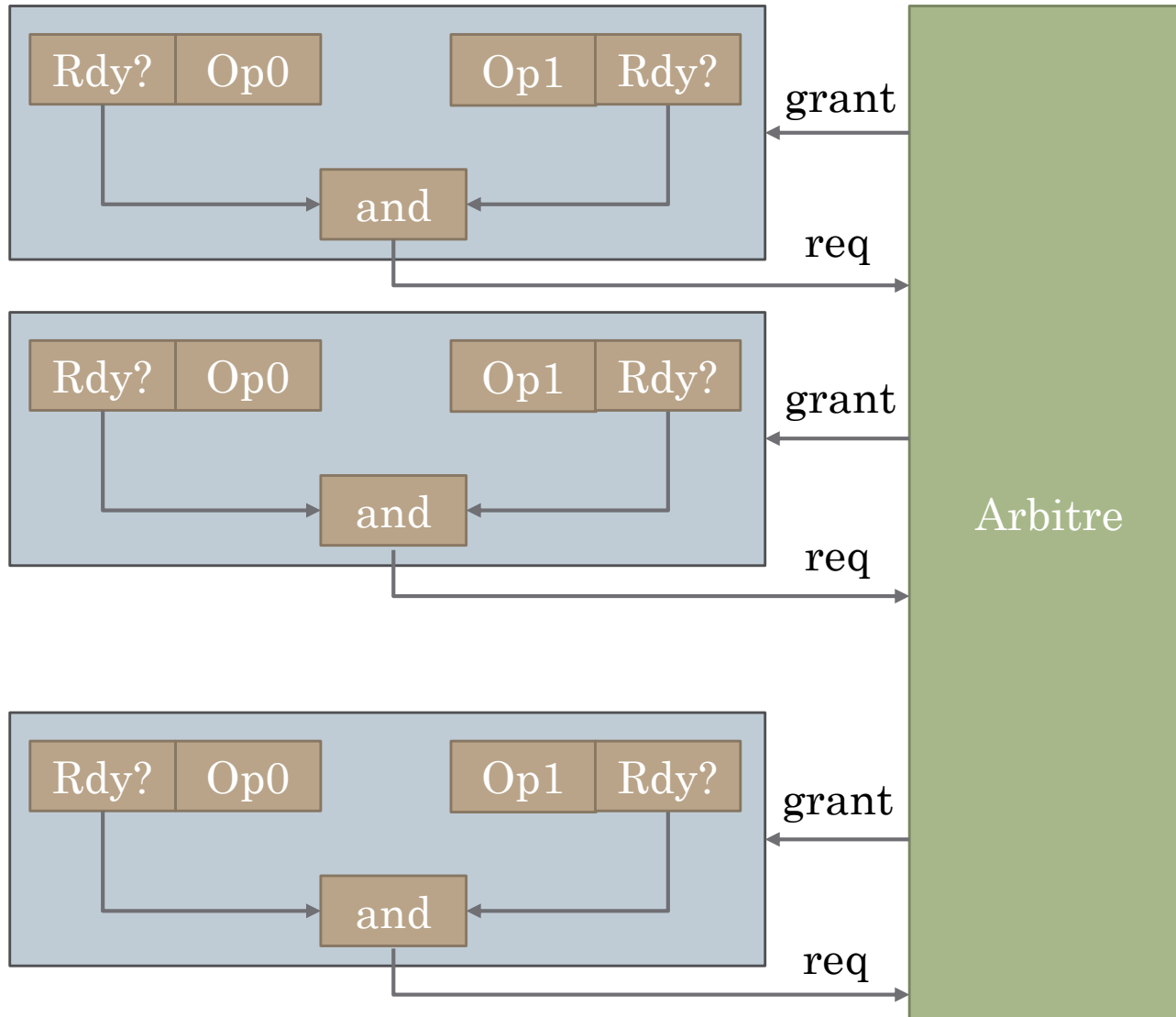
**Chaque cycle**, les reg. phy. Nouvellement prêts sont envoyés à **chaque entrée** de l'ordonnanceur

Chaque reg. est comparé à **chaque opérande source** et met à jour le ready bit correspondant

#Comparateurs =  
#SrcOpParInst \*  
#EntréesOrdo \*  
#RésultatsParCycle  
(ex. 2 \* 90 \* 6 = 1080)

Design **associatif** (CAM)

# Ordonnanceur – Select



**Chaque cycle**, les instructions prêtes envoient une requête à l'arbitre. L'arbitre sélectionne une ou plusieurs instructions à exécuter ("grant"), selon des critères :

- Structurels :
  - Unité fonctionnelle disponible
- Heuristique :
  - Instruction plus vieille d'abord
  - Load d'abord
  - etc.

Les instructions sélectionnées seront exécutées au cycle suivant, et diffuseront leur preg dest. pour réveiller les instructions dépendantes lors de la phase de *Wakeup* suivante.

# Exécution dans le désordre

Conclusion

# Une question de dépendances

- Un programme doit être exécutée en respectant un certains nombres de contraintes ou *dépendances*
  - Ces dépendances limitent grandement la performance d'une implémentation matérielle simple
- On a donc cherché à s'affranchir de certaines dépendances
  - Au niveau microarchitectural uniquement
  - On offre au logiciel une vue de l'état architectural correcte, càd qui respecte toutes les dépendances spécifiées par le jeu d'instructions



Technique matérielle	Etat microarchitectural potentiellement incorrect	Etat architectural rendu correct
Prédiction de branchement (dépendance de contrôle sur les branchements)	Instructions sur le mauvais chemin rentrent dans le pipeline	Détection d'une prédiction incorrecte et suppression des instructions incorrectes du pipeline
Superscalaire	—	—

Technique matérielle	Etat microarchitectural potentiellement incorrect	Etat architectural rendu correct
Prédiction de branchement (dépendance de contrôle sur les branchements)	Instructions sur le mauvais chemin rentrent dans le pipeline	Détection d'une prédiction incorrecte et suppression des instructions incorrectes du pipeline
Superscalaire	—	—
Exécution dans le désordre 1 (pas de renommage, WB dans l'ordre)	Non respect des dépendances RaW à cause des WaW/WaR Deadlock structurel	Exécuter dans le désordre seulement dans certains cas spécifiques (Pas de WaW/WaR ni deadlock structurel possible)



Technique matérielle	Etat microarchitectural potentiellement incorrect	Etat architectural rendu correct
Prédiction de branchement (dépendance de contrôle sur les branchements)	Instructions sur le mauvais chemin rentrent dans le pipeline	Détection d'une prédiction incorrecte et suppression des instructions incorrectes du pipeline
Superscalaire	—	—
Exécution dans le désordre 1 (pas de renommage, WB dans l'ordre)	Non respect des dépendances RaW à cause des WaW/WaR Deadlock structurel	Exécuter dans le désordre seulement dans certains cas spécifiques (Pas de WaW/WaR ni deadlock structurel possible)
Exécution dans le désordre 2 (renommage, WB dans le désordre)	Non respect de l'ordre du programme	Reorder Buffer (ROB) + Commit

Technique matérielle	Etat microarchitectural potentiellement incorrect	Etat architectural rendu correct
Prédiction de branchement (dépendance de contrôle sur les branchements)	Instructions sur le mauvais chemin rentrent dans le pipeline	Détection d'une prédiction incorrecte et suppression des instructions incorrectes du pipeline
Superscalaire	—	—
Exécution dans le désordre 1 (pas de renommage, WB dans l'ordre)	Non respect des dépendances RaW à cause des WaW/WaR Deadlock structurel	Exécuter dans le désordre seulement dans certains cas spécifiques (Pas de WaW/WaR ni deadlock structurel possible)
Exécution dans le désordre 2 (renommage, WB dans le désordre)	Non respect de l'ordre du programme	Reorder Buffer (ROB) + Commit
Exécution dans le désordre 3 (renommage, WB dans le désordre, Ordonnanceur)	Non respect de l'ordre du programme	Reorder Buffer (ROB) + Commit

Technique matérielle	Etat microarchitectural potentiellement incorrect	Etat architectural rendu correct
Prédiction de branchement (dépendance de contrôle sur les branchements)	Instructions sur le mauvais chemin rentrent dans le pipeline	Détection d'une prédiction incorrecte et suppression des instructions incorrectes du pipeline
Superscalaire	—	—
Exécution dans le désordre 1 (pas de renommage, WB dans l'ordre)	Non respect des dépendances RaW à cause des WaW/WaR Deadlock structurel	Exécuter dans le désordre seulement dans certains cas spécifiques (Pas de WaW/WaR ni deadlock structurel possible)
Exécution dans le désordre 2 (renommage, WB dans le désordre)	Non respect de l'ordre du programme	Reorder Buffer (ROB) + Commit
Exécution dans le désordre 3 (renommage, WB dans le désordre, Ordonnanceur)	Non respect de l'ordre du programme	Reorder Buffer (ROB) + Commit
Exécution dans le désordre 3 + Load/Store dans le désordre	Ecriture mémoire OoO Non respect des dépendances RaW Ld/St	SQ + écriture mémoire au Commit Store scanne LQ pour détecter RaW non respectées

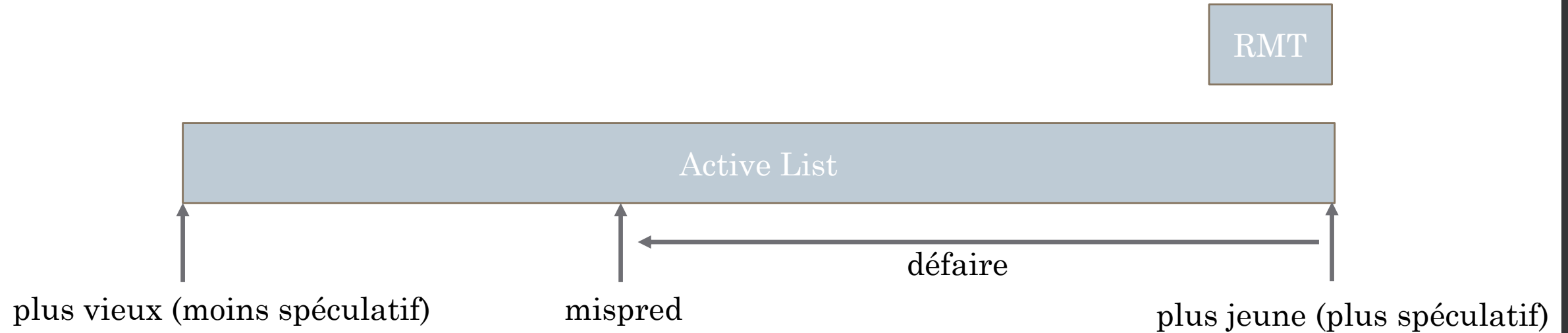
# Questions ?

## Exécution superscalaire Exécution dans le désordre

SEOC3A – CEAMC

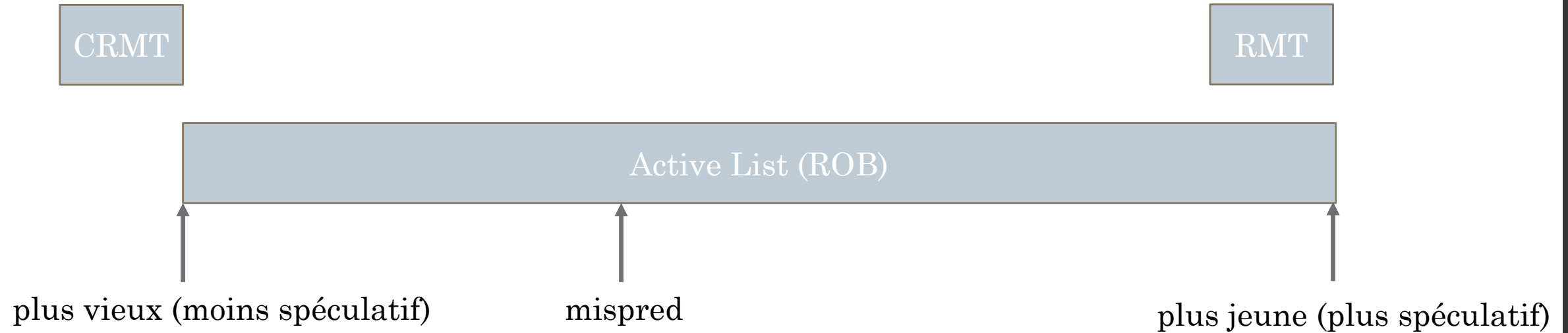
Arthur Perais (arthur.perais@univ-grenoble-alpes.fr)

# Renommage et vidage de pipeline

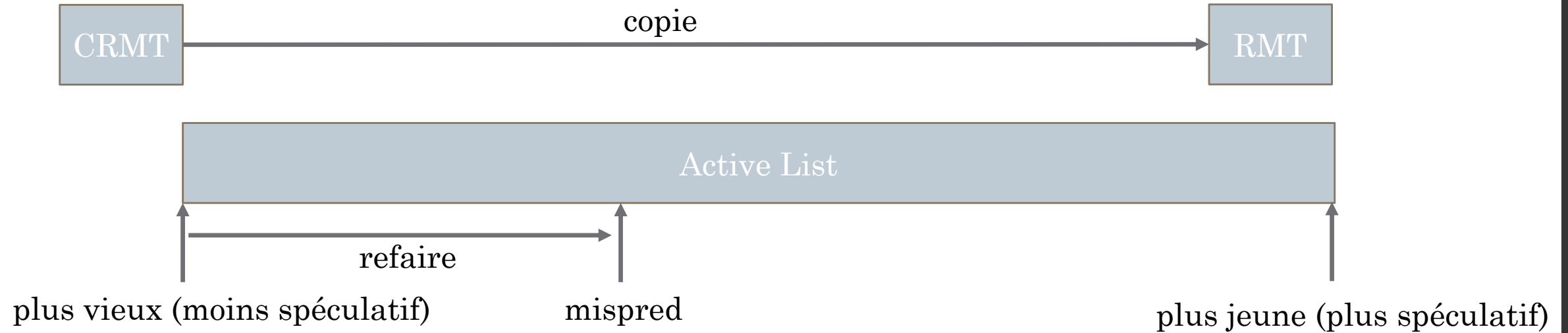


**Algorithme 1 (vu en exemple) :** Défaire les mappings contenus dans la Active List dans la RMT, de l'entrée la plus jeune vers la mauvaise prédiction. **Ne se sert pas de la CRMT.**

# Renommage de registres : Active List (AL)

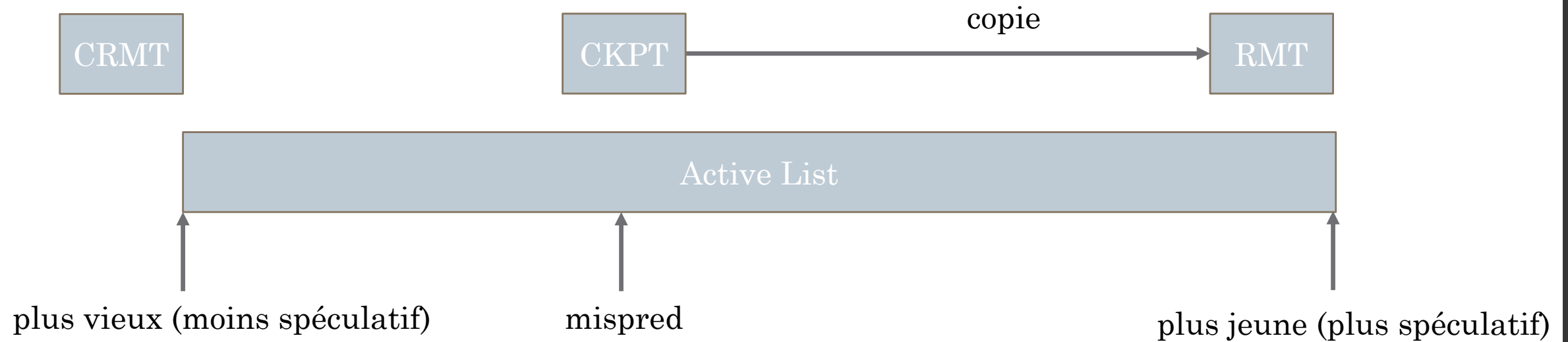


# Renommage et vidage de pipeline



**Algorithme 2** : Copier la Committed RMT (CRMT) dans la RMT, puis rejouer les mappings de la Active List de l'entrée la plus vieille vers la mauvaise prédiction. **Utilise la CRMT.**

# Renommage et vidage de pipeline



**Algorithme 3** : Copier la RMT dans un checkpoint matériel à chaque instruction pouvant causer un vidage de pipeline. Pour réparer la RMT, il suffit de copier le contenu du checkpoint dans la RMT. **Utilise la CRMT** si on voit la CRMT comme le plus vieux checkpoint.



# Renommage et vidage de pipeline

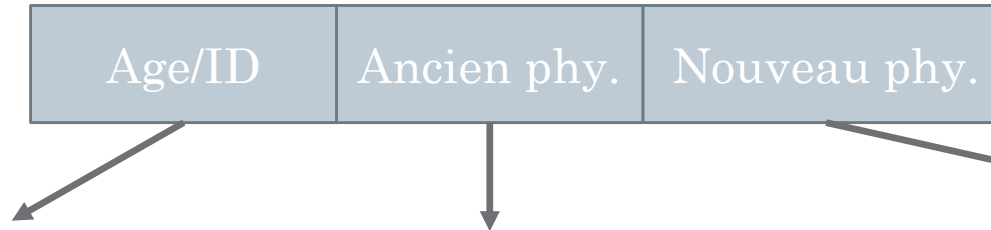
- En admettant qu'on peut traiter une instruction de la Active List par cycle, quel algorithme est le meilleur ?

# Renommage et vidage de pipeline

- En admettant qu'on peut traiter une instruction de la Active List par cycle, quel algorithme est le meilleur ?
- Difficile à dire car cela dépend de la distance moyenne entre la mauvaise prédiction et l'instruction la plus vieille/plus jeune. Dépend de la microarchitecture et du type de programme que l'on exécute.
- Qualitativement, le premier n'a pas besoin de la CRMT, le troisième est plus rapide (mécanisme non itératif) mais a besoin de beaucoup plus de matériel (beaucoup de checkpoints)

# Recyclage de registres physiques

- D'où provient le numéro de registre phy. à recycler?
  - N'est plus dans la RMT car une nouvelle version existe
  - Dans le ROB ? Mais l'instruction dont on recycle le registre phy. a été validée et a donc quittée le ROB
- Le numéro de registre à identifier vient de la Active List. Elle ne sert donc pas qu'à réparer la RMT, mais aussi à recycler les registres



Pour savoir si l'entrée est plus jeune/vieille qu'un événement causant un vidage de pipeline

Pour restaurer la RMT (algorithme 1) et remettre l'ancien registre phy sur la Free List au Commit

Pour restaurer la RMT (algorithme 2) et mettre à jour la CRMT (si présente) au Commit

# Renommage : Techniques avancées

- Le renommage de registres permet aussi d'améliorer la performance via deux techniques :
  - Registres câblés à la valeur  $x$  (*x-idiom elimination*)
  - Exécution des instructions *move* dans l'étage de renommage (*move elimination*)

# Renommage : *x-idiom elimination*

- Quel point commun entre :
  - *xor r0, x1, x1*
  - *mov r0, 0x0*
  - *and r0, r0, 0x0*

# Renommage : *x-idiom elimination*

- Quel point commun entre :
  - *xor r0, x1, x1*
  - *mov r0, 0x0*
  - *and r0, r0, 0x0*
- Le résultat est connu rien qu'en regardant l'instruction (ne dépend pas de la valeurs des registres sources), c'est toujours 0x0
- Idée :
  - Registre physique câblé à 0x0, par exemple le registre numéro 0
  - Le registre architectural de destination de toute instruction dont le résultat est garanti d'être 0x0 est renommé en registre physique 0

# Renommage : *x-idiom elimination*

- Plusieurs instructions « éliminées » peuvent avoir le registre physique 0 comme registre de destination. Mais les registres physiques sont « **single writer multiple readers** » ?
  - Pas un problème car le registre n'est en fait jamais écrit (car câblé à 0x0)
- Le registre physique 0 n'est jamais remis dans la Free List car il n'est jamais « libre » (car câblé à 0x0)
- On peut utiliser la même technique avec une autre valeur que 0x0, mais il y a assez peu d'instructions dont on connaît toujours le résultat avant de les avoir exécutées
- L'instruction est « exécutée » lors du renommage, mais doit toujours occuper une entrée dans le ROB et l'AL

# Renommage : *move elimination*

- En quoi consiste vraiment l'exécution de *move r0, x1* ?
  - Lire le registre physique correspondant à x1
  - Ecrire la valeur dans le registre physique alloué pour r0
- On peut juste allouer à r0 le registre physique qui correspond à x1...
- *move* « exécuté » au renommage, mais doit toujours occuper une entrée dans le ROB et l'AL
- Voyez-vous un autre problème ?



# Renommage : *move elimination*

- Le recyclage devient plus complexe
  - On ne peut recycler un registre que si tous les lecteurs ont lu le registre

## Sans ME

I1: add x1(pr1), r0(pr0), r0(pr0)  
I2: move x2(**pr2**), x1(pr1)  
I3: add x1(pr3), x4(pr5), x5(pr4)  
I4: add x3(pr6), x2(**pr2**), x2(**pr2**)

## Avec ME

I1: add x1(pr1), r0(pr0), r0(pr0)  
I2: move x2(**pr1**), x1(pr1)  
I3: add x1(pr3), x4(pr4), x5(pr5)  
I4: add x3(pr6), x2(**pr1**), x2(**pr1**)

- D'après l'algorithme de recyclage, on peut recycler pr1 une fois que I3 passe le Commit, car il redéfinit x1
  - Avec ME, I4 (plus jeune que I3) peut toujours lire pr1 !

# Renommage : *move elimination*

- Solution possible : comptage de référence, par registre physique
  - Incrémenté lorsqu'un registre physique est alloué à une instruction (inclut lors d'une élimination)
  - Décrémenté lorsque l'algorithme de recyclage « classique » aurait remis le registre dans la Free List
- On ne remet le registre dans la Free List que si le compteur vaut 0
- On doit aussi ajuster les compteurs lorsqu'on doit vider le pipeline par ex. à cause d'une mauvaise prédiction de branchement

# Dépendances mémoire – StAddr/StData

str r0, [x1 + x2]

- Un store a en général deux ou trois registres sources, repartis en deux catégories
  - Adresse
  - Donnée
- Du point de vue du calcul des dépendances mémoire, l'adresse est plus critique que la donnée
  - Il vaut mieux pouvoir identifier une dépendance quitte à ne pas pouvoir la satisfaire (adresse présente, donnée absente), plutôt que l'inverse
- Idée : diviser une instruction store en deux instructions microarchitecturales ( $\mu$ -op) : *store address* et *store data*

# Dépendances mémoire – StAddr/StData

- Les deux  $\mu$ -op partagent la même entrée dans la Store Queue. En général, partagent aussi l'entrée dans le ROB, mais ont chacune leur entrée dans l'ordonnanceur
- Avantages :
  - Résoudre les dépendances mémoire plus rapidement, et notamment éviter des cas où un load lit une mauvaise donnée
  - Permet d'éviter d'avoir une opération avec trois sources (peut participer à simplifier l'ordonnanceur)
- Désavantage
  - Consomme potentiellement plus de ressources (2 entrées dans l'ordonnanceur)

# Dépendances mémoire – StAddr/StData

## Hypothèses :

- x1 prêt, x5 prêt, x0 **non prêt**
- $(x5 + 42) == (x1 + 128)$  dans le L1D

### Sans StAddr/StData

```
str x0, [x1, #128]  
ldr x4, [x5, #42]
```

1. load s'exécute en premier, renvoie la mauvaise donnée depuis le L1D
2. store s'exécute lorsque x0 devient disponible, détermine que load a renvoyé la mauvaise donnée en scannant la LQ, et **vide donc le pipeline**

### Avec StAddr/StData

```
str x0, [x1, #128]  
ldr x4, [x5, #42]
```

1. stAddr s'exécute en premier et stocke l'adresse calculée dans la SQ
2. load s'exécute et constate qu'un store plus vieux stocke à la même adresse en scannant la SQ
3. stData s'exécute lorsque x0 devient disponible. Fournit directement la donnée au load, ou le load attend l'écriture dans le L1D. Dans les deux cas, **évite un vidage de pipeline**

# Dépendances mémoire – Speculative Memory Bypassing

- Retour aux optimisations rendues possibles par le renommage

```
add r0(pr0), x4(pr4), x5(pr5)    // def
str r0(pr0), [x1(pr1) + x2(pr2)] // store
...
ld x8(px8), [x1(pr1) + x2(pr2)] // load
sub x12(pr12), x8(px8), x5(pr5) // use
```



```
add r0(pr0), x4(pr4), x5(pr5)    // def
str r0(pr0), [x1(pr1) + x2(pr2)] // store
...
ld x8(px8), [x1(pr1) + x2(pr2)] // load
sub x12(pr12), x8(pr0), x5(pr5) // use
```

- Idée : Si on est capable de déterminer qu'un load charge la valeur d'un store dont le producteur est connu (ici, add), court-circuiter la chaîne de dépendance *def-st-ld-use* en *def-use*

# Dépendances mémoire – Speculative Memory Bypassing

- Permet de cacher la latence de la paire store-load
- Deux versions :
  - Non-spéculative : Si le store et le load utilisent les mêmes registres d'adresse et qu'on peut garantir qu'ils ne changent pas entre le store et le load
  - Spéculative : On prédit que l'adresse d'un load sera la même que l'adresse d'un store
- Dans les deux cas, nécessite des structures matérielles dédiées

Pour aller plus loin : Moshovos et al., "Speculative memory cloaking and bypassing."  
International Journal of Parallel Programming, 1999

# Ordonnanceur – Wakeup & Select

- L'ordonnanceur a deux missions :
  - *Wakeup* marque les instructions nouvellement prêtes à s'exécuter comme telles
  - *Select* sélectionne  $n$  instructions parmi  $p$  instructions prêtes et les envoie aux unités fonctionnelles ( $n = \#unités\ fonctionnelles$ )
- Note : Wakeup & Select doivent se faire en un seul cycle, sinon, pas d'exécution dos à dos (diminue la performance)





# Ordonnanceur - Scoreboard

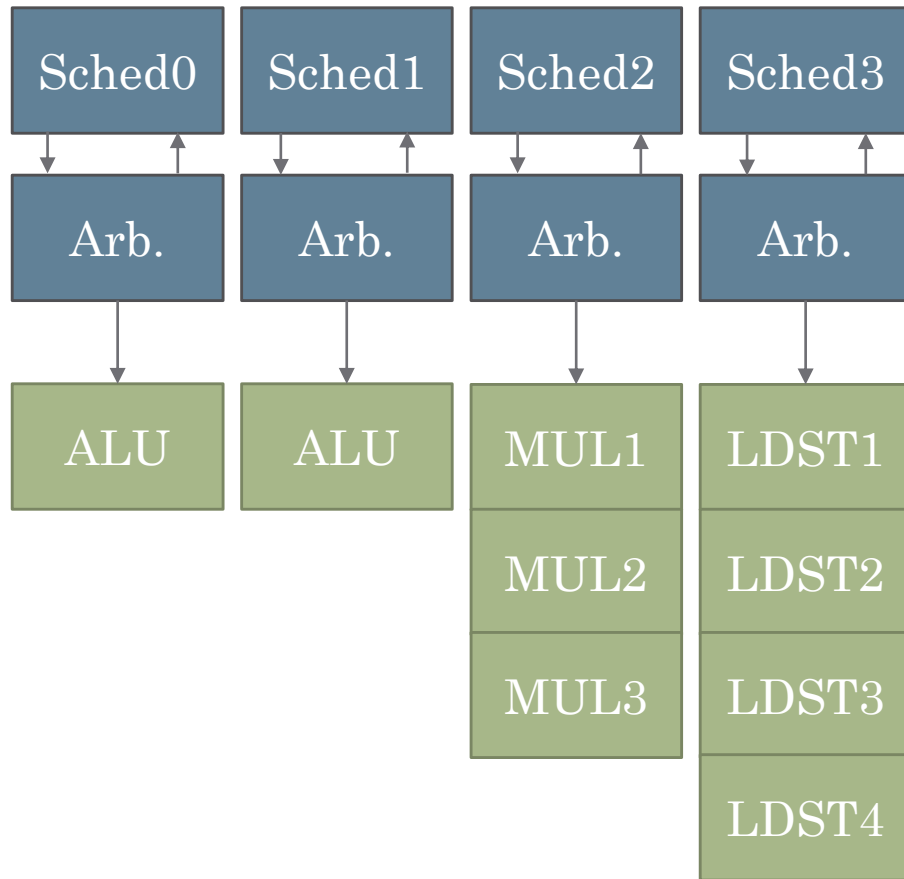
- Une fois dans l'ordonnanceur, une instruction « écoute » quels registres sont produits et devient prête si tous ses registres sources sont prêts
  - Et si les registres ont été produits **avant** que l'instruction ne rentre dans l'ordonnanceur ?
- **Scoreboard** (« tableau des scores »)
  - Un vecteur binaire : 1 bit par registre physique
  - Lorsque le registre phy. est produit à l'exécution, bit mis à 1
  - Lorsque le registre phy. est alloué (renommage de destination), bit mis à 0
  - Lu par l'instruction avant d'être inséré dans l'ordonnanceur (donne la valeur du bit « rdy » lors de l'insertion)

# Ordonnanceur classique - Considérations

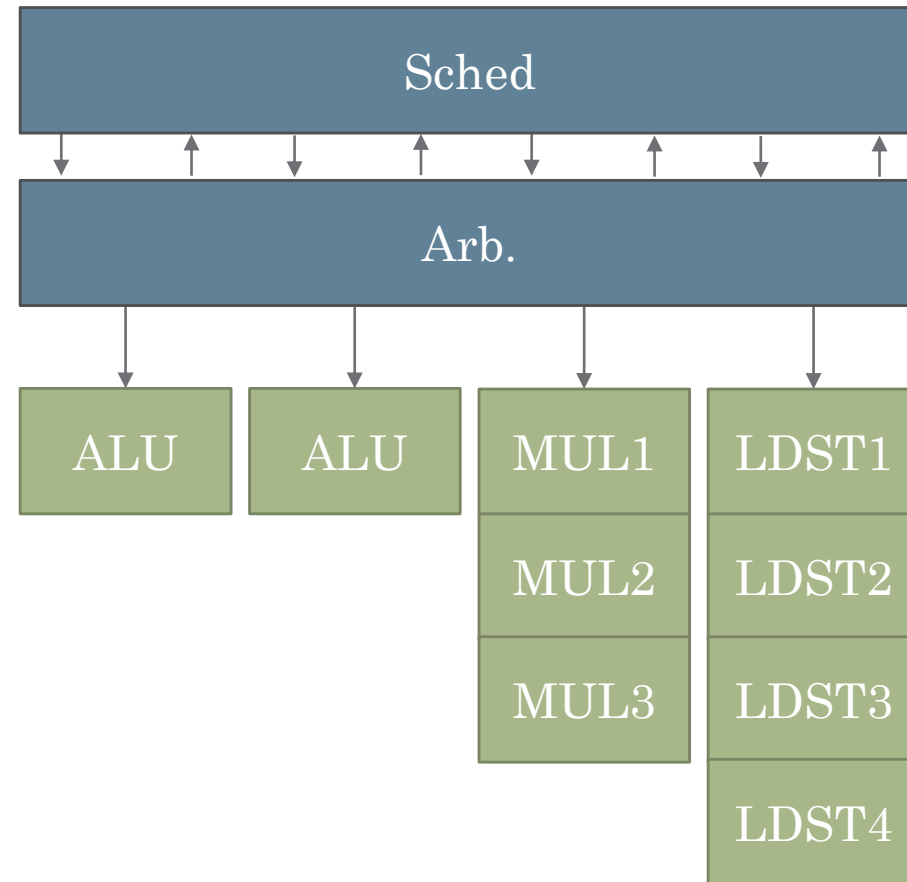
- Distribué ou unifié
  - Distribué : Un ordonnanceur attaché à chaque unité fonctionnelle (ou groupe de FUs) => Plusieurs ordonnanceurs
  - Unifié : Un ordonnanceur attaché à toutes les unités fonctionnelles => Un seul ordonnanceur

# Ordonnanceur classique - Considérations

Distribué (4 x 8 entrées)



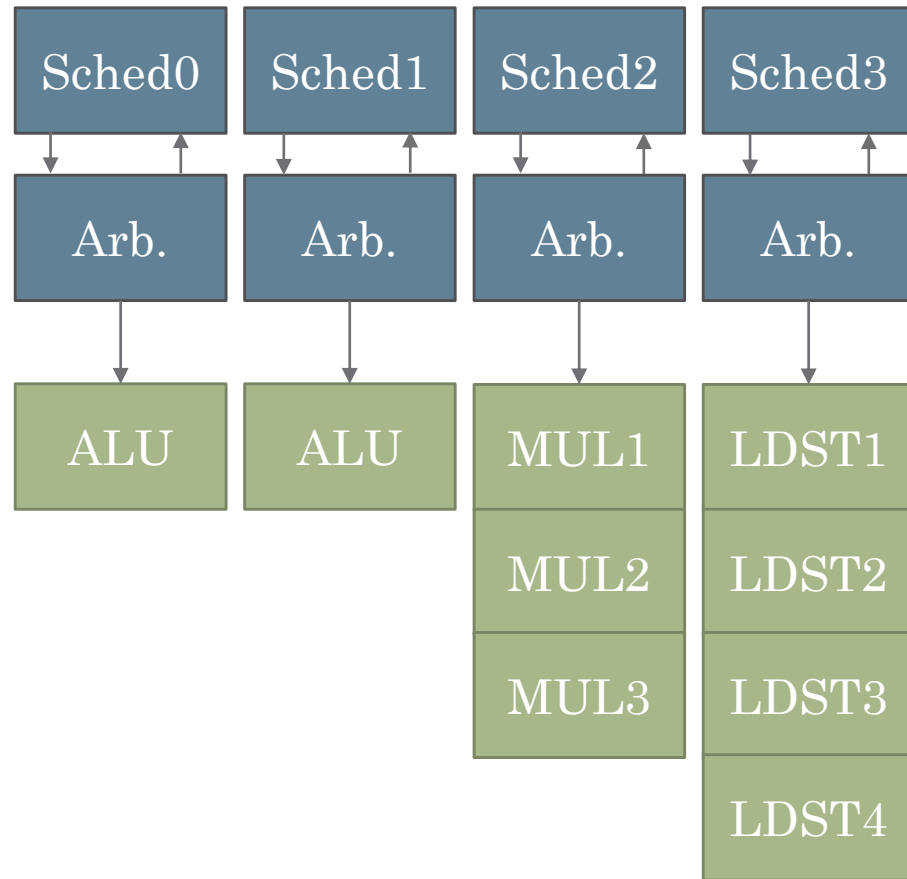
Unifié (32 entrées)



- Risque de déséquilibre (sched0 vide et sched 1 plein, max 8 ld/st en vol même si sched0/1/2 vides)
- Chaque arbitre doit sélectionner une inst/cycle = rapide
- Pas de déséquilibre (32 ld/st en vol)
- L'unique arbitre doit sélectionner 4 instructions à chaque cycle, avec des contraintes de catégorie (2ALU, 1MUL et 1LDST)

# Ordonnanceur classique - Considérations

Distribué (4 x 8 entrées)



Notion de **politique de steering** : L'étage de Dispatch (insertion dans l'ordonnanceur) doit potentiellement choisir entre plusieurs ordonnanceurs

Par exemple, si on veut ajouter un *add*, on peut l'insérer dans Sched0 ou dans Sched1. Comment choisir ?

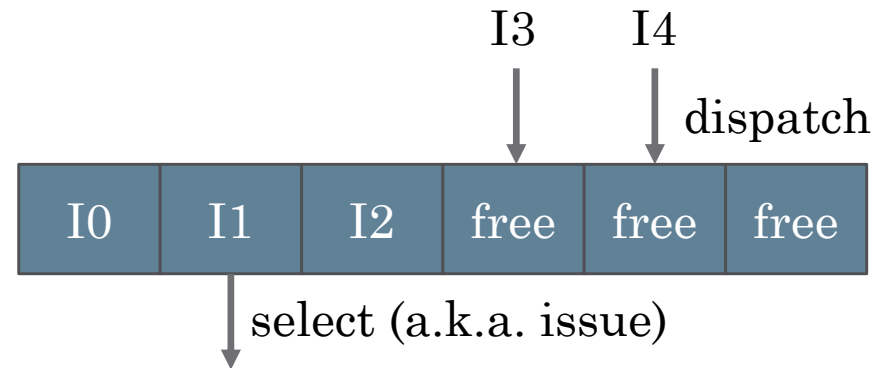
- En fonction du nombre d'entrées occupées ? Difficile d'avoir un nombre exact car au même cycle, une instruction peut aussi quitter un ordonnanceur pour s'exécuter
- En fonction des dépendances entre instructions ?

- Risque de déséquilibre (sched0 vide et sched 1 plein, max 8 ld/st en vol même si sched0/1/2 vides)
- Chaque arbitre doit sélectionner une inst/cycle = rapide

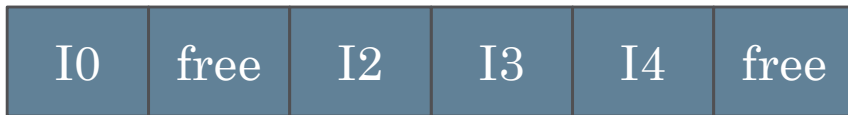
# Ordonnanceur classique - Considérations

- Distribué ou unifié
  - Distribué : Un ordonnanceur attaché à chaque unité fonctionnelle (ou groupe de FUs) => Plusieurs ordonnanceurs
  - Unifié : Un ordonnanceur attaché à toutes les unités fonctionnelles => Un seul ordonnanceur
- *Collapsing* ou non
  - Instructions ordonnées (âge) structurellement ou non

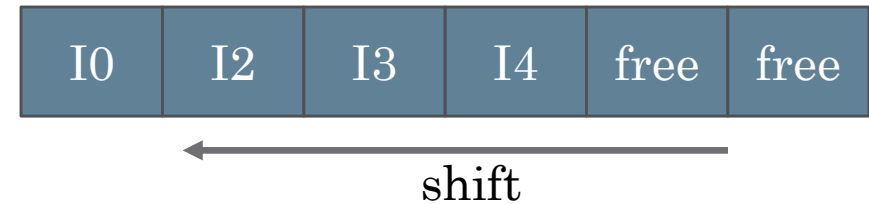
# Ordonnanceur classique - Considérations



**Non-collapsing**



**Collapsing**



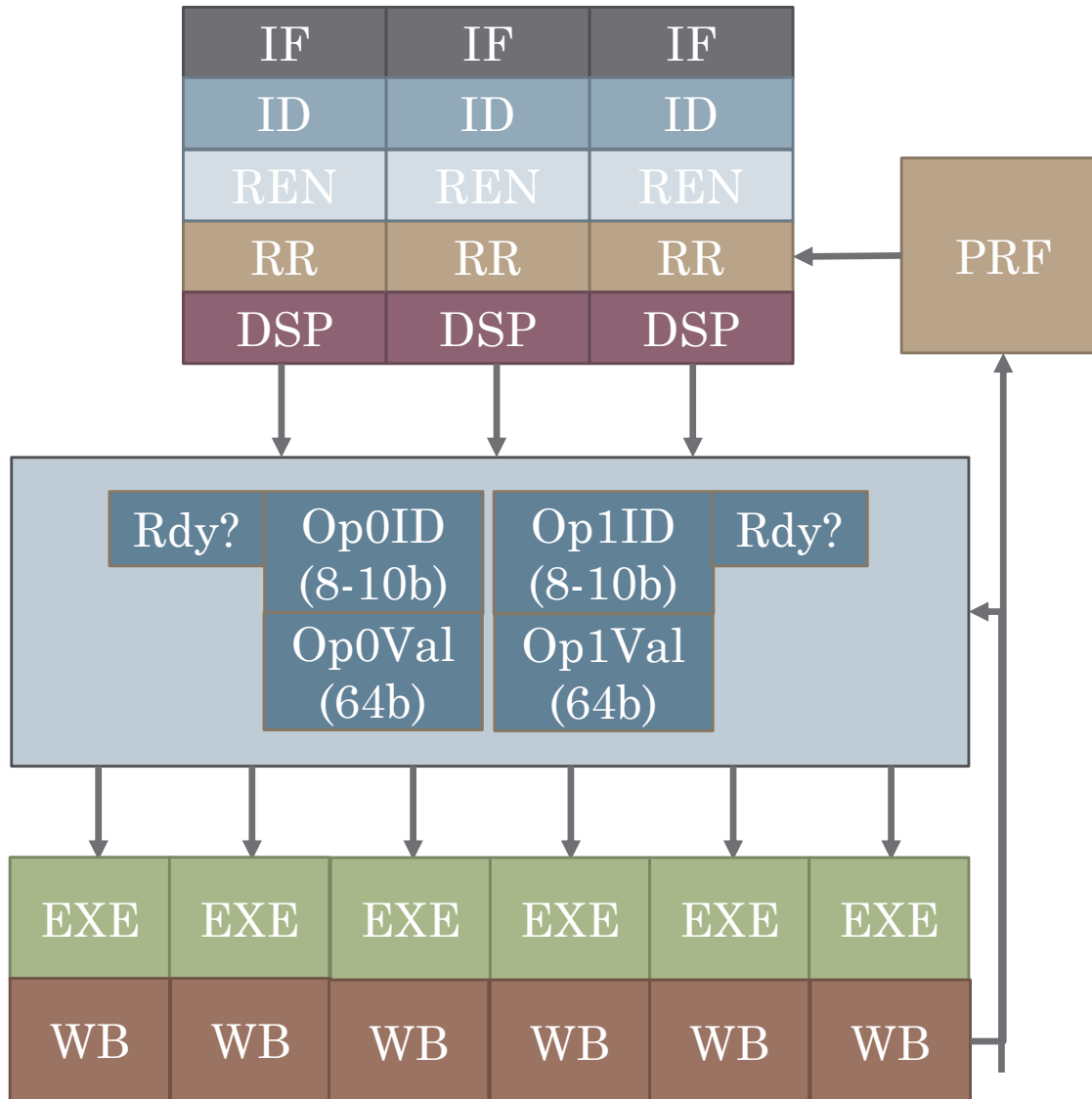
- Plus difficile à « nettoyer » lors d'un vidage de pipeline car l'âge de chaque entrée doit être vérifié

- Plus facile à « nettoyer » lors d'un vidage de pipeline : il suffit de trouver la première entrée plus vieille et d'enlever toutes les entrées à droite
- Décalage entre 0 et  $n$  à chaque cycle prend du temps et de l'énergie

# Ordonnanceur classique - Considérations

- Distribué ou unifié
  - Distribué : Un ordonnanceur attaché à chaque unité fonctionnelle (ou groupe de FUs) => Plusieurs ordonnanceurs
  - Unifié : Un ordonnanceur attaché à toutes les unités fonctionnelles => Un seul ordonnanceur
- *Collapsing* ou non
  - Instructions ordonnées (âge) structurellement ou non
- *Value-capture* ou non
  - Opérandes dans l'ordonnanceur, ou juste un pointeur

# Ordonnanceur classique - Considérations



## Value-capture

### Avantage :

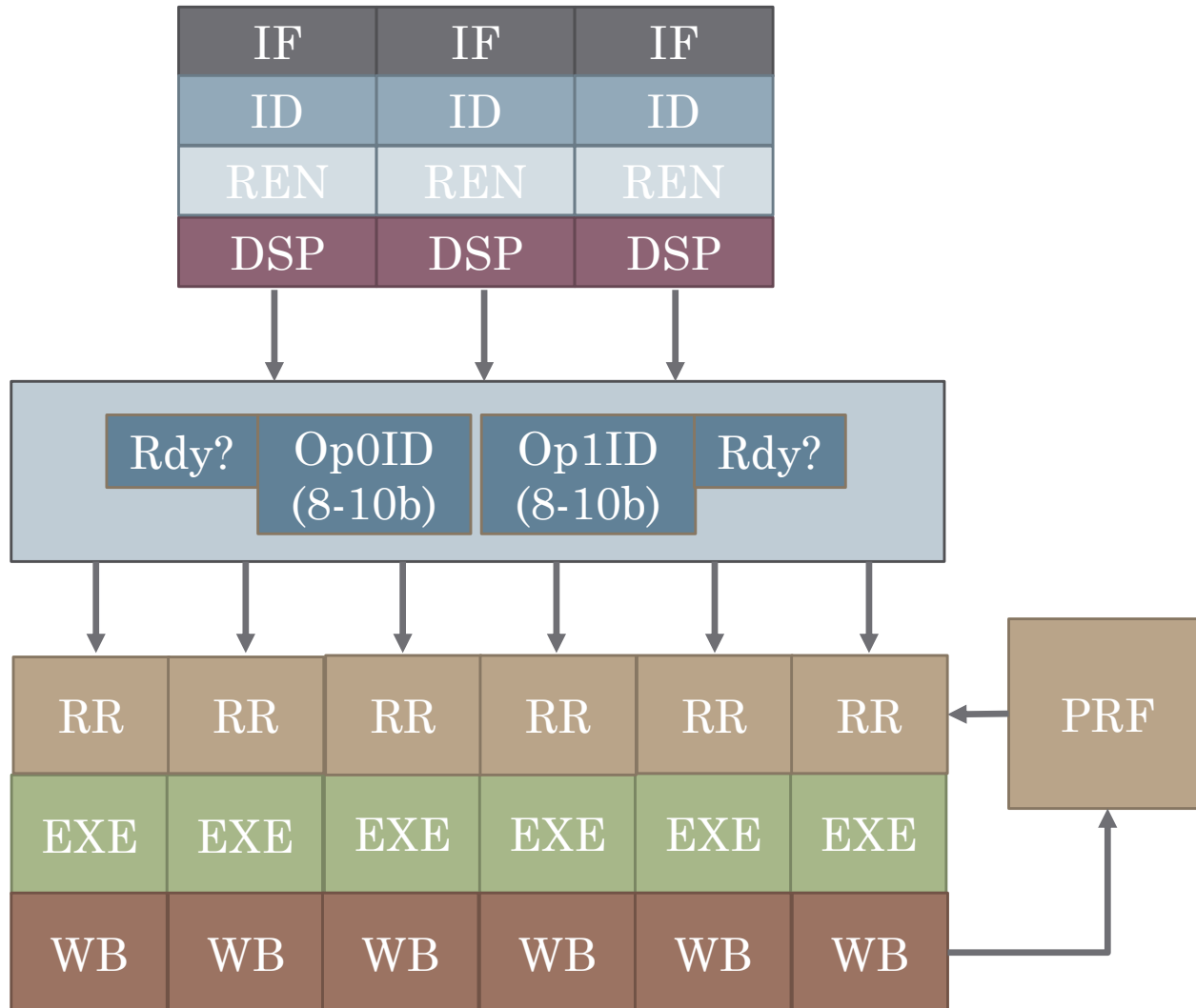
- Lecture du PRF avant l'ordonnanceur où le pipeline est en général moins large : moins de ports en lecture sur le PRF

### Désavantages :

- Chaque entrée est beaucoup plus grosse ce qui ajoute de la difficulté au routage des bus de broadcasts pour le wakeup
- Bus de broadcasts doivent amener le résultat (64b) en plus du tag (8-10b)



# Ordonnanceur classique - Considérations



## Non value-capture

### Avantages :

- Entrée du scheduler plus petite, donc réduit la longueur des fils (wakeup plus rapide)
- Pas besoin d'amener le résultat jusqu'au scheduler

### Désavantages :

- Lecture du PRF lorsque le pipeline est généralement plus large : plus de ports en lecture sur le PRF

# Aparté - Tomasulo

- Algorithme de Tomasulo (du nom de son inventeur) ancêtre de l'exécution dans le désordre « moderne »
- Idée similaire avec quelques différences. Basé sur des ordonnanceurs (« Reservation Stations ») qui contiennent des valeurs et du renommage de registres (mais avec autant de registres physiques que de registre architecturaux)
- [https://en.wikipedia.org/wiki/Tomasulo\\_algorithm](https://en.wikipedia.org/wiki/Tomasulo_algorithm) donne une bonne description du fonctionnement

# Ordonnanceur classique – Latences

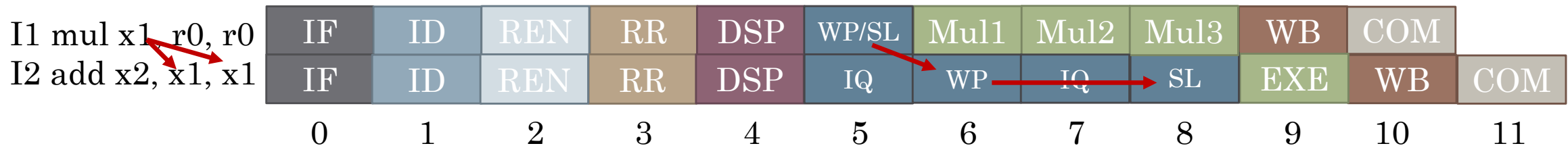
- Hypothèse : RR avant Select, instruction à latence fixe (1 cycle)



- Au cycle 5, on a sélectionné I1, donc on sait que I1 s'exécutera au cycle 6 et que le résultat de I1 sera disponible au cycle 7. Il faut donc réveiller et sélectionner I2 au cycle 6 afin que I2 soit dans EXE au cycle 7 (exécution dos à dos)

# Ordonnanceur classique – Latences

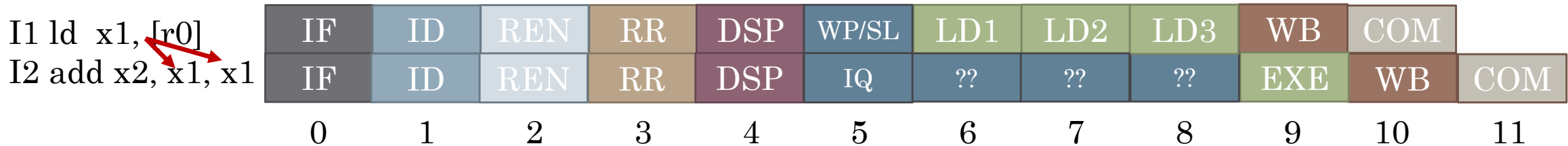
- Hypothèse : RR avant Select, instruction à latence fixe (3 cycles **exactement**)



- Au cycle 5, on a sélectionné I1, donc on sait que I1 s'exécutera au cycle 6/7/8 et que le résultat de I1 sera disponible au cycle 9. Il faut donc réveiller et sélectionner I2 au cycle 8 afin que I2 soit dans EXE au cycle 9 (exécution dos à dos). Pour ça, on peut faire le *Wakeup* de I2 au cycle 6, mais avoir un compteur dans l'entrée de l'ordonnanceur, afin d'ajouter un délai avant d'envoyer « req » à l'arbitre qui fait le *Select* (car la latence est **fixe**)

# Ordonnanceur classique – Latences

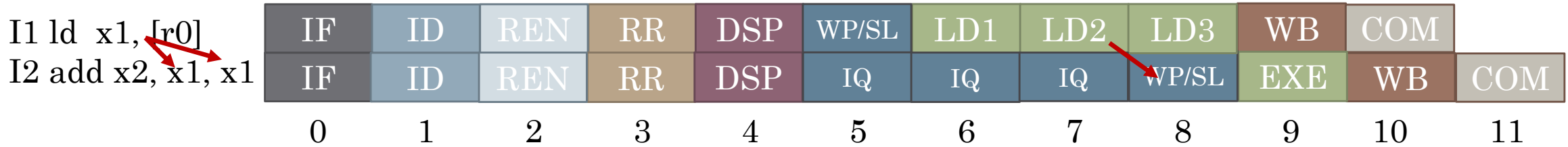
- Hypothèse : RR avant Select, instruction à latence variable (3 cycle **ou plus**)



- Au cycle 5, on a sélectionné I1, mais on ne sait pas à quel cycle le résultat sera disponible (dépend si I1 est un hit ou un miss dans le L1D)
- Si on a un hit, l'objectif est l'exécution dos à dos comme pour le mul de l'exemple précédent. Tout va dépendre du cycle auquel le pipeline peut garantir que
  - I1 est un hit
  - La donnée sera disponible au cycle 9

# Ordonnanceur classique – Latences

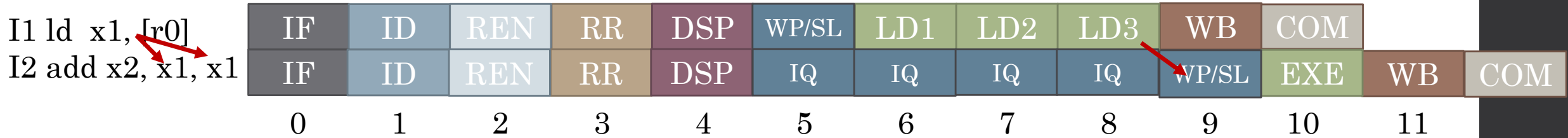
- Hypothèse : RR avant Select, instruction à latence variable (3 cycle **ou plus**), hit connu dans **LD2**



- Si on a ces garanties à LD2 (cycle 7), on peut faire *Wakeup & Select* de I2 au cycle 8, et I2 sera dans EXE au cycle 9
  - Exécution dos à dos

# Ordonnanceur classique – Latences

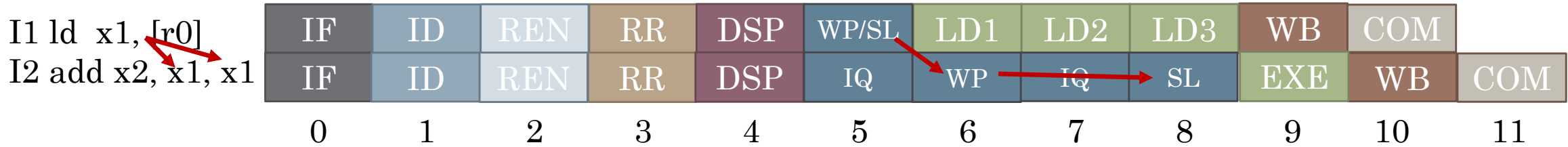
- Hypothèse : RR avant Select, instruction à latence variable (3 cycle **ou plus**), hit connu dans **LD3**



- Si on a ces garanties à LD3 (cycle 8), on peut faire *Wakeup & Select* de I2 au cycle 9, et I2 sera dans EXE au cycle 10
  - Pas d'exécution dos à dos mais une bulle
  - Le load-to-use est passé à 4 cycles alors que le temps d'accès au L1D pour un hit est de 3 cycles

# Ordonnanceur classique – Latences

- Hypothèse : RR avant Select, instruction à latence variable (3 cycles **ou plus**), hit connu dans **LD3**



- Il n'est pas évident de pouvoir garantir hit et donnée présente dans LD2, comment faire pour avoir de l'exécution dos à dos ?
  - Ordonnancement spéculatif : on spécule que I1 sera un hit, ce qui nous fait revenir dans le cas « latence fixe » (mul)
  - Mais si on a un miss ? On va exécuter I2 alors que la donnée ne sera pas présente : Il faut annuler I2 et la rejouer



# Ordonnancement spéculatif

- Spéculatif donc on peut se tromper. Comment réparer l'état microarchitectural ?
  - Vider le pipeline pour rejouer I2 ? Trop coûteux car les miss dans le L1D ne sont pas si rares
- On peut bloquer le pipeline et garder I2 dans EXE en attendant que le résultat de I1 soit disponible.
  - Verrouille un pipeline d'exécution pendant un temps indéterminé, et attention aux *deadlocks*
- *replay* : mécanisme pour marquer I2 comme « non prête » et refaire le *Wakeup & Select* lorsque le résultat de I1 sera disponible
  - Peut rapidement devenir complexe

# Ordonnanceur classique – Latences

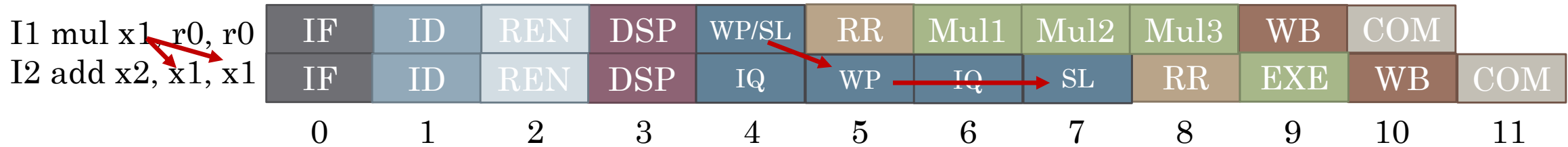
- Hypothèse : RR **après** Select, instruction à latence fixe (1 cycle)



- Pas de différence avec RR **avant** Select

# Ordonnanceur classique – Latences

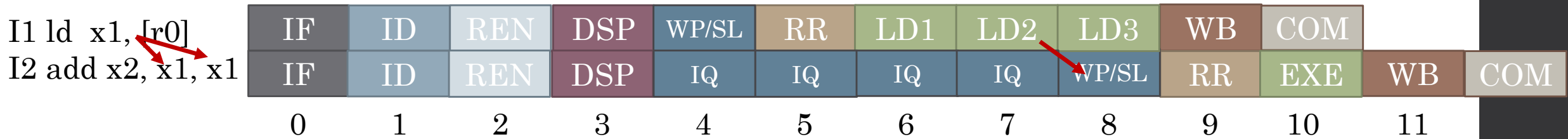
- Hypothèse : RR **après** Select, instruction à latence fixe (3 cycles **exactement**)



- Pas de différence avec RR **avant** Select

# Ordonnanceur classique – Latences

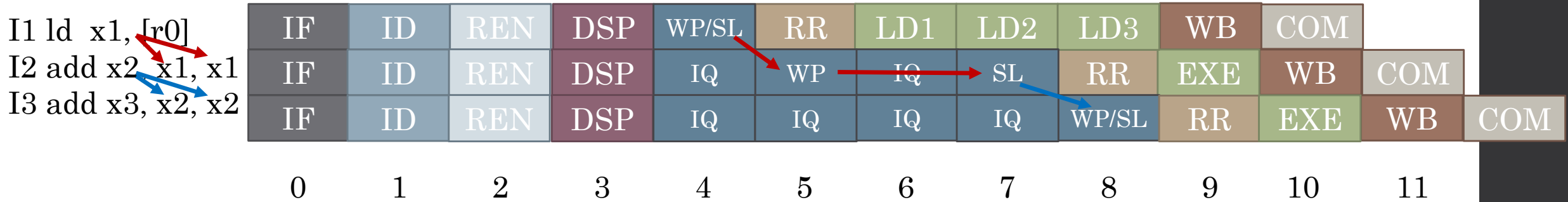
- Hypothèse : RR **après** Select, instruction à latence variable (3 cycles **ou plus**), hit connu dans **LD2**



- Même si on a la garantie d'avoir la donnée produite par I1 au cycle 9, le fait d'attendre le cycle 8 pour faire *Wakeup & Select* fait qu'on a maintenant une « bulle » à cause de RR
  - Exécution dos à dos plus possible même avec hit connu à LD2
  - Il faudrait avoir l'information à LD1 ! Impossible...

# Ordonnanceur classique – Latences

- Hypothèse : RR **après** Select, instruction à latence variable (3 cycle **ou plus**), hit connu dans **LD3**



- Même observation qu’avec LD2, on aurait une bulle en plus (donc 2)
- Si on fait de l’ordonnancement spéculatif, on doit annuler deux instructions (I2 et I3) lorsqu’on se rend compte qu’on a fait un miss (cycle 8) !

# Ordonnancement spéculatif

- Ajouter du délai entre Select et EXE exacerbe le problème des instructions à latence variable vis-à-vis de l'exécution des instructions
- Un mécanisme de *replay* devient nécessaire si on ne peut pas avoir la garantie d'avoir la donnée disponible assez tôt pour faire de l'ordonnancement non spéculatif
  - Ce mécanisme est complexe, car il faut immédiatement arrêter l'ordonnancement de la chaîne de dépendance quand on découvre qu'on a un miss
  - Pas si facile : voir l'article sur le Pentium IV <https://web.archive.org/web/20140408212527/http://www.xbitlabs.com/articles/cpu/display/replay.html> (on fait probablement mieux de nos jours)

# Ordonnanceur classique – Latences

- Hypothèse : RR **après** Select, instruction à latence variable (4 cycles **ou plus**), hit connu dans **LD2**



- Cependant, si le pipeline d'accès au L1D est assez long (ici 4 cycles), on peut compenser l'ajout d'un cycle entre Select et EXE
  - Exécution dos à dos à nouveau possible, mais avec load to use de 4 cycles. OK si il faut de toute façon 4 cycles pour accéder au L1D

# Ordonnanceur - Résumé

- Mémoire pour découpler le *frontend* dans l'ordre et le *backend* dans le désordre
- Implémente la logique nécessaire pour déterminer si
  1. Une instruction peut être exécutée
  2. Sélectionner les instructions à envoyer aux unités d'exécution
- Détails de design
  - Unifié ou distribué
  - Collapsing ou non
  - Value capture ou non
- Suivant le design du pipeline d'accès au L1D et la position de l'accès au PRF, ordonnancement spéculatif pour permettre l'exécution dos à dos des accès mémoires