

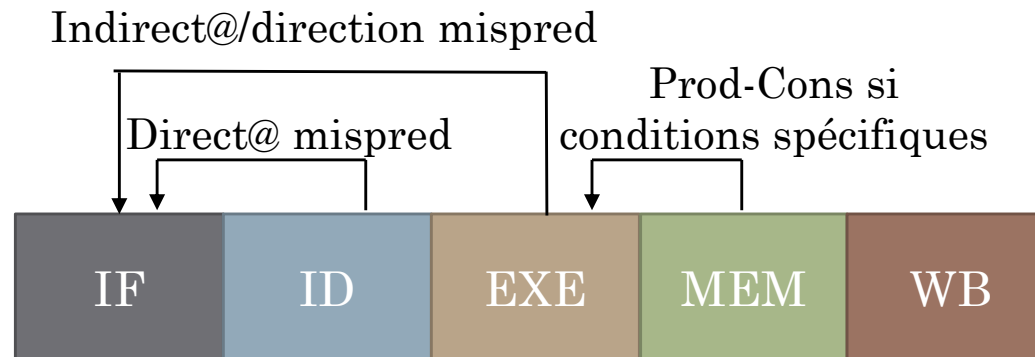
# Exécution superscalaire Exécution dans le désordre

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

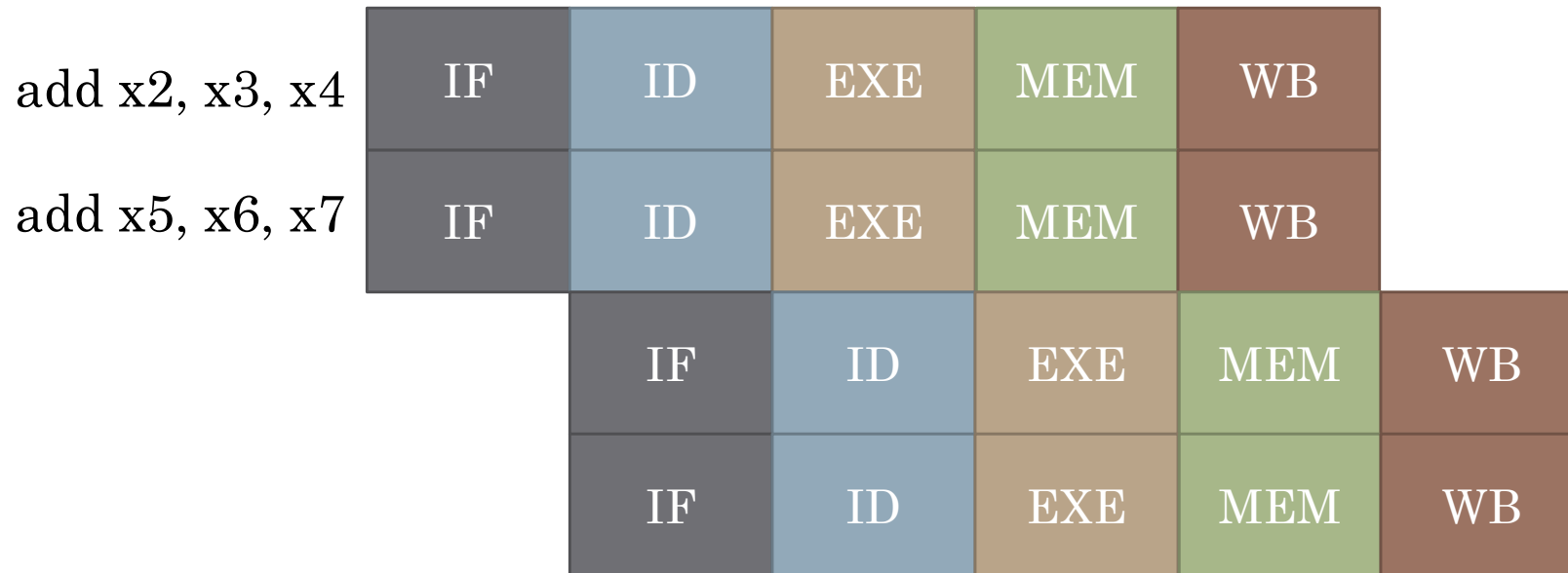
- Dupliquer les ressources pour tirer parti du parallélisme d'instruction (ILP)

```
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)



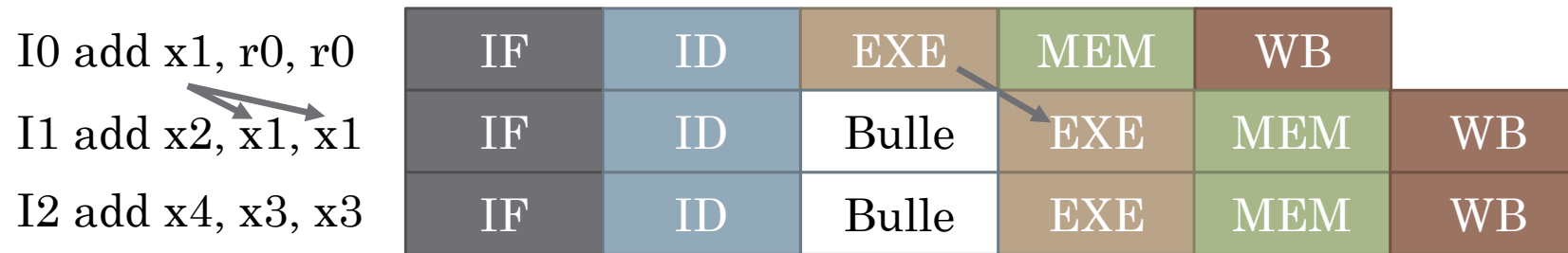
« superscalaire degré 2 »

Latence : 5 CPI

Débit : **2 IPC**

# Pipeline superscalaire

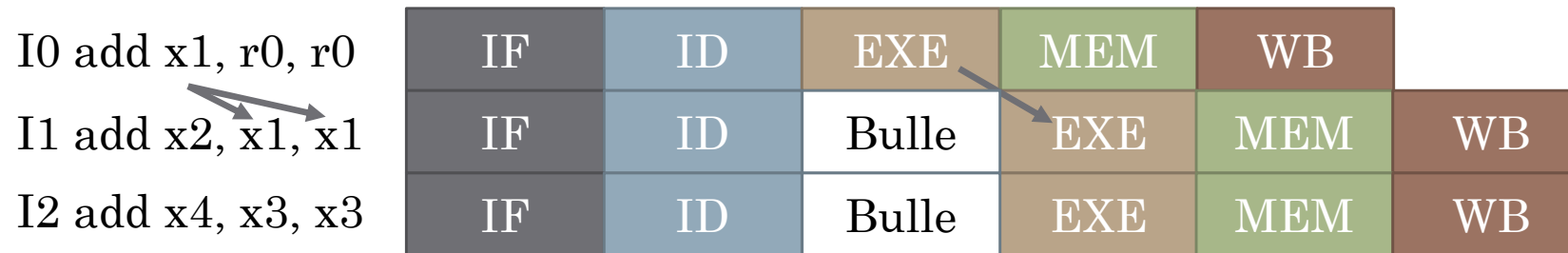
- Degré 4, 8, 16 mais...
  - Dès qu'une instruction bloque, tout le monde attend





# 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

Exercice :

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

I0 add x1, r0, r0

I1 add x2, x1, x1

I2 add x4, x3, x3

I3 ld x8, 0(x10)

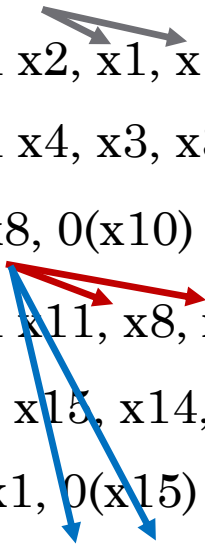
I4 add x11, x8, x8

I5 sub x15, x14, x14

I6 sd x1, 0(x15)

I7 sll x9, x8, x8

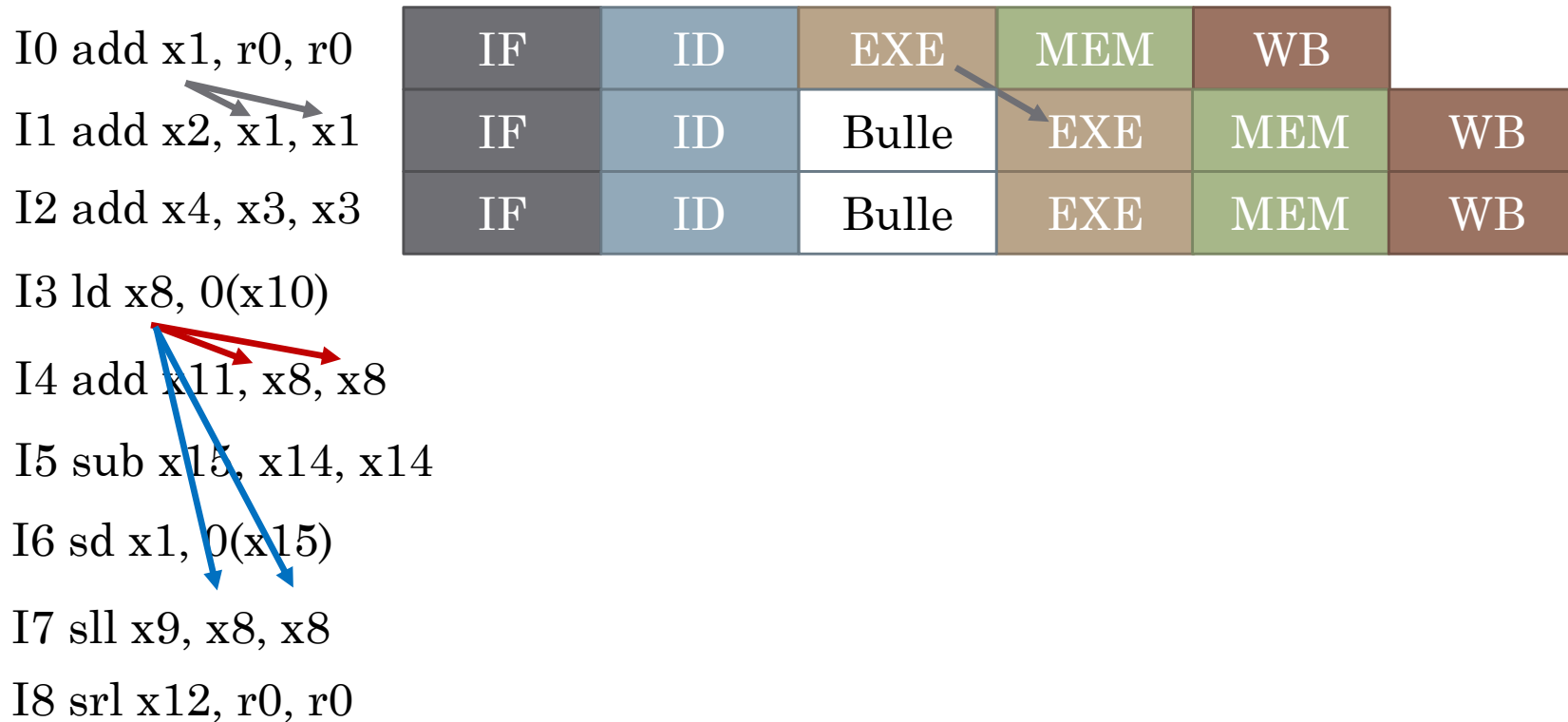
I8 srl x12, r0, r0



# Pipeline superscalaire

Exercice :

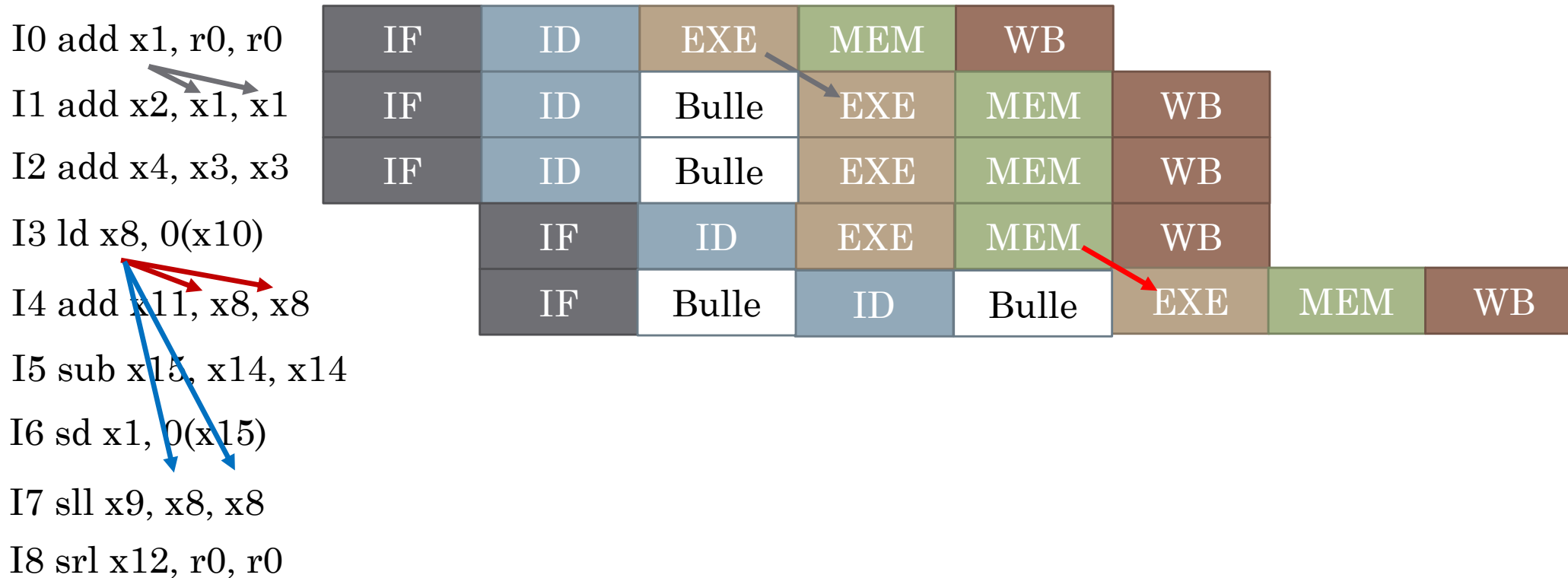
- **Quelle exécution en pipeline pour le code suivant ?**
- Dépendance(s) structurelle(s) vs. architecturale(s) ?



# Pipeline superscalaire

Exercice :

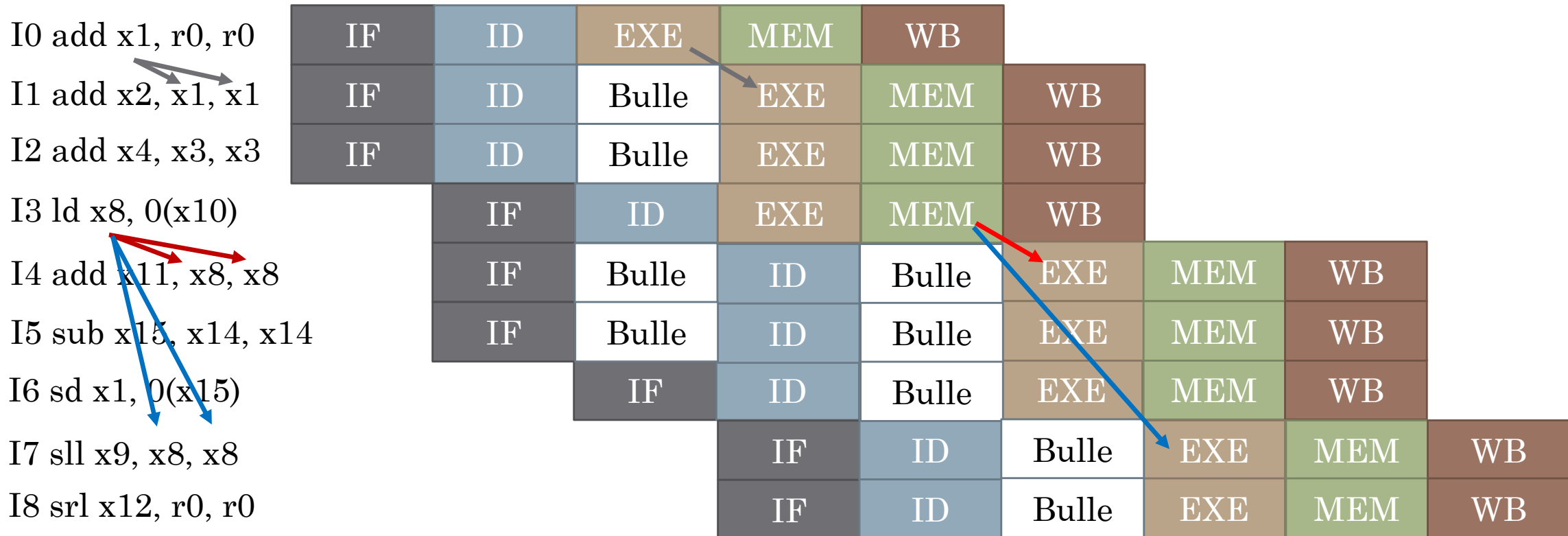
- Quelle exécution en pipeline pour le code suivant ?
- Dépendance(s) structurelle(s) vs. architecturale(s) ?



# Pipeline superscalaire

Exercice :

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

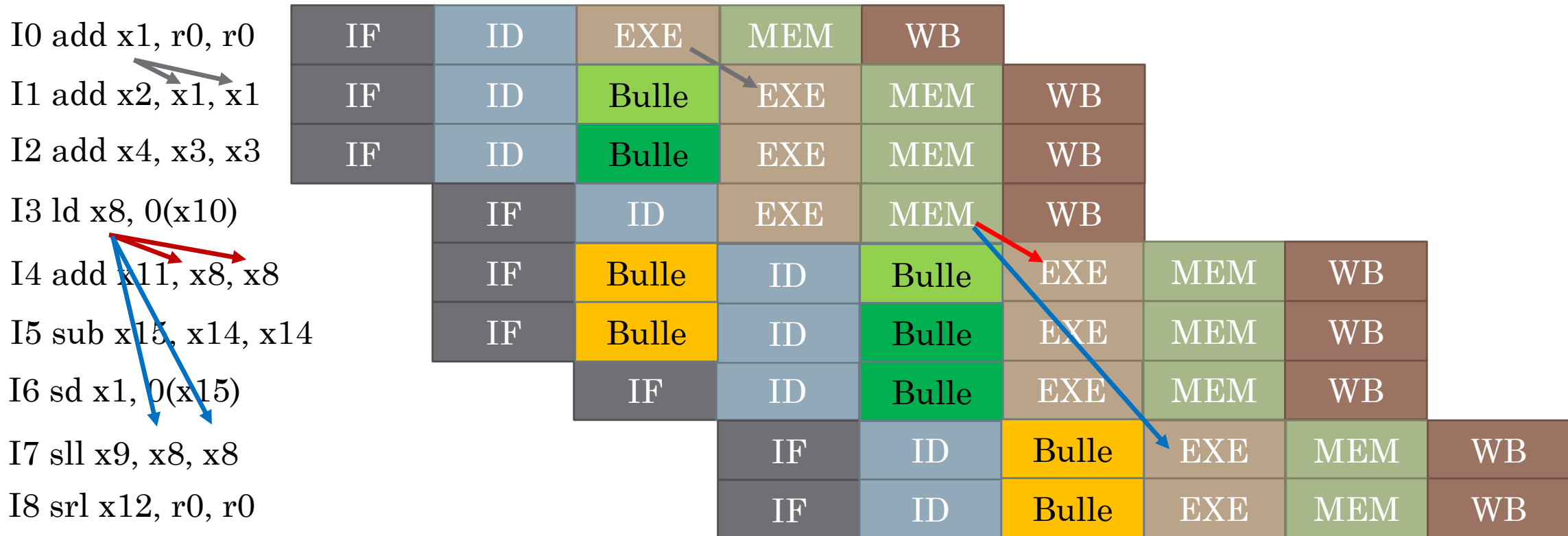


# Pipeline superscalaire

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

Exercice :

- 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, r0, r0

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 *dégré superscalaire* instructions peuvent être traitées par un étage lors d'un cycle donnée

I1 addu x2, x1, x1

I2 addu x4, x3, x3

I3 ld x8, 0(x10)

I4 addu x11, x8, x8

I5 subu 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 : performance crête : 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, r0, r0

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, r0, r0

I1 add x2, x1, x1

I2 add x4, x3, x3

I3 ld x8, 0(x10)

I4 add x11, x8, x8

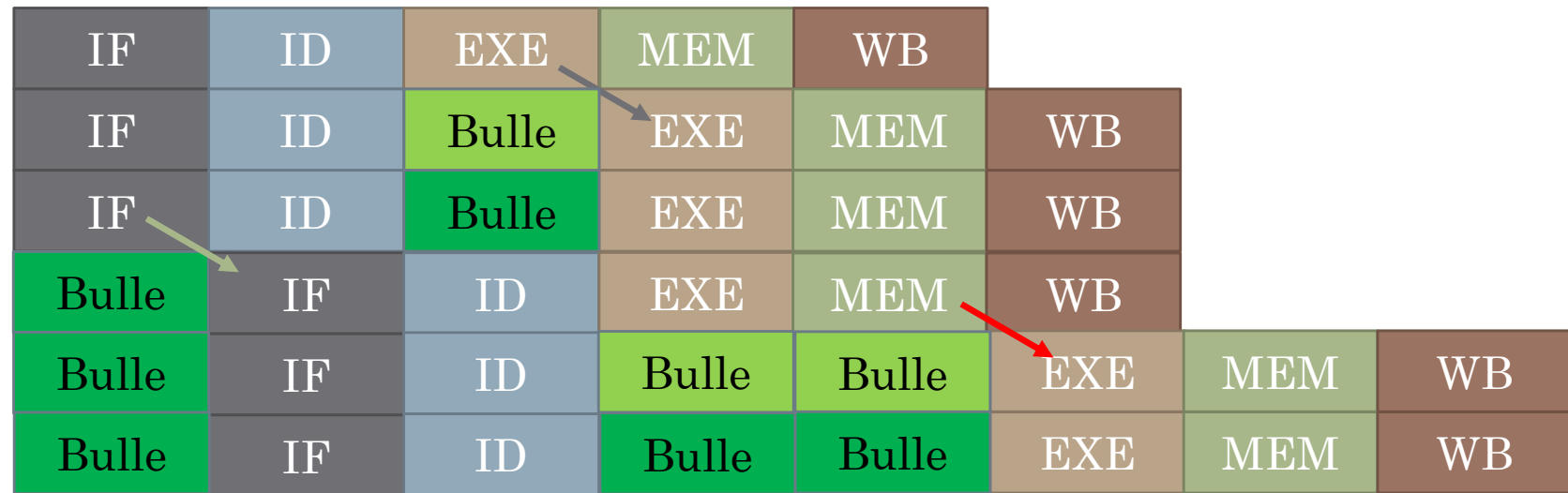
I5 sub x15, x14, x14

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 addu x1, r0, r0  
I1 addu x2, x1, x1  
I2 bnez x4, label  
I3 ld x8, 0(x10)  
I4 addu x11, x8, x8  
I5 subu 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 minimiser 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, r0, r0

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(x15)

I7 sll x9, x8, x8

I8 srl x12, r0, r0

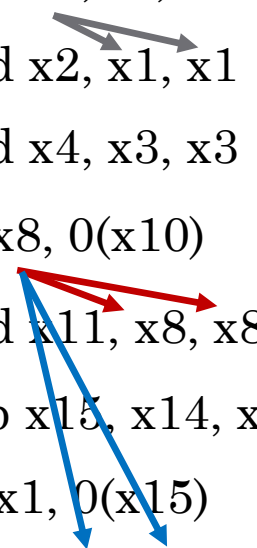
Si on observe l'état architectural après l'exécution d'une instruction Ix, alors :

- 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, r0, r0  
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(x15)  
I7 sll x9, x8, x8  
I8 srl x12, r0, r0



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

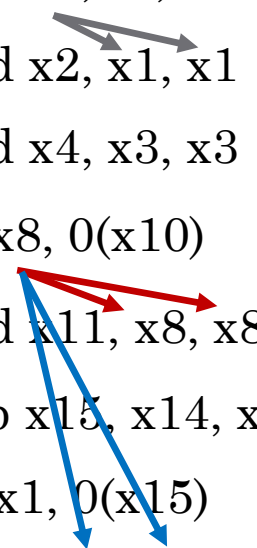
Exemple si on observe l'état arch. après I4 :

- x1 contient  $r0 + r0$  (I0), x2 contient  $x1 + x1$  (I1), x3 contient  $x3 + x3$  (I3), x8 contient  $[x10 + 0]$  (I4)
- x11 ne contient pas encore  $x8 + x8$  (I5), etc.

# Dépendances de contrôle

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

I0 add x1, r0, r0  
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(x15)  
I7 sll x9, x8, x8  
I8 srl x12, r0, r0

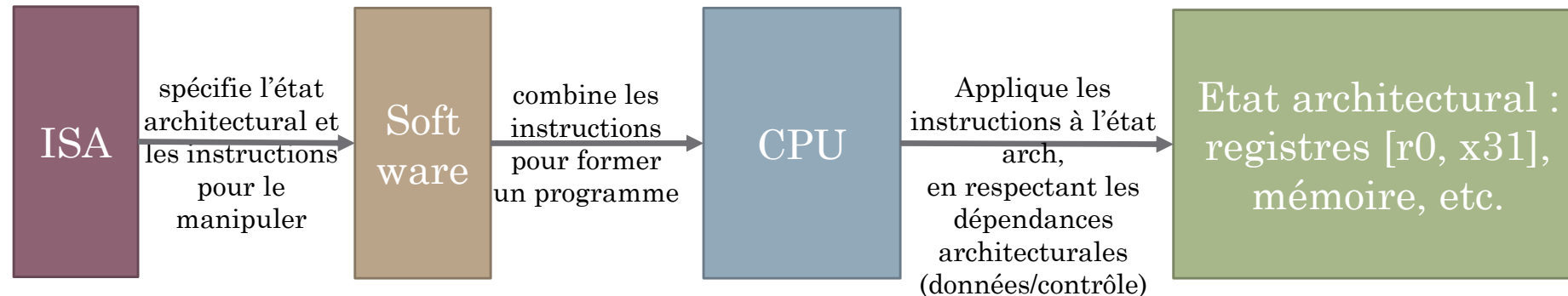


Si on observe l'état architectural après l'exécution d'une instruction Ix, alors :

- 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

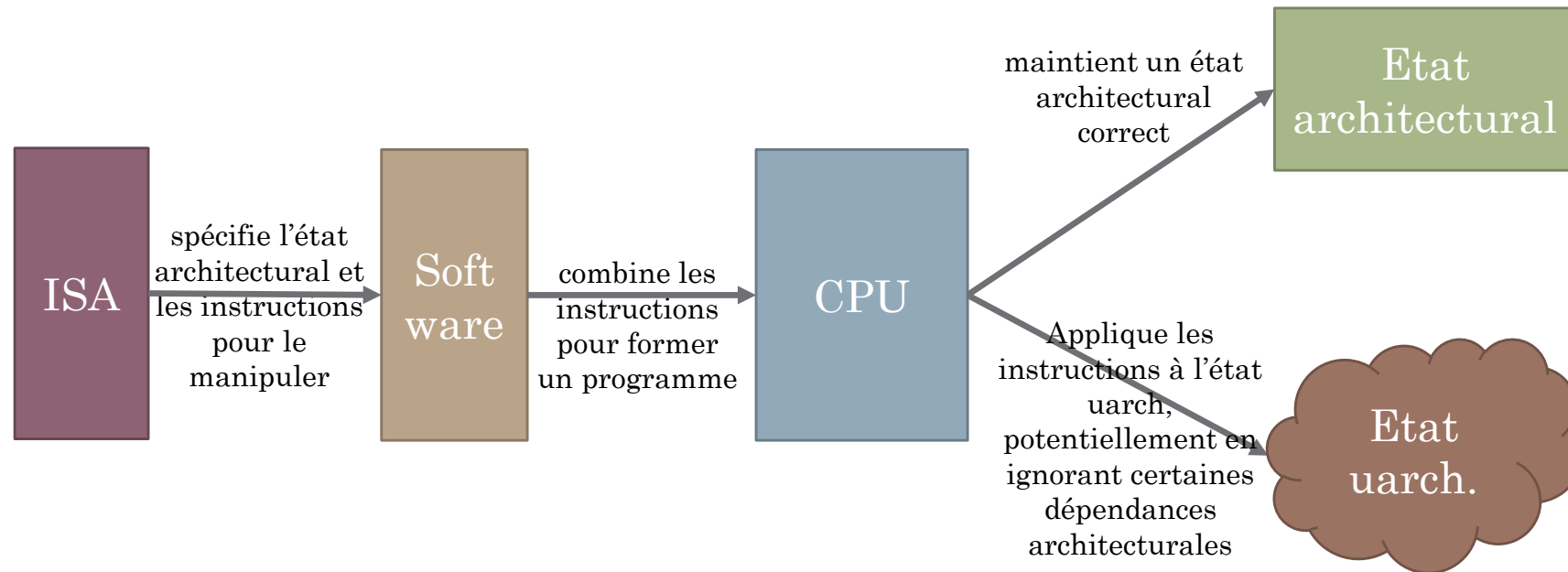
- Etat architectural observable et manipulable par le logiciel
  - Manipulations contraintes par la spécification (notamment ordre du programme)





# 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

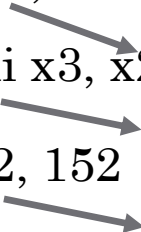
Concept

# 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

Exercice : Superscalaire degré 4. Quelle exécution ? Exécution correcte ?

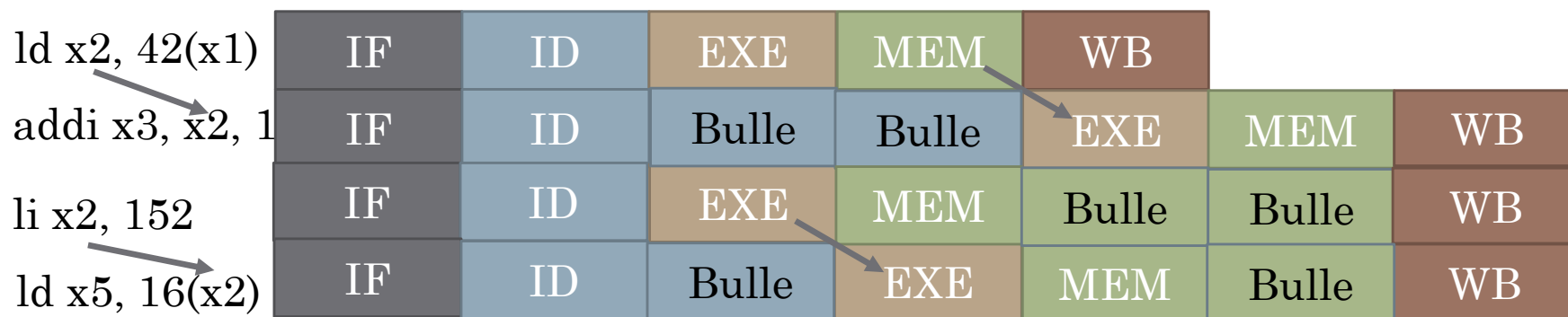
ld x2, 42  
addi x3, x2, 1  
li x2, 152  
ld x5, 16(x3)



# 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

Exercice : Superscalaire degré 4. Quelle exécution ? Exécution correcte ?



Etat arch.

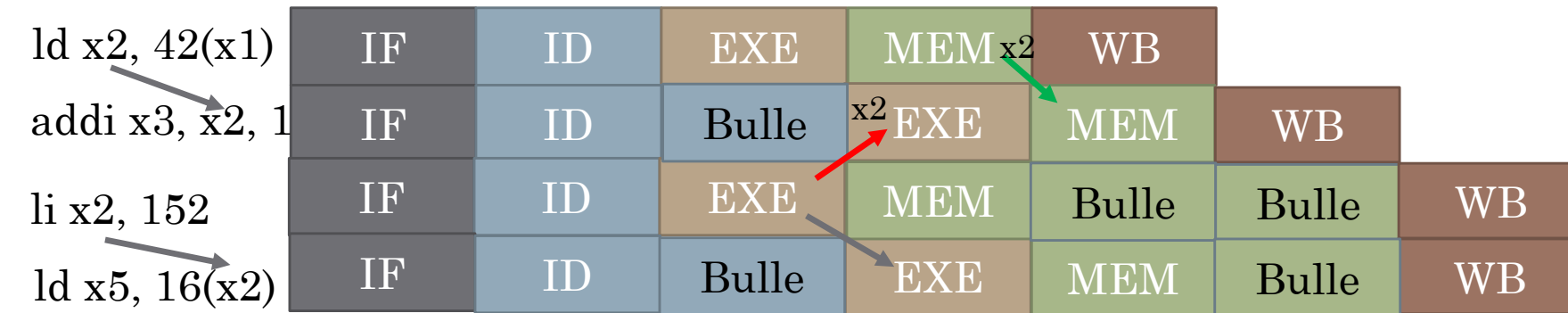
- x2 contient  $[x1 + 42]$

Etat uarch.

- Reg pipeline MEM[2] contient 152
- Reg pipeline MEM[3] contient  $[152 + 16]$

# Exécution dans le désordre

- Problème : x2 produit par *li x2, 152* présent sur le bypass avant x2 de
  - *ld x3, 16(x2)* 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)

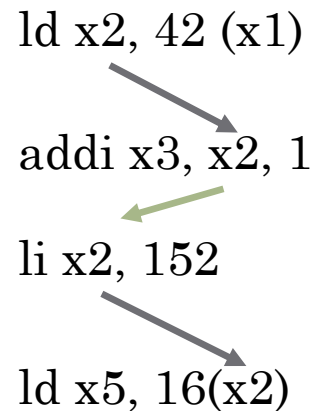
addi x3, x2, 1

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*) 
- Avec exécution dans le désordre, autres dépendances à respecter
  - consommateur-producteur (*Write-after-Read, WaR*) 

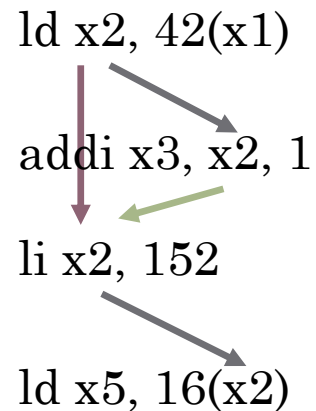


```
ld x2, 42 (x1)
addi x3, x2, 1
li x2, 152
ld x5, 16(x2)
```



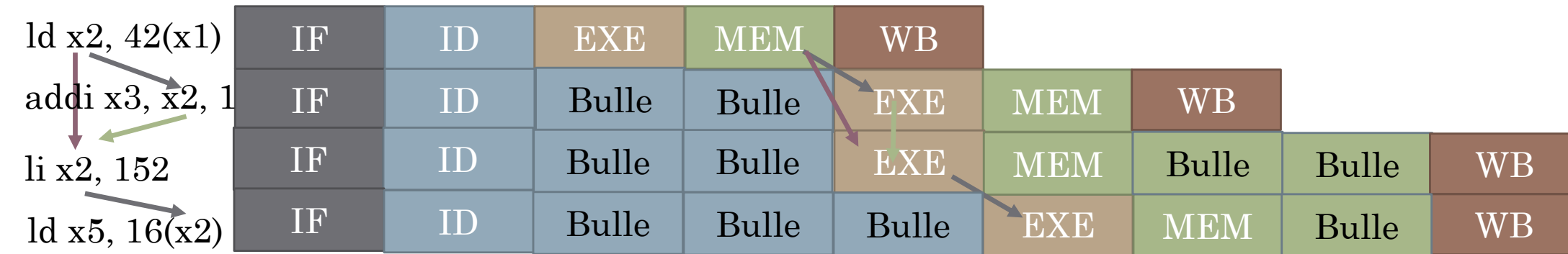
# Exécution dans le désordre

- 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



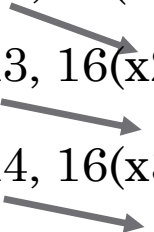
- Aucune exécution dans le désordre...

# 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

Exercice : Superscalaire degré 4. Quelle exécution ? Exécution correcte ?

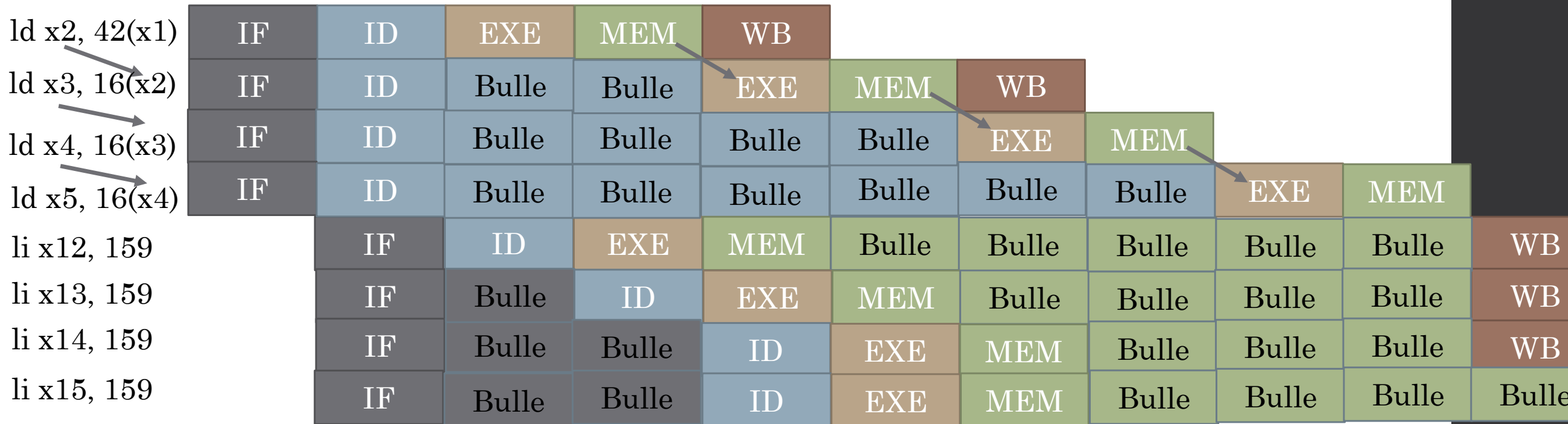
```
ld x2, 42(x1)
ld x3, 16(x2)
ld x4, 16(x3)
ld x5, 16(x4)
li x12, 159
li x13, 159
li x14, 159
li x15, 159
```



# Exécution dans le désordre

- 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

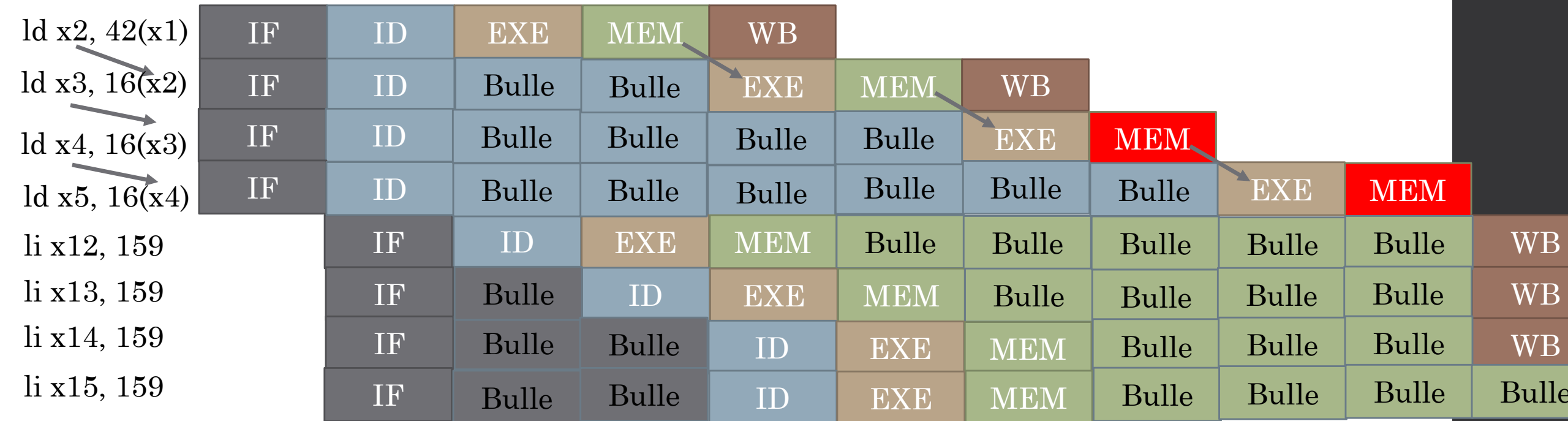
Exercice : Superscalaire degré 4, quelle exécution ? Exécution correcte ?



# Exécution dans le désordre

- **Deadlock matériel :**

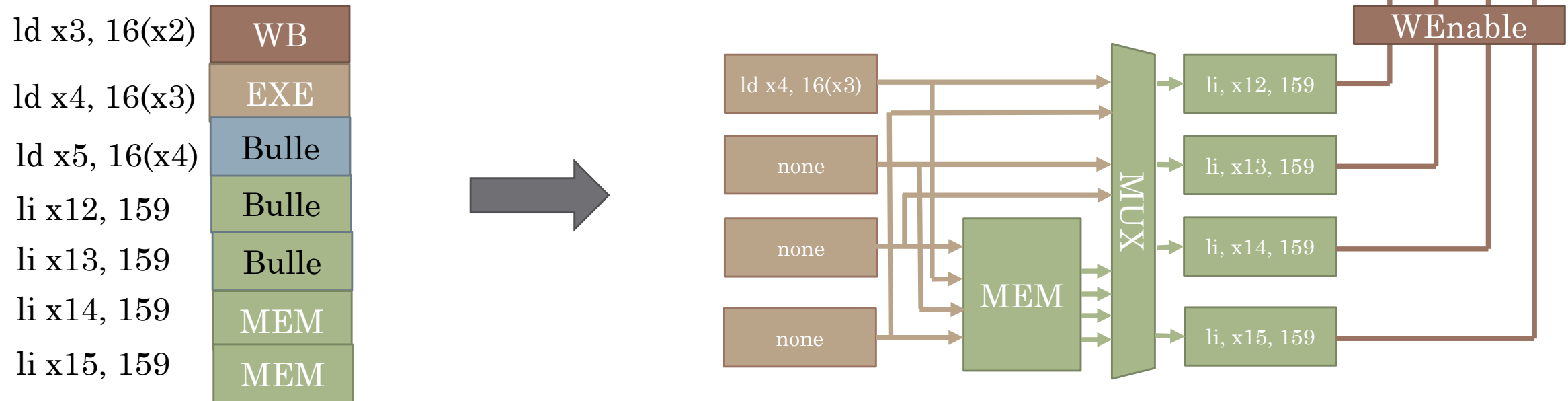
- Instructions plus jeunes attendent que ld x4, 16(x3) quitte WB pour avancer dans WB
- ld x4, 16(x3) ne peut pas entrer dans MEM. **Pourquoi ?**



# Exécution dans le désordre

- **Deadlock matériel :**

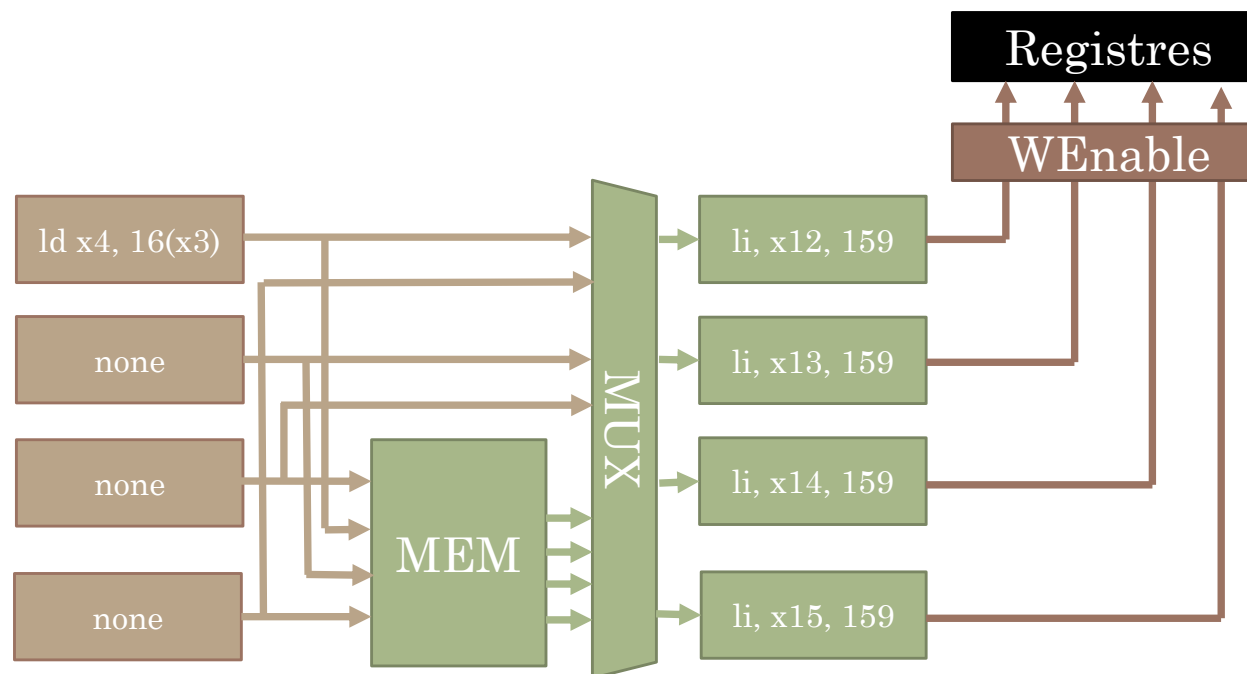
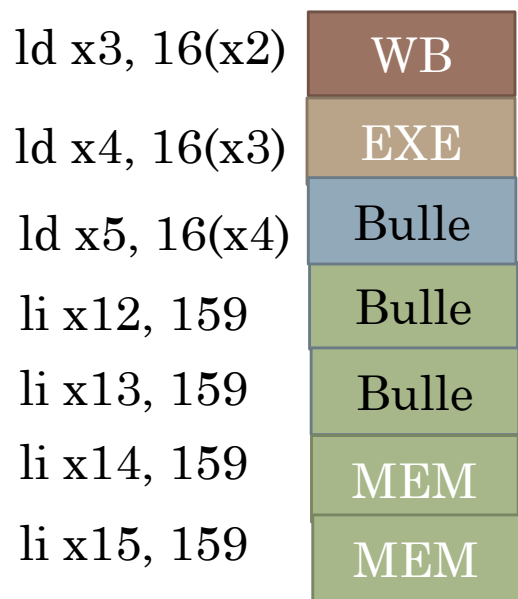
- Car on conserve les résultats calculés dans les registres de pipeline de EXE/MEM en attendant WB



# 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



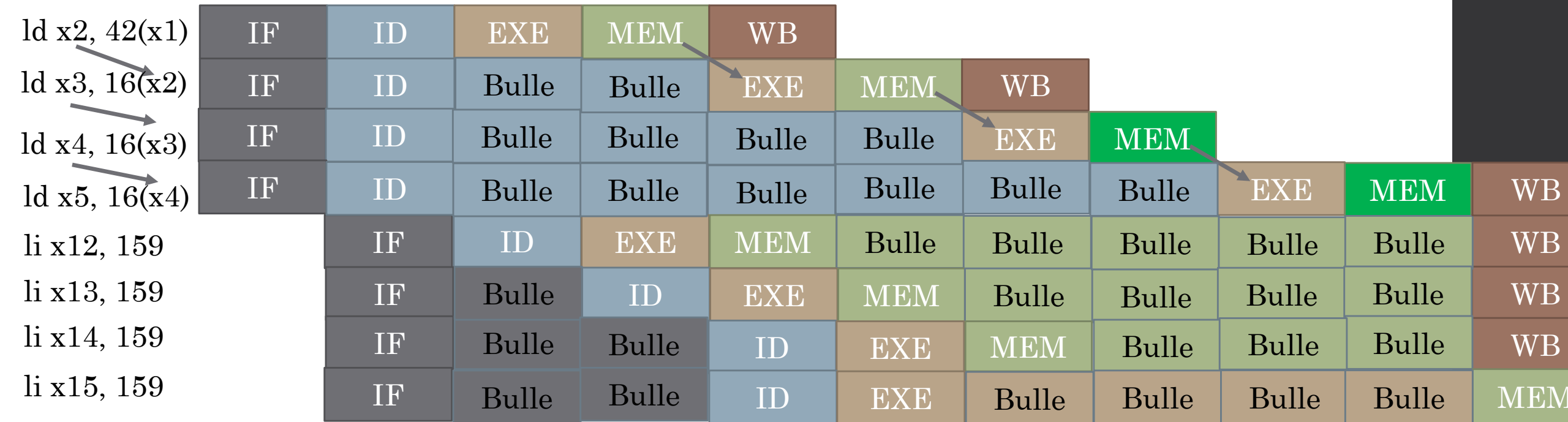
# Exécution dans le désordre

- 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 si elle prend le dernier emplacement ET qu'une instruction plus vieille n'est pas encore passée par EXE



# 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 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*

# Exécution dans le désordre

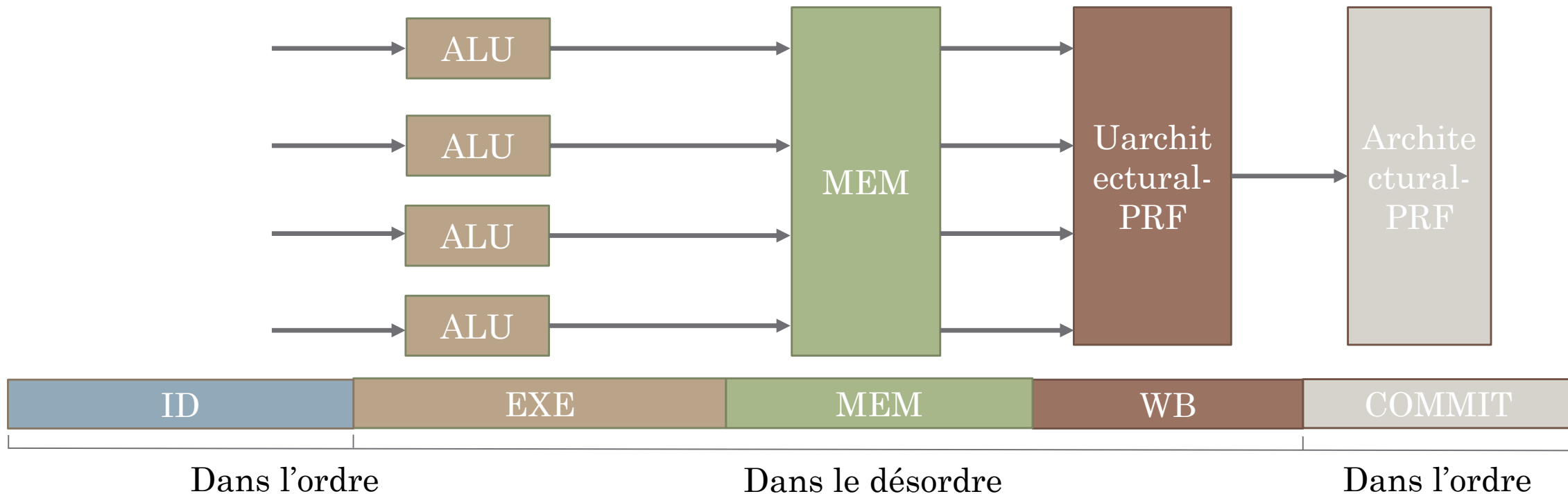
- Concrètement :
  - WaW/WaR : Il peut exister plusieurs version d'un même registre architectural dans le pipeline, mais aucune façon des les différencier, donc on bloque si WaW/WaR
  - Deadlock : il peut y avoir plus de résultats en vol attendant d'être écrit dans WB que de registres de pipelines pour 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*
- On doit toujours copier le registre temporaire vers le registre architectural dans l'ordre du programme

# Exécution dans le désordre

Renommage de registres

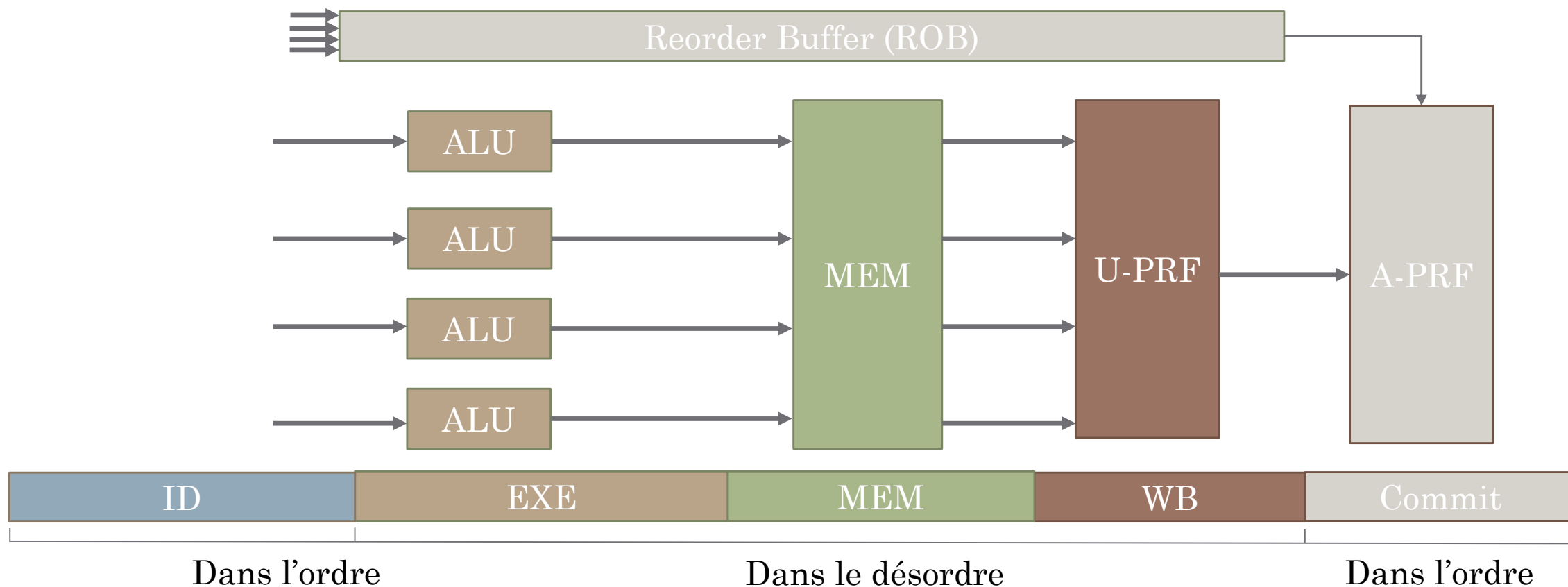
# Exécution dans le désordre – 2<sup>nd</sup>e itération

- On ajoute un fichier de registre microarchitectural (U-PRF)
  - Chaque instruction obtient un registre microarchitectural où écrire à ID, et le libère dans COMMIT (après copie dans le registre architectural)
  - ID détermine dynamiquement si le registre doit être lu depuis le fichier architectural ou microarchitectural



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

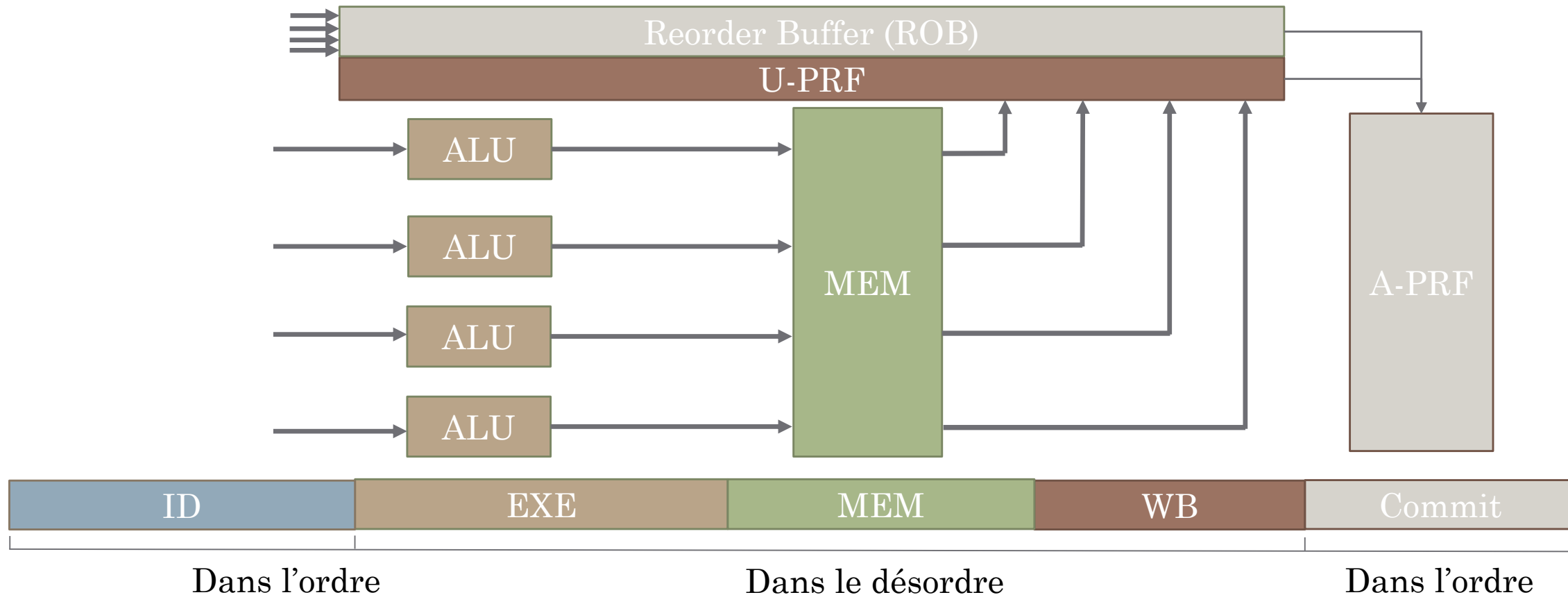
- On ajoute un fichier de registre microarchitectural
  - On doit aussi garder une trace des instructions pour pouvoir les traiter dans l'ordre => ROB (structure First In First Out)





# 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)
  - Ici, le ROB est aussi le fichier de registre microarchitectural



# 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 (= numéro de l'entrée du producteur dans le ROB) 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 (= numéro de l'entrée du producteur dans le ROB) 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 leurs propres registres microarchitecturaux

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 (= numéro de l'entrée du producteur dans le ROB) 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 leurs propres registres microarchitecturaux
  - ld x3, 16(x2) lit le registre microarchitectural 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 (= numéro de l'entrée du producteur dans le ROB) 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 leurs propres registres microarchitecturaux
  - ld x3, 16(x2) lit le registre microarchitectural écrit par ld x2, 42(x1)
  - ld x5, 16(x2) lit le registre microarchitectural écrit par li x2, 152

ld x2, 42(x1)

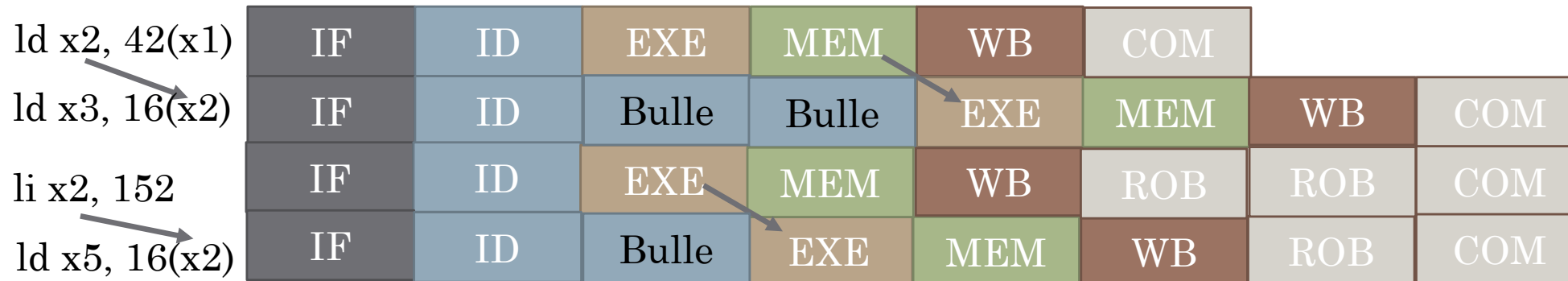
ld x3, 16(x2)

li x2, 152

ld x5, 16(x2)

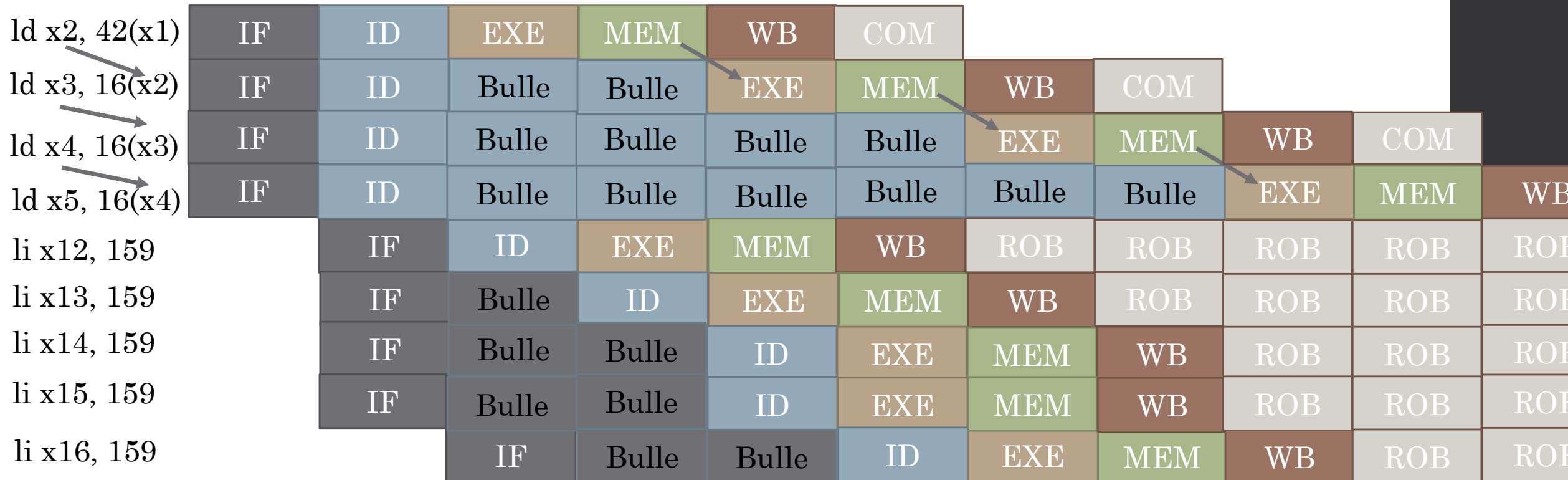
# Exécution dans le désordre – 2<sup>nd</sup>e itération

- On admet pour le moment que chaque instruction détermine le nom des ses registres physiques sources (= numéro de l'entrée du producteur dans le ROB) 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 leurs propres registres microarchitecturaux
  - ld x3, 16(x2) lit le registre microarchitectural écrit par ld x2, 42(x1)
  - ld x5, 16(x2) lit le registre microarchitectural écrit par li x2, 152



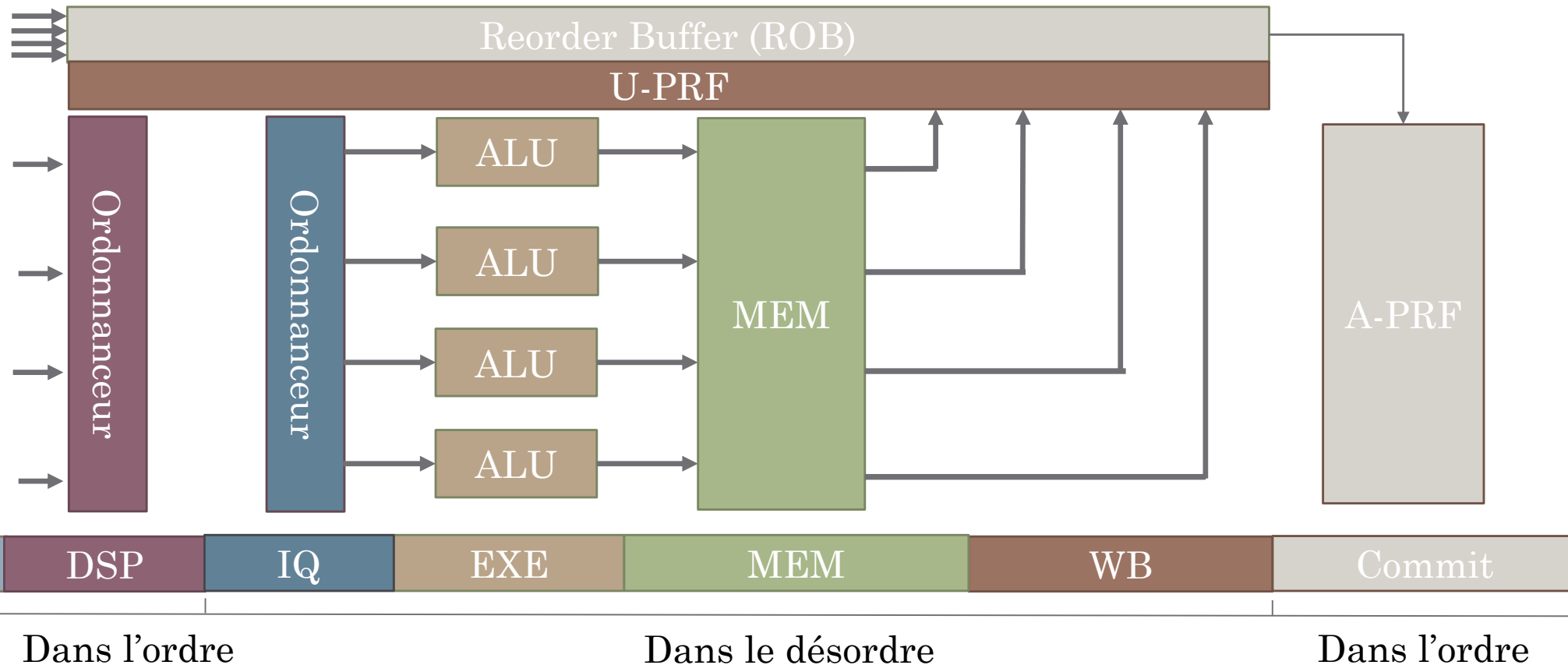
# Exécution dans le désordre – 2<sup>nd</sup>e itération

- 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 »)
    - Permet de stocker plusieurs dizaines d'instructions qui attendent leurs opérandes
  - Chaque cycle, DSP insère 4 instructions, et IQ sélectionne jusqu'à 4 instructions prêtes





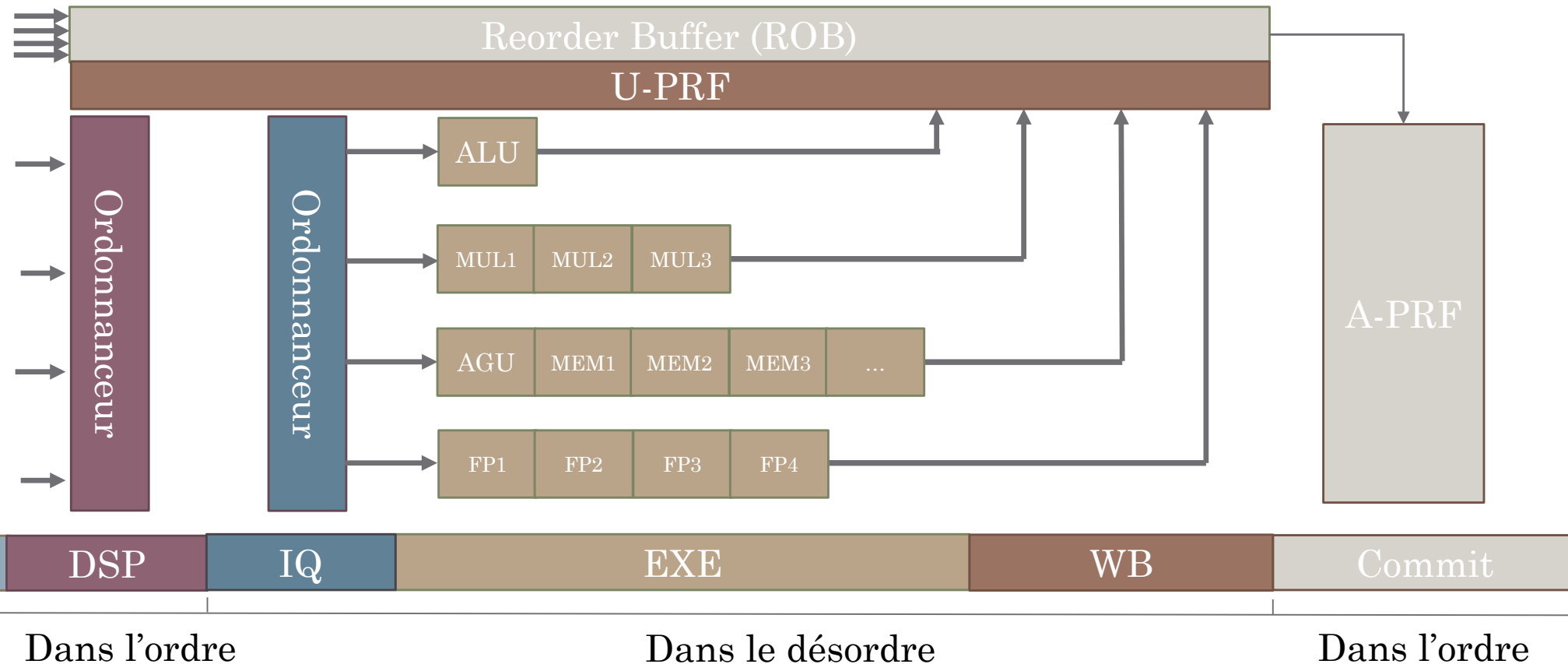
# Exécution dans le désordre – 3ème itération

- 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



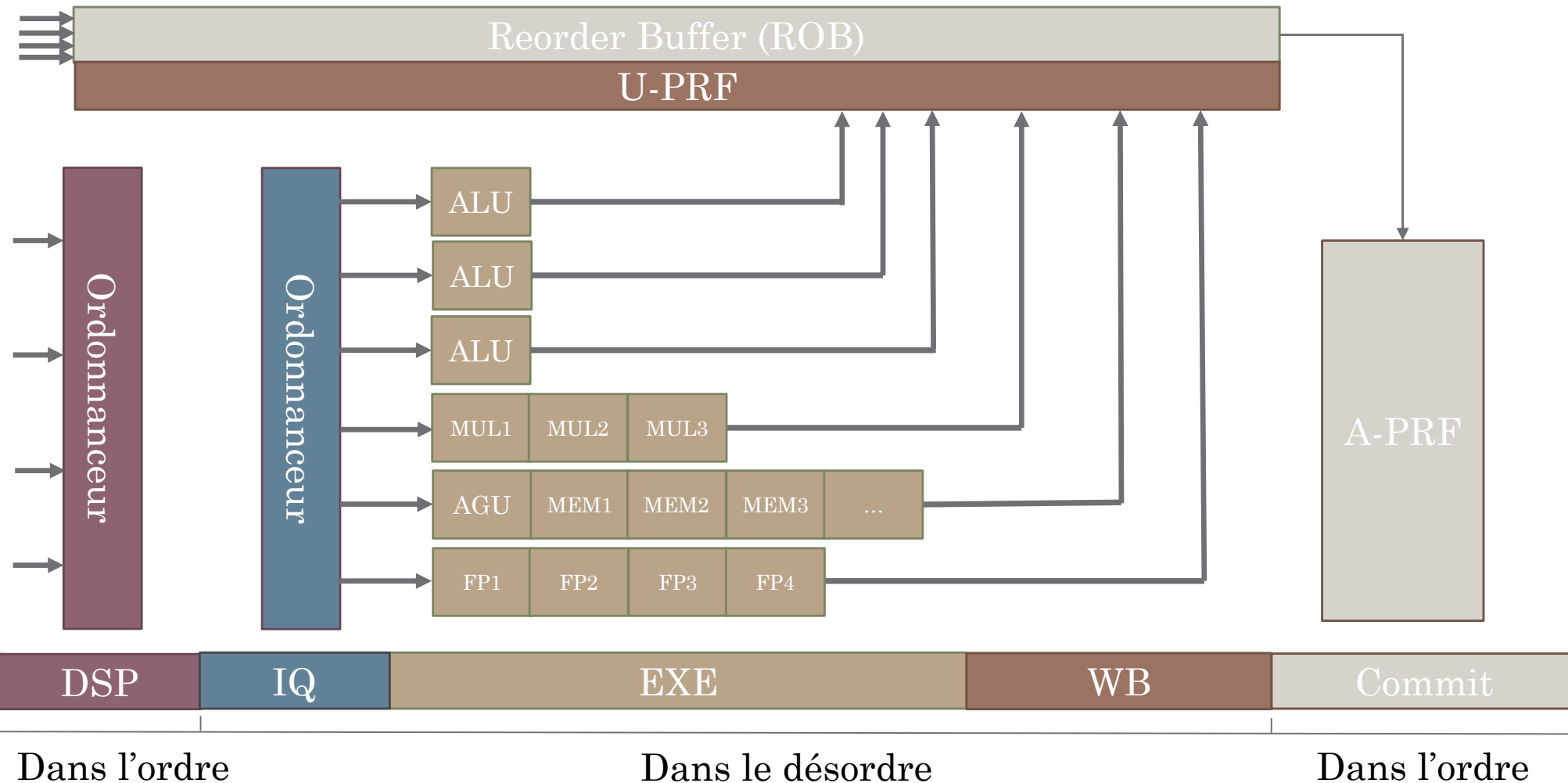
# Exécution dans le désordre – Hétérogénéité

- 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



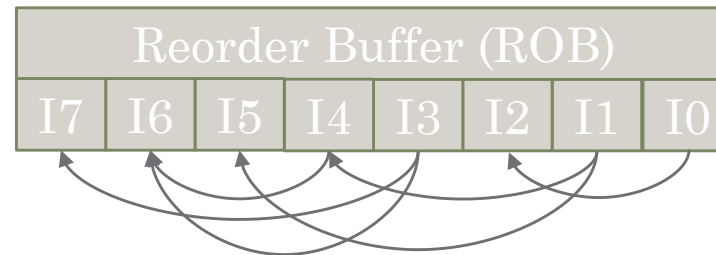
# Exécution dans le désordre - Avantages

- 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 » (= index dans le ROB/PRF) 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



# Renommage de registres

- La microarchitecture implémente  $n$  registres physiques, avec  $n$  >> #regs\_archs (U-PRF)
  - En plus du fichier de registre arch. qui a lui #regs\_archs registres (A-PRF)
  - Pour le moment, U-PRF lié au ROB

# Renommage de registres

- La microarchitecture implémente  $n$  registres physiques, avec  $n >> \#regs\_archs$  (U-PRF)
  - En plus du fichier de registre arch. qui a lui  $\#regs\_archs$  registres (A-PRF)
  - Pour le moment, U-PRF lié au ROB
- Chaque instruction se voit attribuer un nouveau registre physique pour écrire son résultat (insertion dans le ROB)
  - 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  $n \gg \#regs\_archs$  (U-PRF)
  - En plus du fichier de registre arch. qui a lui  $\#regs\_archs$  registres (A-PRF)
  - Pour le moment, U-PRF lié au ROB
- Chaque instruction se voit attribuer un nouveau registre physique pour écrire son résultat (insertion dans le ROB)
  - 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)
  - Afin qu'une instruction puisse déterminer quelle entrée du ROB contient son opérande source

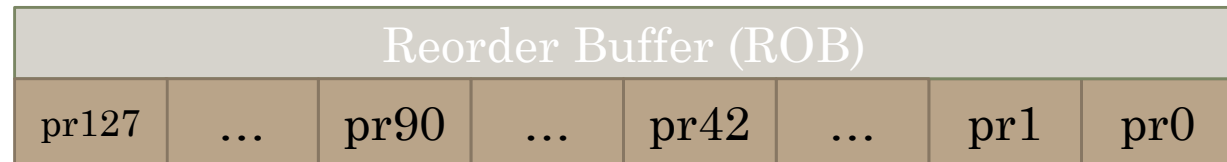


# Renommage de registres

Rename Map Table (RMT)

x0	pr1
x1	pr90
...	...
x30	pr42
x31	pr127

Physical Register File (PRF)  
ici, c'est le ROB



La RMT indique le registre physique contenant la version la plus récente du registre architectural

# Renommage des registres sources

Rename Map Table (RMT)

add x0, x2, x1

The diagram illustrates the mapping of source registers from a code snippet to physical registers using a Rename Map Table (RMT). The code snippet is "add x0, x2, x1". Arrows indicate the flow of data: x1 is mapped to pr90 and x2 is mapped to pr127. The RMT table is a 4x2 grid with the following entries:

x0	pr2
x1	<b>pr90</b>
x2	<b>pr127</b>
x3	pr42

Arrows from the code point to the RMT table: one from x1 to the row containing x1 and pr90, and another from x2 to the row containing x2 and pr127. Arrows also point from the pr90 and pr127 cells to the right, indicating the physical registers used for the operation.

# Renommage du registre de destination

- On crée une nouvelle version de x0, et on invalide l'ancienne dans la RMT

add x0, x2, x1



Rename Map Table (RMT)

x0	<del>pr2</del>
x1	pr90
x2	pr127
x3	pr42

# Renommage du registre de destination

Insertion dans ROB

...	add x0, x2, x1	I127	...	I1	I0
...	pr128	pr127	...	pr1	pr0

Rename Map Table (RMT)

x0	<del>pr2</del> pr128
x1	pr90
x2	pr127
x3	pr42

add x0, x2, x1

# Renommage du registre de destination

Insertion dans ROB

...	add x0, x2, x1	I127	...	I1	I0
...	pr128	pr127	...	pr1	pr0

Rename Map Table (RMT)

x0	<del>pr2</del> pr128
x1	pr90
x2	pr127
x3	pr42

add x0, x2, x1

En substance:

- Toutes les instructions **plus vieilles que add** iront lire la valeur de x0 dans **pr2**
- Toutes les instructions **plus jeunes que add** iront lire la valeur de x0 dans **pr128**

# Renommage de registres : RaW

ROB/PRF

...	...	add x0, x2, x1
...	pr128	pr127

add x0(**pr127**), x2(pr4), x1(pr90)  
sub x3, x0(**pr127**), x2(**pr4**)

Rename Map Table (RMT)

x0	<b>pr127</b>	→
x1	pr90	
x2	<b>pr4</b>	→
x3	pr42	

# Renommage de registres : RaW

ROB/PRF

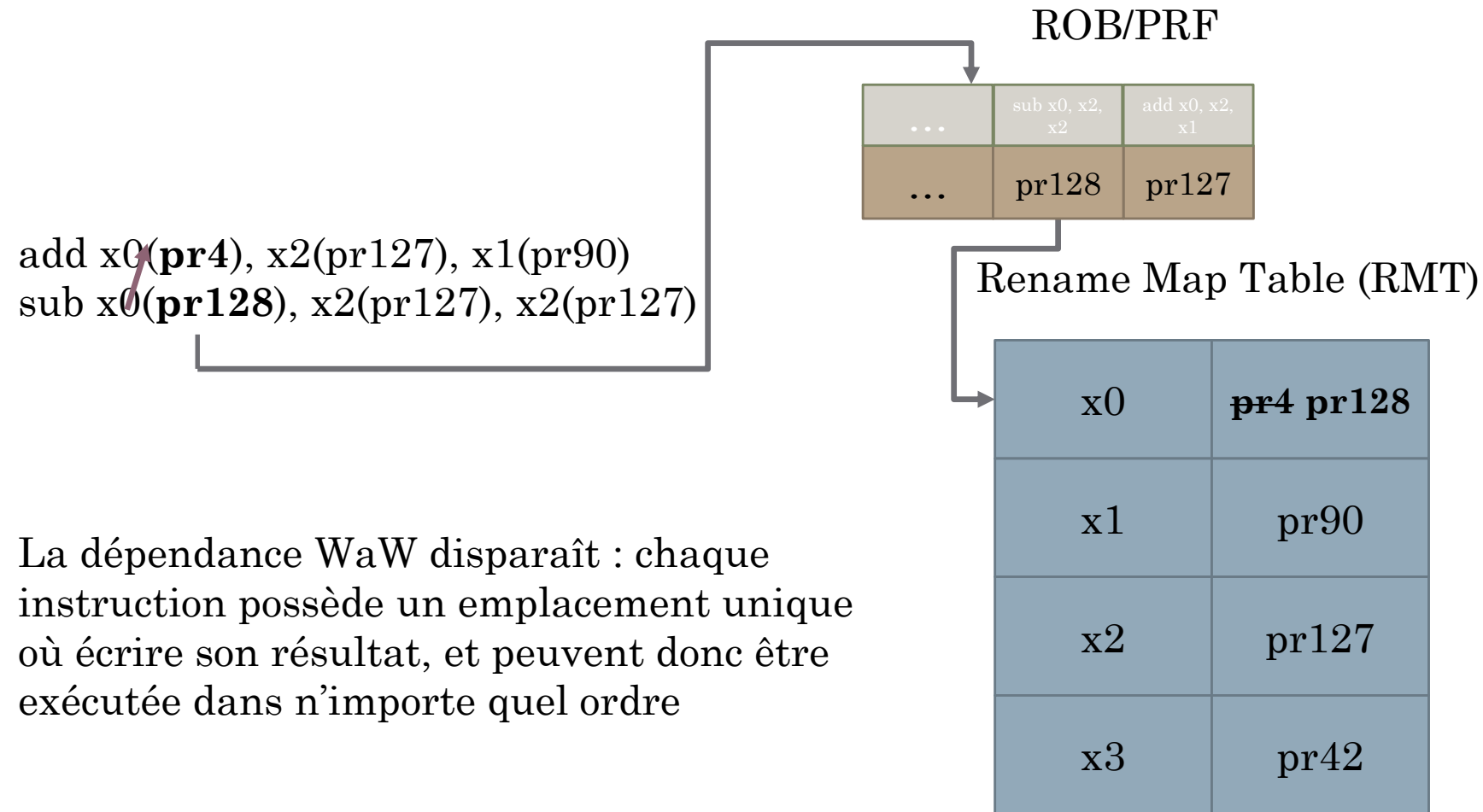
...	sub x3, x0, x2	add x0, x2, x1
...	pr128	pr127

add x0(**pr127**), x2(pr4), x1(pr90)  
sub x3(**pr128**), x0(**pr127**), x2(**pr4**)

Rename Map Table (RMT)

x0	pr127
x1	pr90
x2	pr127
x3	<del>pr42</del> pr128

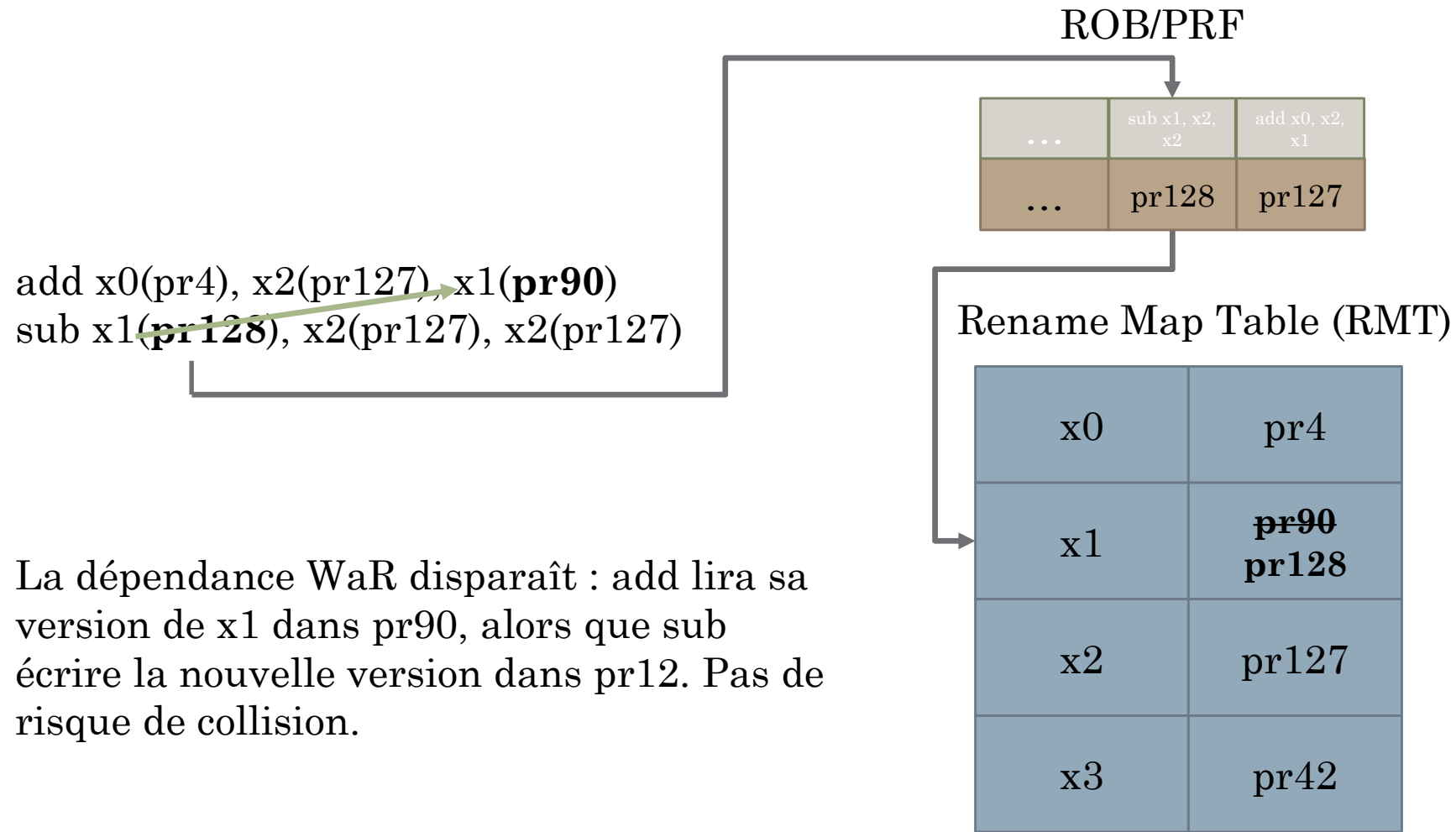
# Renommage de registres : WaW



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



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

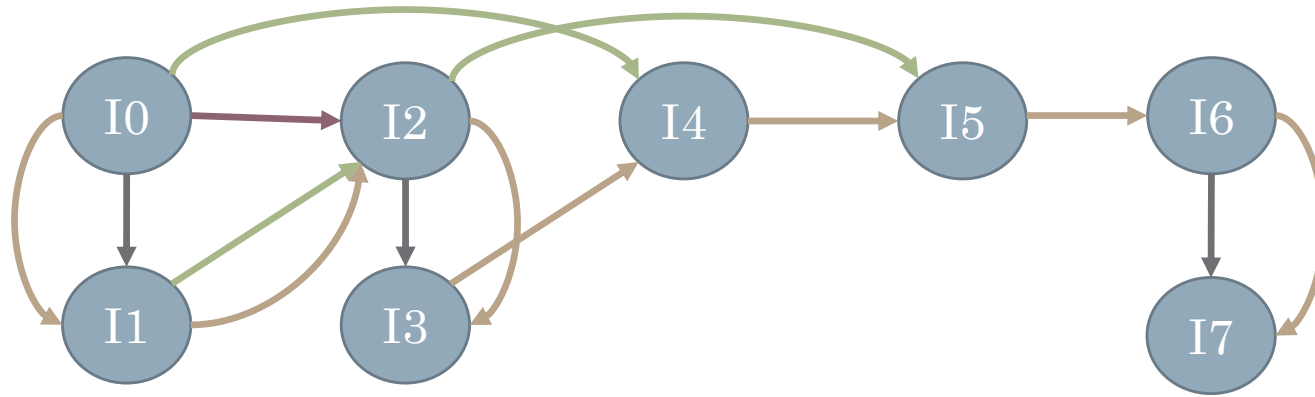
- Exercice

```
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.
  - Calculer l'IPC maximum sur une machine OoO avec des ressources illimitées et prédiction de branchement parfaite
- Calculer l'IPC une fois le pipeline rempli
  - superscalaire degré 4, dans l'ordre
  - superscalaire degré 4, dans le désordre

# 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

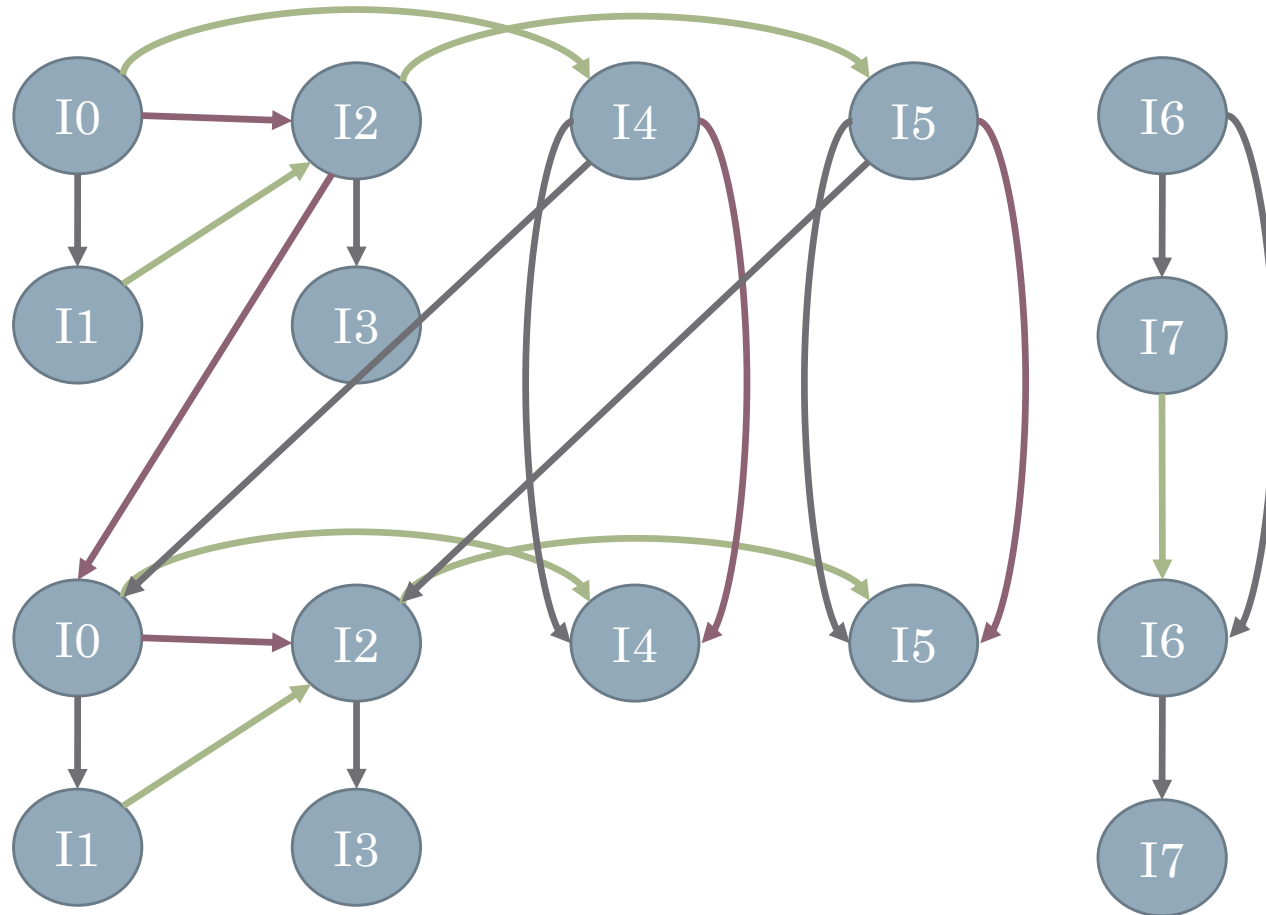
I5 addi, x3, x3, 16

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

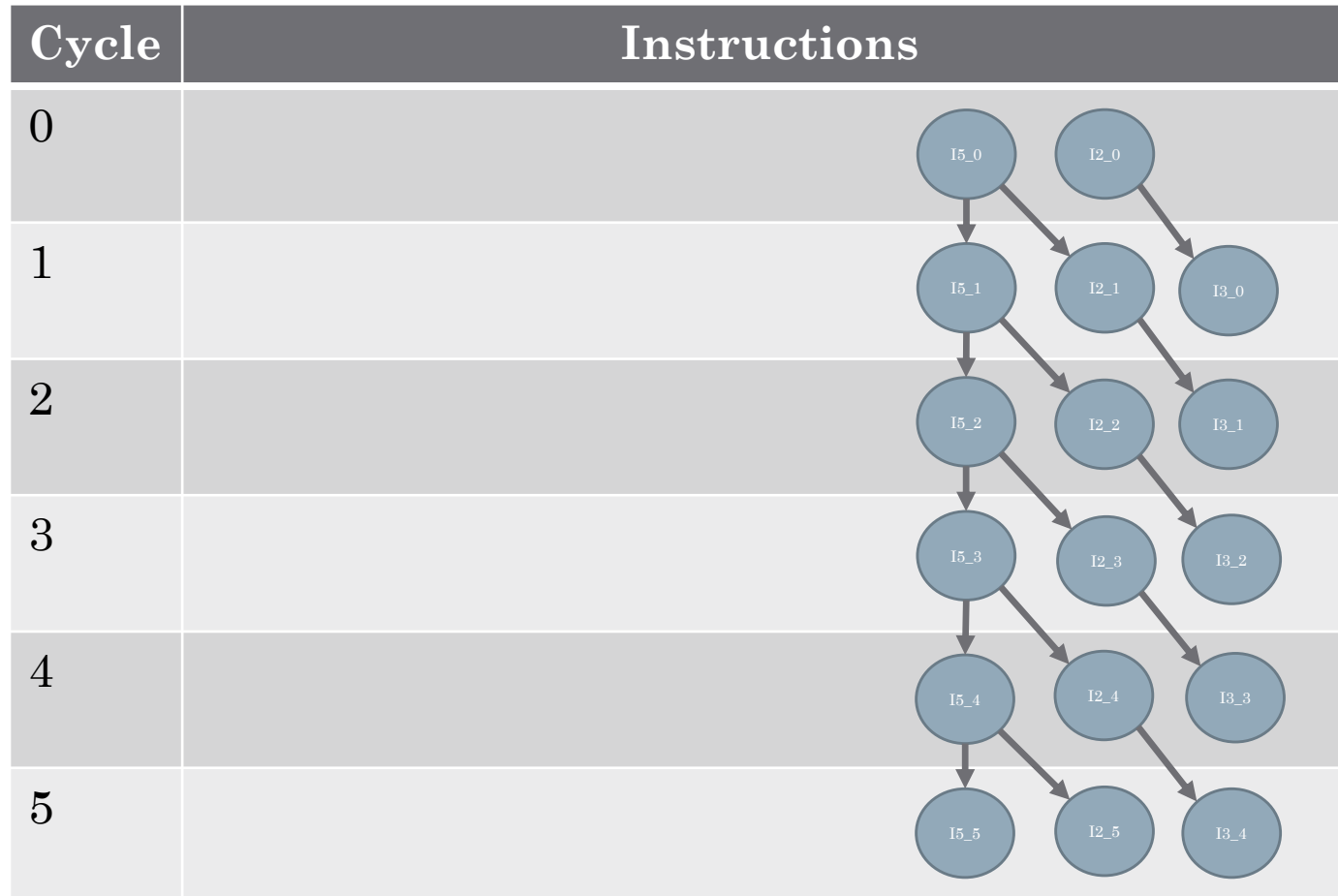
# Exécution dans l'ordre (InO) vs. Désordre (OoO)

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

Cycle	Instructions
0	
1	
2	
3	
4	
5	

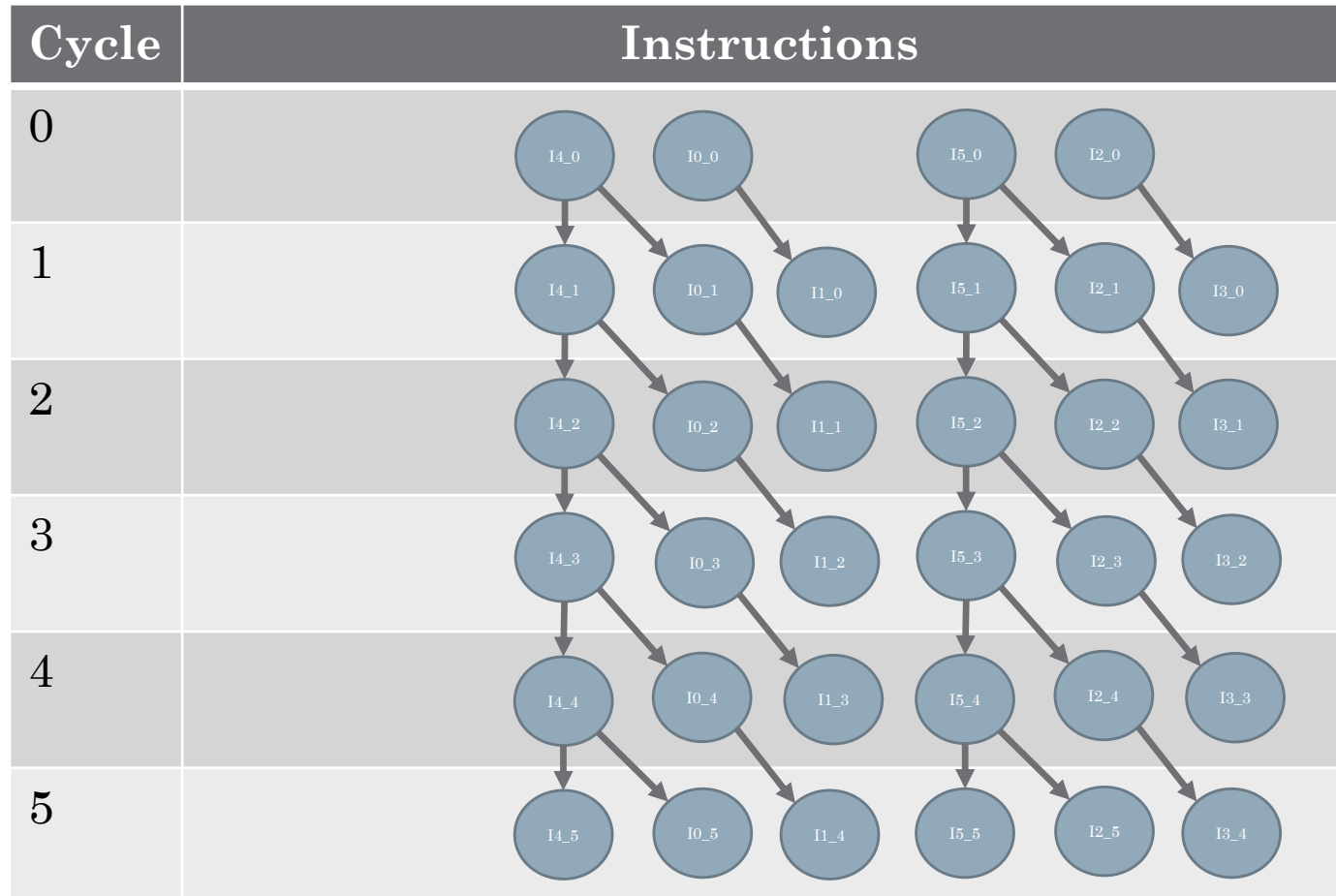
# Exécution dans l'ordre (InO) vs. Désordre (OoO)

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



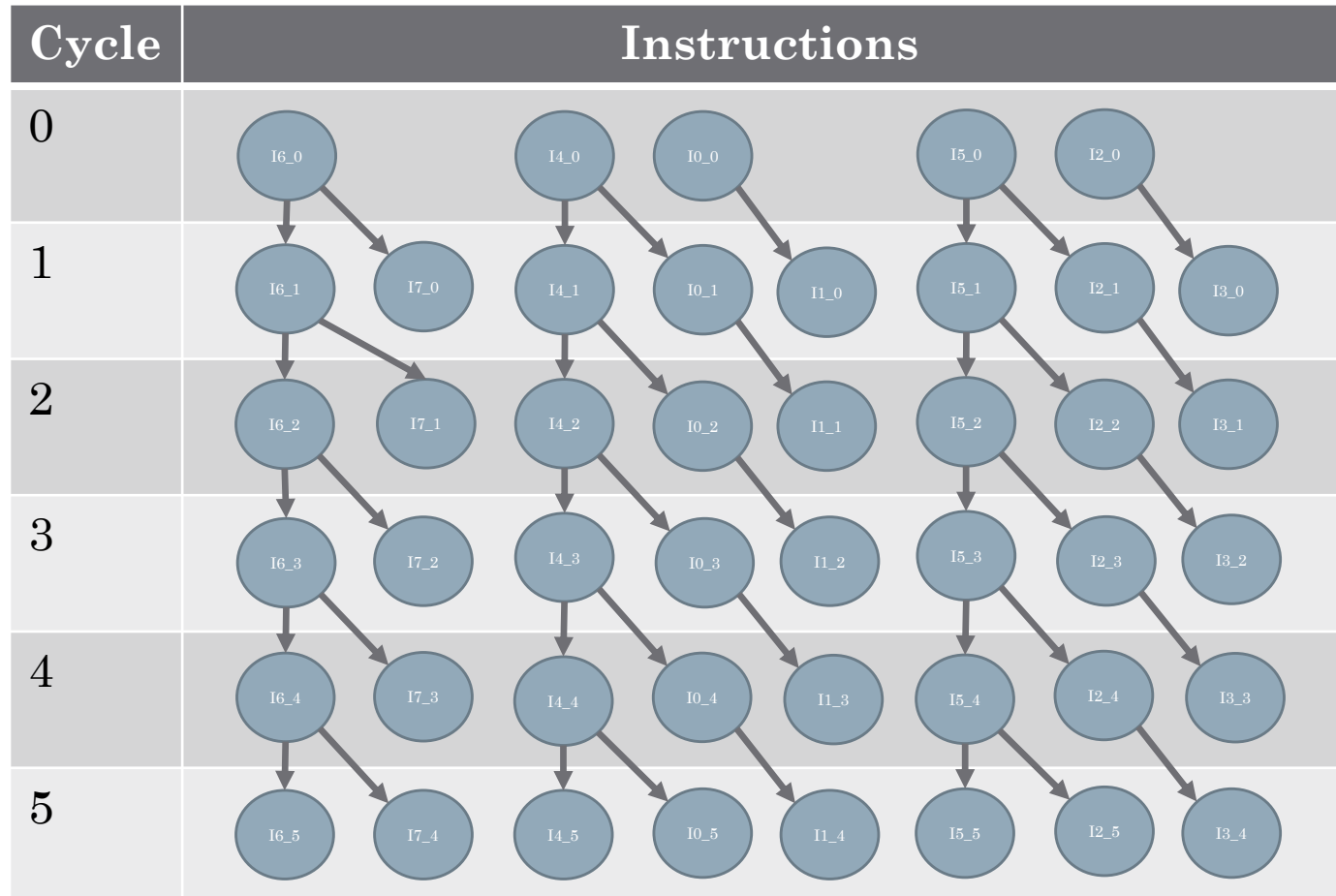
# Exécution dans l'ordre (InO) vs. Désordre (OoO)

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



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

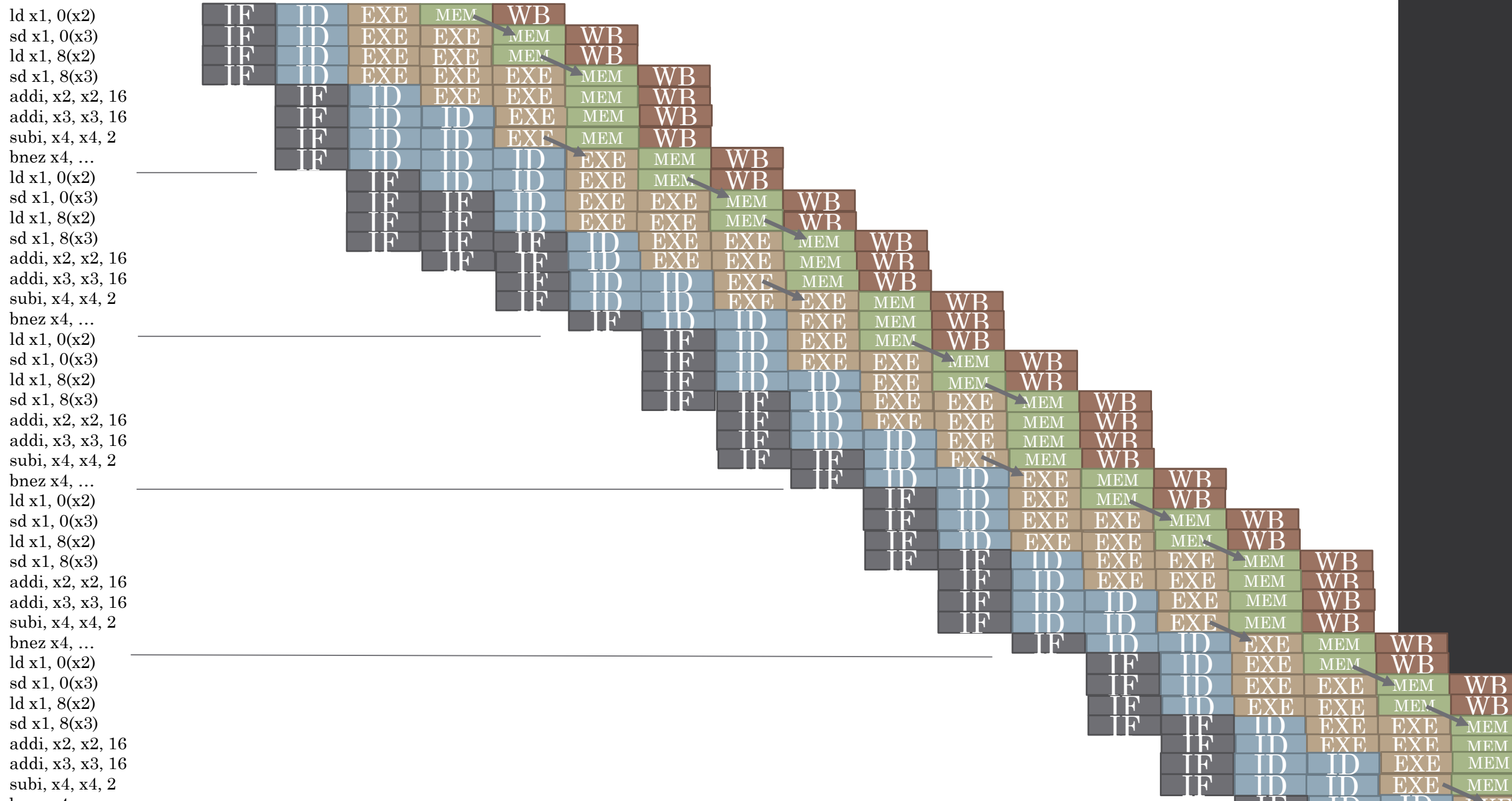
- IPC maximum sur machine OoO illimitée (= uniquement RaW) : on ordonnance



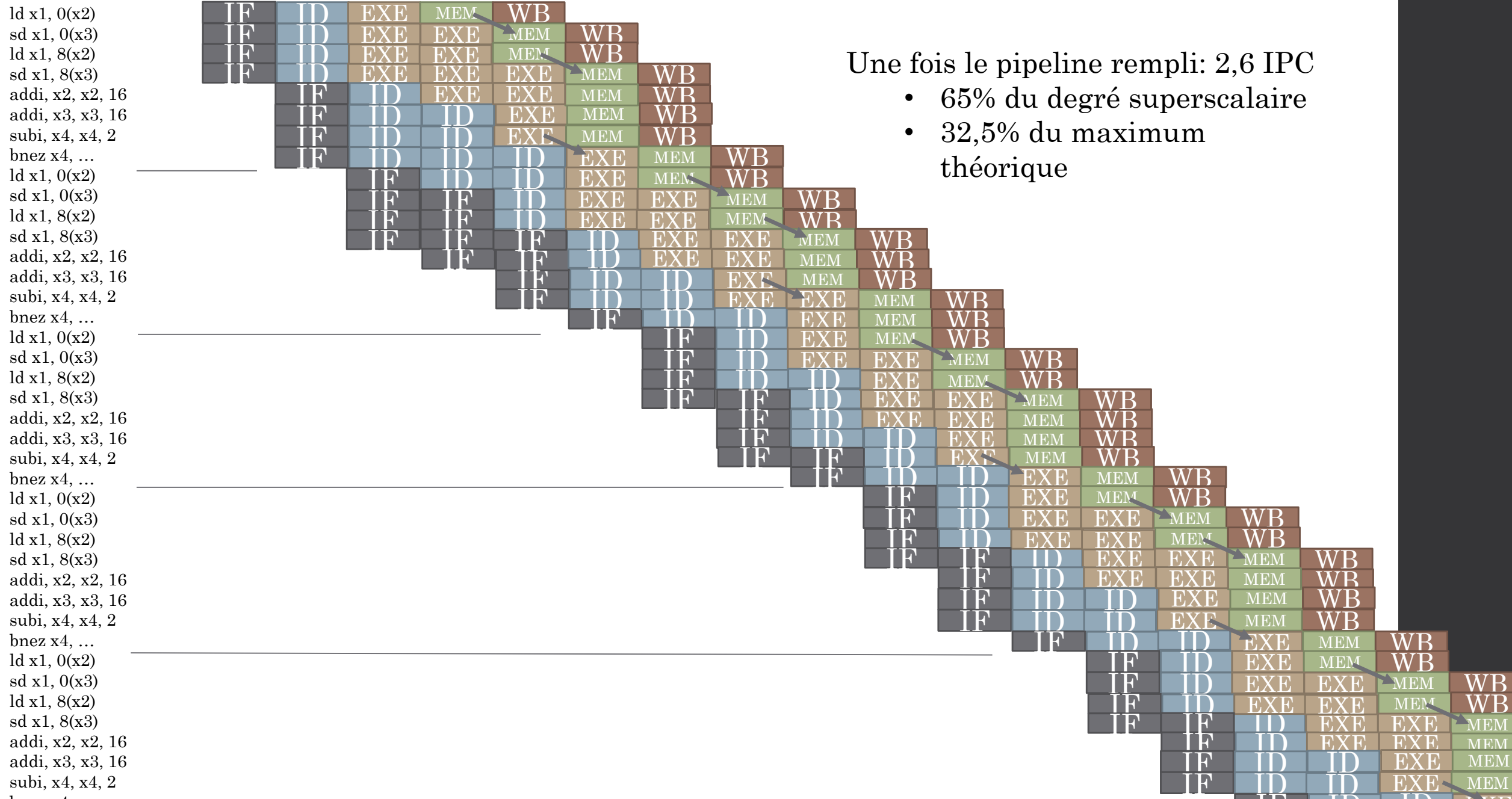
IPC max  
(intrinsèque à  
l'algorithme) = 8



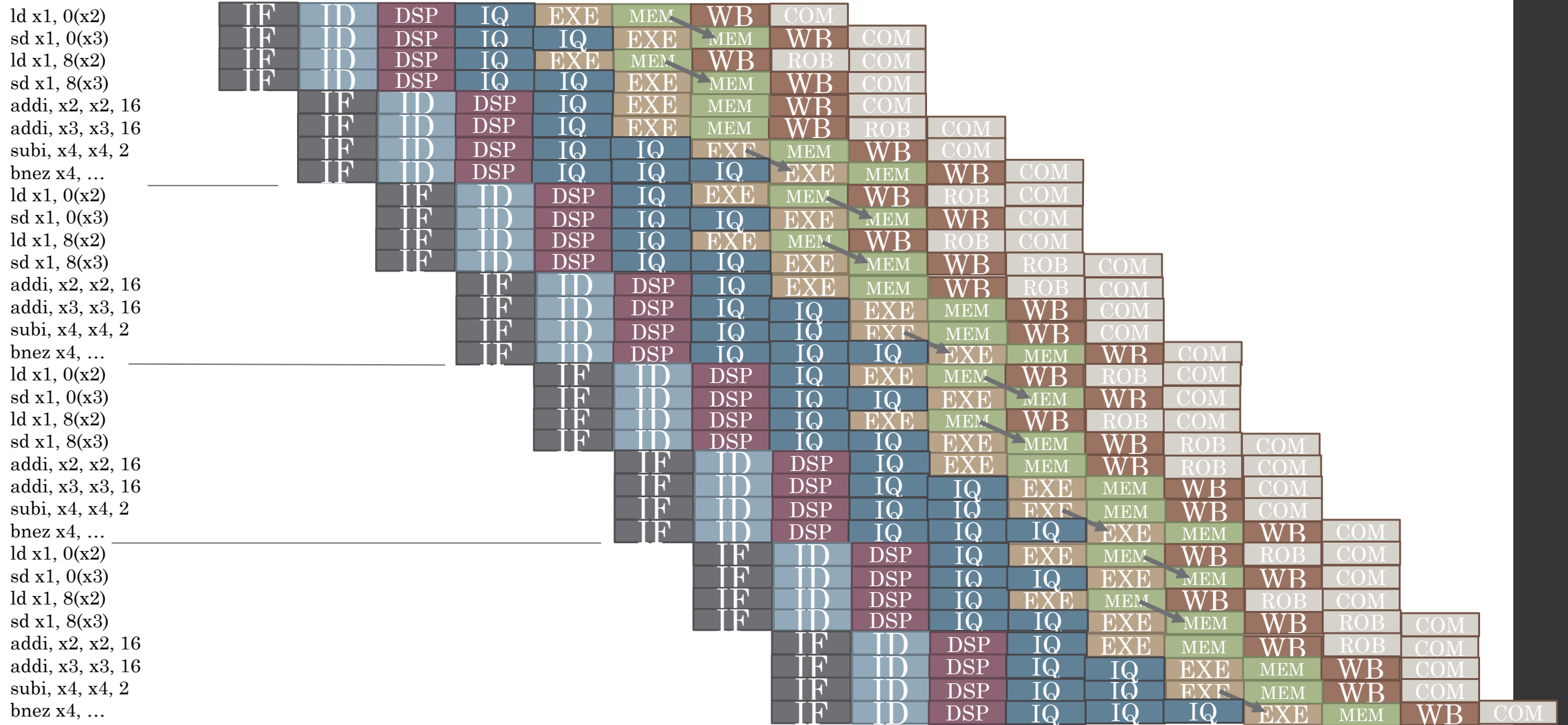
## Exécution dans l'ordre (InO) vs. Désordre (OoO) - InO



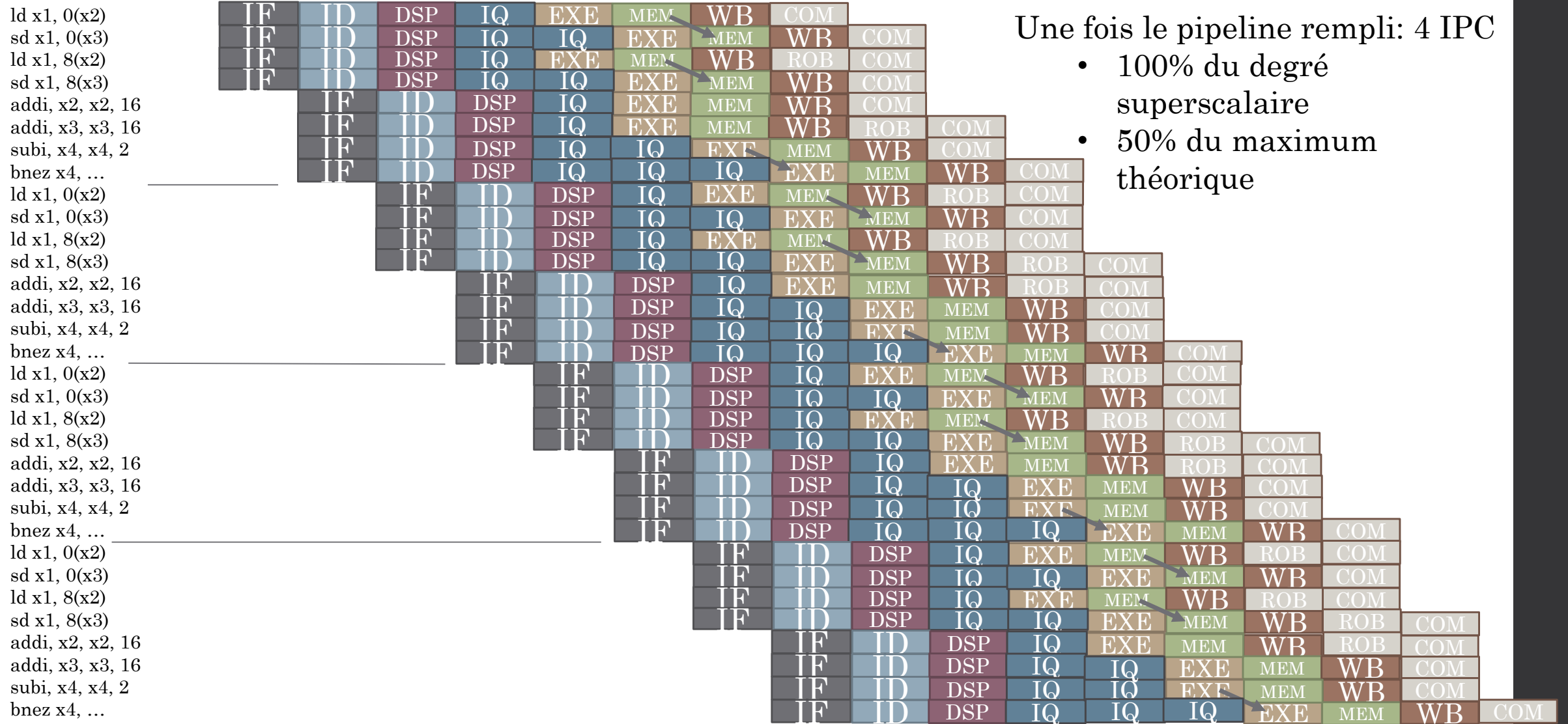
# Exécution dans l'ordre (InO) vs. Désordre (OoO) - InO



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



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



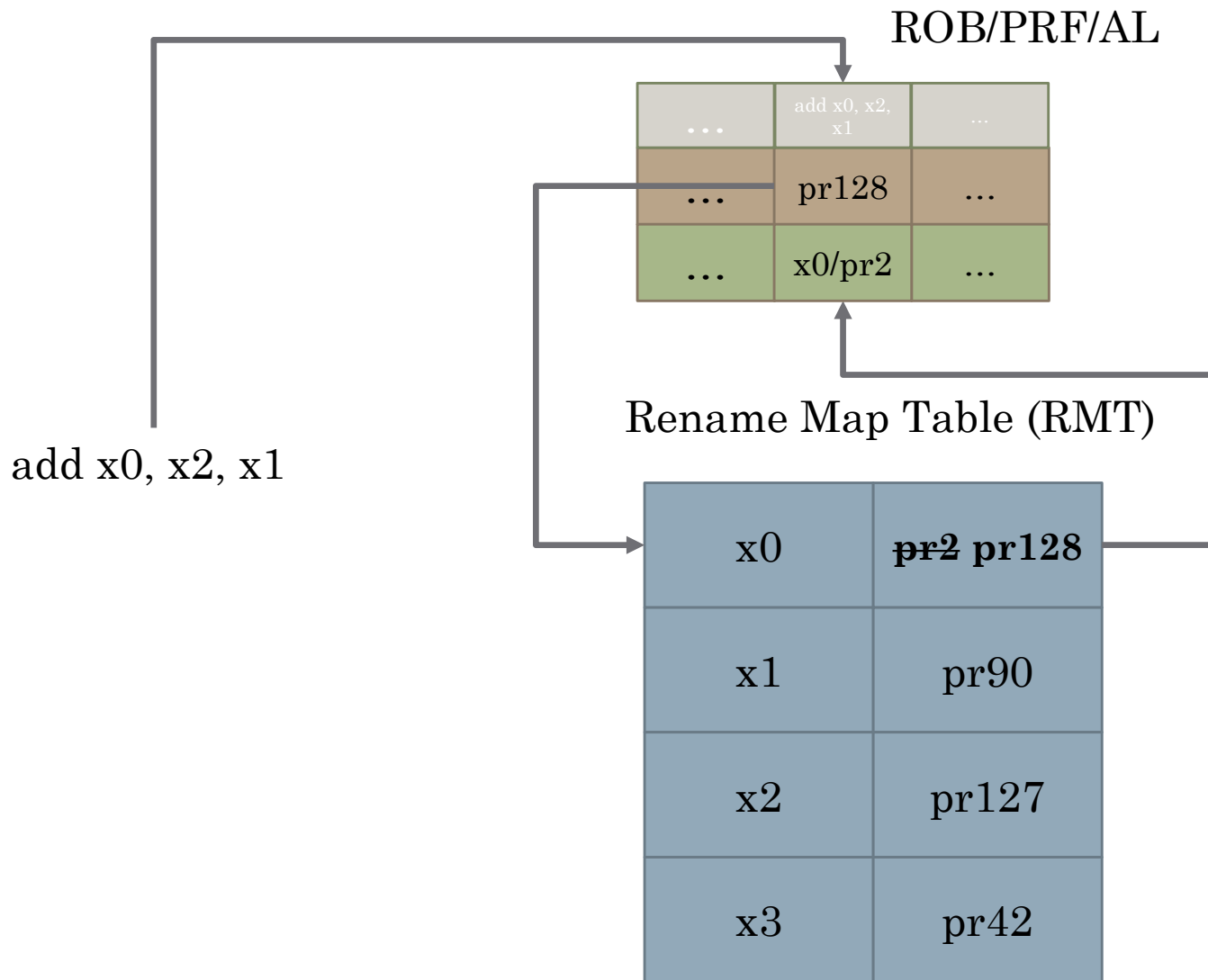
# Renommage de registres

- 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)

L'AL contient l'association arch/phy précédente

# Renommage de registres : Active List (AL)

Active  
List (ROB)

RMT

x0
x4

add x0, x2, x1

subi x0, x0, 1

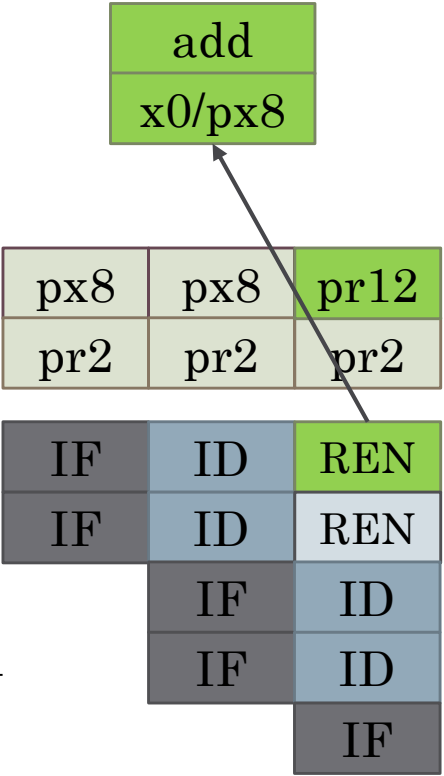
bnez x0, A

sub x0, x12, x1

mul x4, x0, x1

A:

add x4, x0, x1





# Renommage de registres : Active List (AL)

Active  
List (ROB)

add	subi
x0/px8	x0/pr12

RMT

x0
x4

px8	px8	pr12	pr19
pr2	pr2	pr2	pr2

add x0, x2, x1  
subi x0, x0, 1  
bnez x0, A  
sub x0, x12, x1  
mul x4, x0, x1

IF	ID	REN	RR
IF	ID	REN	RR
	IF	ID	REN
	IF	ID	REN
		IF	ID

A:  
add x4, x0, x1



# Renommage de registres : Active List (AL)

Active  
List (ROB)

add	subi	mul
x0/px8	x0/pr12	x4/pr2

RMT

x0
x4

px8	px8	pr12	pr19	pr19
pr2	pr2	pr2	pr2	pr13

add x0, x2, x1  
subi x0, x0, 1  
bnez x0, A  
sub x0, x12, x1  
mul x4, x0, x1

IF	ID	REN	RR	DSP
IF	ID	REN	RR	DSP
	IF	ID	REN	RR
	IF	ID	REN	RR
		IF	ID	REN

A:  
add x4, x0, x1

# Renommage de registres : Active List (AL)

Active  
List (ROB)

add	subi	mul
x0/px8	x0/pr12	x4/pr2

Br mispred (bnez x0, A)

RMT

x0
x4

px8	px8	pr12	pr19	pr19
pr2	pr2	pr2	pr2	pr13

pr19	pr19	pr19
pr13	pr13	pr13

add x0, x2, x1  
subi x0, x0, 1  
bnez x0, A  
sub x0, x12, x1  
mul x4, x0, x1

IF	ID	REN	RR	DSP	IQ	EXE	WB	CO
IF	ID	REN	RR	DSP	IQ	IQ	EXE	WB
	IF	ID	REN	RR	DSP	IQ	IQ	EXE
	IF	ID	REN	RR	DSP	IQ	EXE	WB
		IF	ID	REN	RR	DSP	IQ	EXE

A:  
add x4, x0, x1

subi et mul renommée  
sur le mauvais chemin

Après le branchement,  
on devrait avoir :  
x0 = pr12  
x1 = pr2

On a :  
x0 = pr19  
x1 = pr13

# Renommage de registres : Active List (AL)

Active  
List (ROB)

add	subi	mul
x0/px8	x0/pr12	x4/pr2

Br mispred (bnez A)

RMT

x0
x4

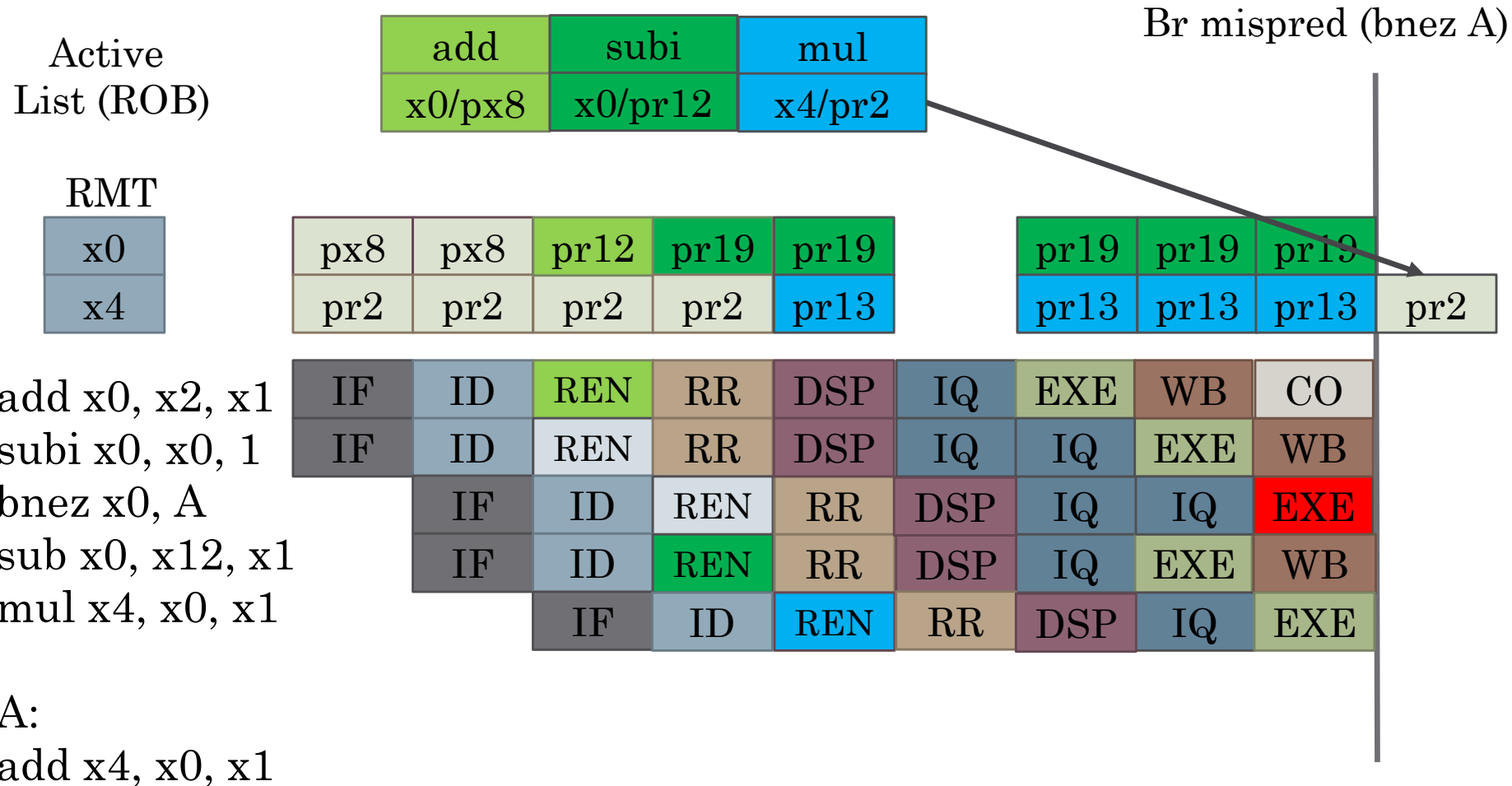
px8	px8	pr12	pr19	pr19		pr19	pr19	pr19
pr2	pr2	pr2	pr2	pr13		pr13	pr13	pr13
IF	ID	REN	RR	DSP	IQ	EXE	WB	CO
IF	ID	REN	RR	DSP	IQ	IQ	EXE	WB
	IF	ID	REN	RR	DSP	IQ	IQ	EXE
	IF	ID	REN	RR	DSP	IQ	EXE	WB
		IF	ID	REN	RR	DSP	IQ	EXE

add x0, x2, x1  
subi x0, x0, 1  
bnez x0, A  
sub x0, x12, x1  
mul x4, x0, x1

A:  
add x4, x0, 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)



# Renommage de registres : Active List (AL)

Active  
List (ROB)

add
x0/px8

Br mispred (bnez A)

RMT

x0
x4

px8	px8	pr12	pr19	pr19
pr2	pr2	pr2	pr2	pr13

pr19	pr19	pr19	pr12
pr13	pr13	pr13	pr2

add x0, x2, x1

subi x0, x0, 1

bnez x0, A

sub x0, x12, x1

mul x4, x0, x1

IF	ID	REN	RR	DSP	IQ	EXE	WB	CO
IF	ID	REN	RR	DSP	IQ	IQ	EXE	WB
	IF	ID	REN	RR	DSP	IQ	IQ	EXE
	IF	ID	REN	RR	DSP	IQ	EXE	WB
		IF	ID	REN	RR	DSP	IQ	EXE

A:

add x4, x0, x1

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

# Renommage de registres : Active List (AL)

Active  
List (ROB)

add
x0/px8

Br mispred (bnez A)

RMT

x0
x4

px8	px8	pr12	pr19	pr19
pr2	pr2	pr2	pr2	pr13

pr19	pr19	pr19	pr12	pr12	pr12
pr13	pr13	pr13	pr2	pr2	pr2

add x0, x2, x1

subi x0, x0, 1

bnez x0, A

sub x0, x12, x1

mul x4, x0, x1

IF	ID	REN	RR	DSP	IQ	EXE	WB	CO
IF	ID	REN	RR	DSP	IQ	IQ	EXE	WB
	IF	ID	REN	RR	DSP	IQ	IQ	EXE
	IF	ID	REN	RR	DSP	IQ	EXE	WB
		IF	ID	REN	RR	DSP	IQ	EXE

	CO
WB	CO

A:

add x4, x0, x1

IF	ID
----	----

On reprend ensuite l'exécution

Active  
List (ROB)

add	add
x0/px8	x4/pr2

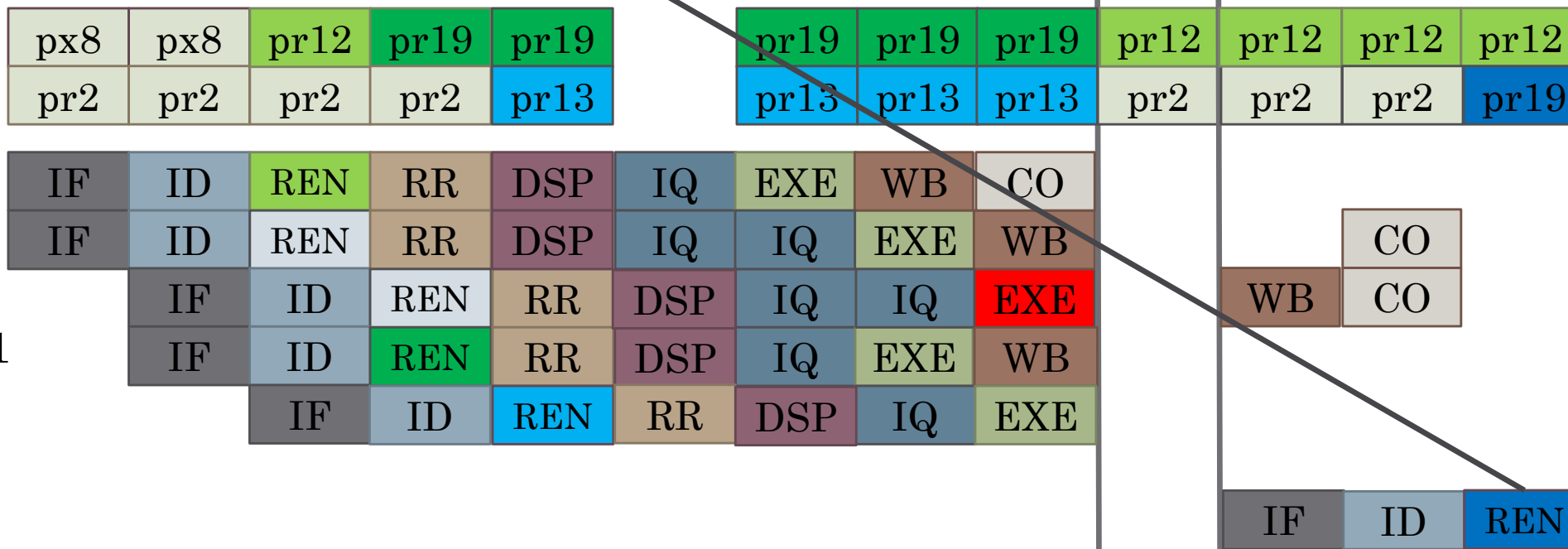
Br mispred (bnez A)

RMT

x0
x4

add x0, x2, x1  
subi x0, x0, 1  
bnez x0, A  
sub x0, x12, x1  
mul x4, x0, x1

A:  
add x4, x0, x1





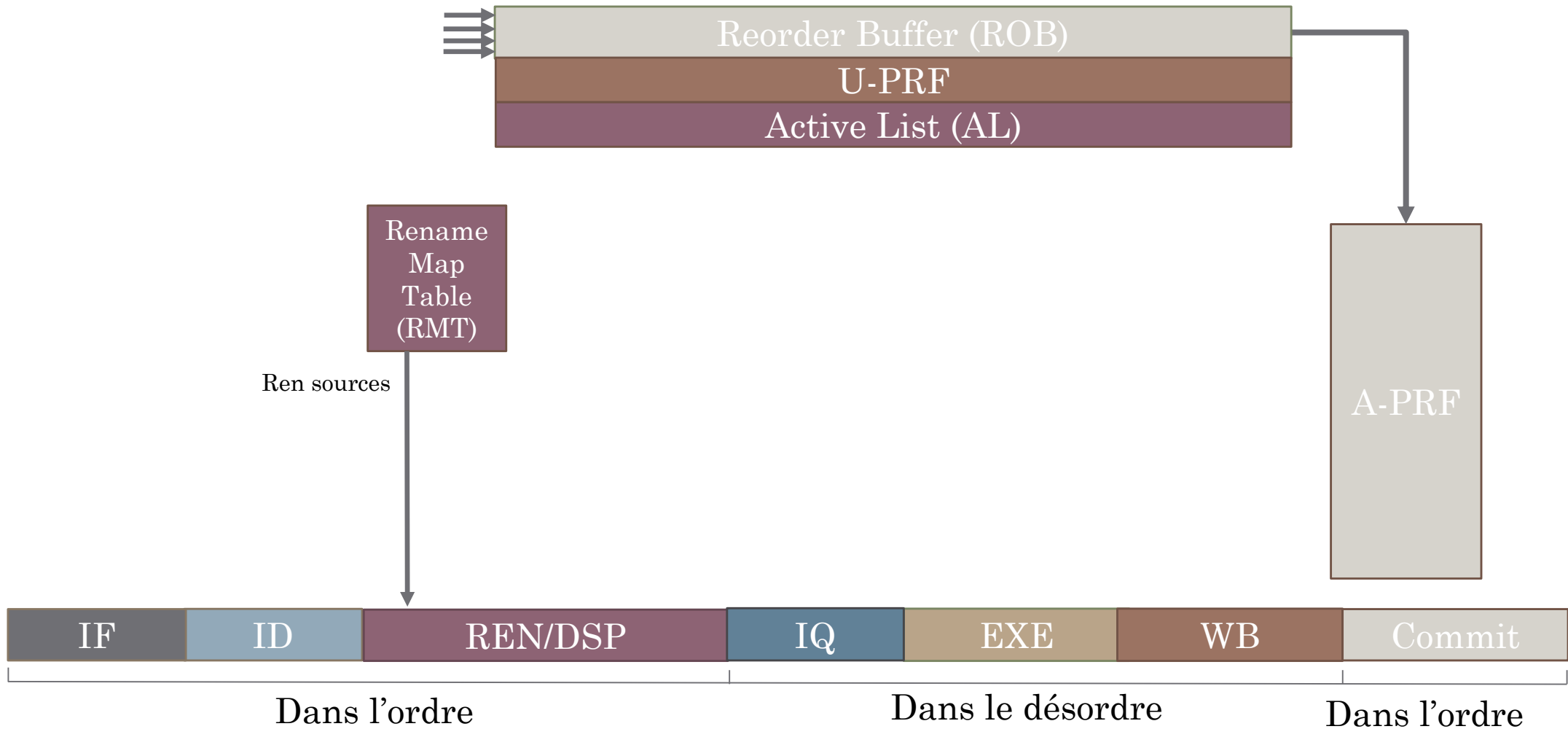
# Renommage de registres : Active List (AL)

- 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

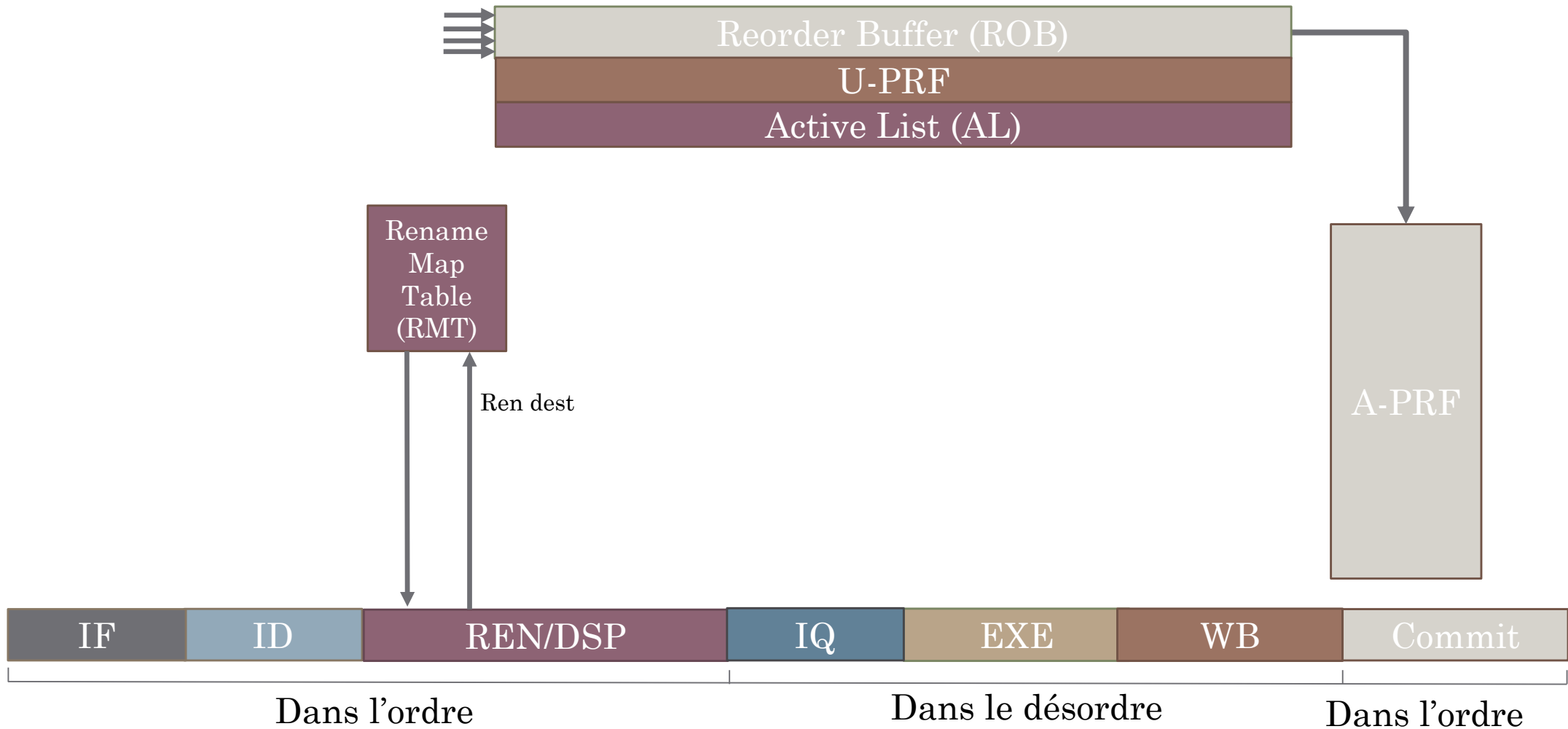
# Recyclage de registres physiques

- Jusqu'ici, on a parlé de l'attribution des registres physiques
  - En même temps que l'insertion de l'instruction dans le ROB
- Libération du registre ?
  - En même temps que l'instruction quitte le ROB, au Commit
  - Copie du registre vers le fichier de registres *architectural*
  - Variations possibles (pas discutées ici)

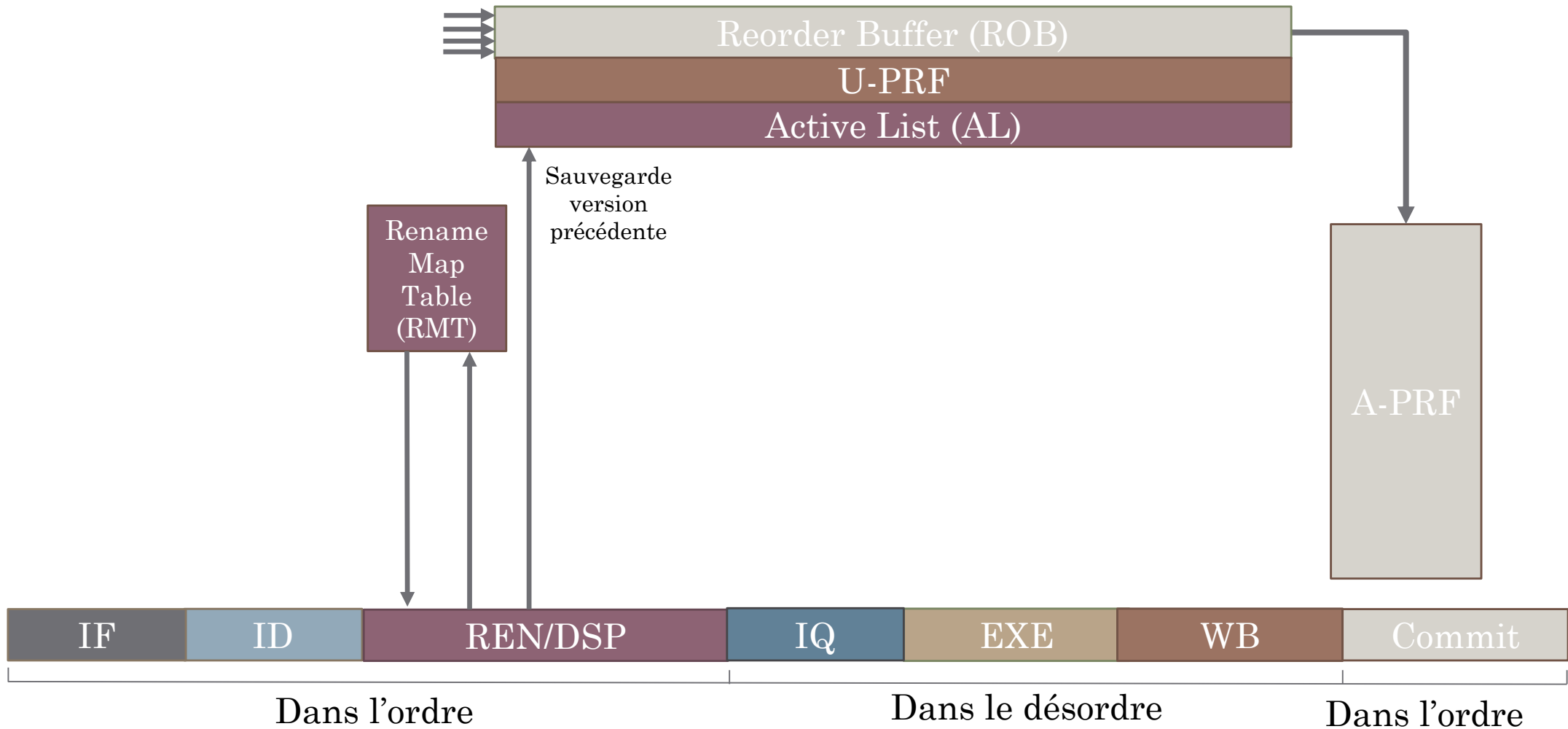
# Renommage de registres



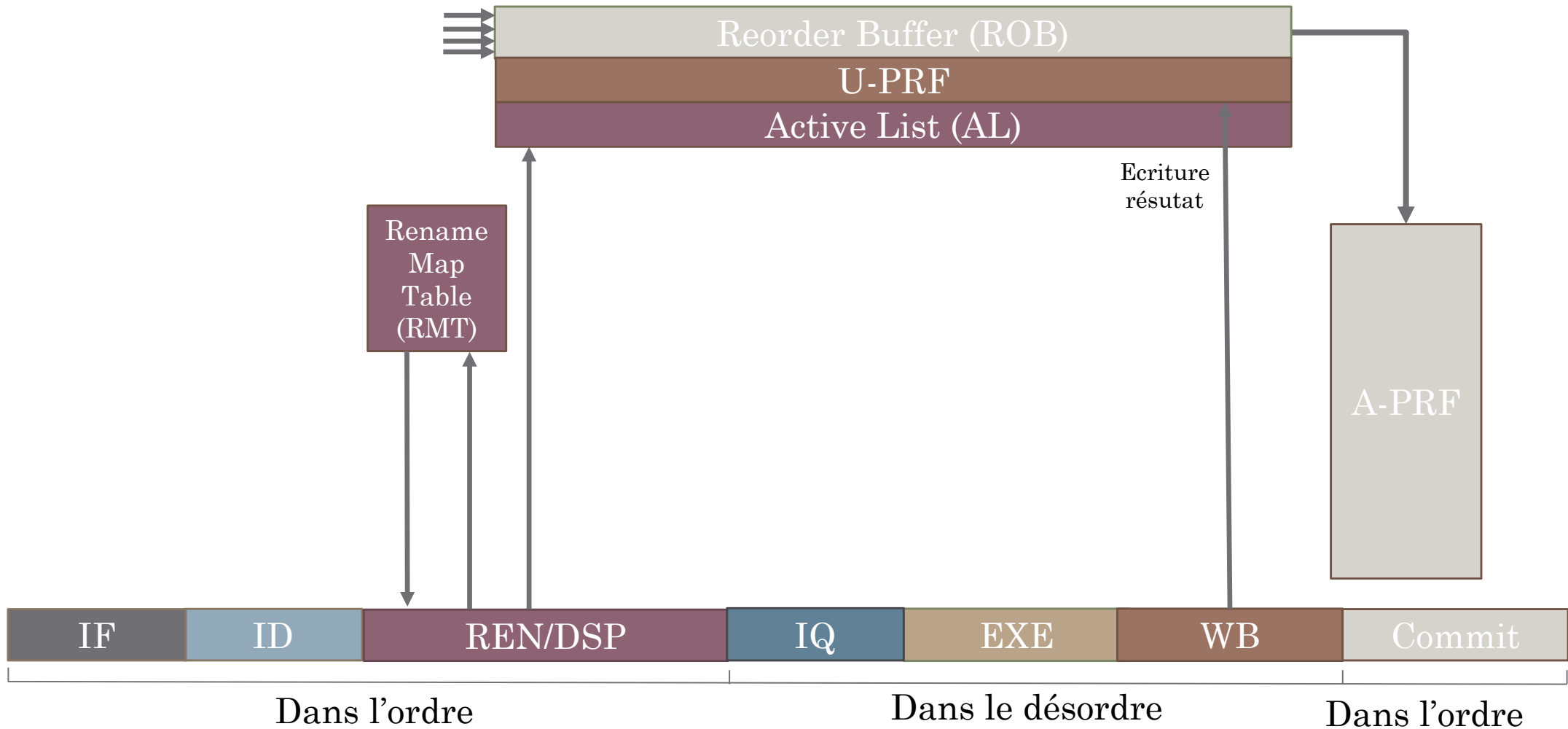
# Renommage de registres



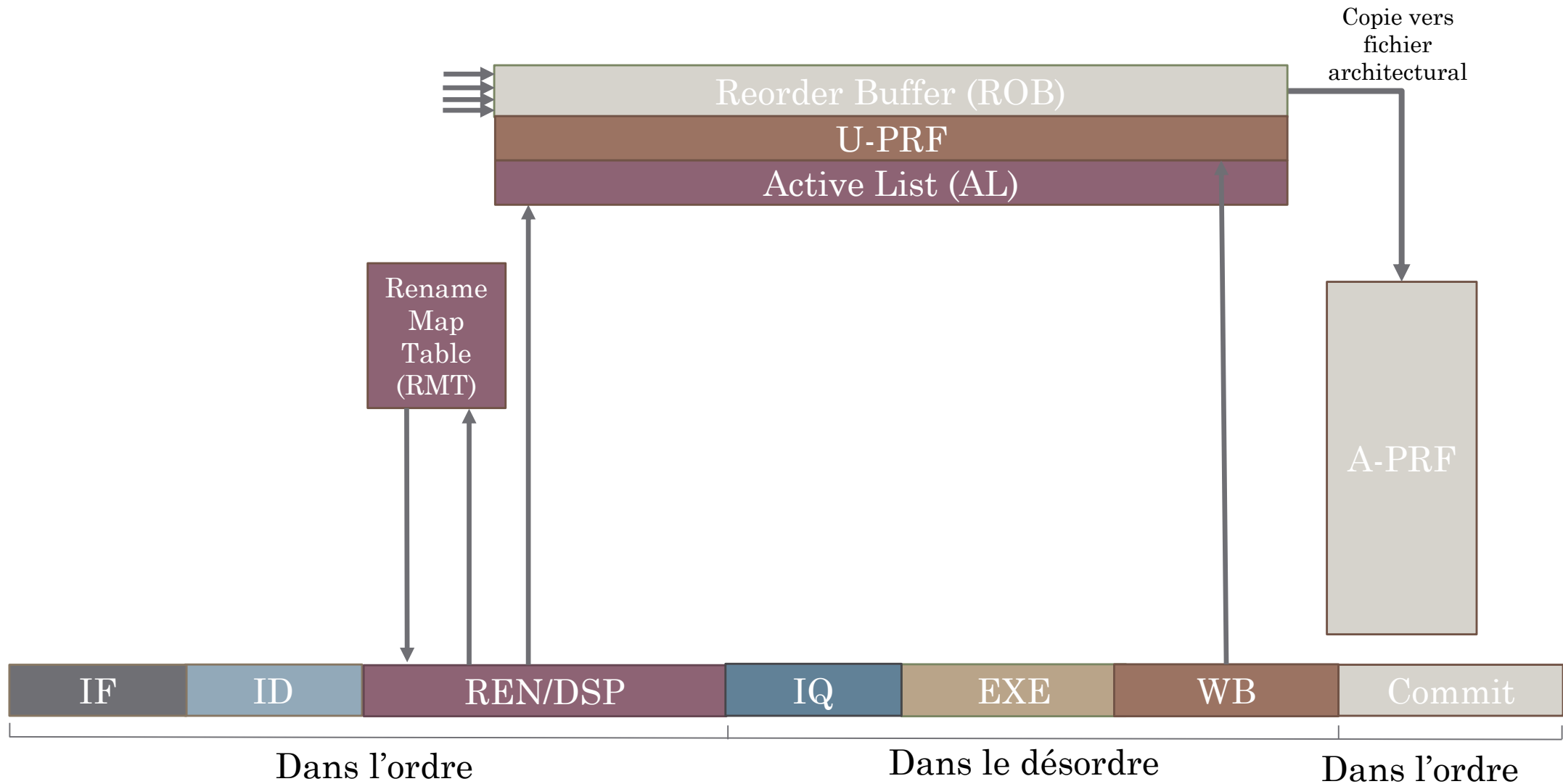
# Renommage de registres



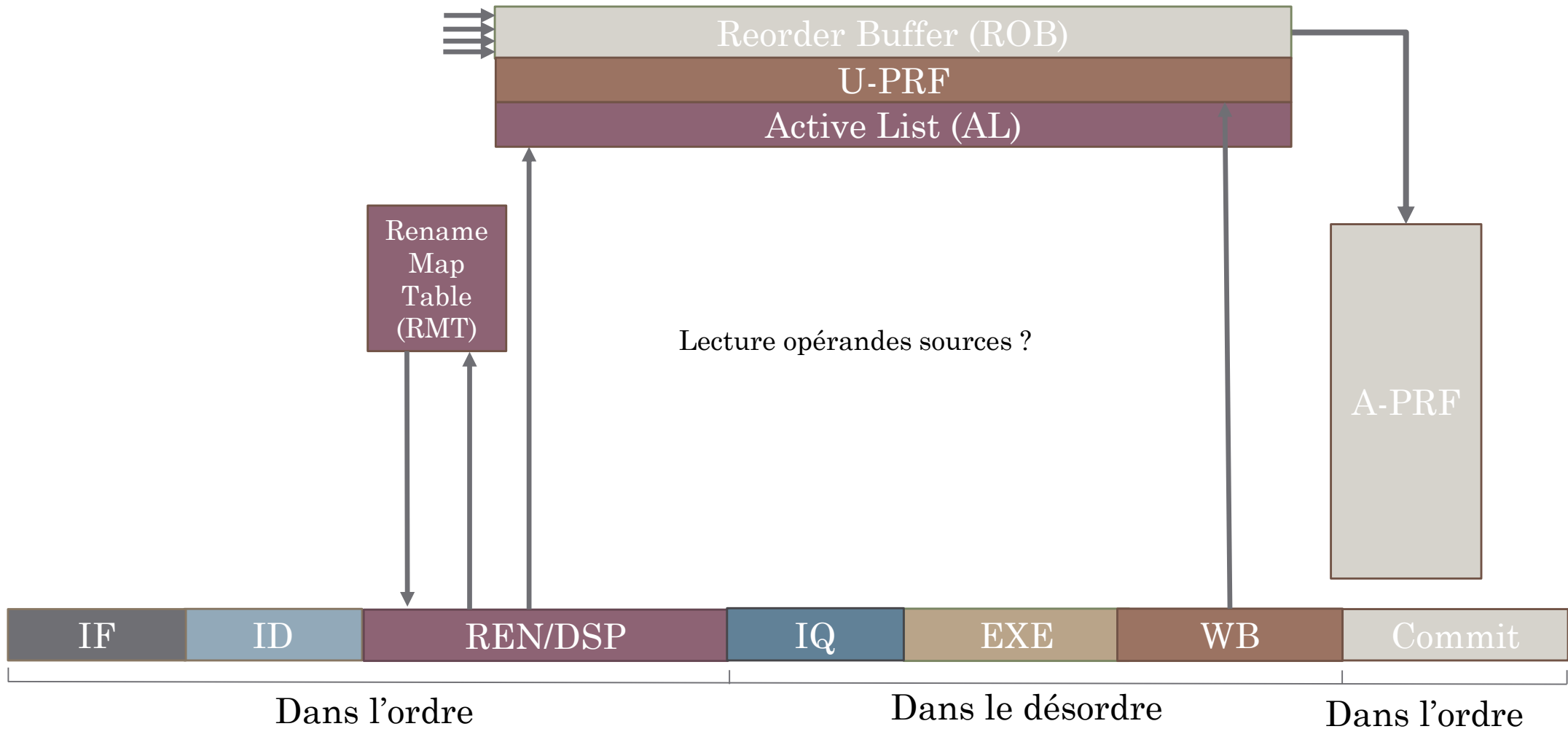
# Renommage de registres



# Renommage de registres



# Renommage de registres

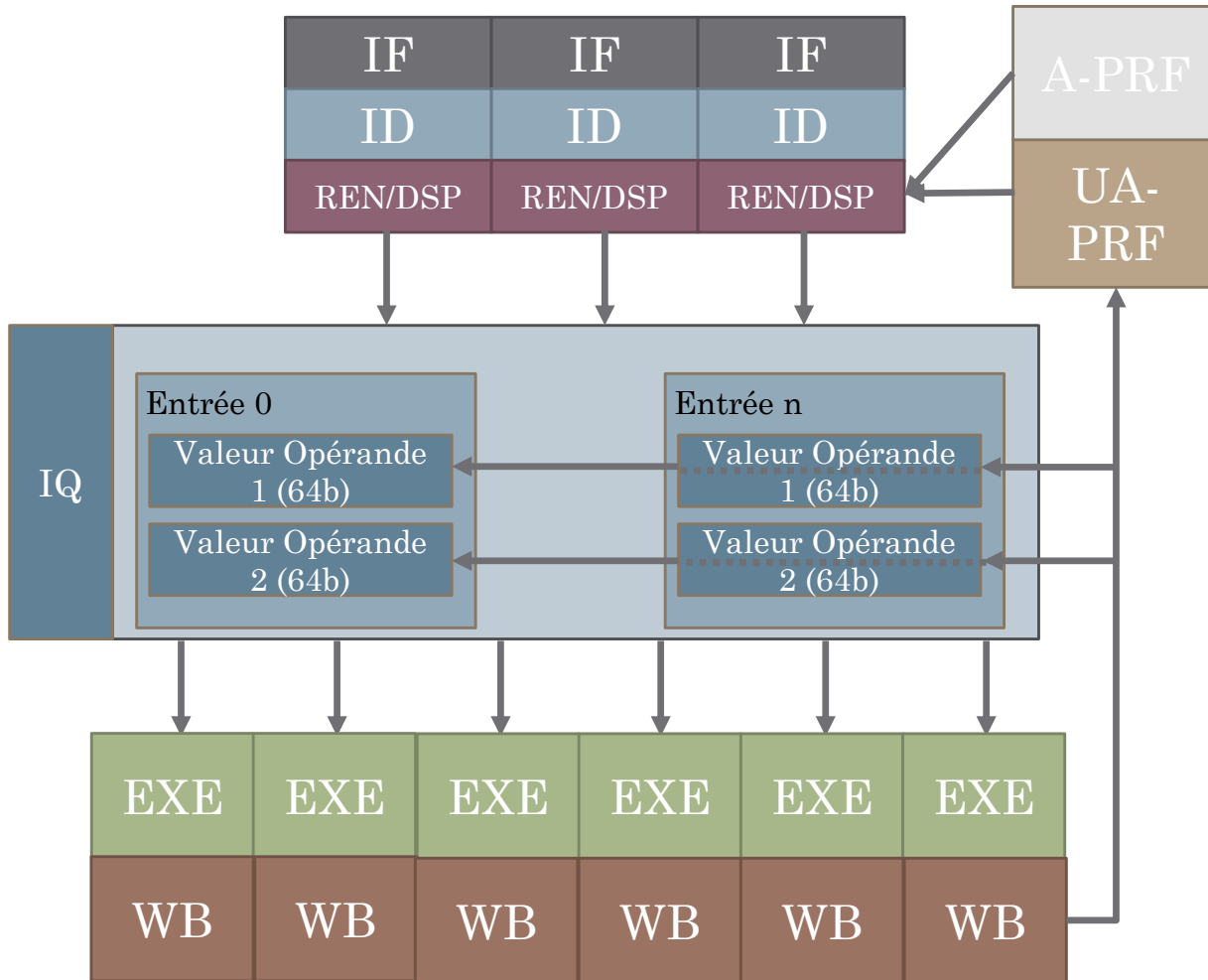




# Lecture des sources

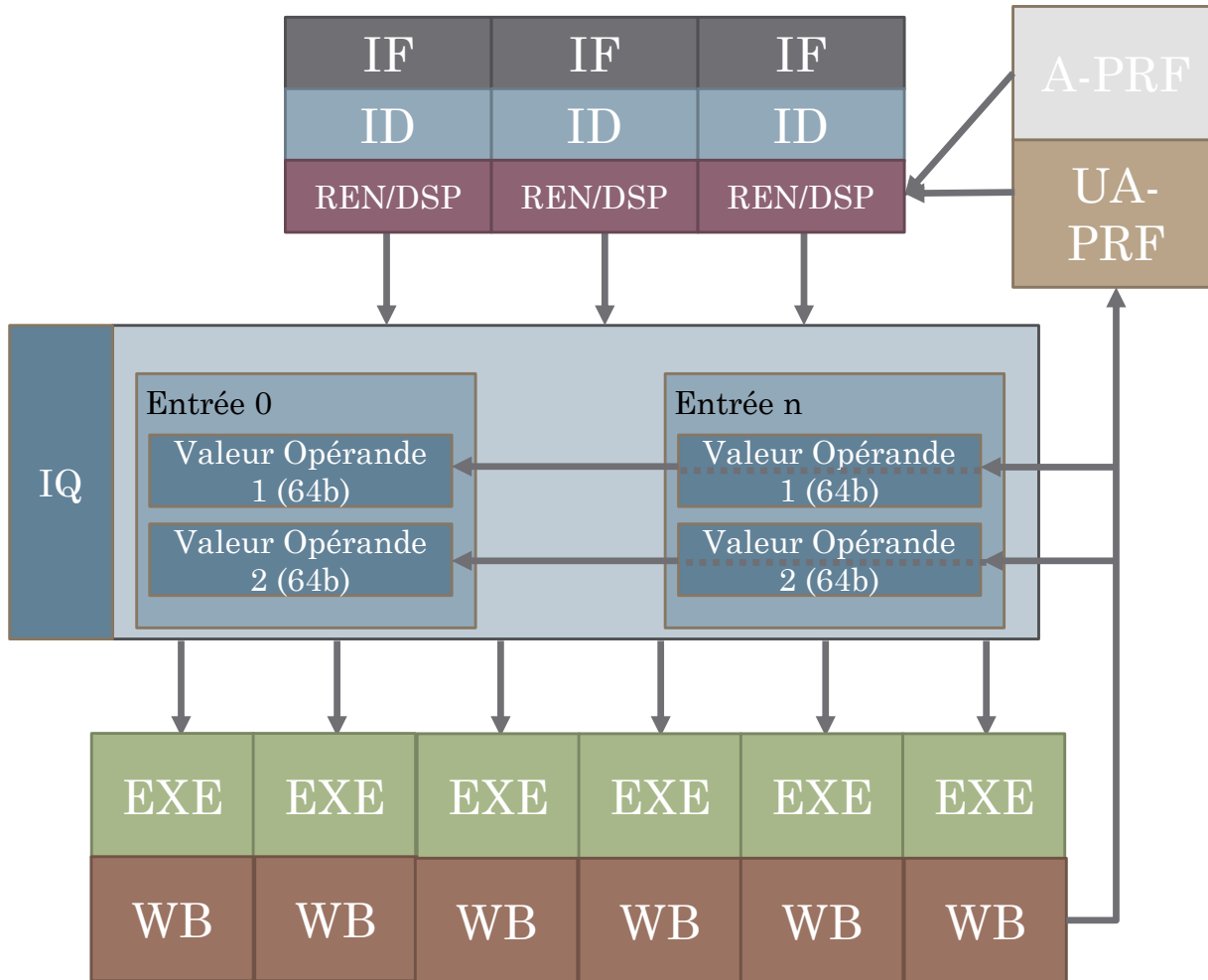
- Pipeline dans l'ordre :
  - Dans ID
  - Sur le bypass
- Dans le désordre :
  - Dans REN/DSP, après avoir renommé les sources
  - Sur le bypass
- L'ordonnanceur doit aussi capturer les résultats qui arrivent sur le réseau de bypass, sinon le résultat est perdu pour les consommateurs

# Ordonnanceur « value capture »



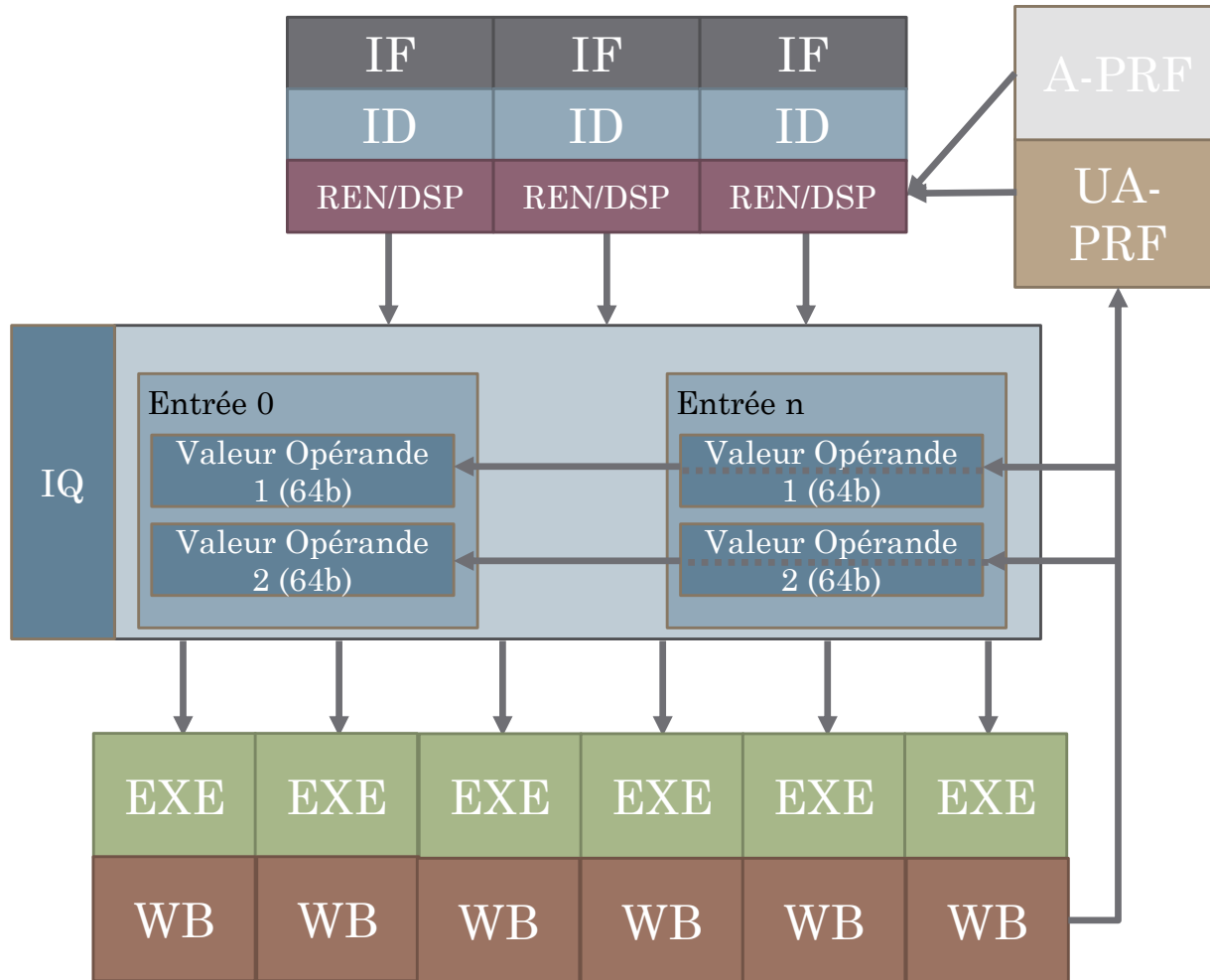
- L'opérande est lue depuis un registre arch. ou uarch. dans DSP **si le registre est prêt**
  - La RMT informe si la valeur est toujours dans un registre uarch. ou a été copiée dans un registre arch.

# Ordonnanceur « value capture »



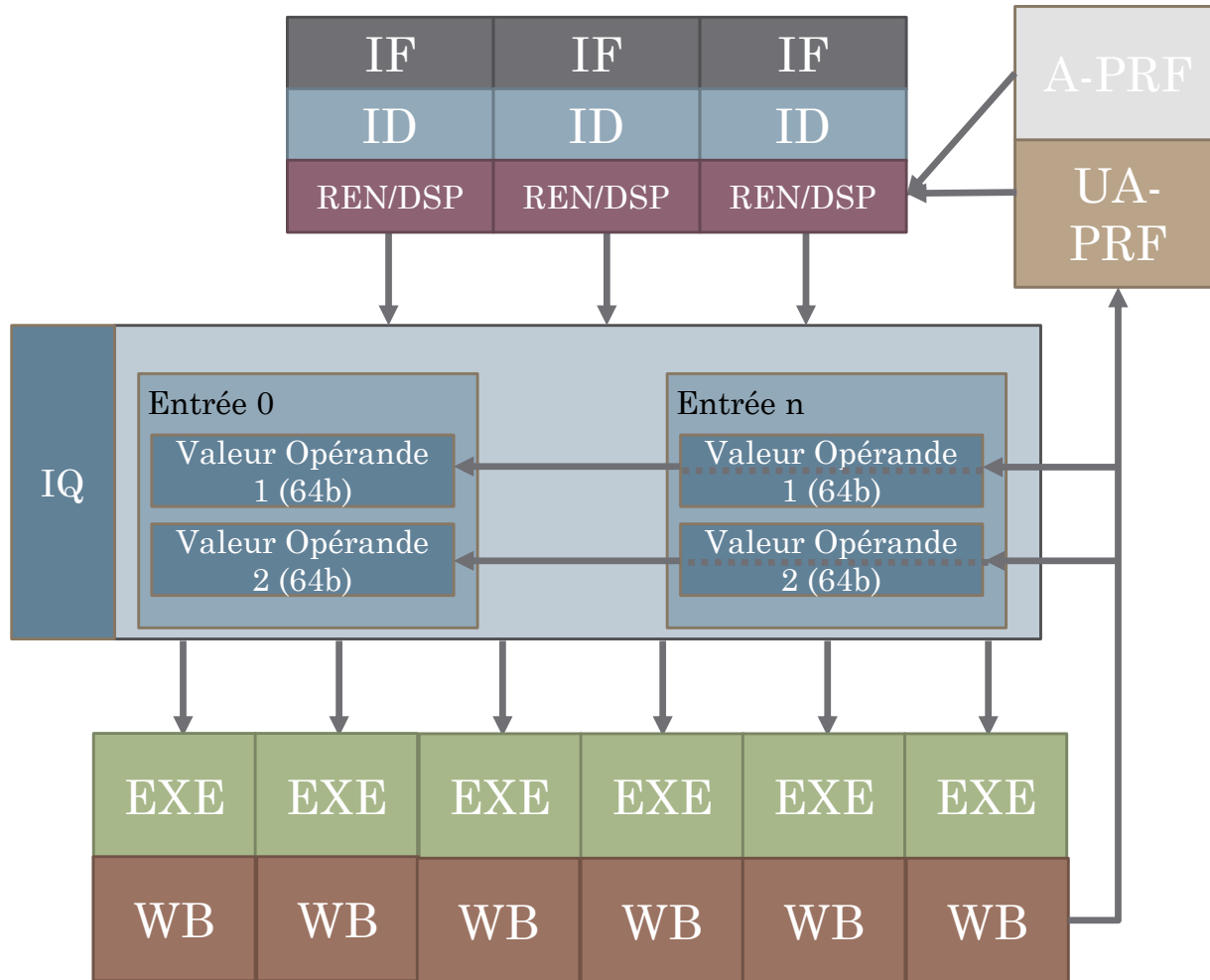
- L'opérande est lue depuis un registre arch. ou uarch. dans DSP **si le registre est prêt**
  - La RMT informe si la valeur est toujours dans un registre uarch. ou a été copiée dans un registre arch.
- Sinon, le registre est produit alors que le consommateur attend dans IQ

# Ordonnanceur « value capture »



- L'opérande est lue depuis un registre arch. ou uarch. dans DSP **si le registre est prêt**
  - La RMT informe si la valeur est toujours dans un registre uarch. ou a été copiée dans un registre arch.
- 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 arch. ou uarch. dans DSP **si le registre est prêt**
  - La RMT informe si la valeur est toujours dans un registre uarch. ou a été copiée dans un registre arch.
- 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 arch. ou uarch. 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 (arch ou uarch)
  - Sera capturé par l'entrée de l'IQ/directement lu sur le bypass

# Lecture des sources

- « La RMT informe si la valeur est toujours dans un registre uarch. ou a été copiée dans un registre arch. »

ROB/PRF/AL

...	sub x2, x8, x9
...	pr2
...	x2/pr12

- sub x2, x8, x9 est la plus vieille instruction dans le pipeline, est exécutée

Rename Map Table (RMT)

arch	phy
x0	pr4
x1	pr90
x2	pr2
x3	pr42

# Lecture des sources

- « La RMT informe si la valeur est toujours dans un registre uarch. ou a été copiée dans un registre arch. »

ROB/PRF/AL

...	Invalid
...	pr2
...	Invalid

- sub x2, x8, x9 est la plus vieille instruction dans le pipeline, est exécutée
- Commit :
  - pr2 copié dans x2

Rename Map Table (RMT)

x0	pr4
x1	pr90
x2	pr2
x3	pr42





# Lecture des sources

- « La RMT informe si la valeur est toujours dans un registre uarch. ou a été copiée dans un registre arch. »

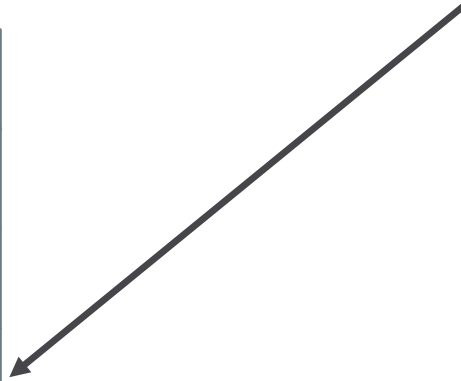
ROB/PRF/AL

...	Invalid
...	pr2
...	Invalid

- sub x2, x8, x9 est la plus vieille instruction dans le pipeline, est exécutée
- Commit :
  - pr2 copié dans x2
  - Si pr2 est la version la plus récente de x2, la RMT doit pointer vers x2, et plus vers pr2

Rename Map Table (RMT)

arch	phy
x0	pr4
x1	pr90
x2	x2
x3	pr42



# Lecture des sources

- « La RMT informe si la valeur est toujours dans un registre uarch. ou a été copiée dans un registre arch. »

ROB/PRF/AL

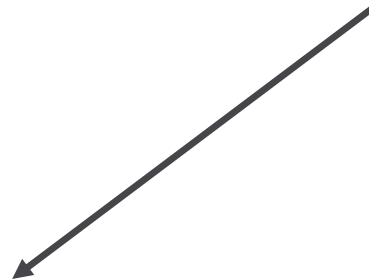
addi x2, x1, 1	...
pr256	...
x2/pr2	...

- sub x2, x8, x9 est la plus vieille instruction dans le pipeline, est exécutée
- Commit :
  - pr2 copié dans x2
  - Si pr2 est la version la plus récente de x2, la RMT doit pointer vers x2, et plus vers pr2

Rename Map Table (RMT)

arch	phy
x0	pr4
x1	pr90
x2	<b>pr256</b>
x3	pr42

- Si une nouvelle instruction écrivant x2 est renommée, la RMT pointe de nouveau vers un registre microarchitectural



# Lecture des sources

- « La RMT informe si la valeur est toujours dans un registre uarch. ou a été copiée dans un registre arch. »

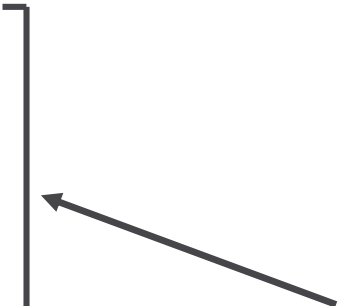
ROB/PRF/AL

Invalid	...
pr256	...
Invalid	...

- sub x2, x8, x9 est la plus vieille instruction dans le pipeline, est exécutée
- Commit :
  - pr2 copié dans x2
  - Si pr2 est la version la plus récente de x2, la RMT doit pointer vers x2, et plus vers pr2

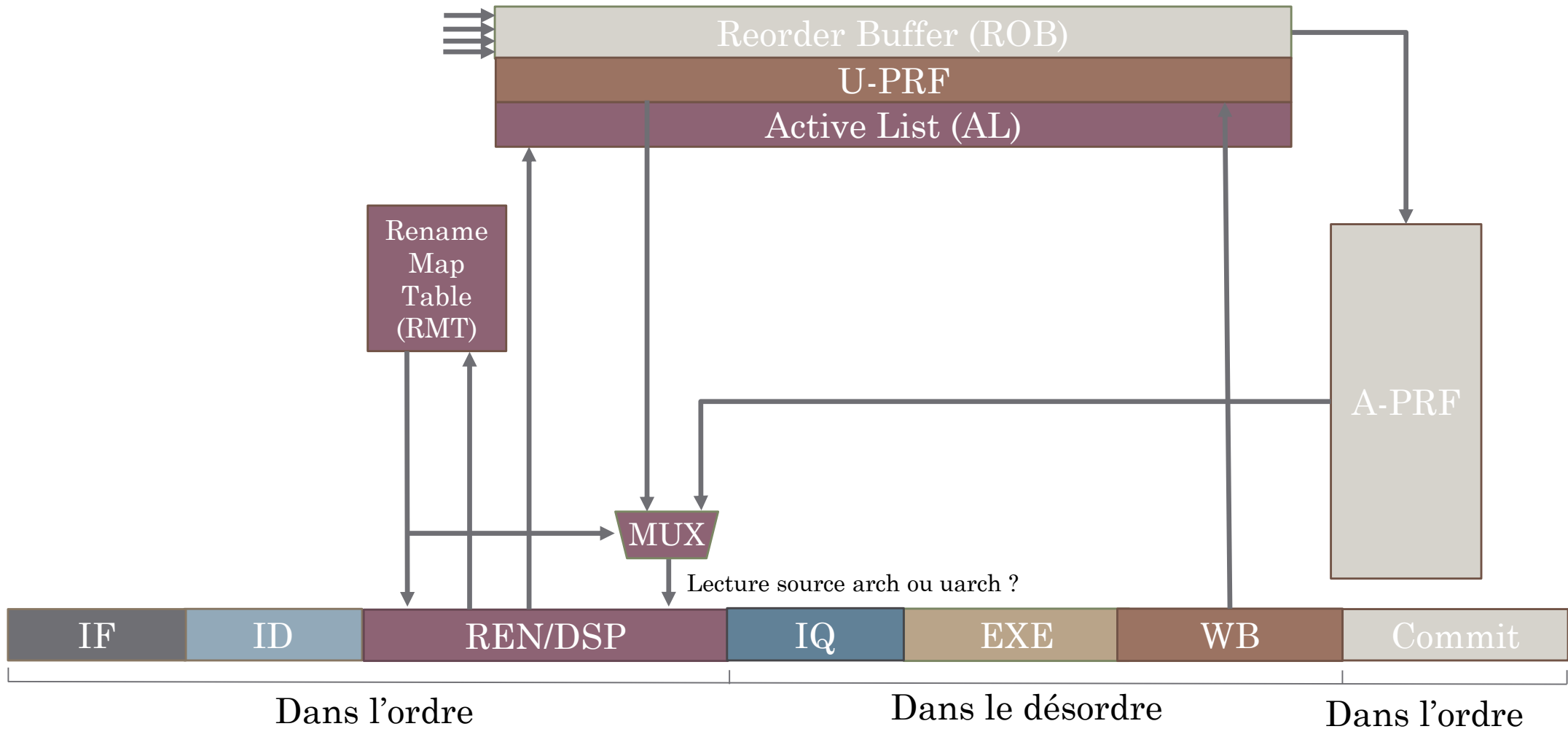
Rename Map Table (RMT)

arch	phy
x0	<b>x0</b>
x1	<b>x1</b>
x2	<b>x2</b>
x3	<b>x3</b>

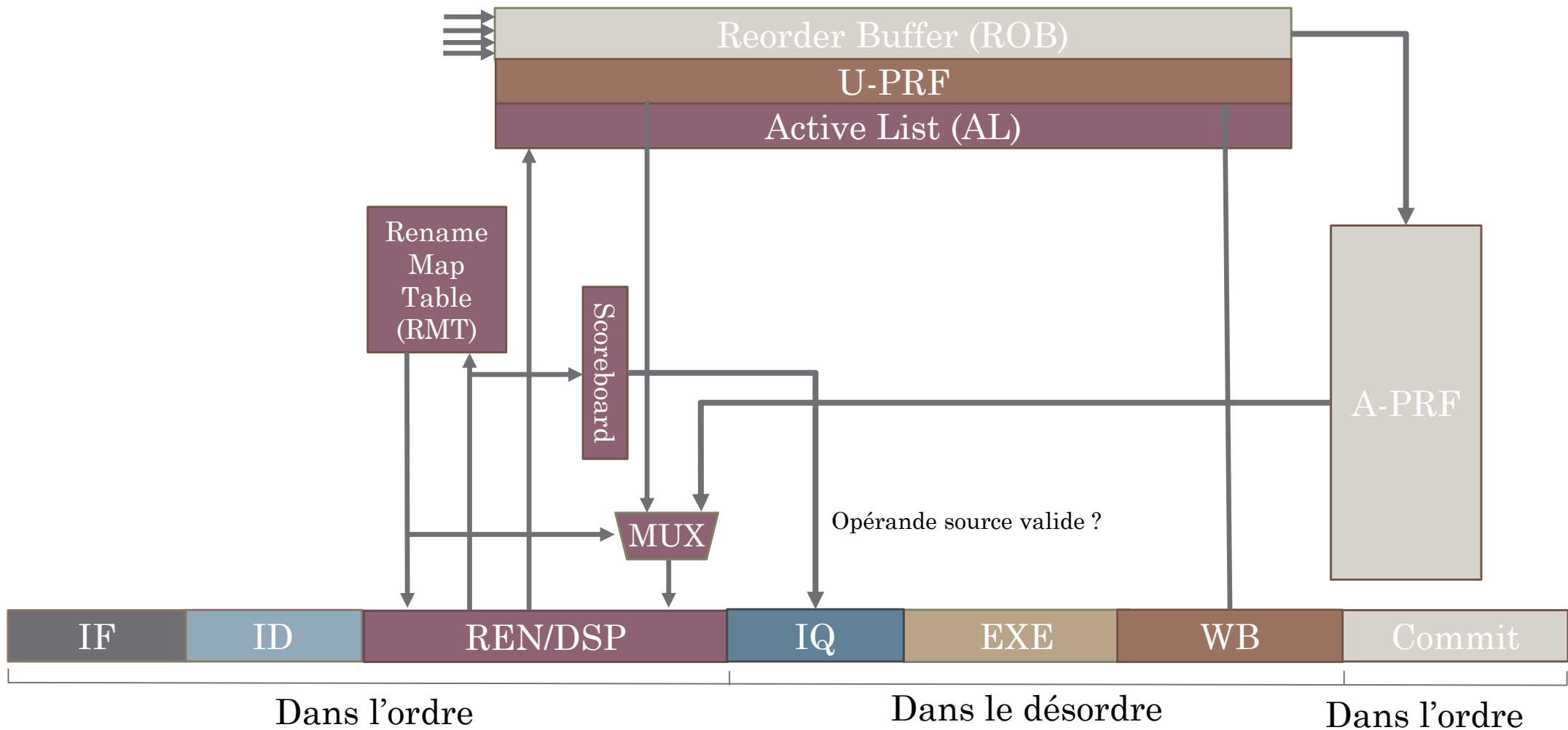


- Si une nouvelle instruction écrivant x2 est renommée, la RMT pointe de nouveau vers un registre microarchitectural
- Quand le pipeline est complètement vide, tous les registre sont dans le fichiers de registres architectural

# Renommage de registres



# Renommage de registres



# Travaux pratiques

## ROB/AL/U-PRF

### RMT

r0	pr10	1
x1	pr4	1
x2	pr15	1
x3	pr1	1
x4	pr2	1
x5	pr13	1
r6	pr0	1
r7	pr5	1

### Code

```
I1: add x1, r0, r0
I2: add x2, x1, x1
I3: add x4, x3, x3
I4: sub x5, r6, r7
I5: add x3, x4, x2
I6: sub x1, x1, x2
I7: or x1, r0, r0
I8: sll r6, x4, r7
I9: srl r7, r6, r0
```

### Questions :

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

\*On considère que le ROB est rempli avant que I1 passe Commit

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

↑  
Pointeur de tête : on commence à insérer les instructions ici

# Exécution dans le désordre

Mémoire

# Exécution dans le désordre et accès mémoires

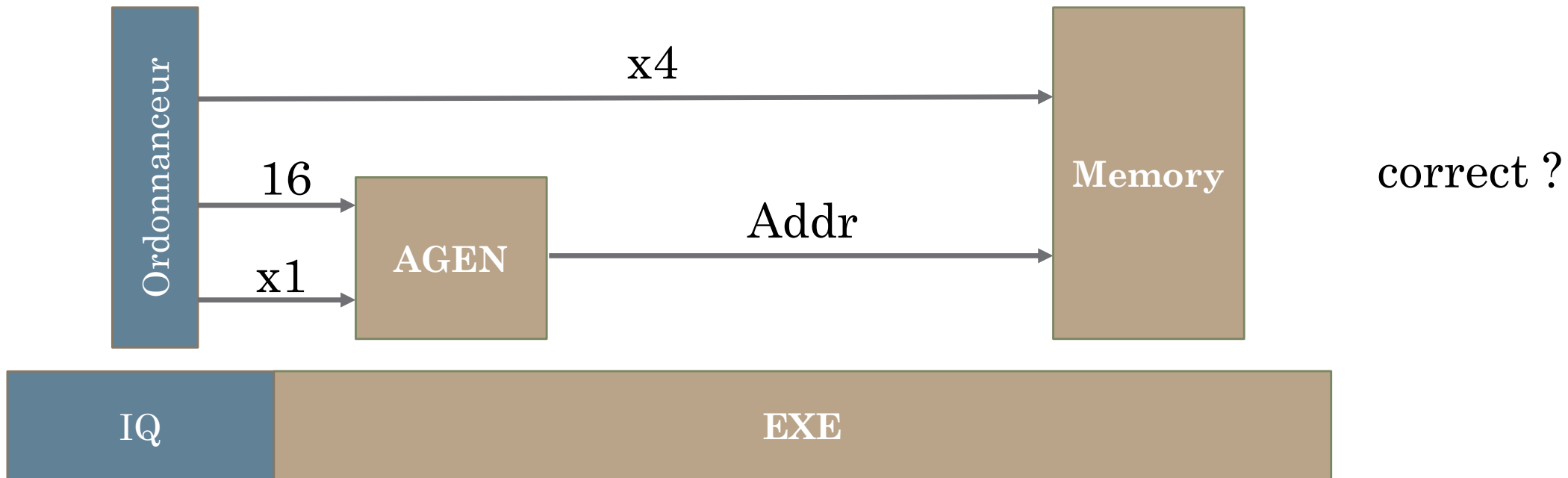
- Jusqu'ici, on s'est intéressé uniquement à la gestion des registres et aux dépendances de données via les registres
- Pas suffisant pour les accès mémoires
  - Ecritures
  - Lectures



# Exécution dans le désordre – Stores

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

sd x4, 16(x1)



# Exécution dans le désordre – Stores

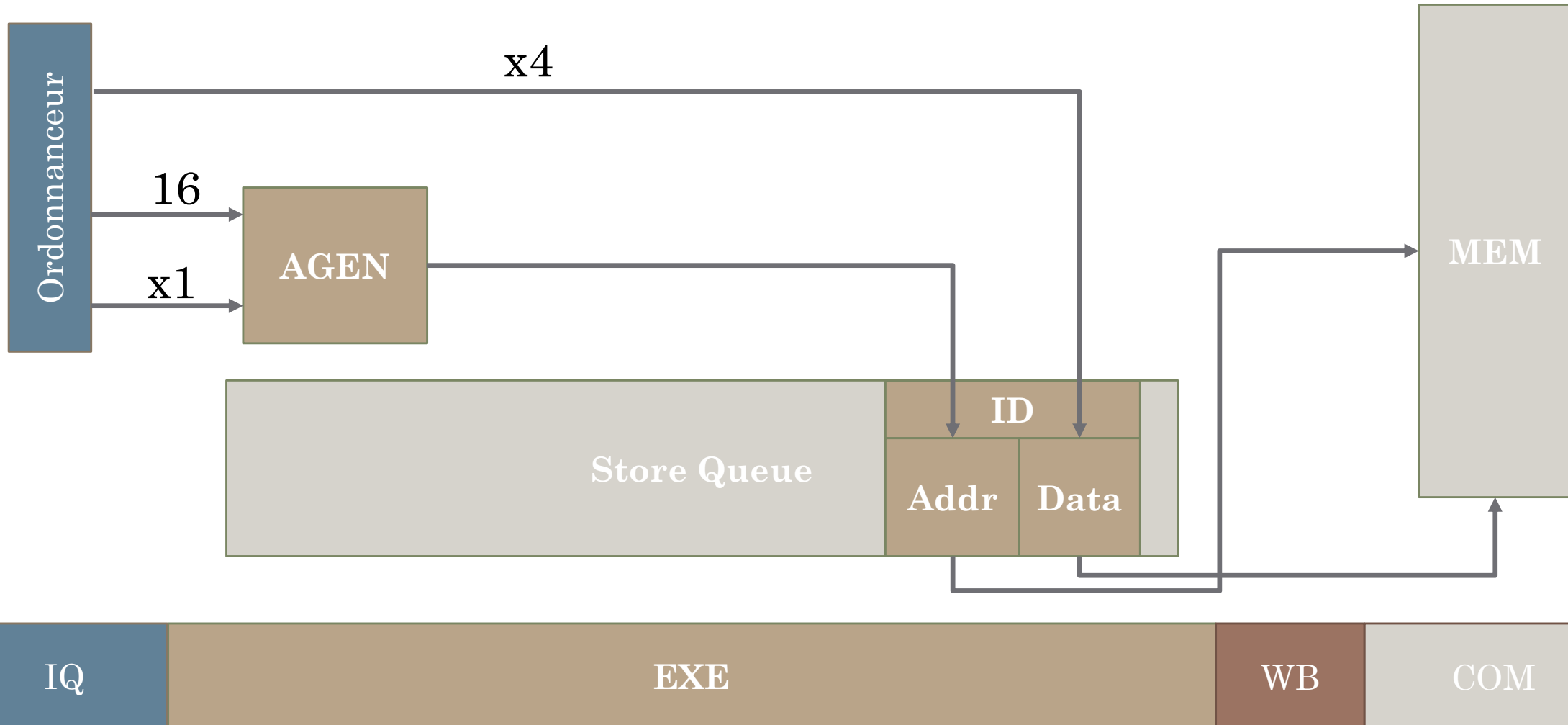
- La mémoire fait partie de l'état architectural (que le programmeur peut observer)
- Tant qu'une instruction n'a pas passé le Commit, elle est spéculative : elle sera peut-être jetée du pipeline
- Ecrire dans la mémoire spéculativement = effets du store observables architecturalement avant Commit
 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

# Exécution dans le désordre – Stores

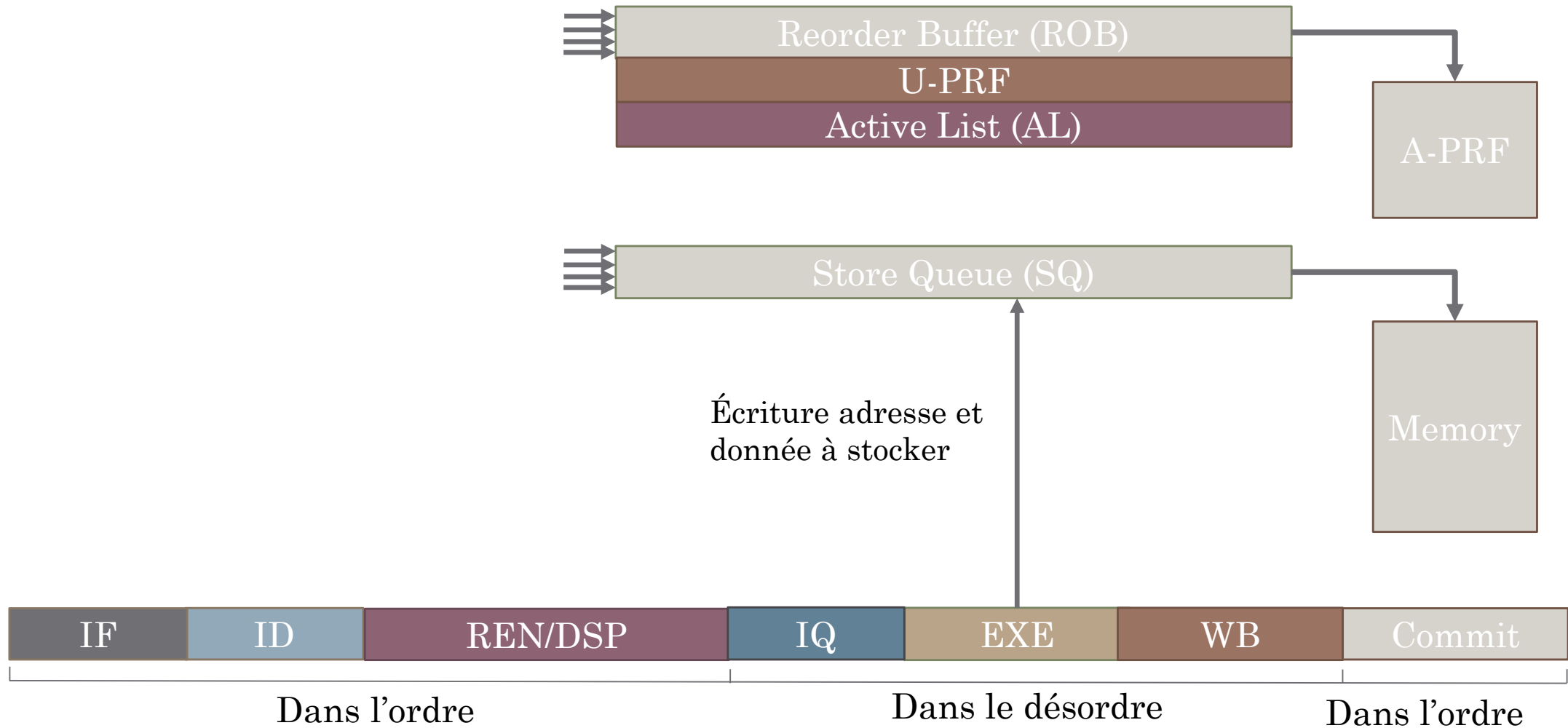
- Exécuter les stores au Commit
  - Dans l'ordre donc on ne peut pas cacher la latence d'exécution via exécution dans le désordre
- On introduit donc une nouvelle mémoire matérielle : La Store Queue (FIFO)
  - Peut être vue comme un sous-ensemble du ROB spécialisé pour les stores
  - Stocke l'adresse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

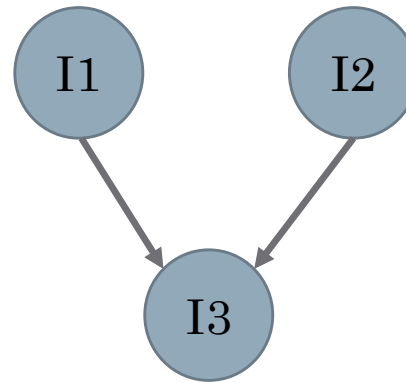
# Exécution dans le désordre – Mémoire

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

I1 : sd x0, 0(x1)

I2 : sd x4, 16(x2)

I3 : ld x5, 0(x6)



→ Dépendances RaW  
*potentielle*

- Quelle est la bonne valeur pour x5 ?



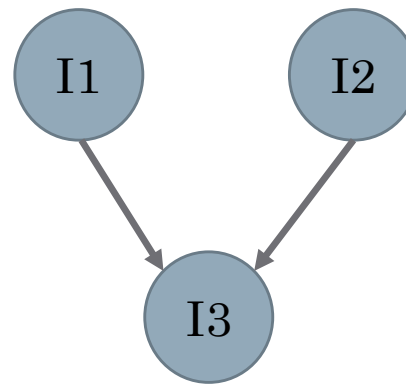
# Exécution dans le désordre – Mémoire

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

I1 : sd x0, 0(x1)

I2 : sd x4, 16(x2)

I3 : ld x5, 0(x6)



→ Dépendances RaW  
*potentielle*

- Quelle est la bonne valeur pour x5 ?
  - Dépend de si  $(x2 + 16) == x1$  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 : une instruction ne peut s'exécuter que si ses dépendances sont satisfaites, i.e. :
  - Après l'exécution de tous les stores plus vieux
  - Après l'exécution du load

# Exécution dans le désordre – Mémoire

- Problème 1 : une instruction ne peut s'exécuter que si ses dépendances sont satisfaites, i.e. :
  - Après l'exécution de tous les stores plus vieux
  - Après l'exécution du load
- 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 les stores spécifiques seront exécutés

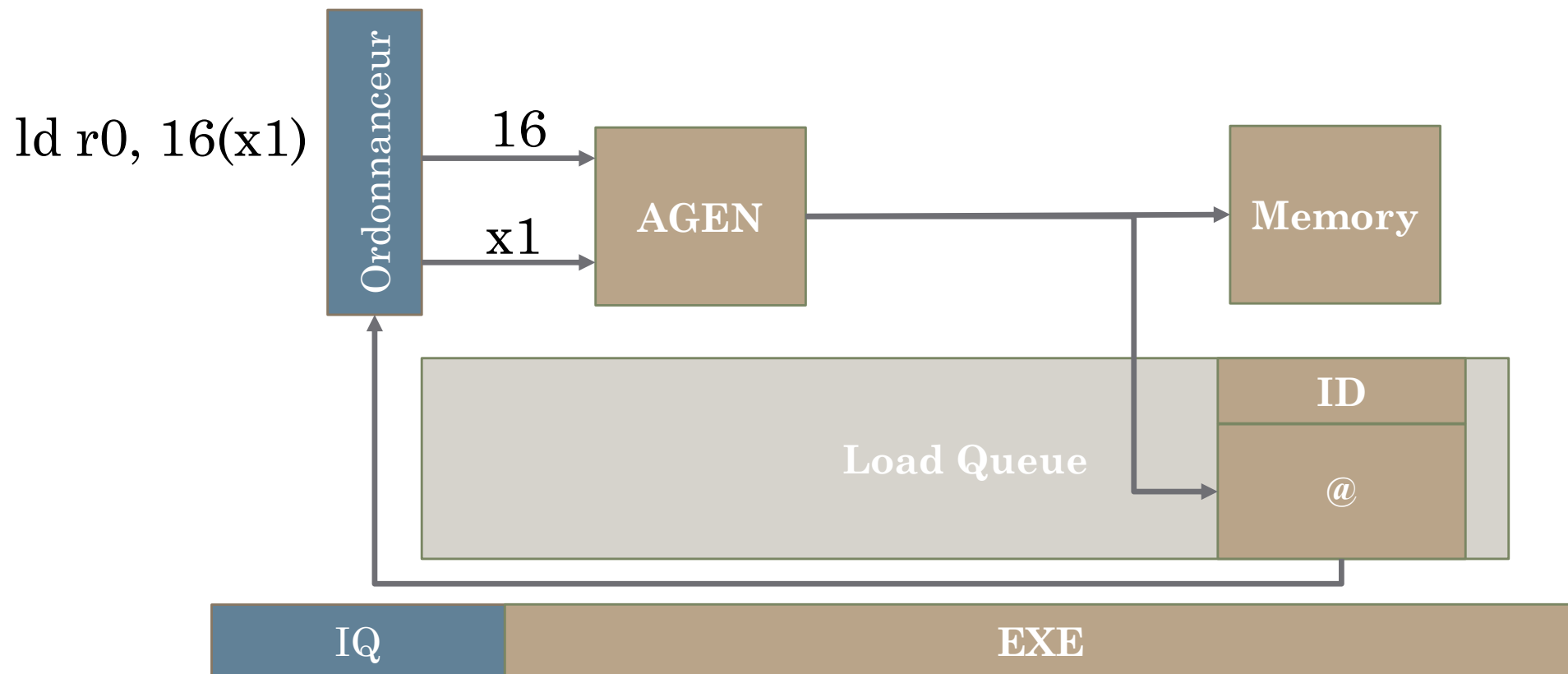
# Dépendances mémoire

- 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 ?

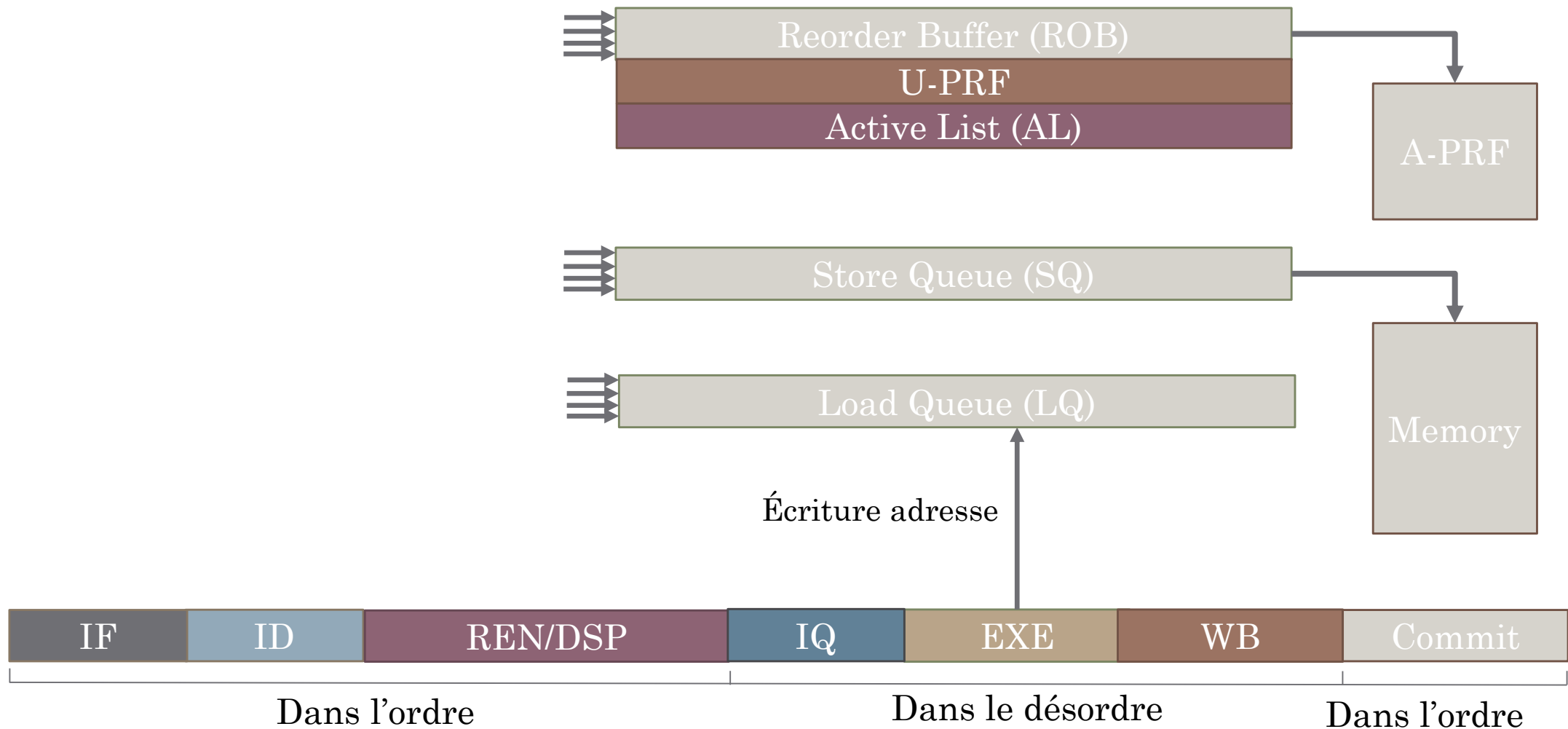


# Dépendances mémoire – Load Queue

- 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

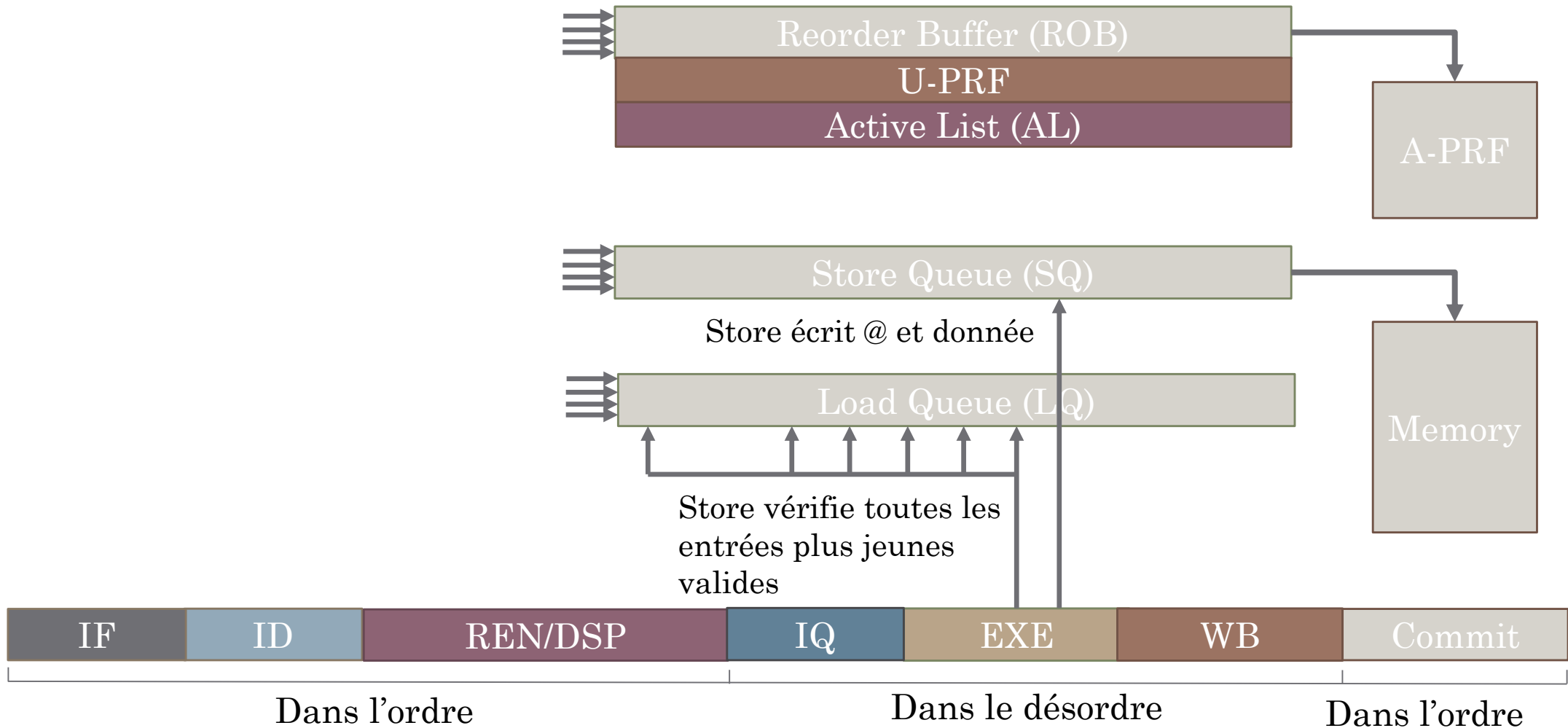


# Dépendances mémoire

- 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 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 ?
  - Le load doit de toute façon attendre que le store écrive la mémoire (= Commit) avant de lire la mémoire

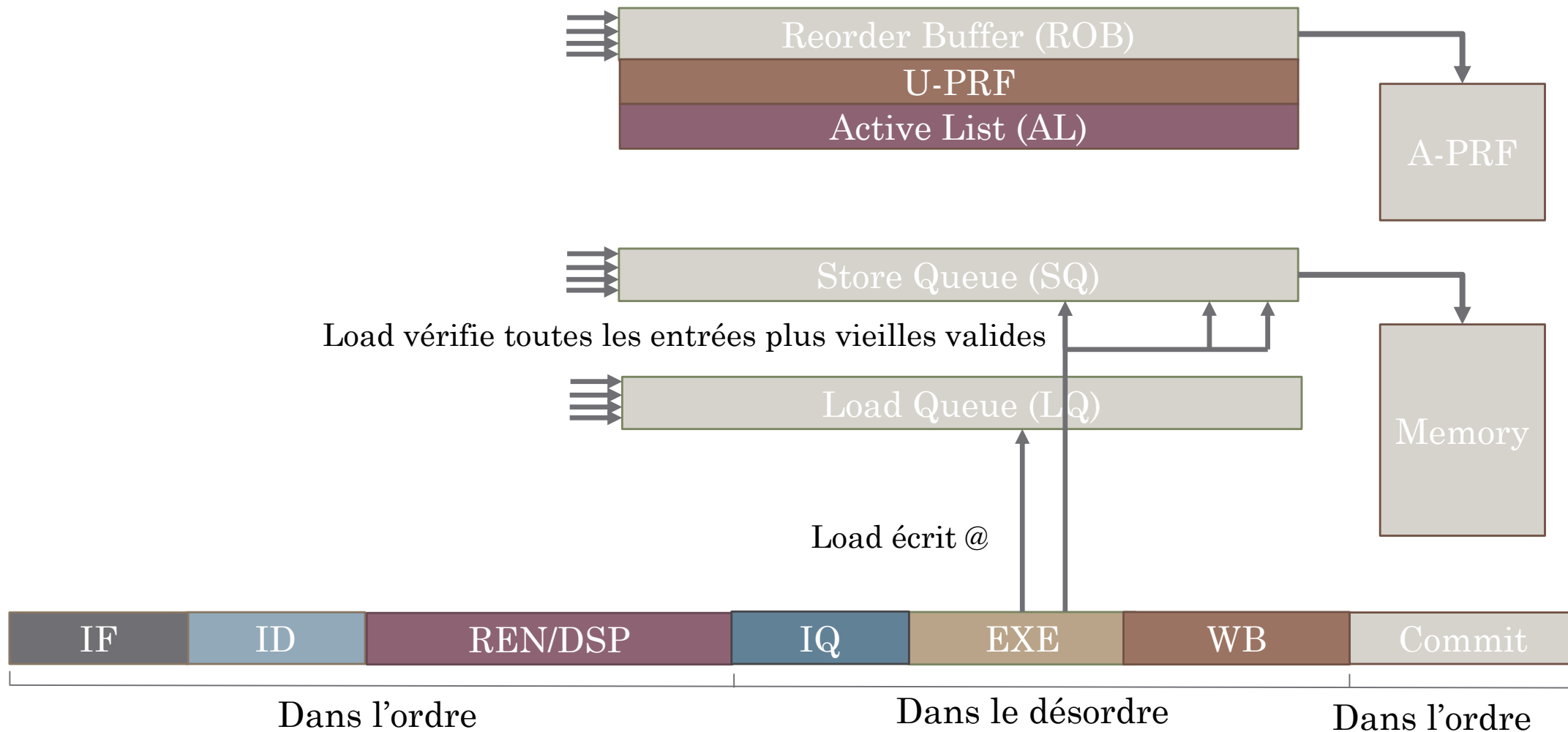
# Dépendances mémoire - Spéculation

- 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 ?

# 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

# Dépendances mémoire - Spéculation





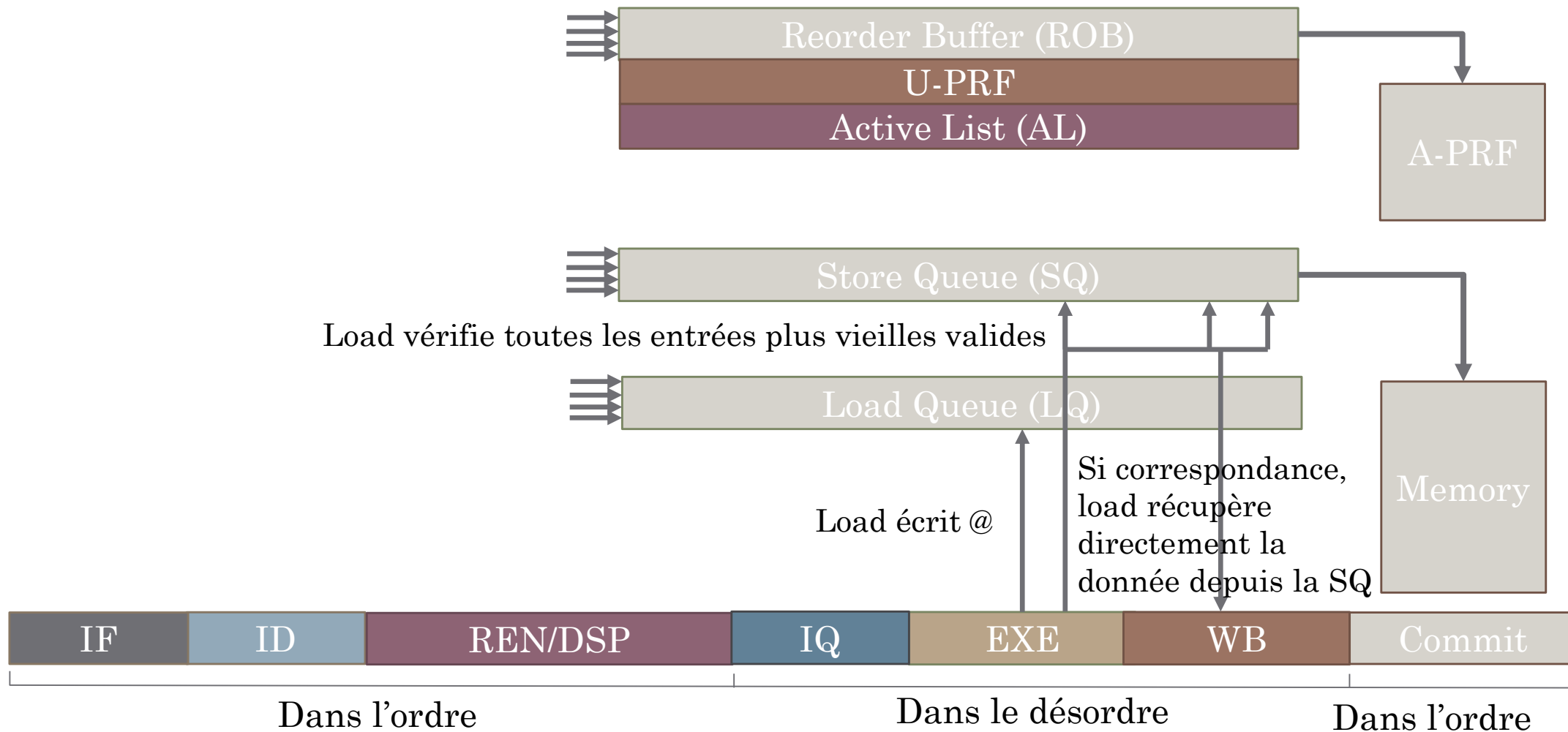
# Dépendances mémoire - Spéculation

- 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 ?

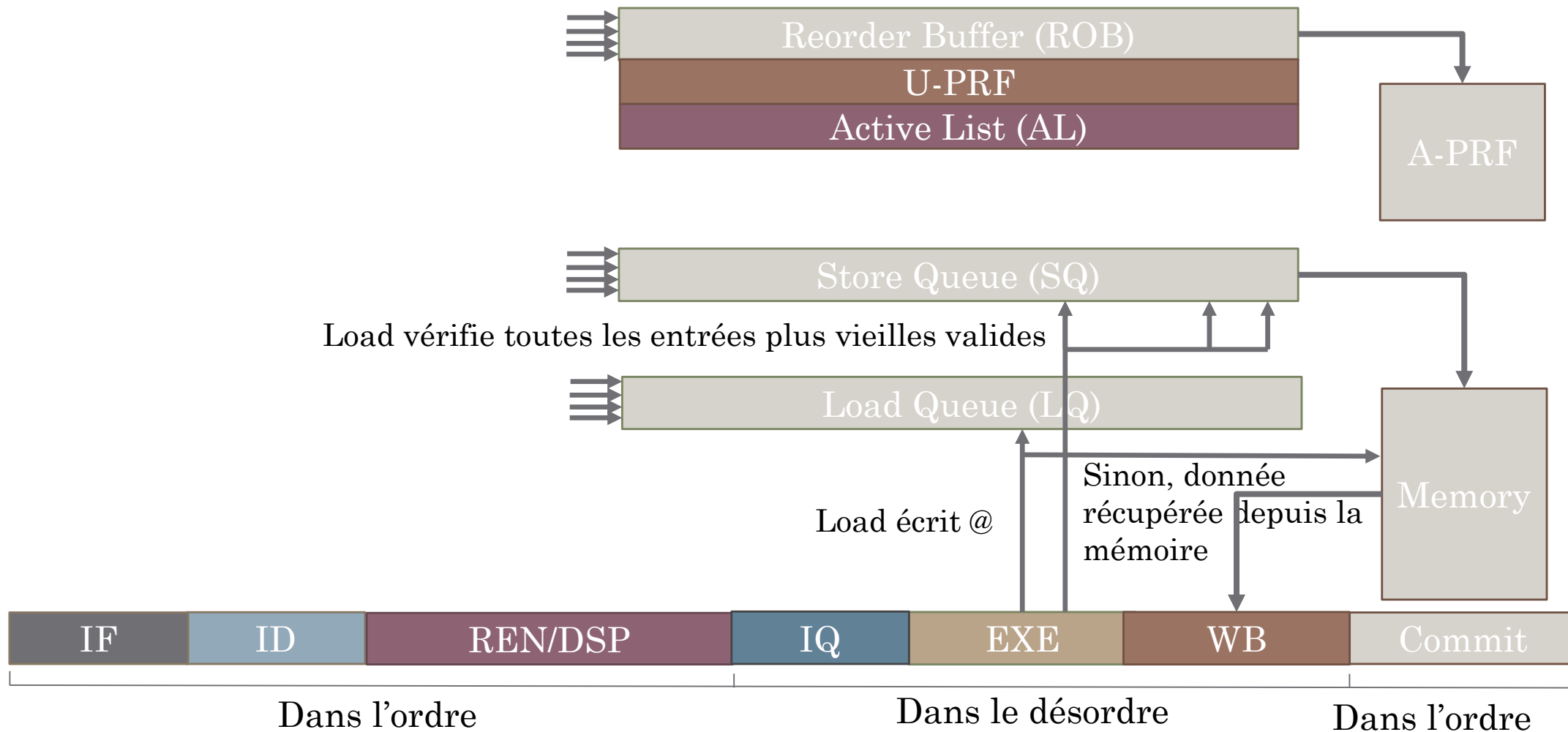
# Dépendances mémoire - Spéculation

- 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



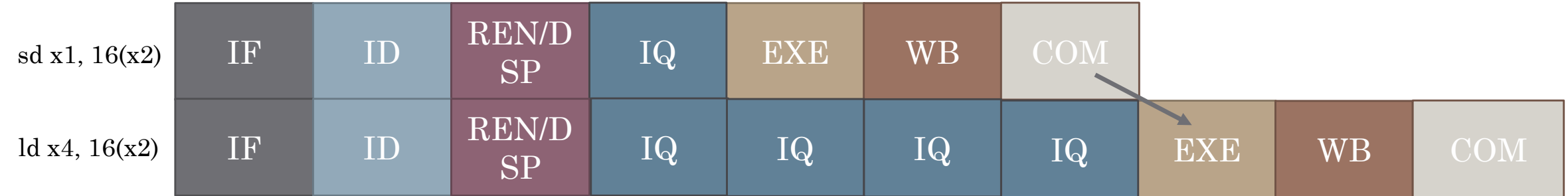
# Dépendances mémoire - Spéculation



# STLDF = Bypass mais pour la mémoire

Sans STLDF

Écrit mémoire



Lit mémoire

Avec STLDF

Écrit SQ



Lit 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 ?



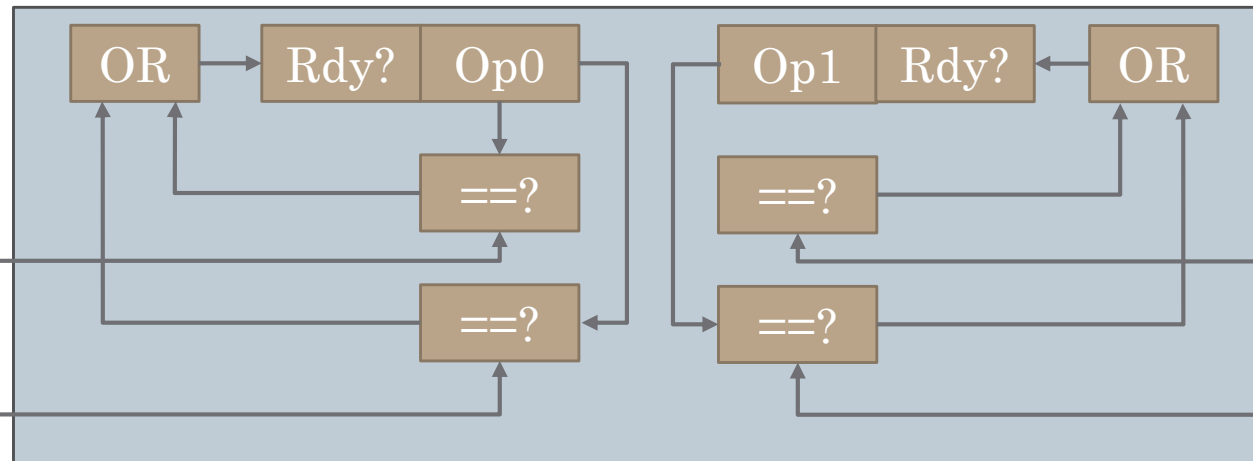
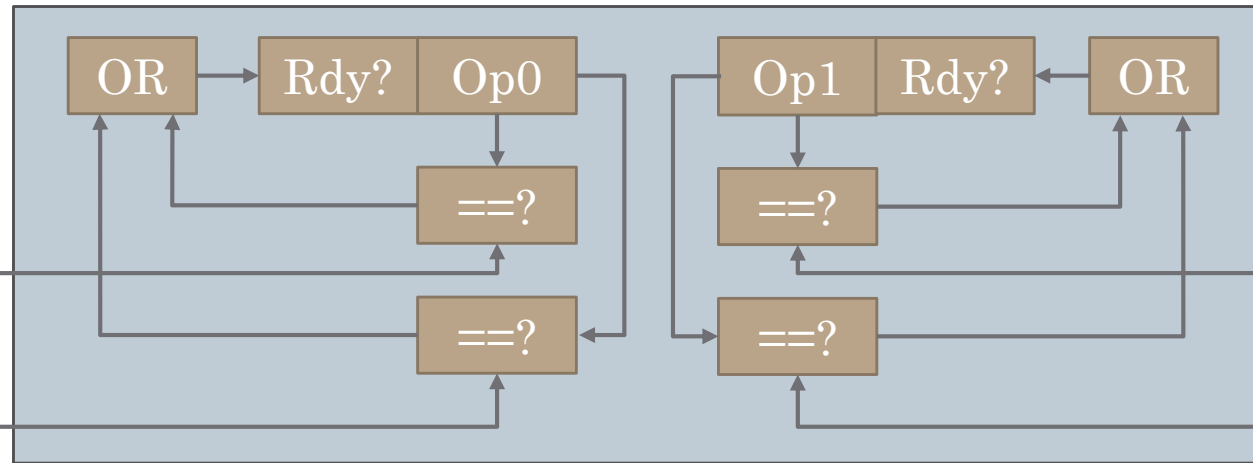
# Ordonnanceur

- 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 fonctionelles

# Ordonnanceur – Wakeup

pregx pregx

pregx pregx



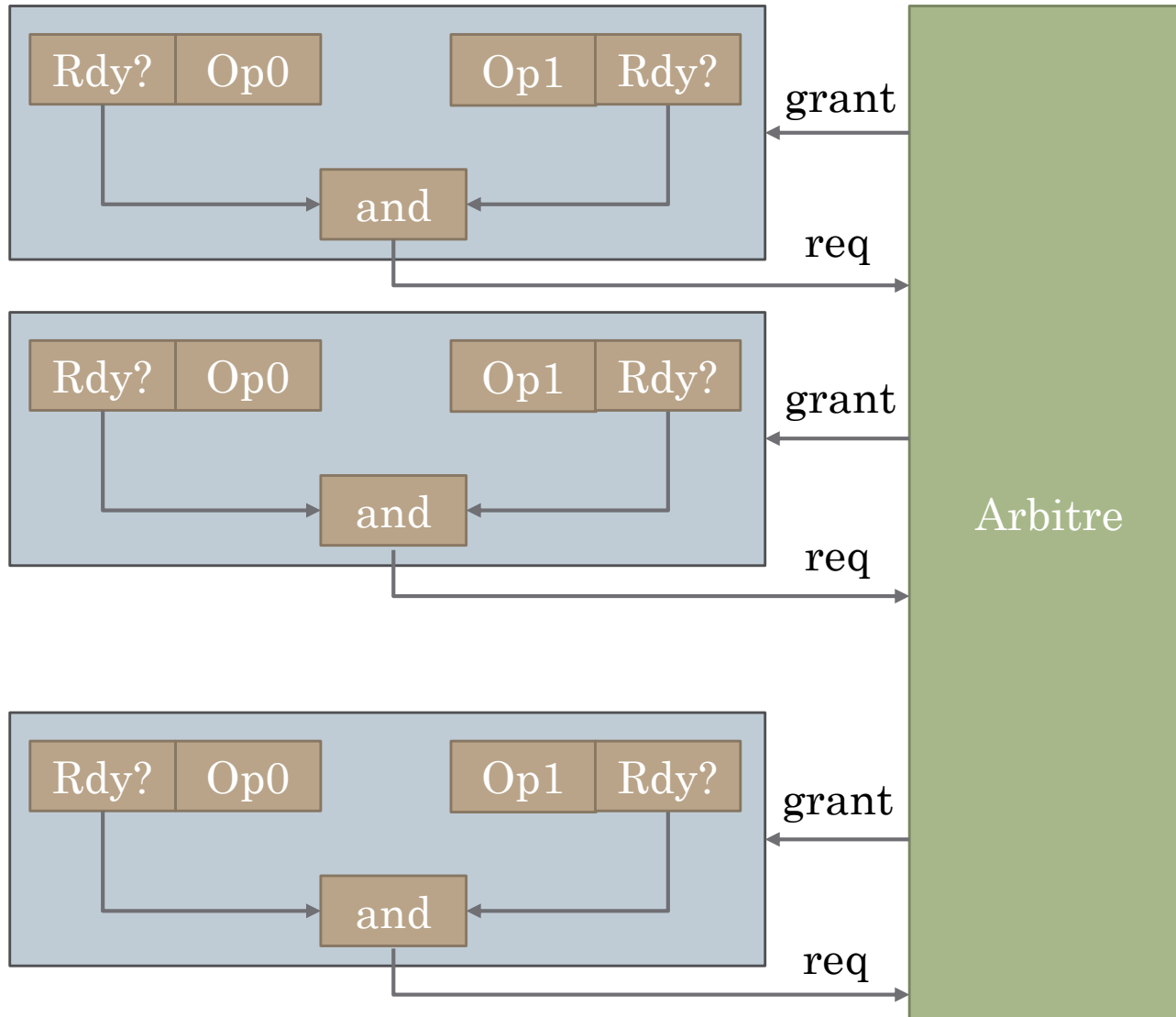
**Chaque cycle**, les reg. phy. Nouvellement prêts sont envoyés à **chaque entrée** de l'ordonnanceur

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.

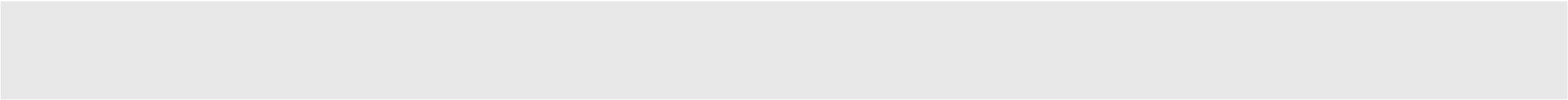
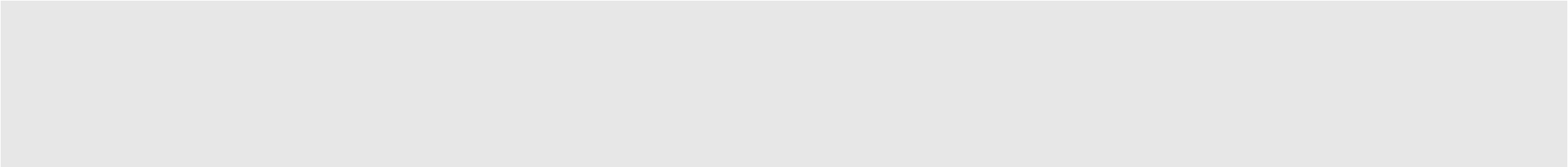
# Exécution dans le désordre

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









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

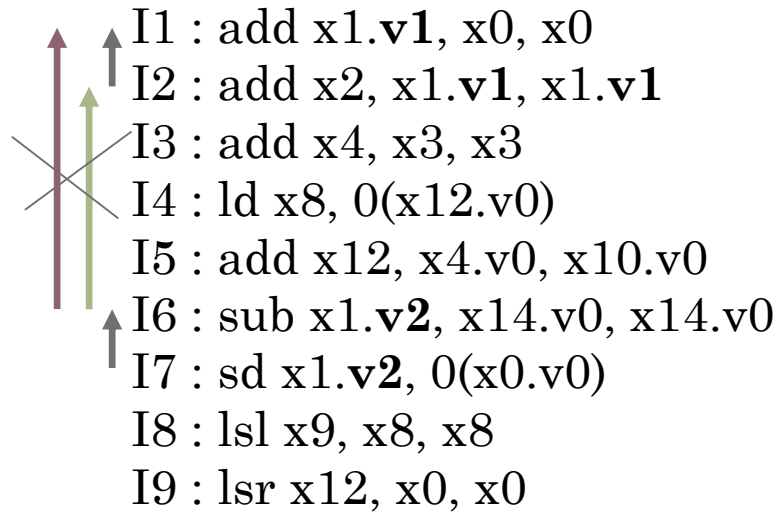
# Questions ?

# Exécution superscalaire Exécution dans le désordre

SEOC3A – CEAMC

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

# Renommage de registres



Chaque écrivain de x1 (I1, I6) obtient un registre physique afin d'écrire une nouvelle version de x1. Durant sa durée de vie, un registre physique est **écrit une fois, lu  $n$  fois**

