Mise en oeuvre pipeline du processeur NIOS II

Alain MÉRIGOT

Université Paris Saclay

Principe du traitement pipeline

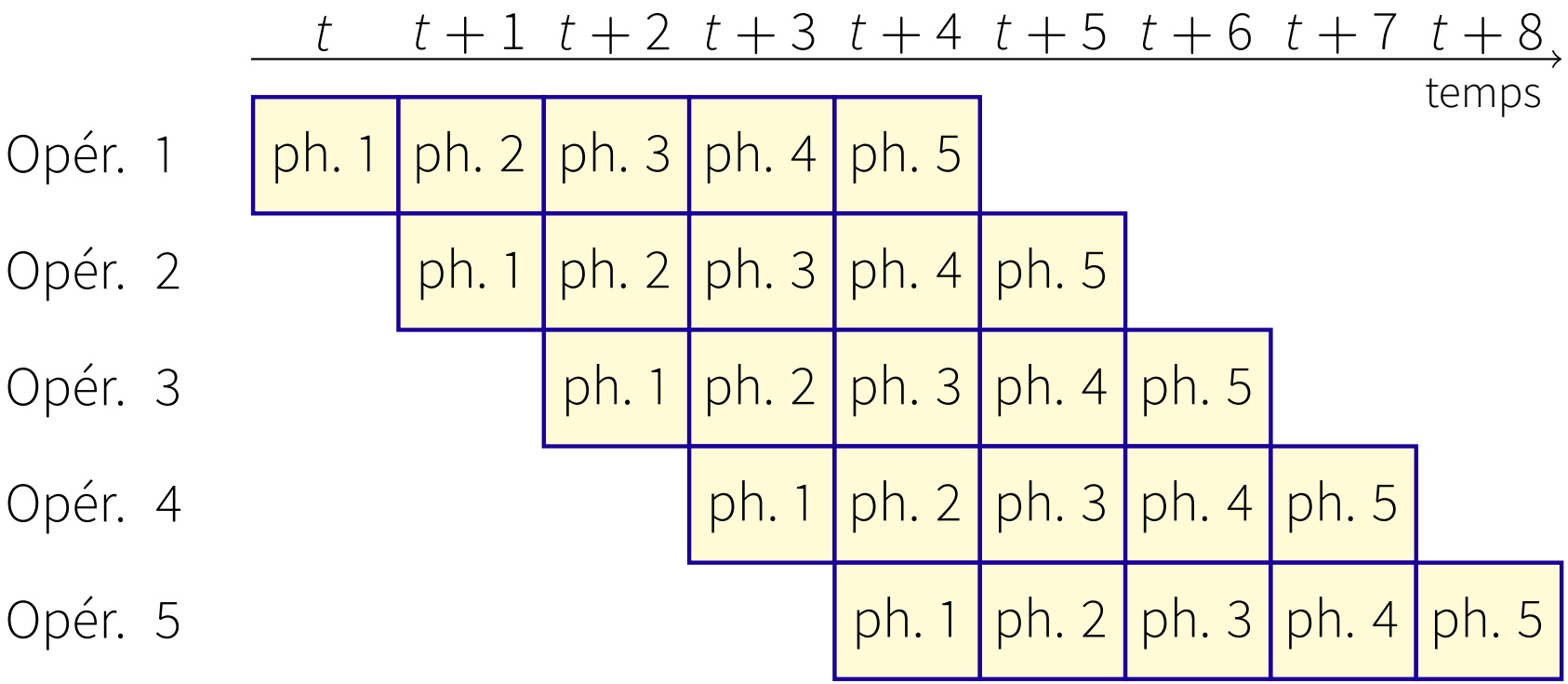
- traitement décomposé en n phases
- tous les traitements ont le même nombre de phases
- pas d'interférences entre phases

$\rightarrow$ possibilité de pipeline

On démarre une instruction, sans attendre la fin de l'exécution de la précédente.

Lancement d'une opération par cycle
(mais latence inchangée)

Principe du traitement pipeline (cont.)



Après une phase de démarrage, 5 opérations sont simultanément effectuées.

Principe du traitement pipeline (cont.)

Si le pipeline a *n* étages, les performances sont (*presque*) multipliées par *n*.

Contraintes :

- *Toutes* les instructions doivent comporter le même nombre de phases.
- Aucune phase ne doit utiliser des ressources dont une autre phase aurait aussi besoin. (*conflit structurel*)
  Chaque ressource (opérateur, bus, registre) est liée à une phase spécifique.

Nécessité d'adapter l'architecture.

Nécessité de gérer les intéractions éventuelles

Version *pipeline* du nios

Dans le cas du Nios II, différentes durées d'instructions (3 phases pour `br, jmp, ...`, 4 pour instructions ALU `add, addi,...` et `st` et 5 pour `ld`).
→ On passe toutes les instructions à **5 phases**

Des ressources sont utilisées plusieurs fois :

mémoire utilisée par *toutes* les instructions en phase *fetch* et par `ld,st` en phase 4
→ séparation mémoire instructions – mémoire données

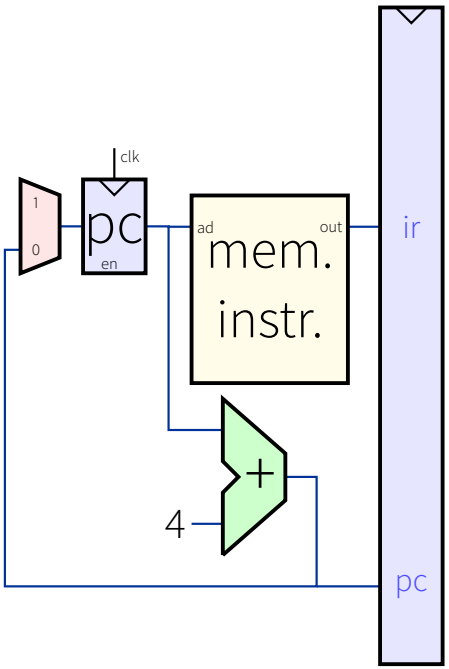
ALU utilisées pour la phase *execute*, et les calculs `pc+4`, `pc+imm16`...
→ ajout d'additionneurs pour les calculs sur `pc`

registre d'instruction `ri` nécessaire à chaque phase pour le contrôle

compteur de programme `pc` nécessaire en phase *fetch* et dans d'autres phases pour `br, jmpi`
→ on copie tous les registres à préserver (`ri`, `pc`, registres tampons `A`, `B`, etc) dans de grands ***registres pipeline***

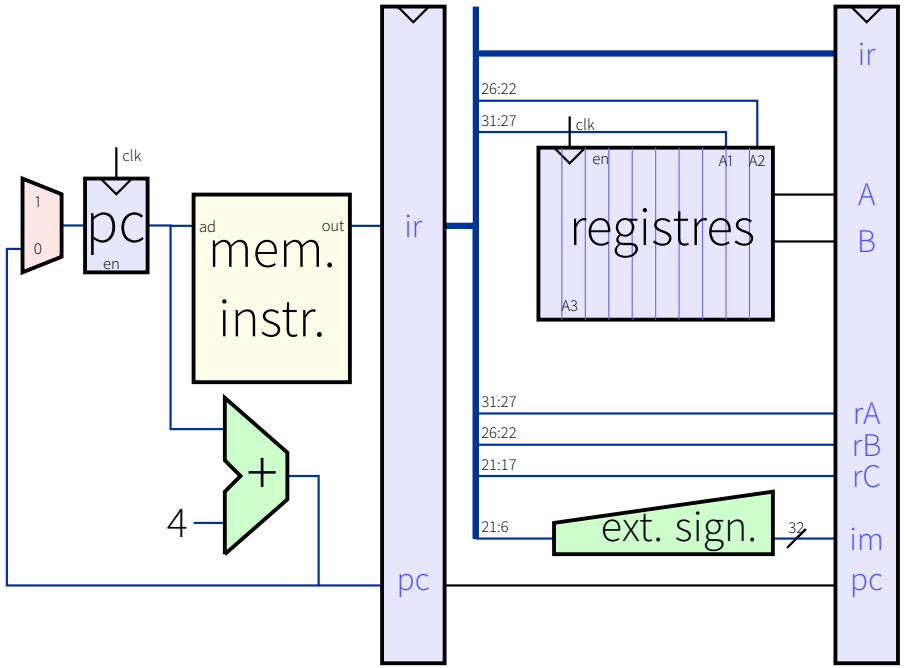
Acquisition de l'instruction

Phase *fetch* ou **lecture instruction** (LI)
On ajoute un additionneur pour l'incrémentation de `pc`.
`ir` et `pc` sont copiés dans un registre pipeline à la fin de cette phase.



Décodage de l'instruction

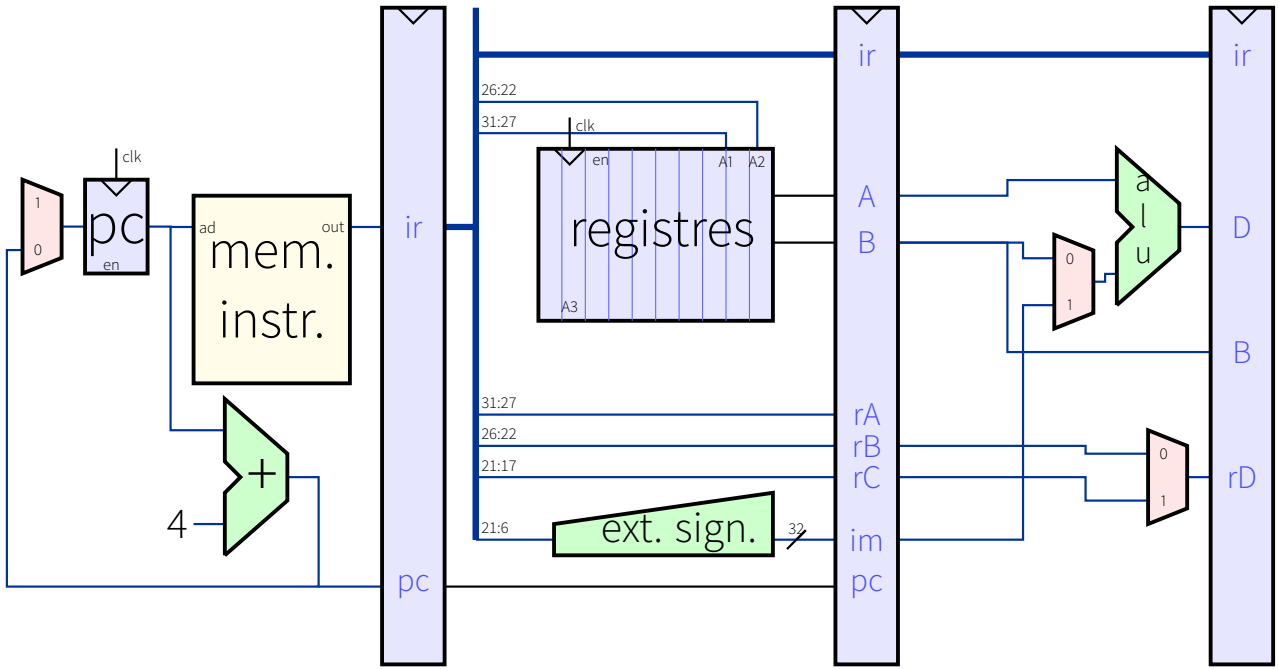
Phase *decode* ou **décodage instruction** (DI)
Extraction des opérandes (`A`, `B`, `imm16`) et copie dans le registre pipeline
Les numéros des registres concernés par l'instruction (`ra`, `rb`, etc) sont également transmis.
`ir` est copié dans le registre pipeline sous forme décodée (commandes pour chaque opérateur/registre)



Exécution de l'instruction

Phase *execute* (EX)
On effectue le calcul requis par l'instruction entre les opérandes : A et soit B, soit `imm16`.
On détermine également le numéro du registre où sera écrit le résultat :

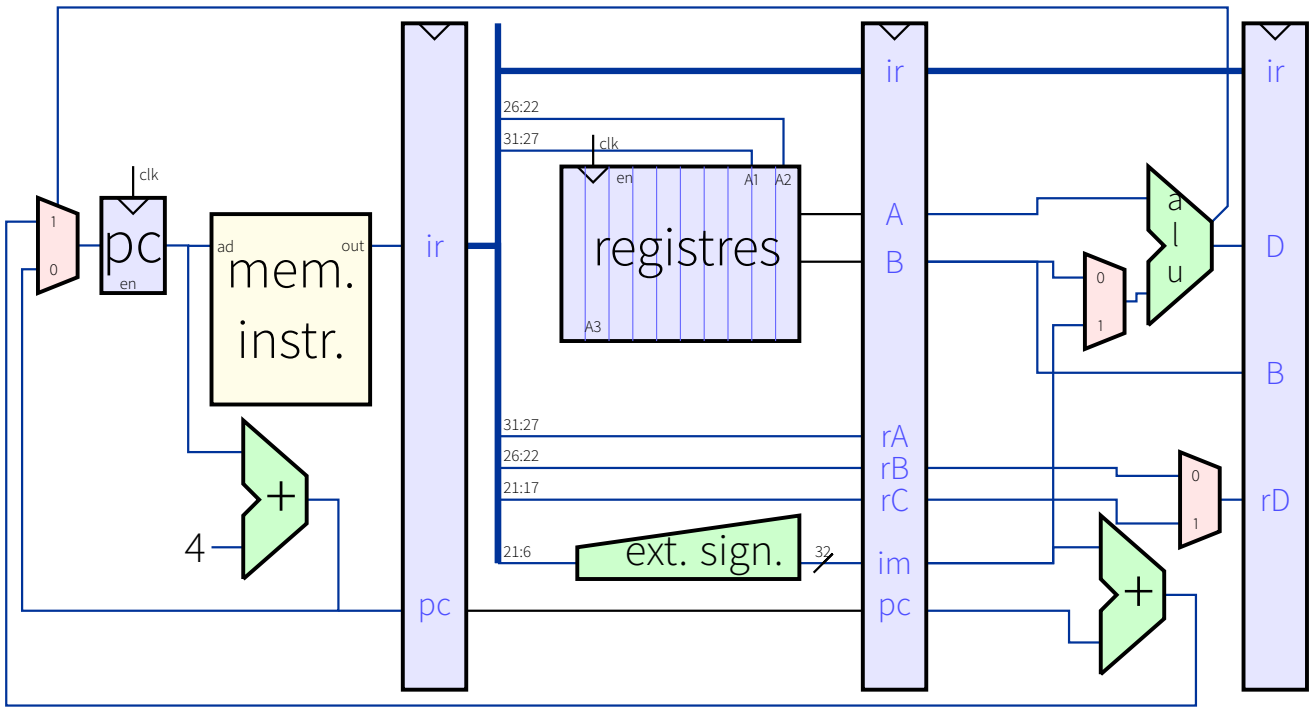
- `rC` pour une instruction de type R
- `rB` pour une instruction de type I



Exécution de l'instruction (cont.)

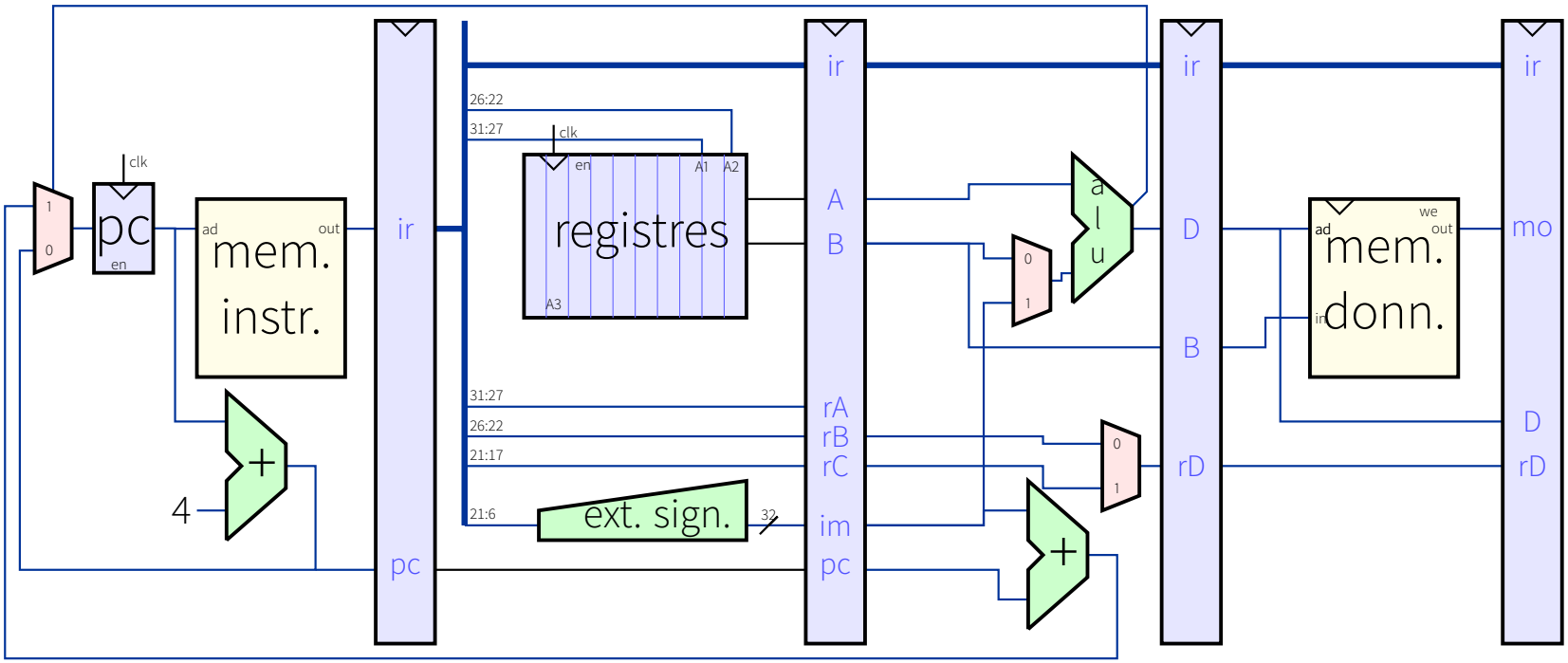
Durant la phase EX, on effectue également les opérations nécessaires pour un branchement :

- calcul de l'adresse de branchement `pc+imm16`
- comparaison entre les opérandes **A** et **B**. Le résultat détermine si `pc` est mis à jour par le mécanisme normal (`pc<-pc+4`) ou par l'adresse de branchement (`pc<-pc+imm16`) suivant la comparaison (ALU).



Accès mémoire

Tous les accès à la mémoire de données se font durant la phase MM.
L'adresse mémoire pour `ld` ou `st` résulte du calcul `A+imm16` et est dans `D`.
Pour un `st`, la donnée à écrire en mémoire est dans `B`
Pour les instructions autres que `ld` ou `st`, on recopie durant cette phase le registre `D` et le numéro du registre destination `rD`.

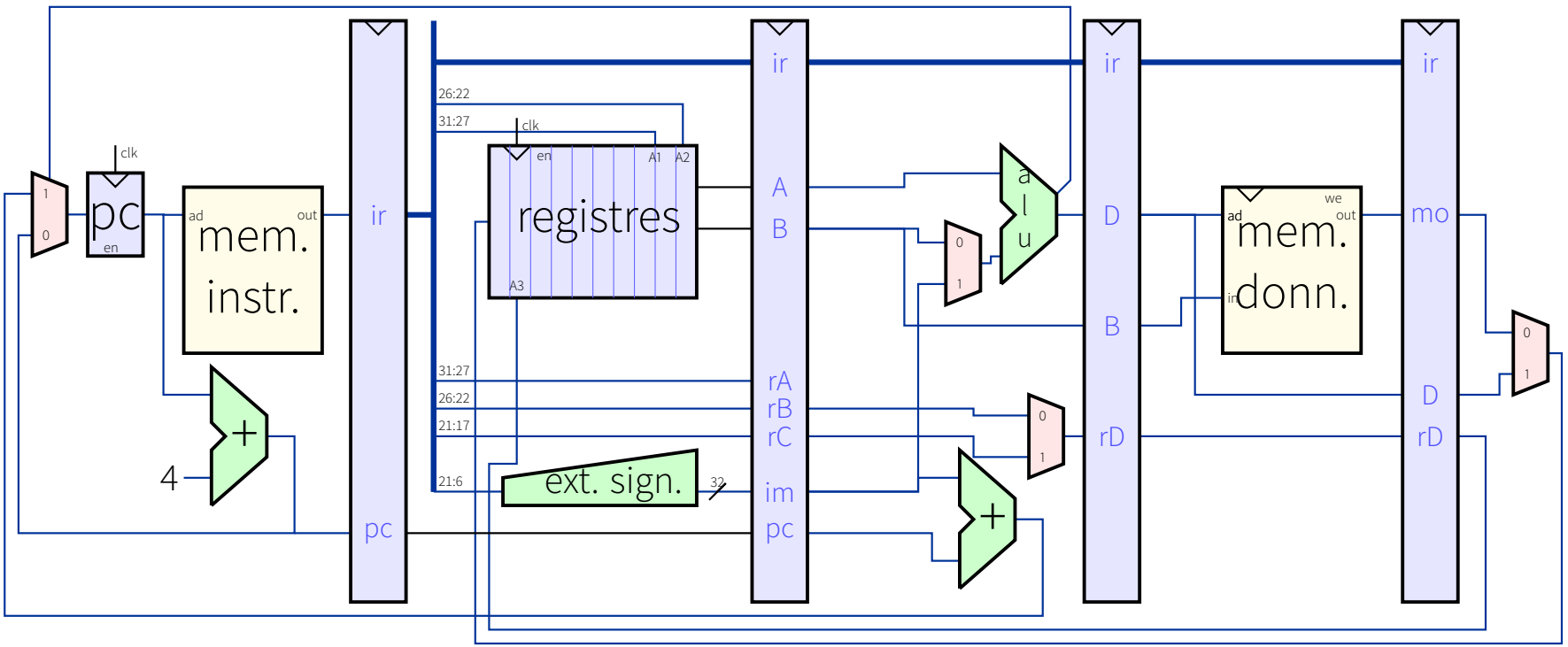


Réécriture des résultats

La phase RR permet la réécriture des résultats vers le banc de registres.
La donnée à réécrire provient :

- de la mémoire pour un `ld`
- de l'ALU (via **D**) pour une instruction arithmétique et logique

`rD` contient le numéro de registre à écrire.



Récapitulatif

Pipeline en 5 étages

LI Lecture de l'instruction. On va chercher l'instruction dans `Mem[pc]` et on incrémente `pc`

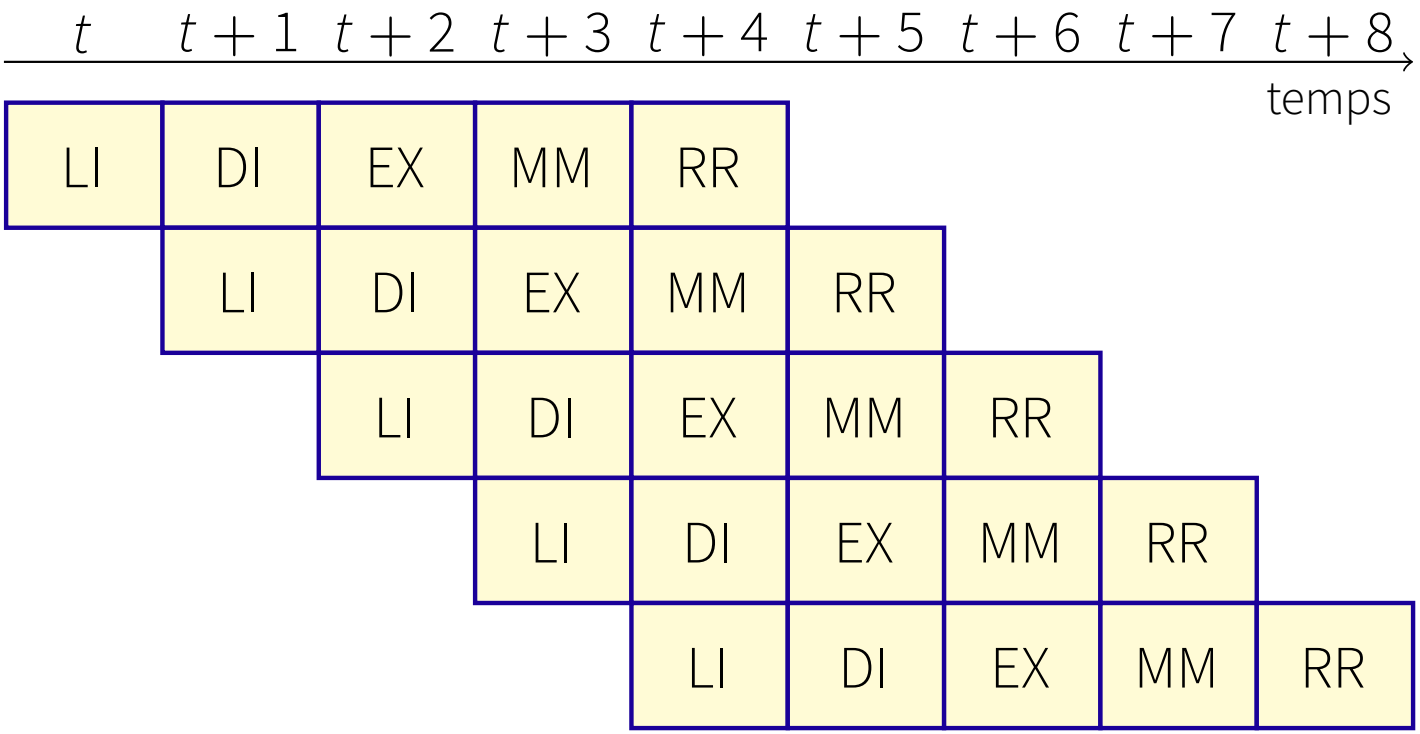
DI Décodage de l'instruction. Pendant ce temps, on lit les données dont on aura besoin dans le banc de registres et dans le registre d'instruction

EX Exécution de l'instruction. On utilise l'unité arithmétique pour effectuer le calcul nécessaire ou calculer l'adresse mémoire (adressage basé).

MM Accés MéMoire. Pour les instructions mémoire on lit ou on écrit la mémoire. Pour les autres instructions, aucune opération, mais transfert des registres tampon.

RR Rangement des résultats. On réécrit les données calculées ou lues en mémoire dans le registre destination.

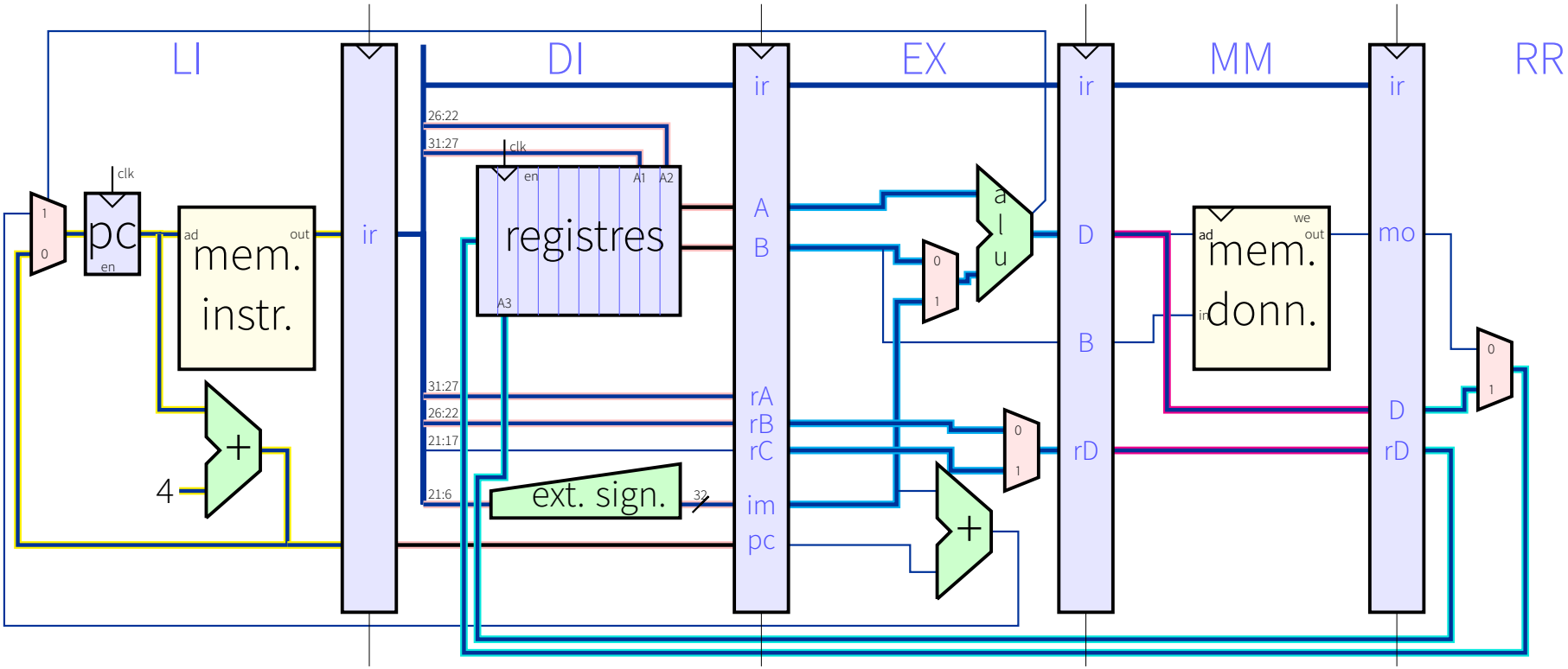
Le pipeline nios



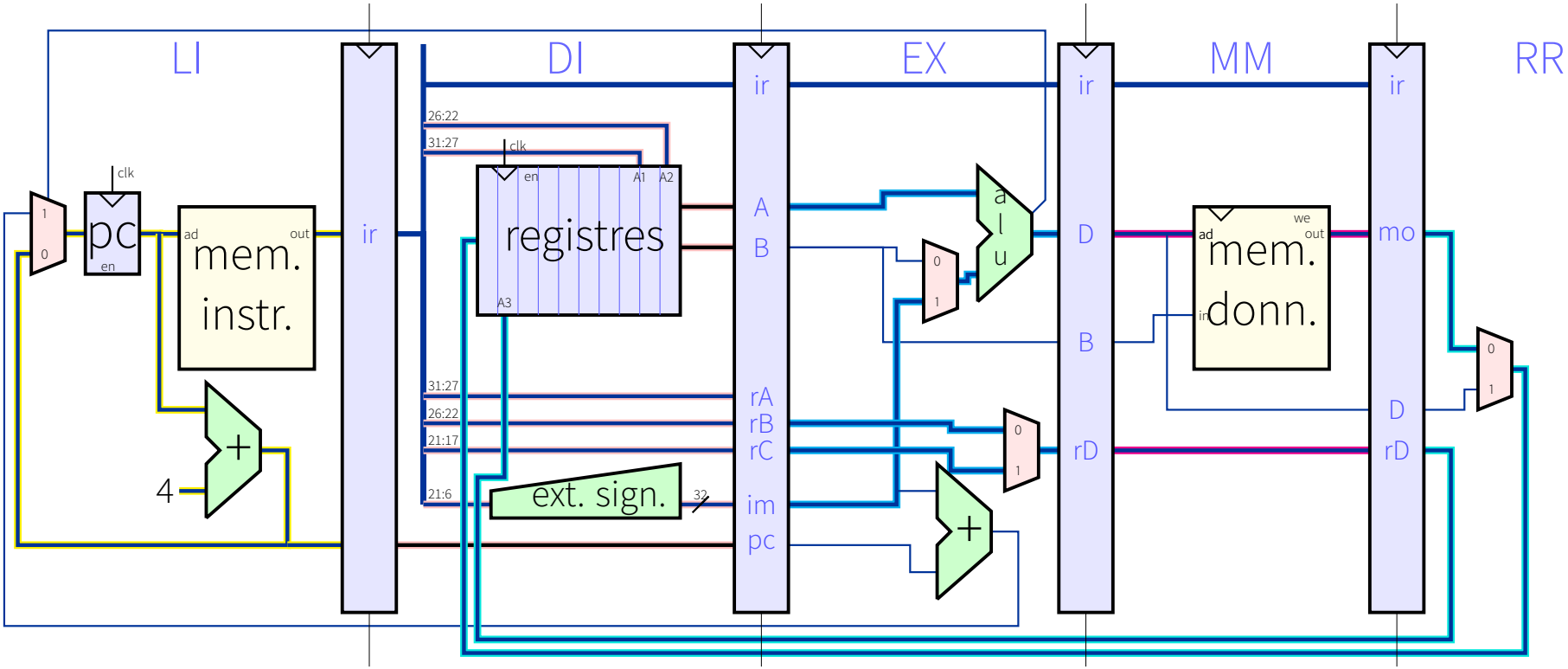
« pipeline RISC simple »

Instructions arithmétiques et logiques

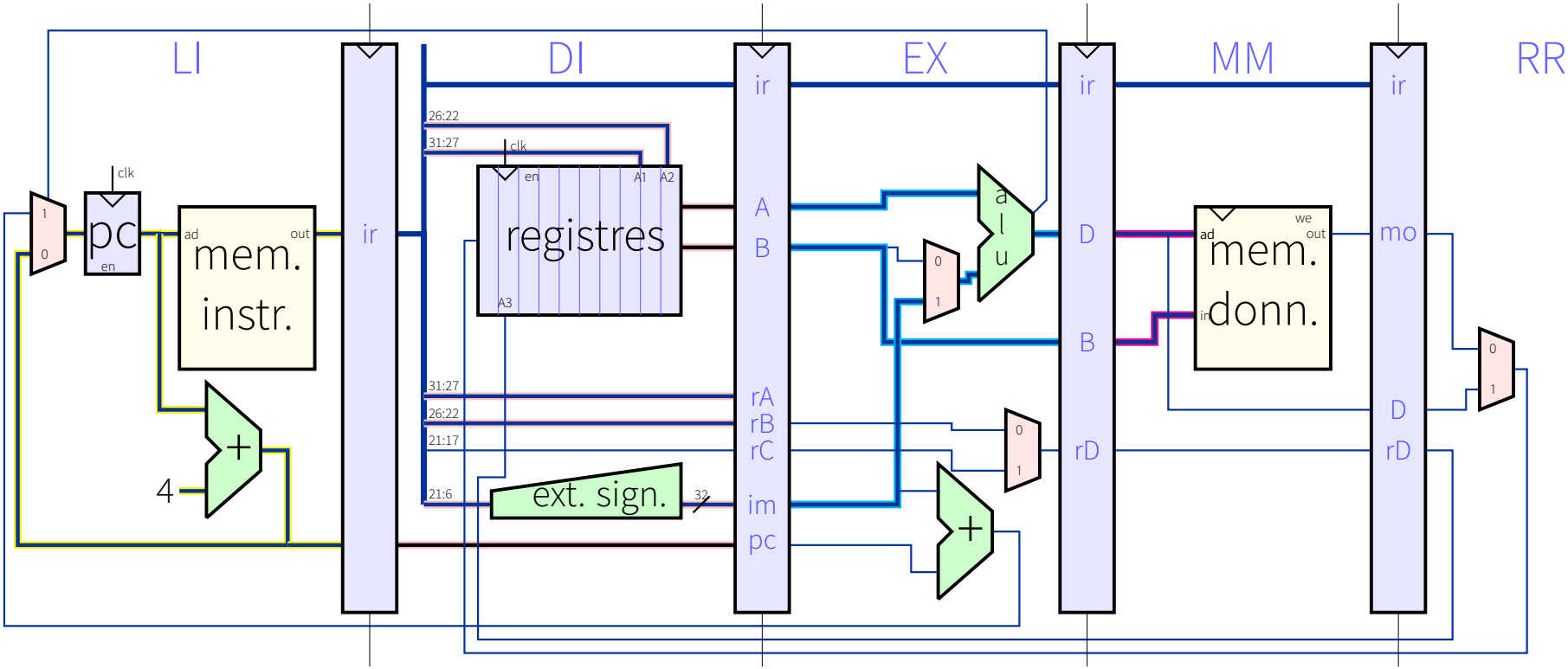
(LI ___, DI ___, EX ___, MM ___, RR ___)



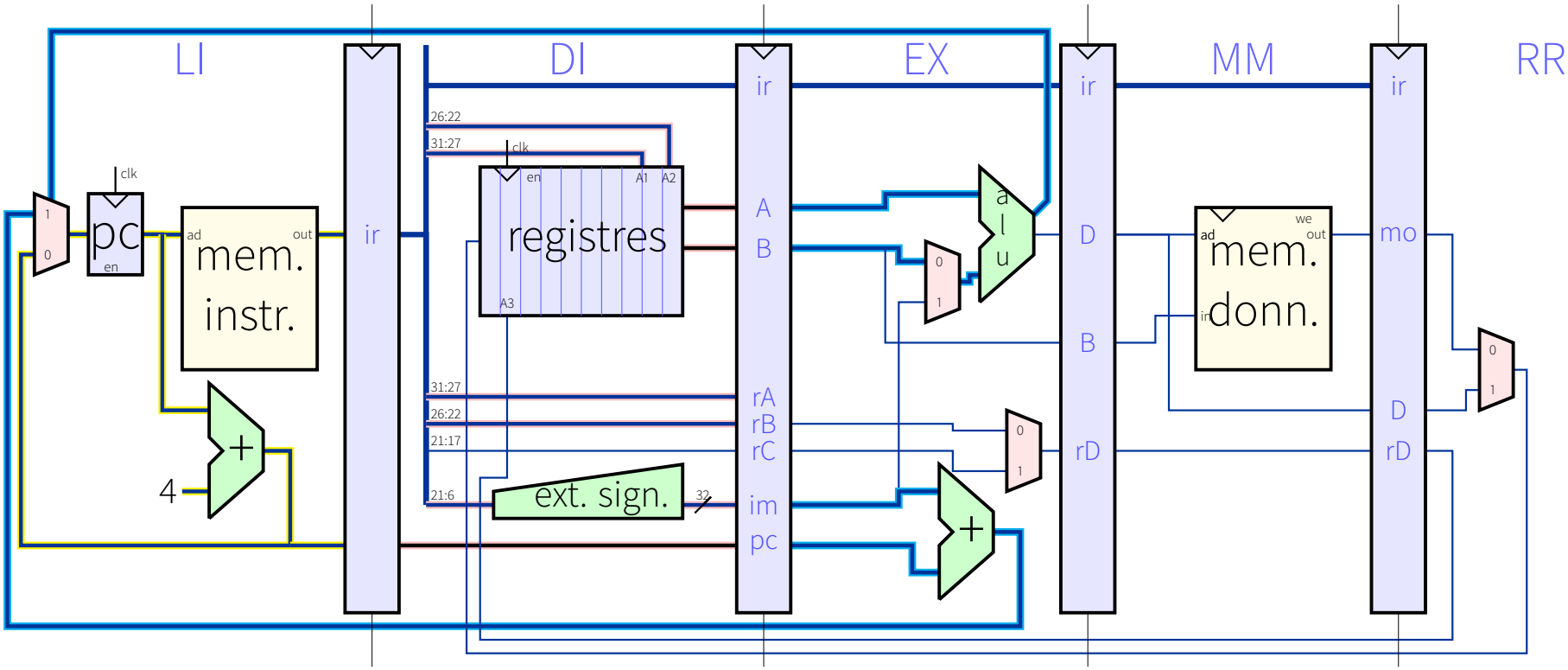
Lectures mémoire (`ld`)



Ecritures mémoire (`st`)



Branchements



Récapitulatif (cont.)

Le nios pipeliné existe en deux versions dans les FPGA Atera/Intel: nios II/s (5 étages de pipeline) et nios II/f (6 étages). On retrouve des pipelines similaires de 5 à 7 étages dans les processeurs embarqués simples actuels (arm-v6, mips4).

Pour les processeurs plus performants (pentium, power-PC, arm-v7, arm-v8, etc), le pipeline est plus important (10 à 20 étages).

- permet la mise en oeuvre d'opérations complexes (flottant)
- permet d'augmenter la fréquence de fonctionnement

Aléas dans les pipeline

Un *aléa* est un événement qui perturbe le bon fonctionnement d'un pipeline.

Trois types d'aléas :

aléa de données lié à la non disponibilité d'une donnée pour éxécuter une certaine opération à cause du délai d'exécution

aléa de contrôle lié aux ruptures du flot d'exécution d'un programme (branchements, sauts, etc)

aléa structurel lié à l'existence d'une ressource matérielle partagée par deux étages ou plus

Aléa de données

Du fait qu'une instruction est lancée alors que la précédente n'est pas achevée, il y a un risque d'utiliser une donnée qui n'est pas encore mise à jour.

```
add  r1, r2, r3
add  r5, r4, r1
```

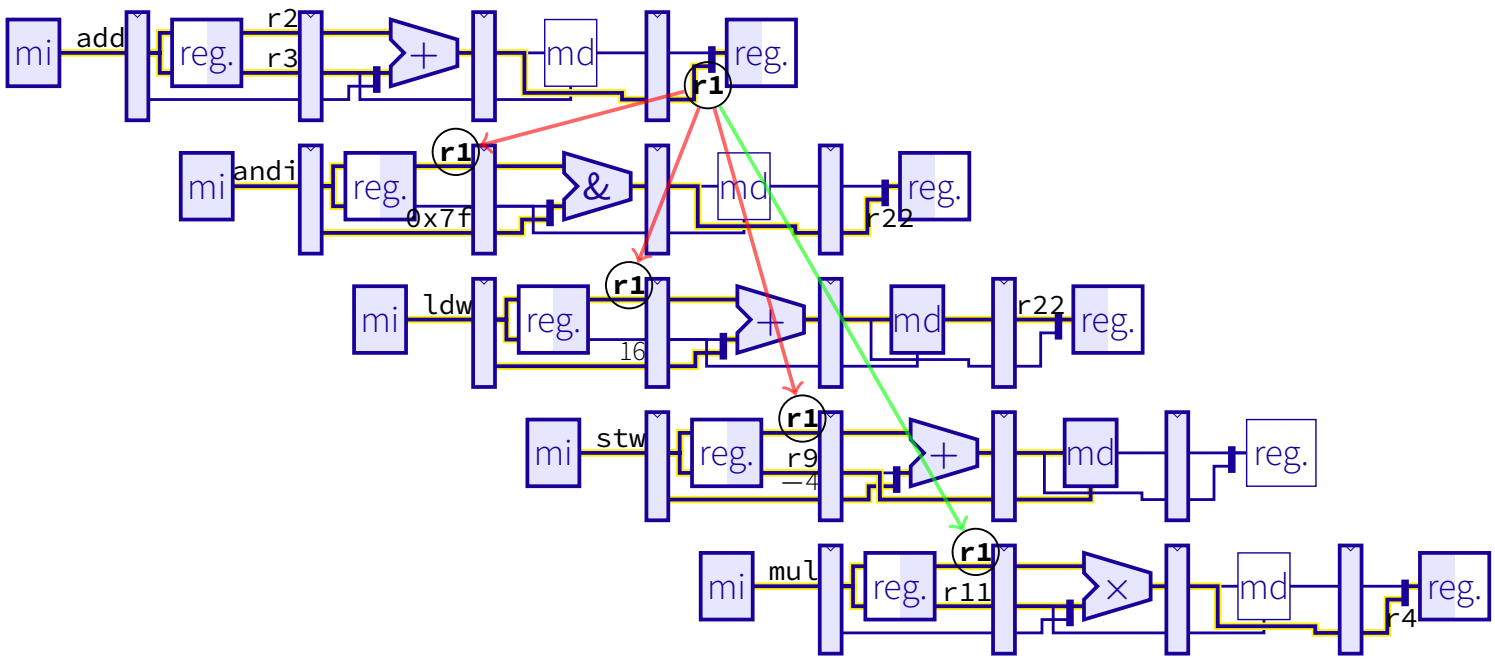
L'instruction 2 va utiliser *l'ancienne* valeur de **`r1`** et non celle qui vient d'être calculée.

Aléa de données

Aléa de données (cont.)

Le contenu des registres est chargé dans les registres pipeline **A** et **B** à la fin de l'étage **LI**.
Les registres **r1–r31** sont écrits à la fin de la phase **RR**.

1 add **r1**,r2,r3
2 andi r22,**r1**,0x7f
3 ldw r22,(16)**r1**
4 stw r9,(-4)**r1**
5 mul r4,**r1**,r11



L'instruction 1 intéragit sur les instructions 2, 3 et 4.
NB : *Tous* les registres sont :

- lus pendant une phase
- écrits à la fin de la phase

Aléa de données (cont.)

On peut supprimer certains aléas de données en allant chercher la nouvelle donnée là où elle est présente avant de faire le calcul lors la phase `EX`.

```
1 add r1,r2,r3
2 add r3,r1,r2
3 add r4,r1,r4
4 add r5,r1,r5
```

`add r1,r2,r3`

`add r3,r1,r3` la nouvelle valeur de `r1` est dans le registre pipeline `EX/MM.D`

`add r4,r1,r4` la nouvelle valeur de `r1` est dans le registre pipeline `MM/RR.D`

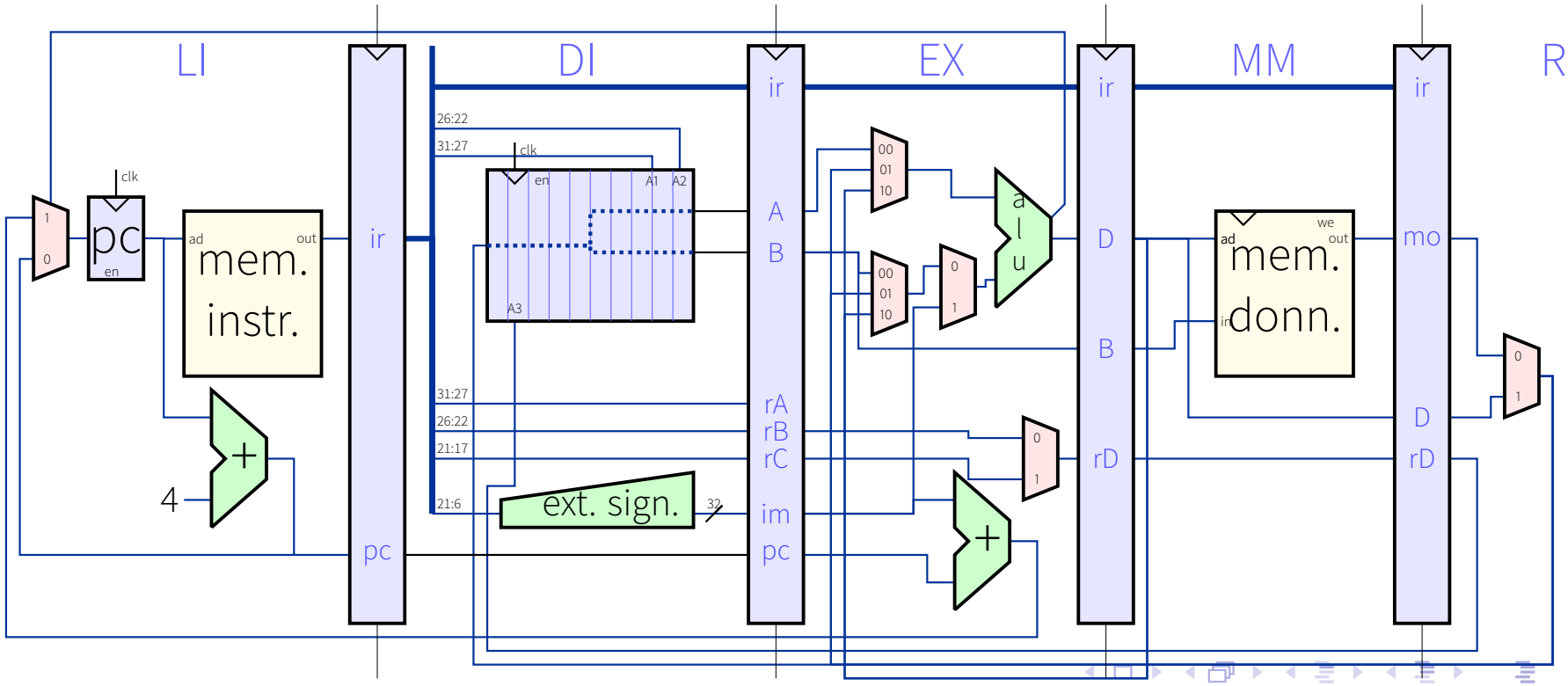
`add r5,r1,r5` le registre `r1` est *en train* d'être écrit dans le banc de registre

Mécanisme d'**envoi** (*forwarding*) (également appelé court-circuit (*bypass*)). On rajoute des chemins entre les registres pipeline où le résultat peut être présent et l'entrée de l'ALU.

Aléa de données (cont.)

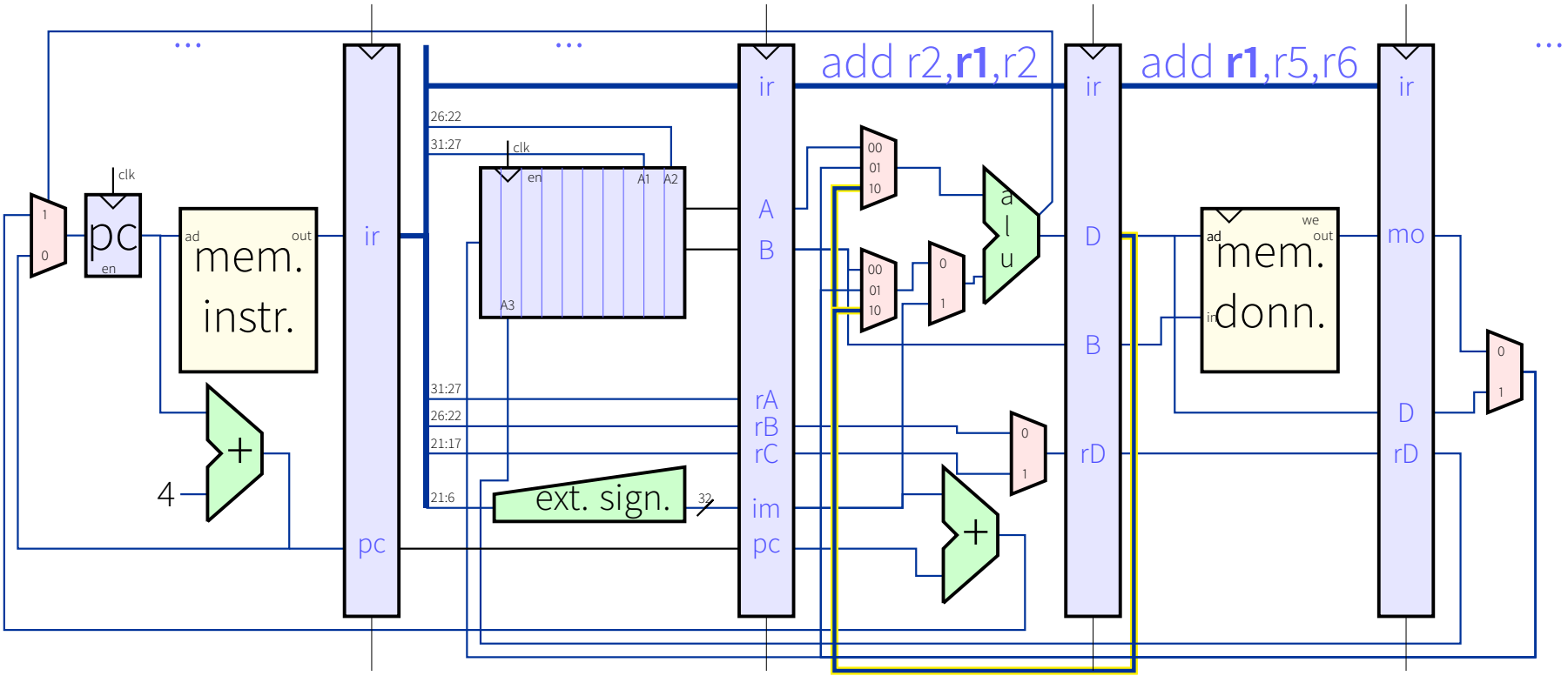
Modifications architecturales :

- ajouts de multiplexeurs aux entrées **A** et **B** de l'ALU
- chemin de EX/MM.D vers les entrées **A** et **B** de l'ALU
- chemin de MM/RR.D vers les entrées **A** et **B** de l'ALU
- transfert direct entre entrée du banc de registres et ses sorties



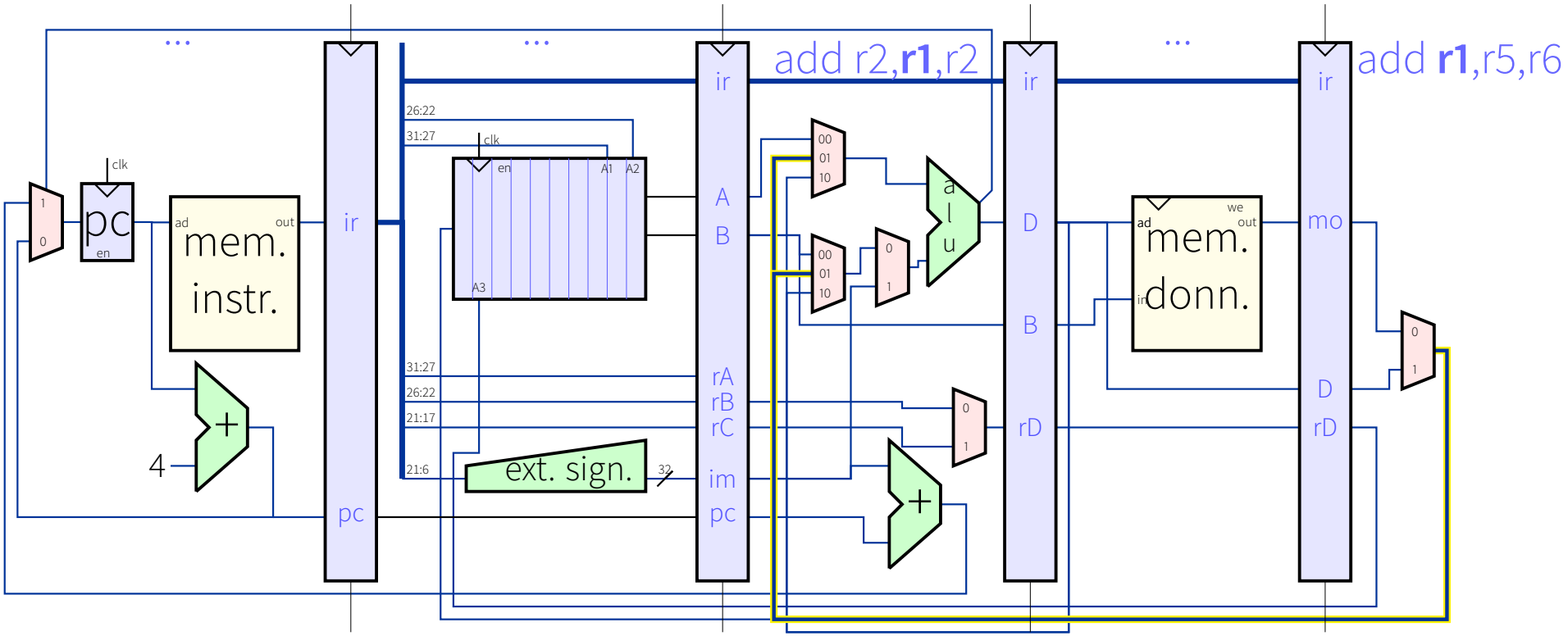
Aléa de données (cont.)

Cas d'une intéraction entre les étages MM et EX.



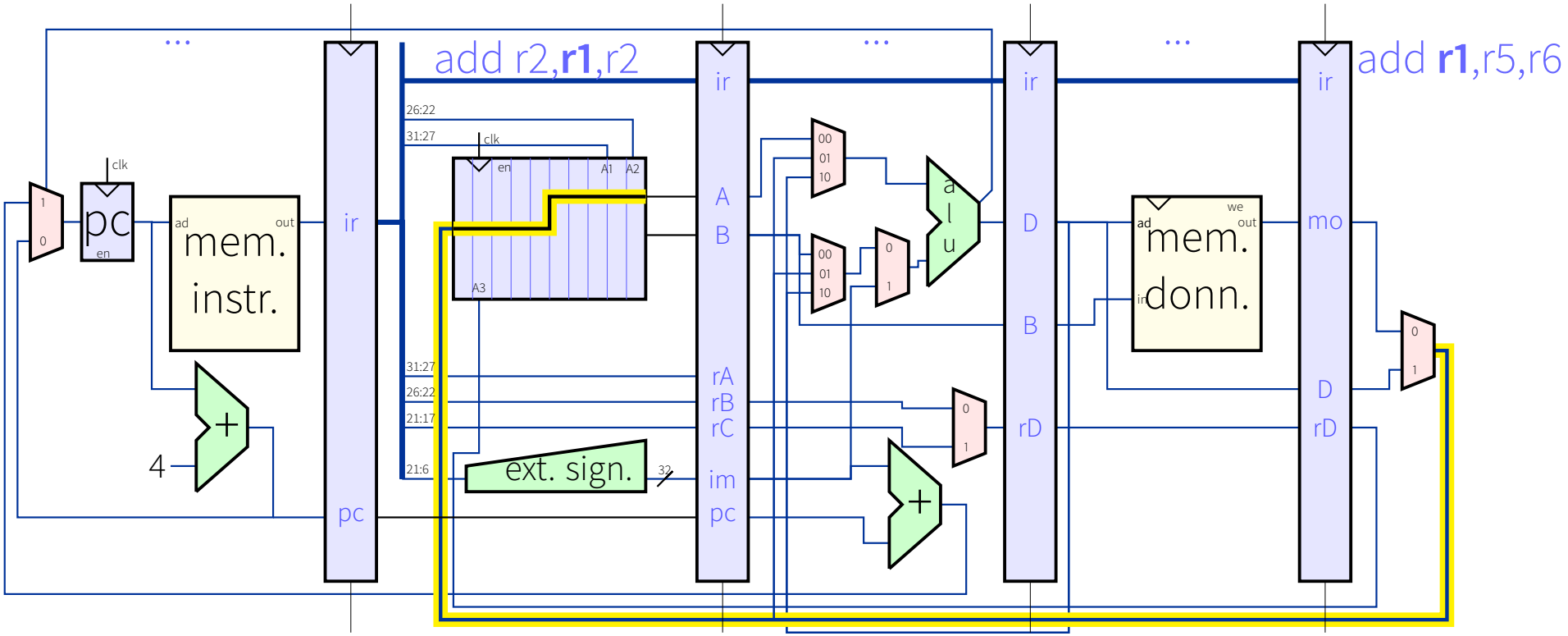
Aléa de données (cont.)

Cas d'une intéraction entre les étages RR et EX.



Aléa de données (cont.)

Cas d'une intéraction entre les étages RR et LI.



Le transfert entrée → sortie pour le banc de registres peut se faire :

- en ajoutant des multiplexeurs aux sorties du banc de registres pour sélectionner entre ce qui est écrit et ce qui est en train d'être écrit
- en écrivant dans le banc de registre sur la phase descendante de l'horloge. On lit ainsi dans tous les cas ce qui vient d'être écrit.

Aléa de données (cont.)

L'envoi est réalisé en comparant les champs `rD` des registres pipeline `EX/MM` et `MM/RR` et les champs `rA` (ou `rB`) du registre pipeline `DI/EX`.

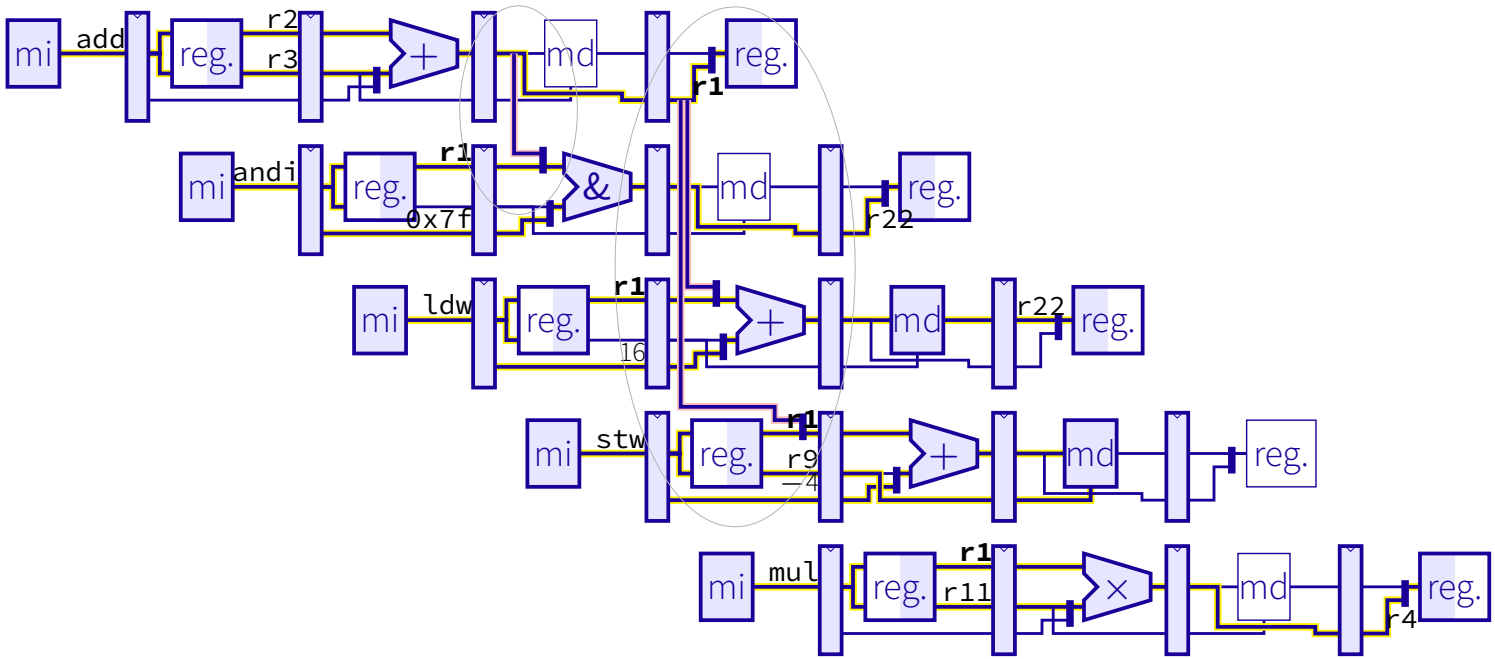
si `EX/MM.rD` == `DI/EX.rA` alors
`muxFwdASel`=10
sinon si `MM/RR.rD` == `DI/EX.rA` alors
`muxFwdASel`=01
sinon
`muxFwdASel`=00

NB1 : `r0` est toujours à 0 et n'est jamais *envoyé*.
NB2 : il faut aussi vérifier que les instructions dans les étages `MM` ou `RR` vont bien réécrire dans le banc de registres (toutes sauf `st` et sauts/branchements).

Aléa de données (cont.)

Avec les mécanismes d'envoi le fonctionnement est correct, sans perte de performances

```
1 add  r1,r2,r3
2 andi r22,r1,0x7f
3 ldw  r22,(16)r1
4 stw  r9,(-4)r1
5 mul  r4,r1,r11
```



Mais...

Aléa de données (cont.)

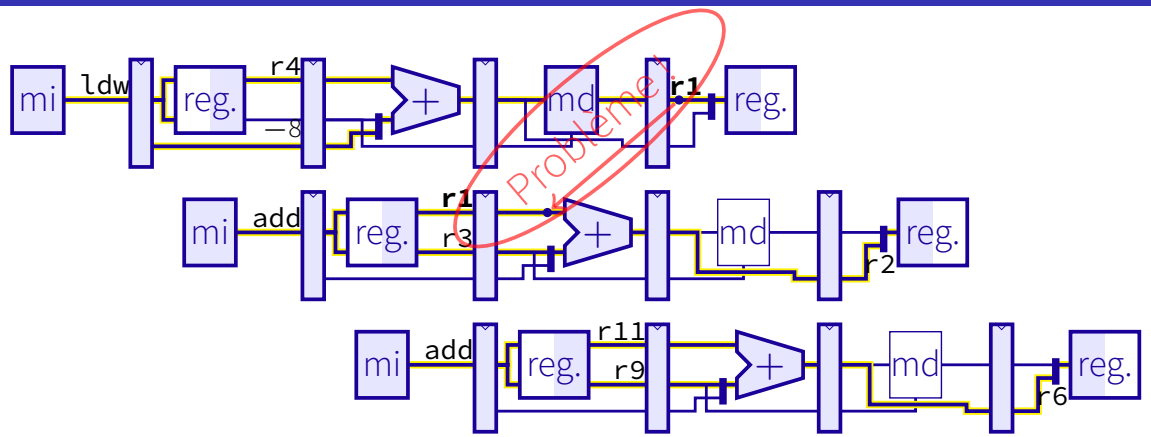
...le mécanisme d'envoi n'est possible que quand la donnée est *dans le processeur*.
Lors d'une lecture mémoire (**ld**), elle n'est *pas* accessible au processeur avant la fin de la phase **MM**.

```
ld    r1,4(r2)
add   r3,r1,r4
```

La solution dans ce cas est de détecter le problème et de faire une **suspension** (*stall*) du pipeline.
L'acquisition et la progression des instructions sont bloquées tant que la donnée n'est pas présente dans le processeur (dans **MM/RR.mo**).
Les mécanismes d'envoi prennent alors le relai.

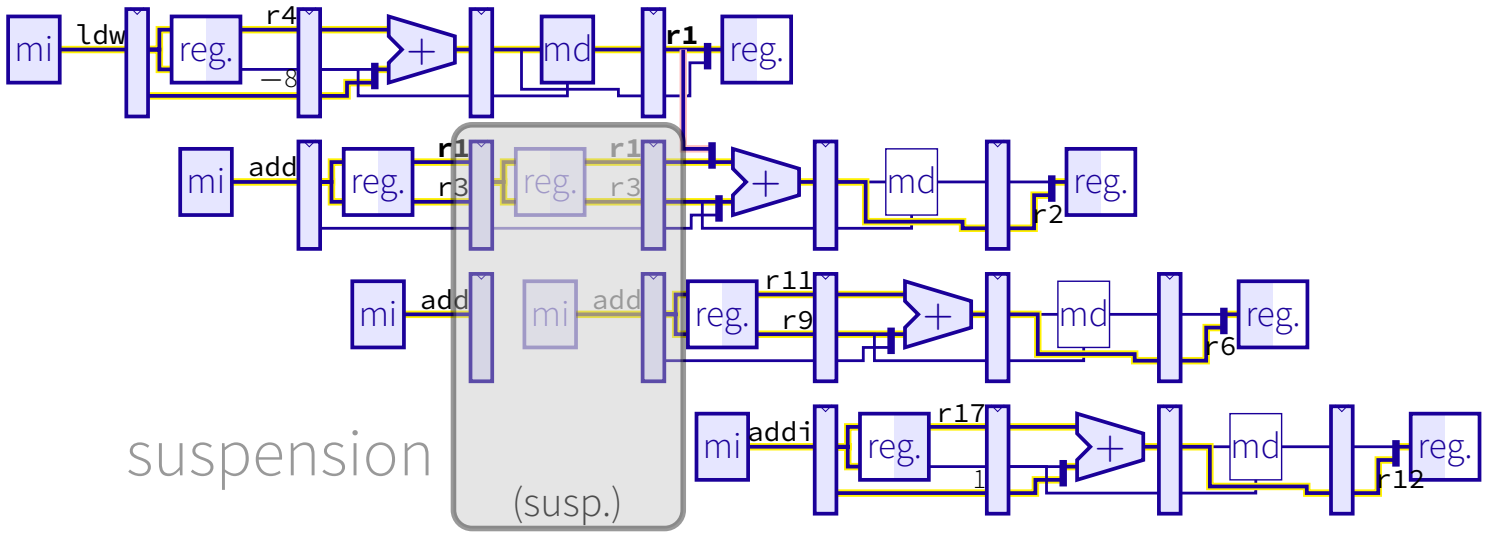
Aléa de données (cont.)

1 ldw **r1**,(-8)r4
2 add r2,**r1**,r3
3 add r6,r11,r9



On injecte une suspension.

1 ldw **r1**,(-8)r4
2 add r2,**r1**,r3
3 add r6,r11,r9
4 addi r12,r17,1



Une suspension est mise en oeuvre en interdisant toute écriture dans les registres pipeline ou **pc**, ce qui empêche la progression du pipeline.
Duplication des phases suspendues.

Aléa de données (cont.)

Un `ld` suivi d'une instruction utilisant comme source le registre écrit par le `ld` prendra alors 2 cycles.

En plus des accès mémoire, il existe d'autres situations où la résolution d'un aléa de données nécessite une suspension.

C'est le cas quand les opérations ont une **latence** importante (>1).
Exemple : opérations flottantes

```
fmul r1,r2,r3
fadd r5,r1,r6
```

Si la multiplication flottante a une latence de 4, 3 cycles de suspension seront nécessaires.

Les compilateurs essaient de *réordonnancer* les instructions pour réduire ces suspensions.

Aléas de controle

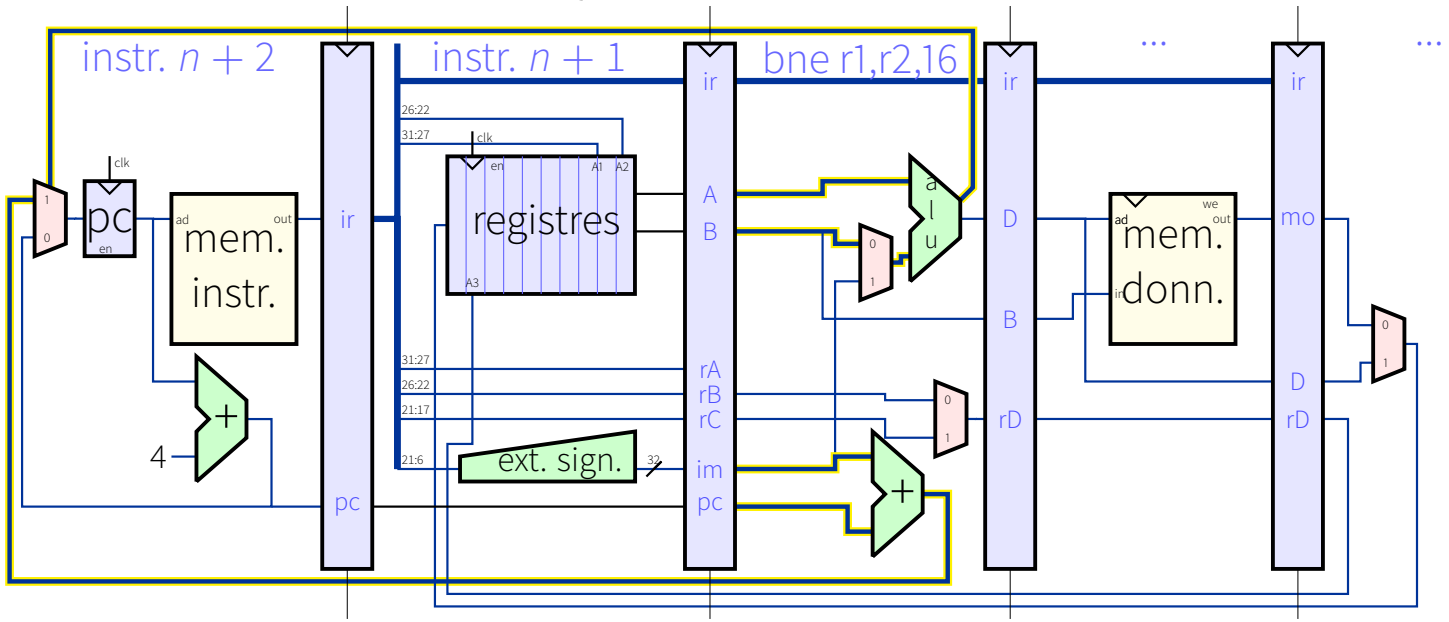
Lors d'une rupture de séquence (branchement, saut), on ne connait le type de l'instruction qu'*après* la phase de décodage d'instruction (`DI`).
Or on a déja acquis l'instruction suivante...

aléa de contrôle

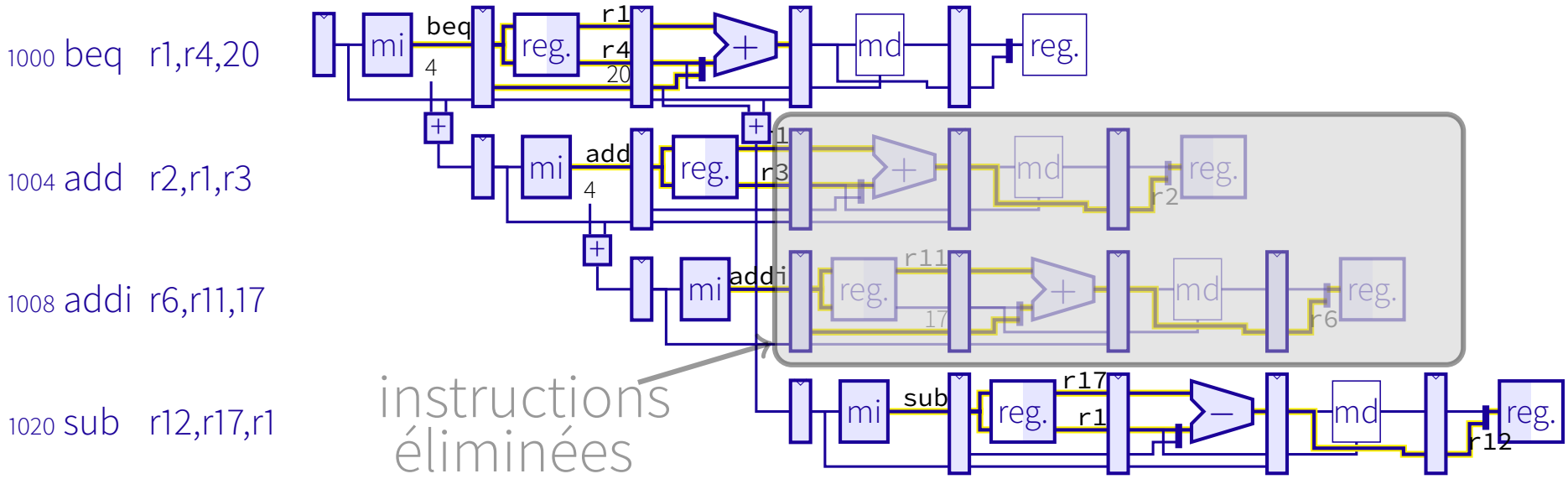
Perte systématique de cycle(s) lors d'un branchement ou d'un saut.
Dans la cas du nios, le branchement se fait en phase `EX`.

Pénalité de branchement de deux cycles.

Instructions $n+1$ et $n+2$ déja acquises au moment du branchement.



Aléas de controle (cont.)



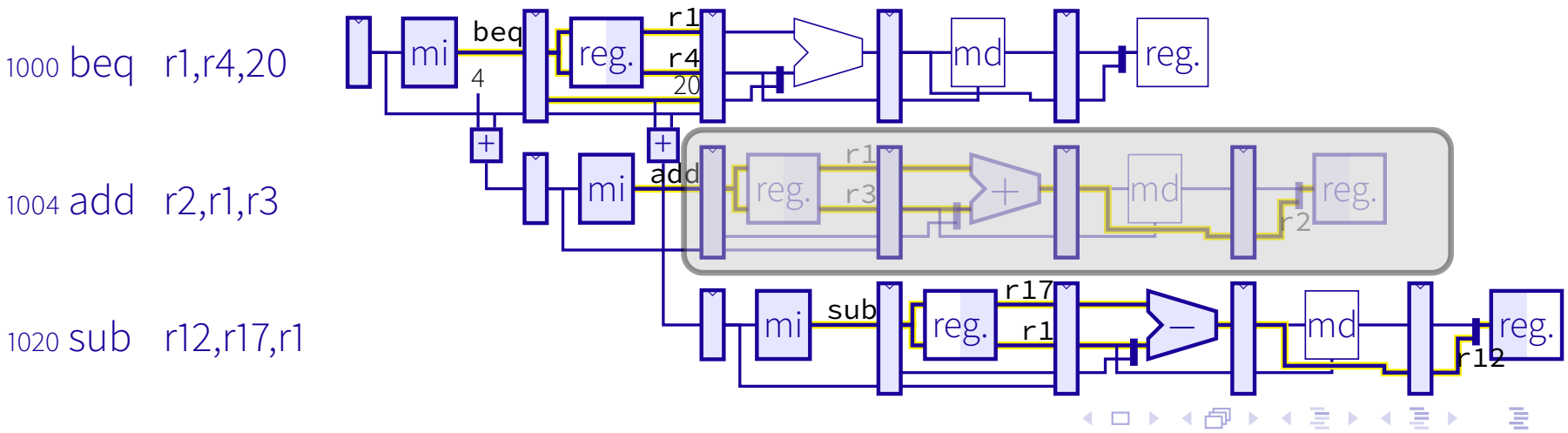
Les instruction aux adresses 1004 et 1008 ont été chargées à tort.
Elles sont éliminées (*flushed*).
En pratique, on met à zéro les registres pipeline pour remplacer les instructions éliminées par des **nop**.
Pénalité de branchement de 2 cycles.

Aléas de controle (cont.)

On peut réduire la pénalité de branchement à un cycle en effectuant le branchement en phase `DI`.
Il faut modifier l'architecture pour :

- calculer `pc+imm16` en phase `DI` (déplacement de l'additionneur de `EX` vers `DI`)
- effectuer les comparaisons en phase `DI` (ajout d'un comparateur aux sorties `A` et `B` du banc de registres)

Mais on ne peut pas supprimer totalement la pénalité par une action faite durant de la phase `DI`.



Aléas de controle (cont.)

Solutions possibles pour réduire la pénalité de branchement :

– Changement de l'action réalisée par un branchement : `brx` (*branch and execute*)

On exécute toujours l'instruction qui suit le `brx`, puis on fait le branchement (*branchement retardé*).

Si on peut exécuter une instruction utile après le branchement, pas de pénalité.

Sinon, on met un **nop** après le branchement

Utilisé dans MIPS et les premiers processeurs pipeline.

Aléas de controle (cont.)

Solutions possibles (suite) :

– Méthodes sophistiquées de *prédiction de branchement* basées sur l'*adresse* de l'instruction lors de la phase `LI`.

Les branchements ont souvent un comportement répétitif (boucles).

Quand on détecte un branchement, on note son adresse et son comportement (pris/non pris).

Quand on détecte une instruction à cette adresse, on cherche à prédire son comportement en fonction de son comportement passé.

Si la prédiction est correcte, pas de pénalité.

Si la prédiction est incorrecte, on efface les instructions acquises et on a la pénalité habituelle.

Aléas de controle (cont.)

Solutions possibles : (suite)
– Instructions conditionnelles
Les instructions conditionnelles présentes dans certaines architectures permettent d'éviter des branchements.

```
if(R1)
  R3=R2;
```

Traduit sans instructions conditionnelles

```
    breq r1,r0,L1
    add r3,r0,r2
L1:  ...
```

Avec instructions conditionnelles

```
movcond r3,r2,r1
```

Très efficace pour de petits branchements.
`cmov` dans pentium, instructions conditionnelles dans ARM

Aléas structurels

Quand deux étages différents d'un pipeline ont besoin d'une même ressource il y a *aléa structurel*.

Idéalement, la plupart sont supprimés par l'architecture.

Mais, certains peuvent arriver.

Exemples :

- si on utilise une mémoire unique pour instructions et données.
  Présence d'un aléa structurel lors de la phase `MM` d'une instruction `ld` ou `st` et de la phase `LI` d'une autre instruction.
- certains opérateurs complexes et rarement utilisés (division flottante, racine carrée) sont *non pipelinés*.
  Ils sont alors indisponibles pendant plusieurs cycles. Aléa structurel potentiel en cas d'autre demandes.
- file d'attente pleine, etc

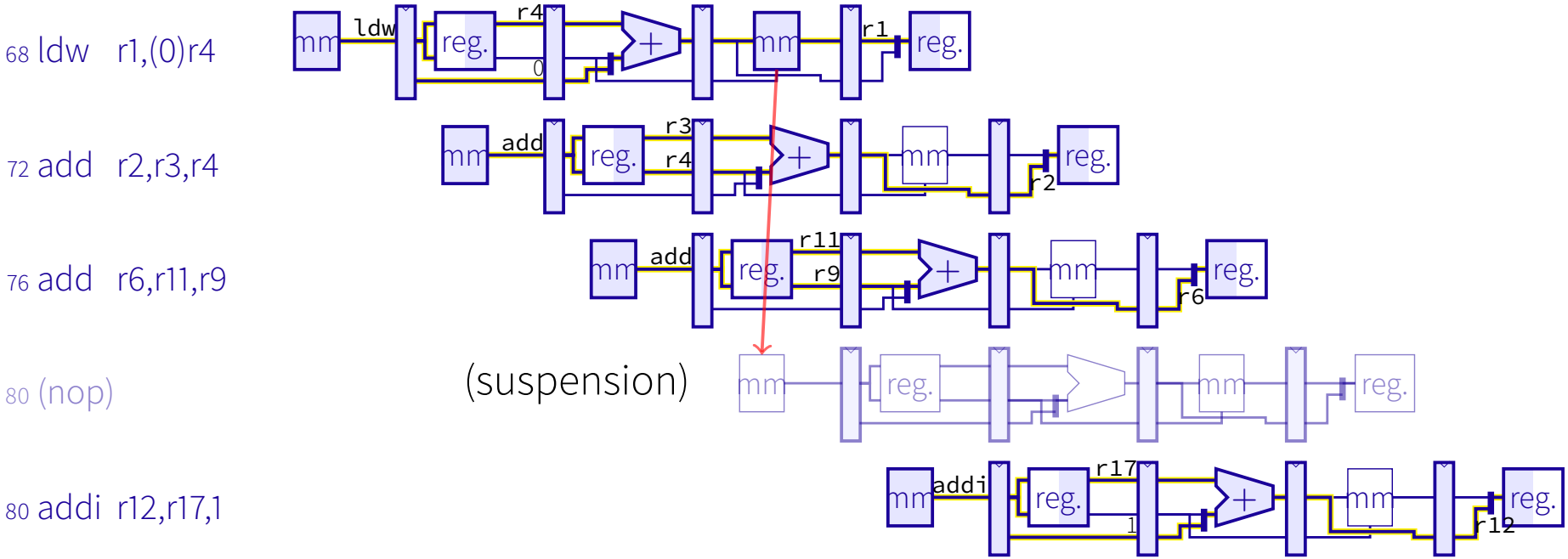
Aléas structurels (cont.)

Solutions :

- duplication de la ressource.
  - mémoire de données et mémoire d'instructions séparées (au moins au niveau des mémoires cache).
  - mise en pipeline d'opérateurs
  - tampons, files d'attentes, etc, plus importantes
- *suspension* (interruption) (*stall*) du pipeline
  La suspension permet d'attendre que la ressource se libère et entraîne la perte d'un ou plusieurs cycles

Aléas structurels (cont.)

Exemple : cas d'une *mémoire unique instructions/données*



Utilisation simultanée de la mémoire par LI et MM.
La phase LI est remplacée par l'injection d'un **nop** sans lecture de la mémoire, ni incrémentation de **pc**.
On reprend ensuite le cours normal d'exécution.
Tout se passe comme si **ld** (ou **st**) prenait 2 cycles.