Mise en oeuvre multicycle du processeur NIOS II

Alain MÉRIGOT

Université Paris sud

Introduction

Le formalisme *multicycle* a été employé pour réaliser des processeurs jusque dans les années 80.

Aujourd'hui, il est encore employé pour des microcontroleurs ou même des processeurs en version basses performances (comme le Nios II/e).

On définit un *chemin de données* et chaque instruction est décomposée en étapes élémentaires sur ce chemin.

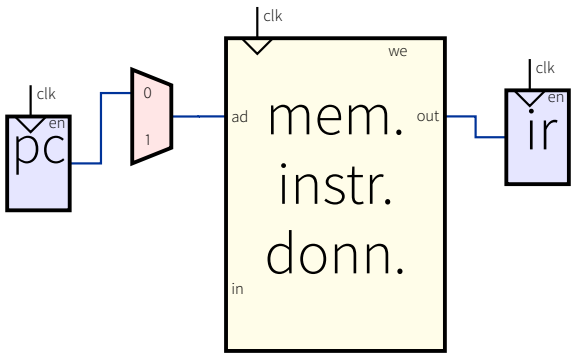
L'exécution des instructions est pilotée par un automate qui va contrôler les différents éléments du chemin de données (registres, multiplexeurs, opérateurs) en fonction des caractéristiques de l'instruction.

Phase d'acquisition

La première phase de toute instruction est la phase d'**acquisition des instruction** (*fetch*).

Nous avons besoin d'introduire :

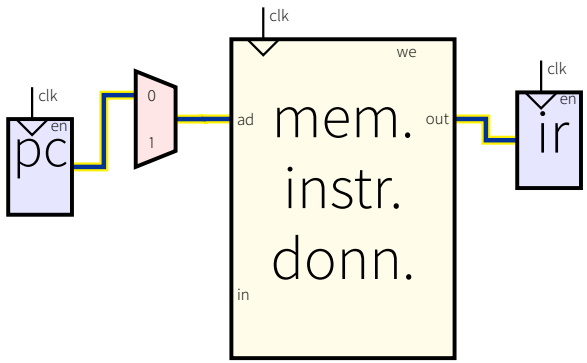
- la *mémoire*, qui contiendra instructions et données
- le *compteur de programme* `pc`, qui contiendra l'adresse de la prochaine instruction à exécuter
- le *registre d'instruction* `ir`, qui stocke le contenu de l'instruction en cours d'exécution



Phase d'acquisition (cont.)

L'action à réaliser durant cette phase est : `ir <- mem[pc]`
Pour cela, il faut :

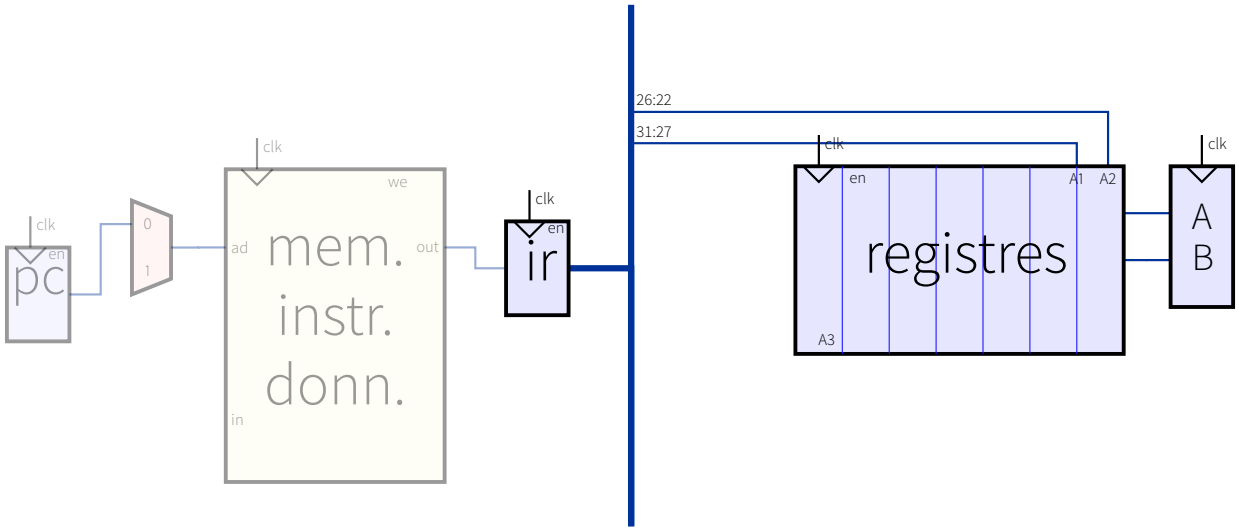
- aller lire l'instruction dans la mémoire à une adresse contenue dans le compteur de programme (**pc**)
- copier la sortie de la mémoire dans le registre d'instructions (**ir**)



Il est également nécessaire de préparer l'instruction suivante en incrémentant le compteur de programme.
Nous utiliserons pour ce calcul l'unité arithmétique et logique.

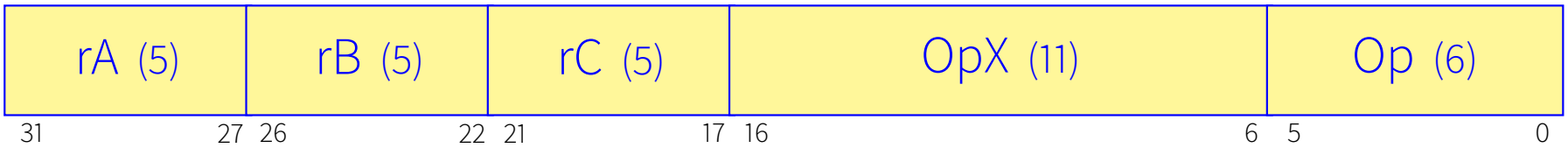
Décodage des instruction

Au début de la deuxième phase, l'instruction est écrite dans `ir`.
Durant cette phase, le contenu de l'instruction est disponible sur le bus instruction, mais la phase **doit** être identique pour toutes les instructions.
Réalisation du *décodage des instructions* (phase *decode*).
Pour préparer les phases d'exécution suivantes, extraction des opérandes du *banc de registre*
A chaque cycle, le banc de registres peut lire deux opérandes (`A1, A2`) et en écrire un troisième (`A3`).

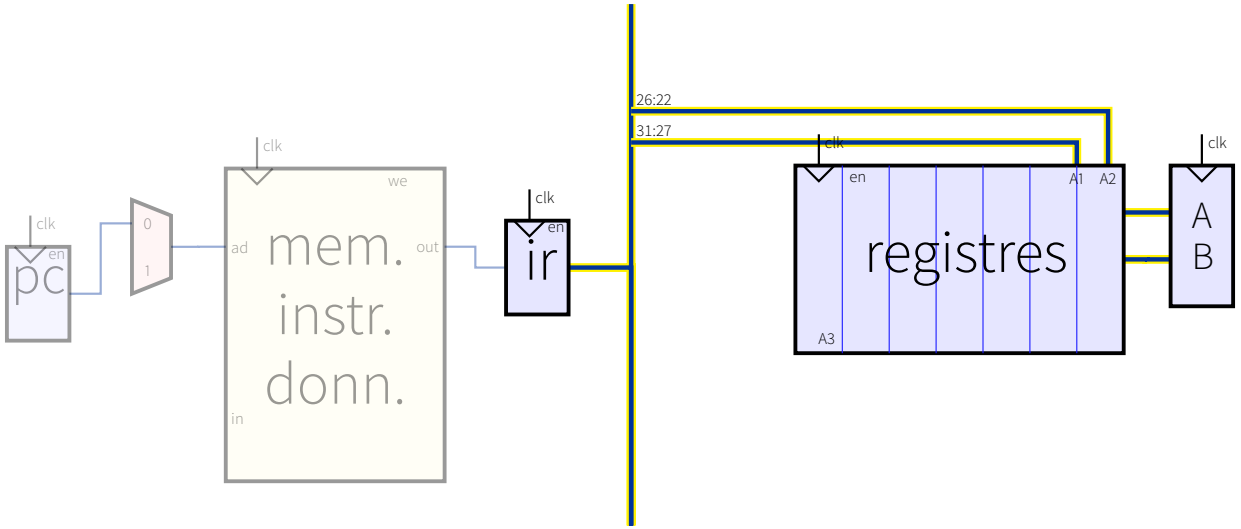


Décodage des instruction (cont.)

Les opérandes spécifiés par les champs **rA** (31:27) et **rB** (26:22) des instructions sont copiés dans des *registres tampon* **A** et **B**.

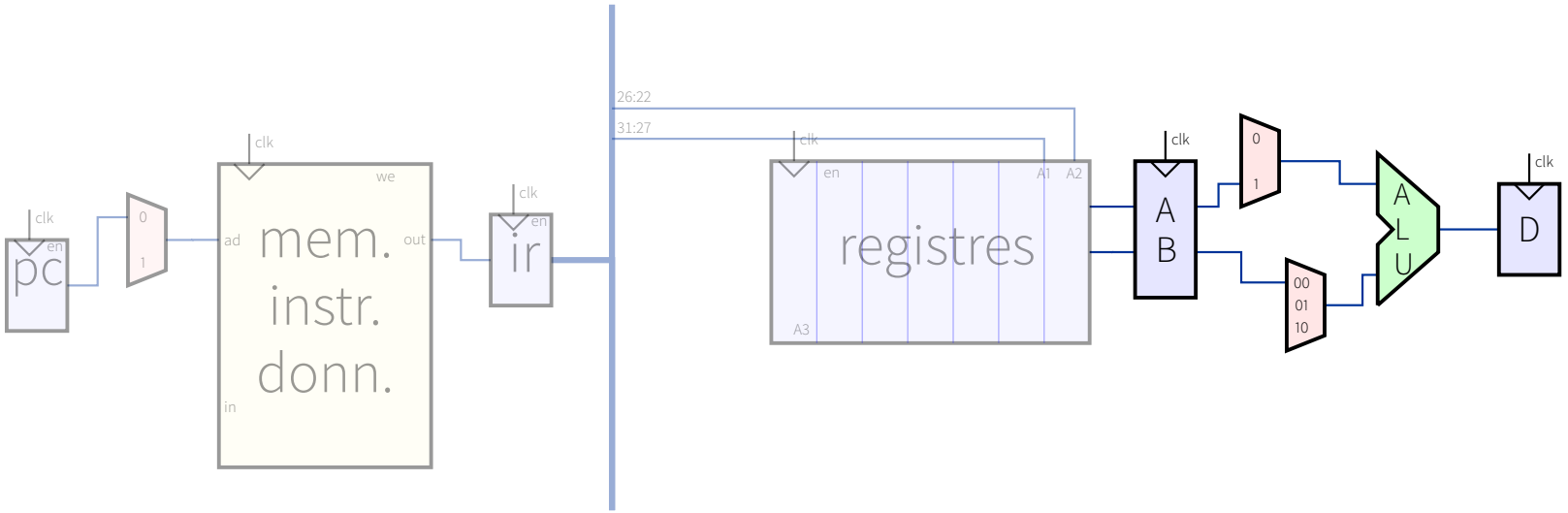



```
A <- rA
B <- rB
```



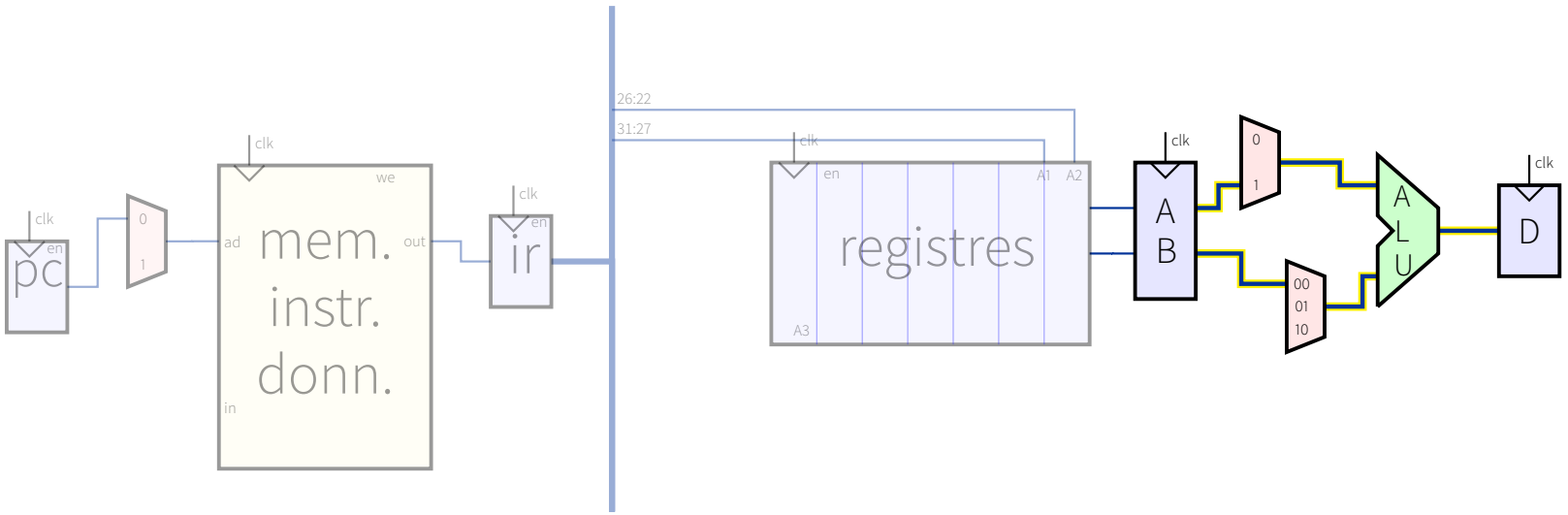
Instructions arithmétiques et logiques

La troisième phase est la phase d'*exécution* des instructions.
Pour les instructions arithmétiques et logiques, il faut appliquer l'opération spécifiée par l'instruction (**add**, **sub**, **and**, etc) aux opérandes copiés dans la phase *decode*.
Introduction de l'unité arithmétique et logique **ALU** et du registre tampon **D**.



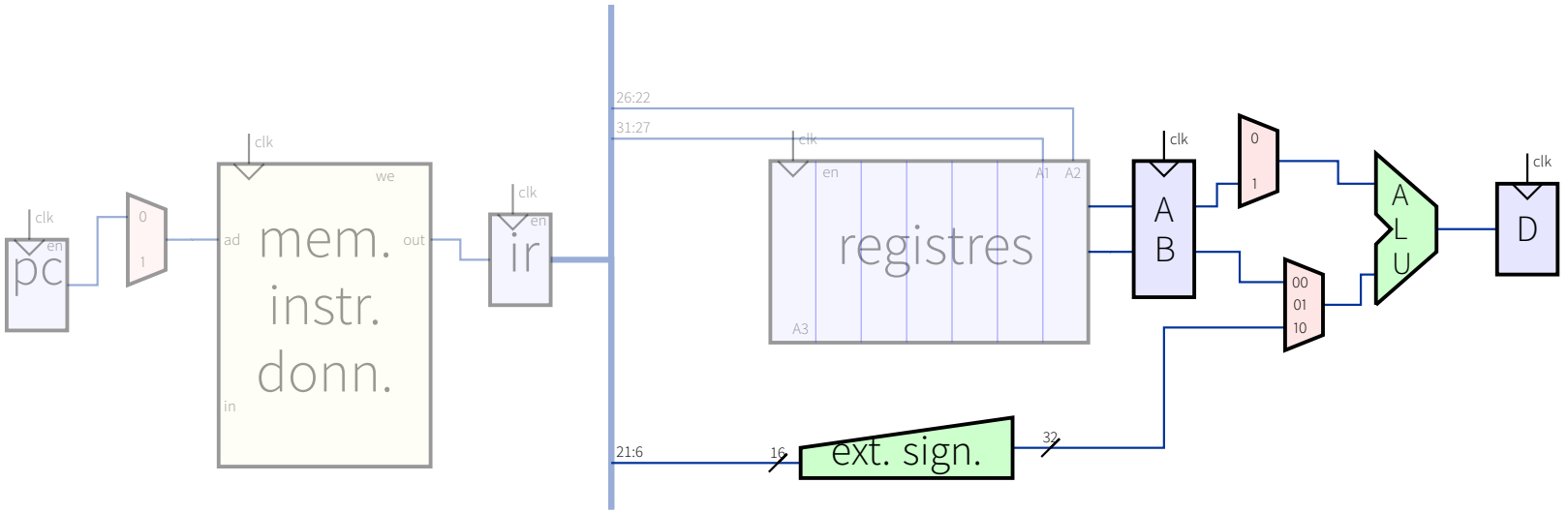
Instructions arithmétiques et logiques (cont.)

L'unité arithétique et logique utilise comme opérandes les registres tampons **A** et **B**.
Le résultat de l'opération effectuée par l'ALU est copié dans le registre tampon **D**.



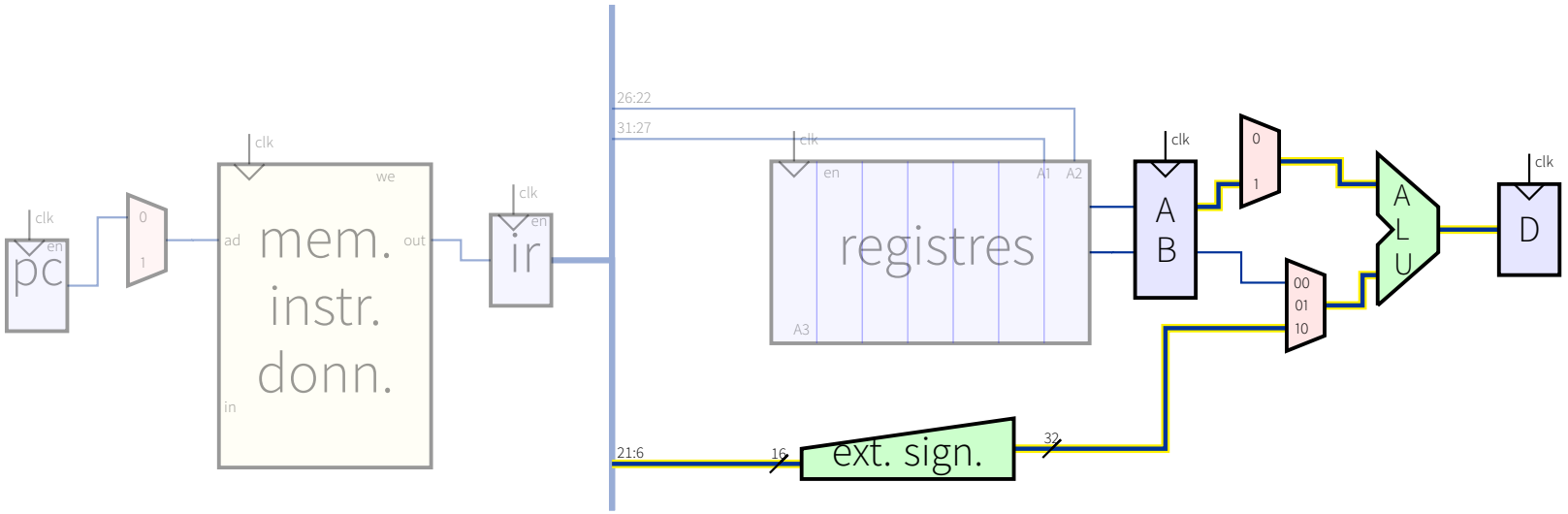
Instruction avec immédiat

Pour les instructions avec un immédiat (**`addi, subi,`** etc), l'opération doit porter entre le registre **`rA`** et l'immédiat présent sur les 16 bits (21:6).
Extraction des bits 21:6 de **`ir`** et extension signée à 32 bits.



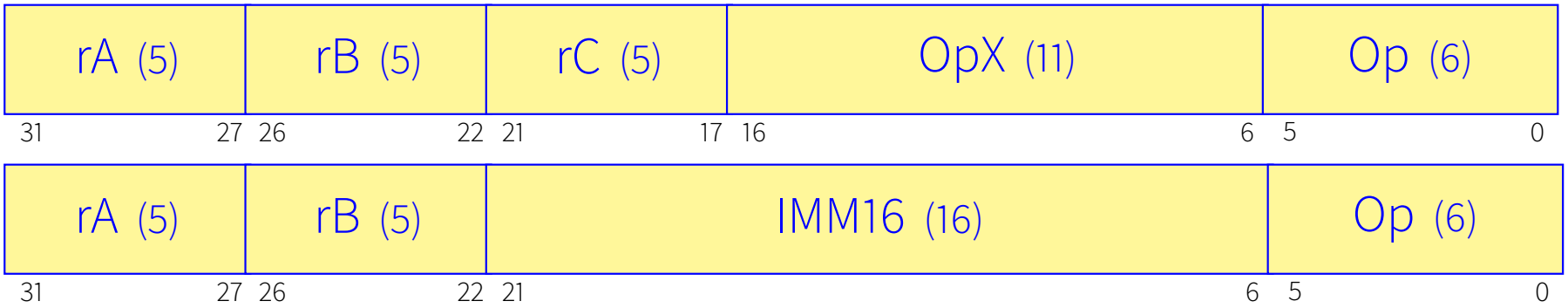
Instruction avec immédiat (cont.)

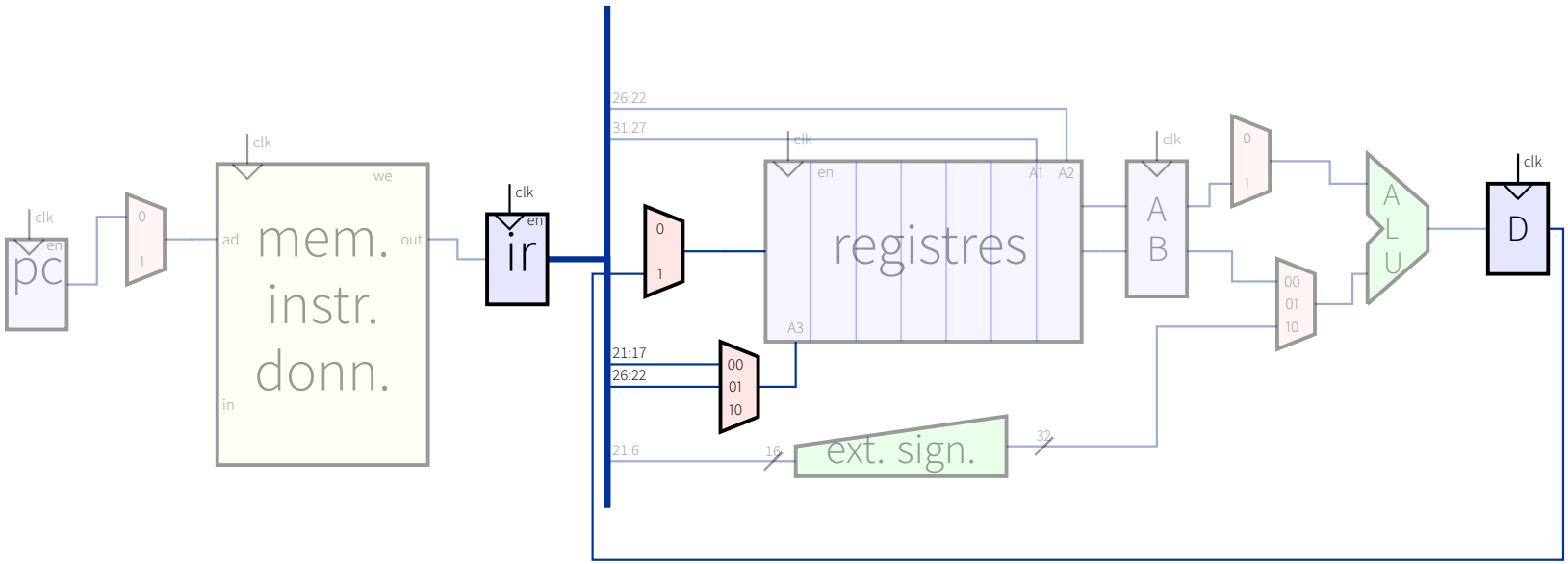
Comme précédemment le résultat du calcul est copié dans le registre **D**.



Rangement du résultat

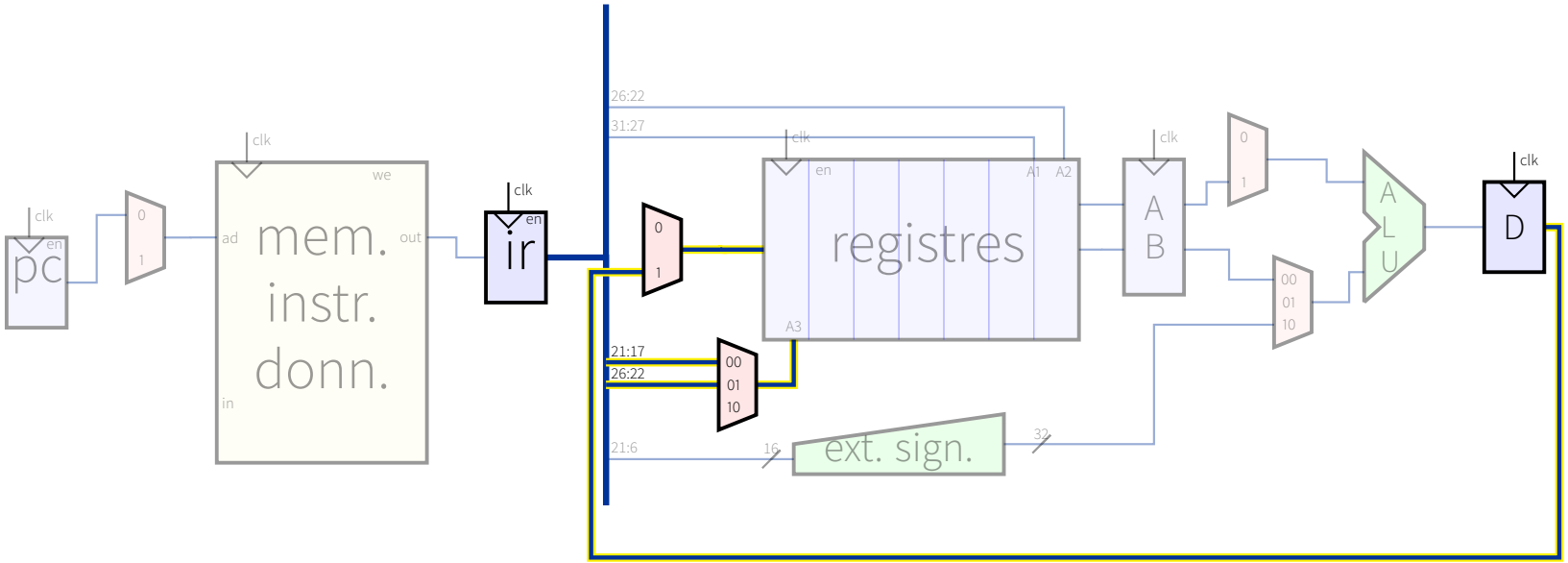
Une fois le calcul effectué et copié dans **D**, il faut ranger le résultat dans le banc de registres.
Suivant le type d'instruction, le registre destination est décrit par `rC` ou `rB`.





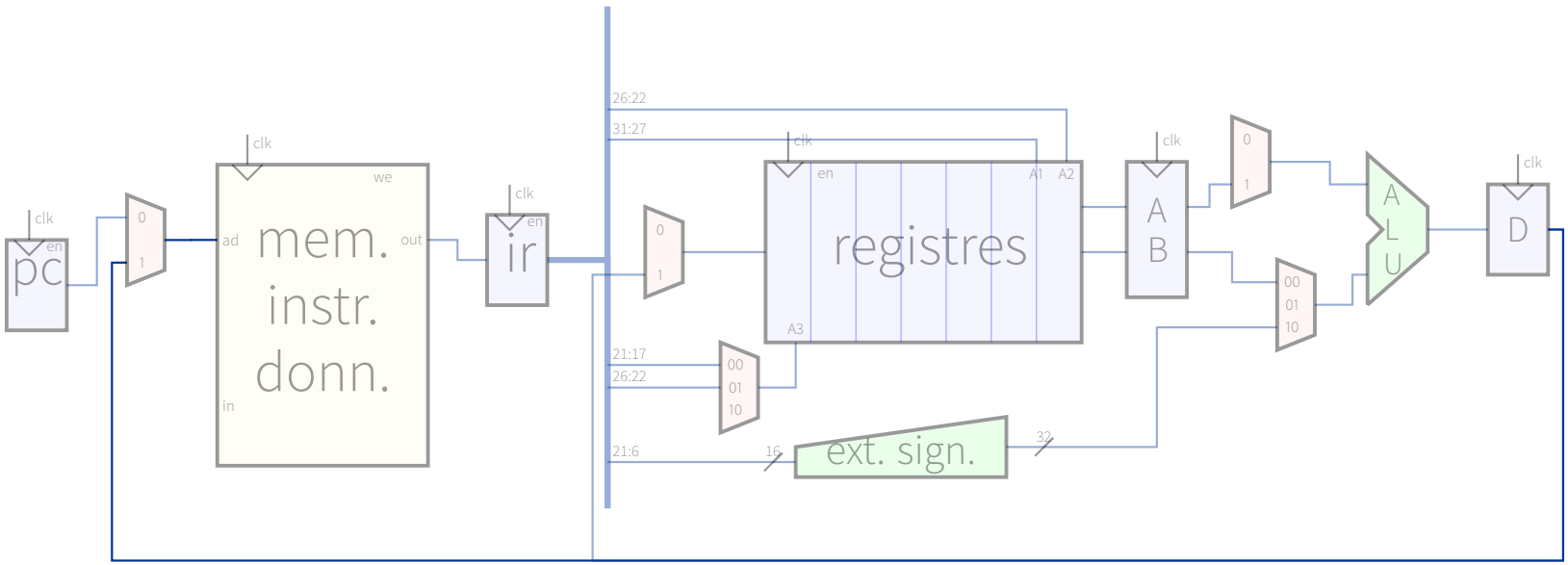
Rangement du résultat (cont.)

Pour les instructions entre registres, recopie du résultat dans le registre `rC` (décrit par les bits 21:17 de l'instruction)
Pour les instructions avec un immédiat, recopie du résultat dans le registre `rB` (décrit par bits 26:22 de l'instruction)



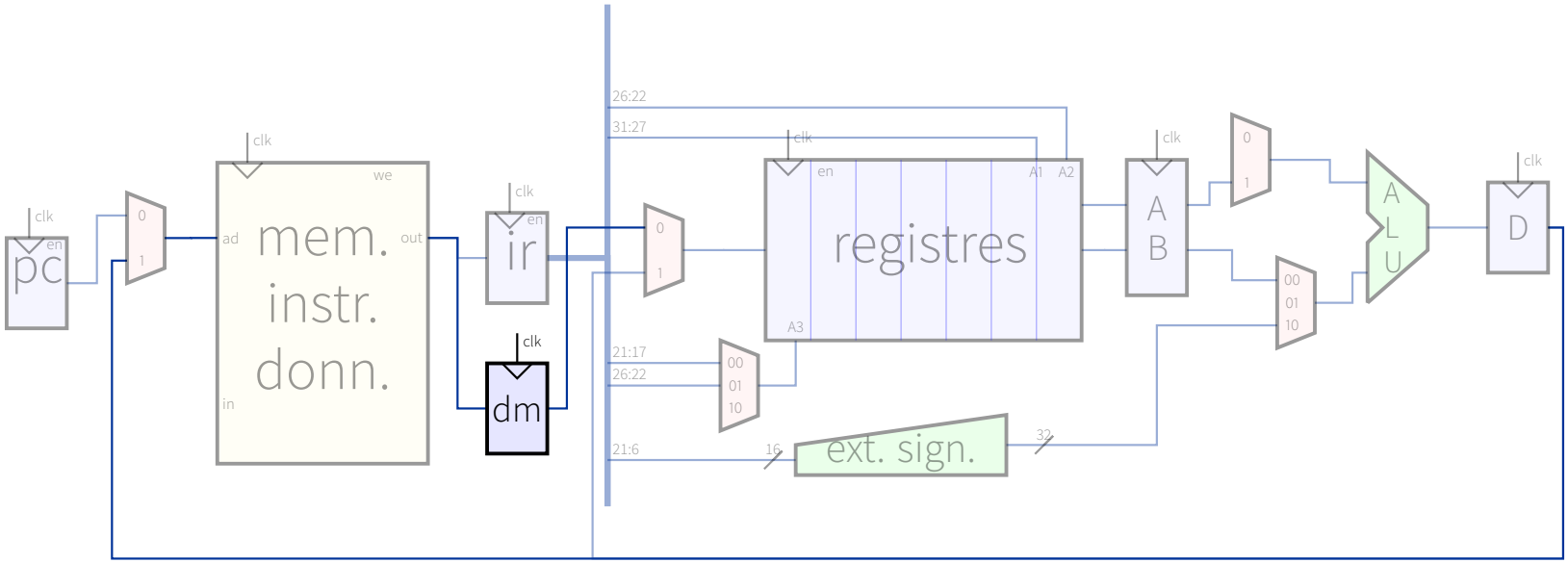
Accès mémoire

Les accès mémoire nécessitent de calculer l'adresse en ajoutant le registre `rA` à un immédiat.
Le mécanisme de calcul est identique aux instructions ALU avec un immédiat.
Le résultat du calcul servira ensuite à adresser la mémoire.
Ajout d'un chemin entre **D** et l'entrée **ad** de la mémoire.



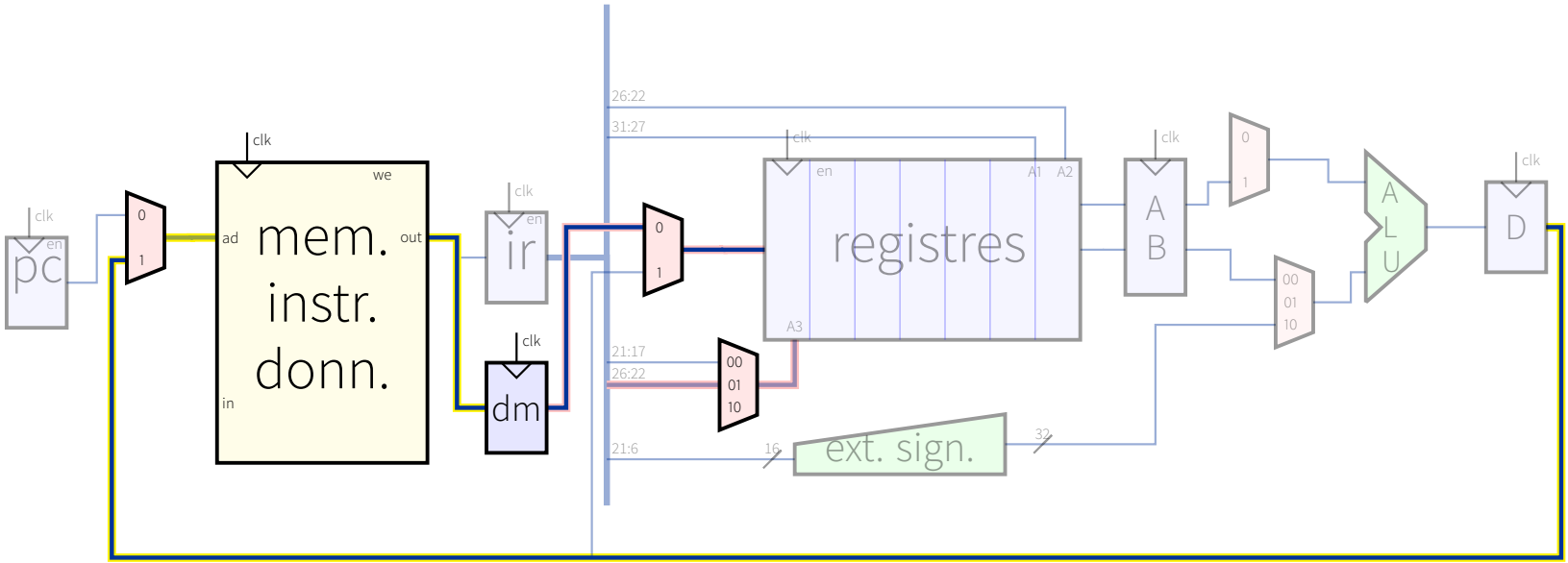
Accès mémoire : chargements

Dans le cas d'un chargement (**`ldb, ldh, ldw, etc`**), la mémoire est ensuite lue et son résultat est écrit dans un *registre de données mémoire* (**`dm`**).



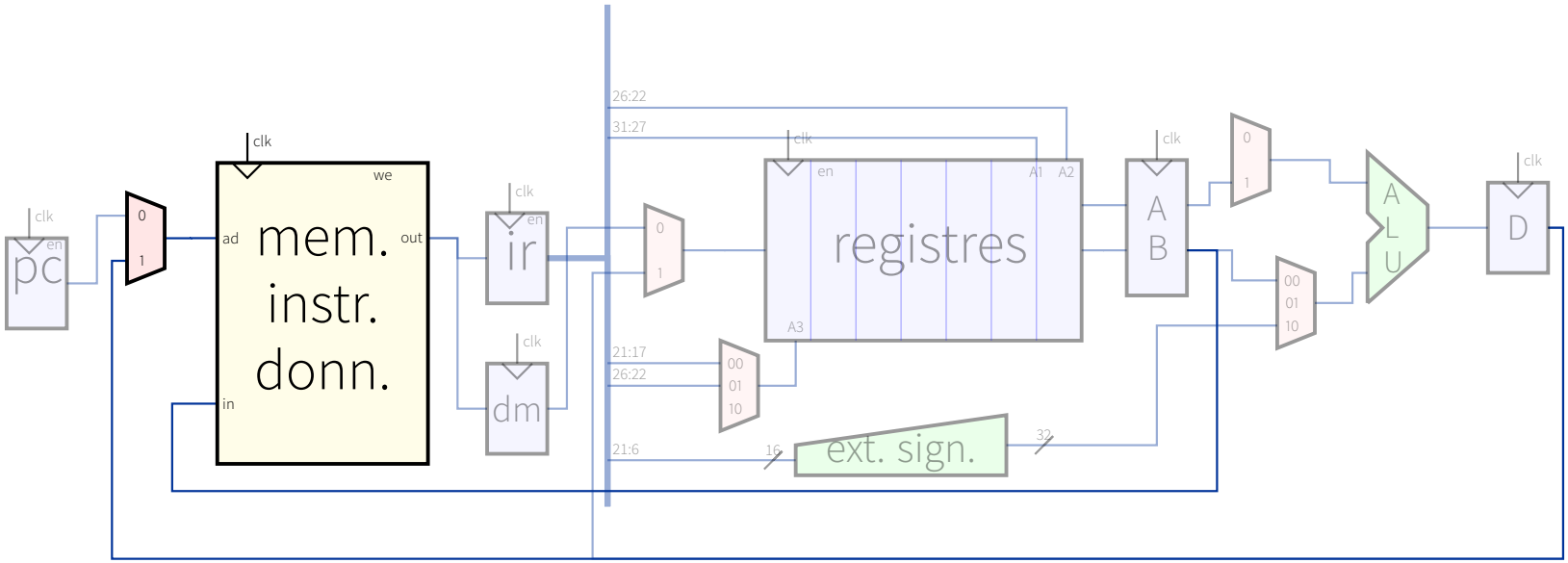
Accès mémoire : chargements (cont.)

Une fois la mémoire lue, dans un deuxième temps, le contenu du registre **dm** est copié vers le registre **rB** du banc de registres.



Accès mémoire : rangements

La partie calcul d'adresse est identique pour les instructions de rangement *store* (`stb, sth,` etc).
Il faut ensuite copier le registre de numéro `rB` dans la mémoire.
Celui-ci a été chargé dans le registre tampon **B** lors de la phase *decode*.
Chemin de **B** vers l'entrée de la mémoire.

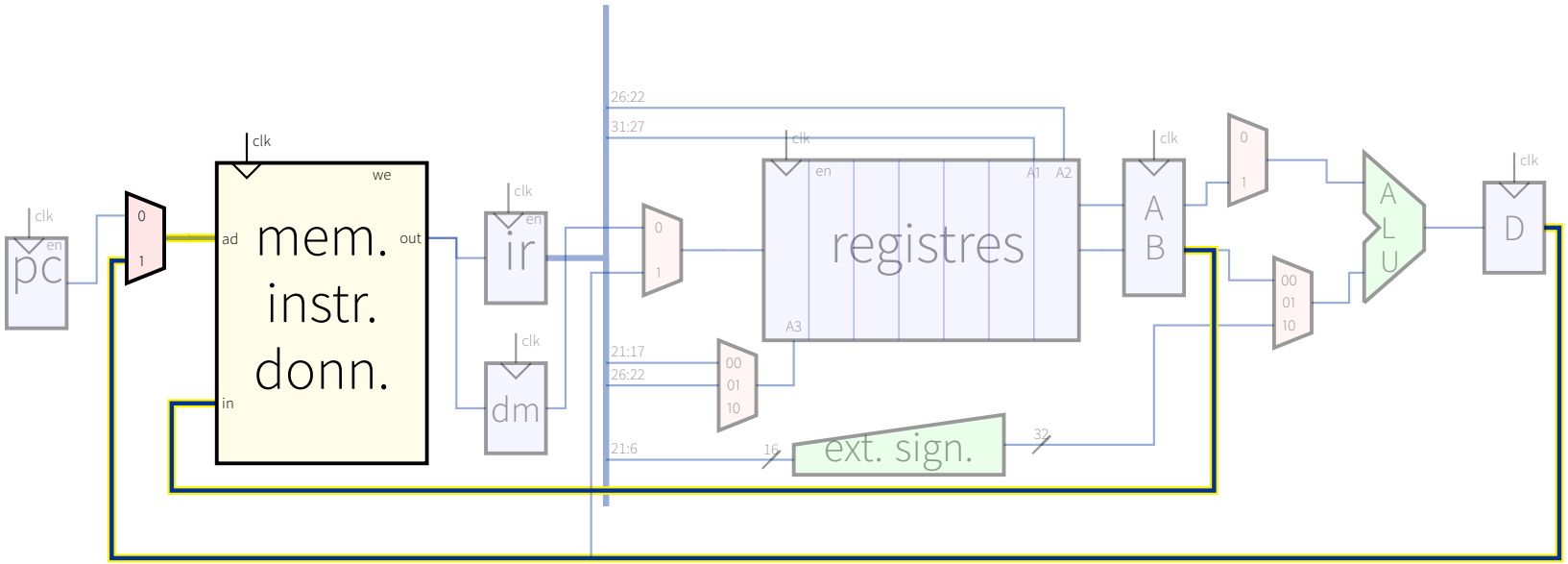


Accès mémoire : rangements (cont.)

Pour effectuer l'écriture, on transfère simultanément :

- le registre **D** qui contient le résultat `rA + immₙ₆` vers l'entrée d'adresse de la mémoire.
- le registre **B** (qui est une copie de `rB`) vers l'entrée de données de la mémoire.

et on donne un ordre d'écriture à la mémoire.



Mise à jour du compteur de programme

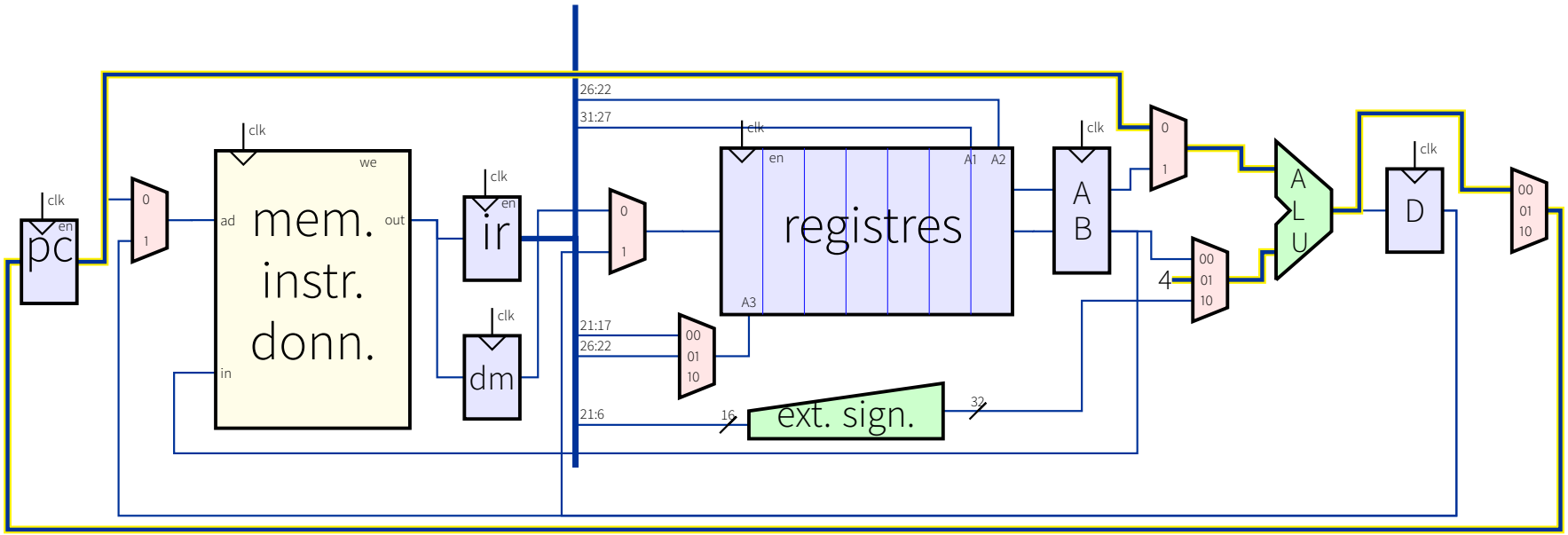
Lors de la phase de *fetch*, incrémentation du compteur de programme **pc**.

`pc <- pc+4`

Chemin de **pc** vers l'entrée de l'ALU

Immédiat "4" à la deuxième entrée de l'ALU

Chemin de la sortie de l'ALU vers l'entrée de **pc**

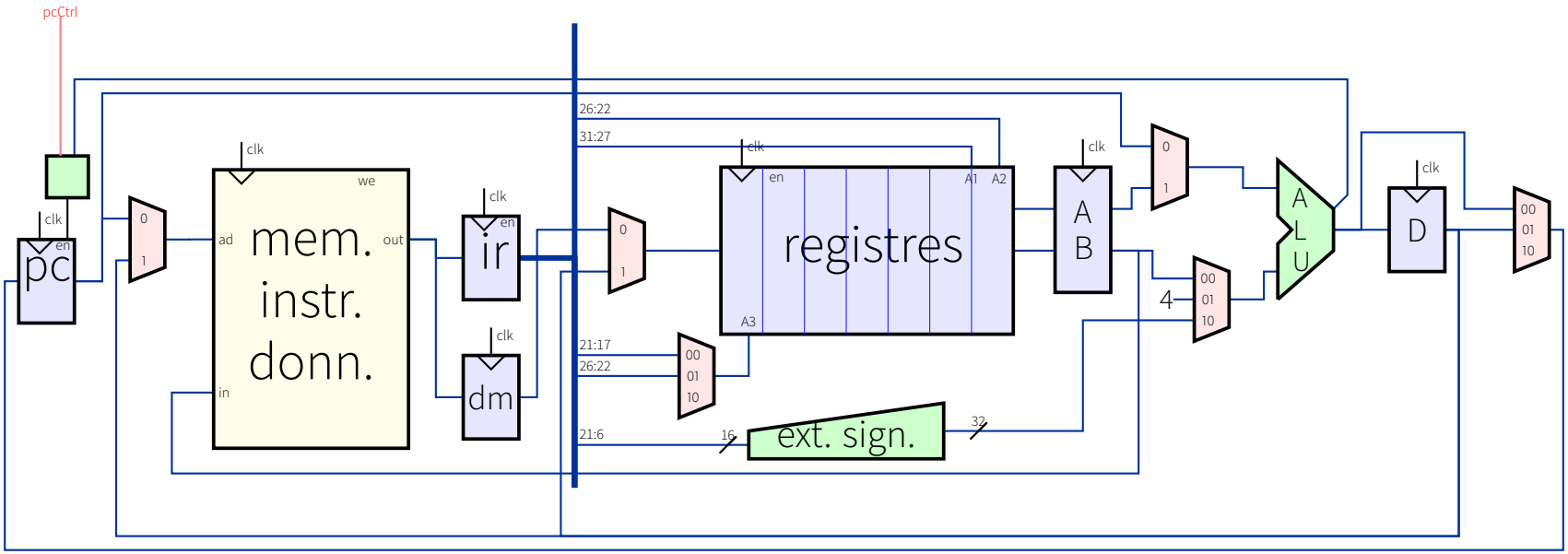


Branchements

Les branchements nécessitent :

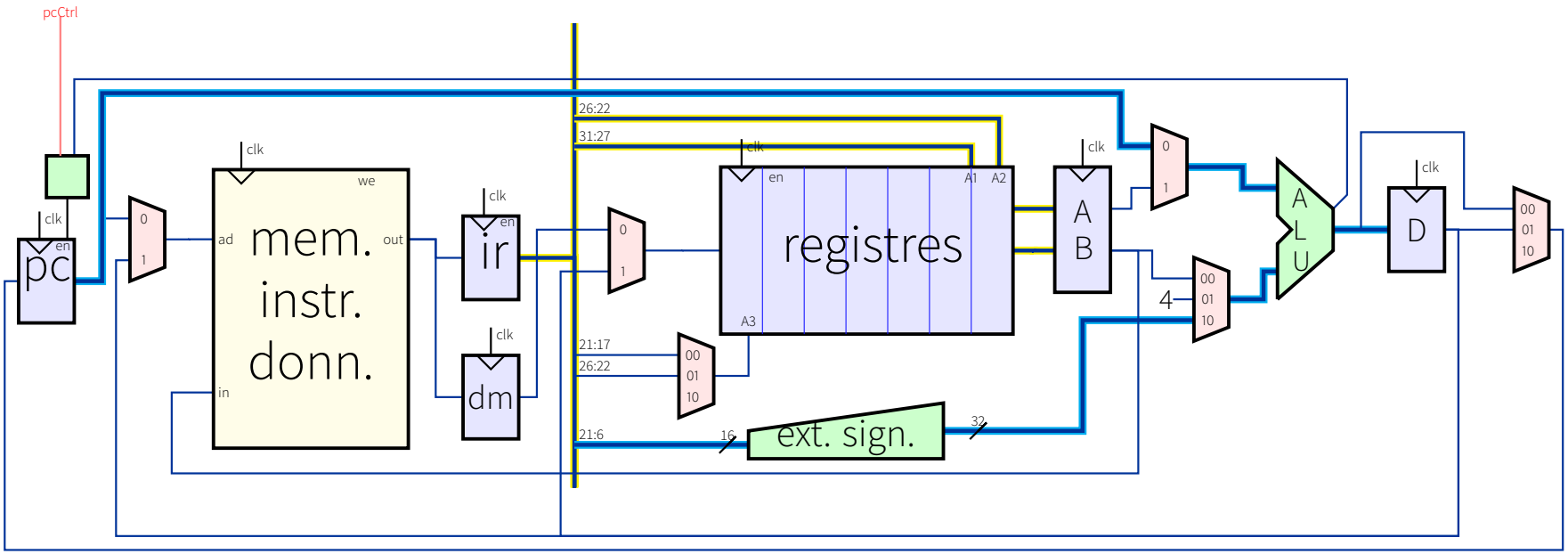
- de calculer la somme de `pc` et d'un $\texttt{imm}_{16}$ en recopiant le résultat dans `pc`. Ceci peut se faire par l'ALU avec les chemin de données existant.
- d'utiliser l'ALU pour effectuer une comparaison. On sortira un bit indiquant le résultat de la comparaison (`eq,ne, gt,` etc) de l'ALU.

Par contre, comme l'ALU est utilisée deux fois, il n'est pas possible de faire ces deux opérations dans une même phase.



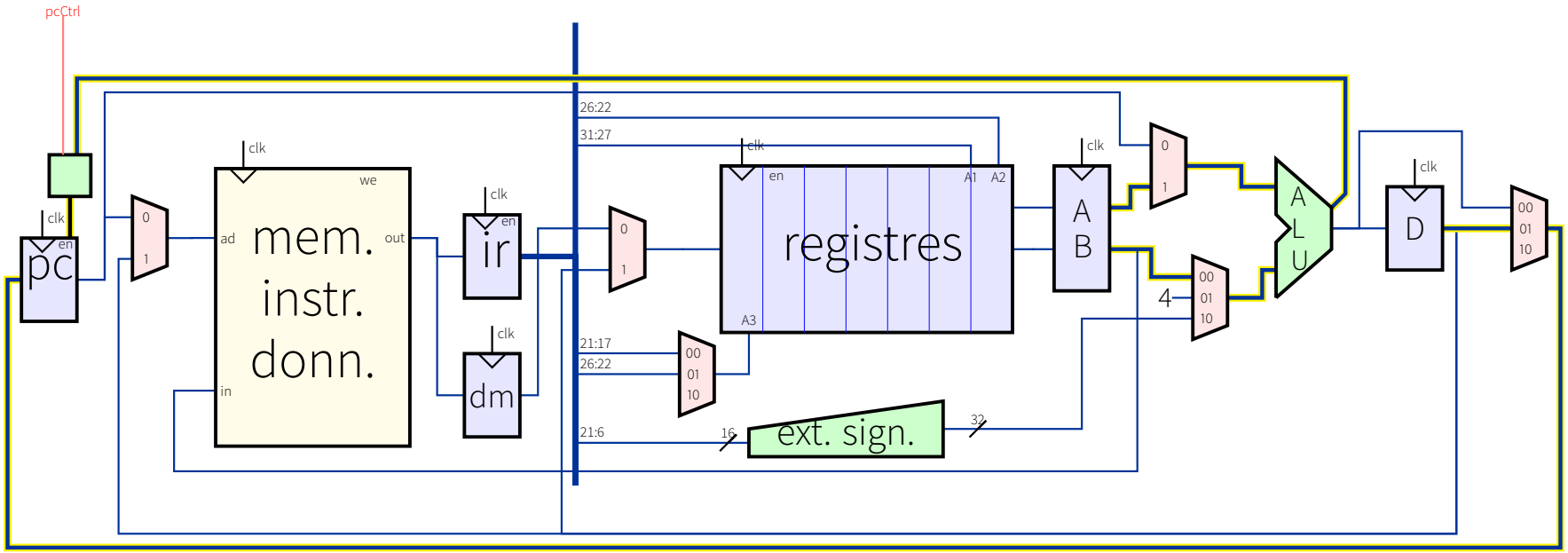
Branchements (cont.)

On modifie la phase *decode*.
En plus des opérations normales de la phase *décode* (___)
on calcule *toujours* la somme de *pc* et de l'immédiat (___)
L'adresse (éventuelle) de branchement est dans le registre **D**.



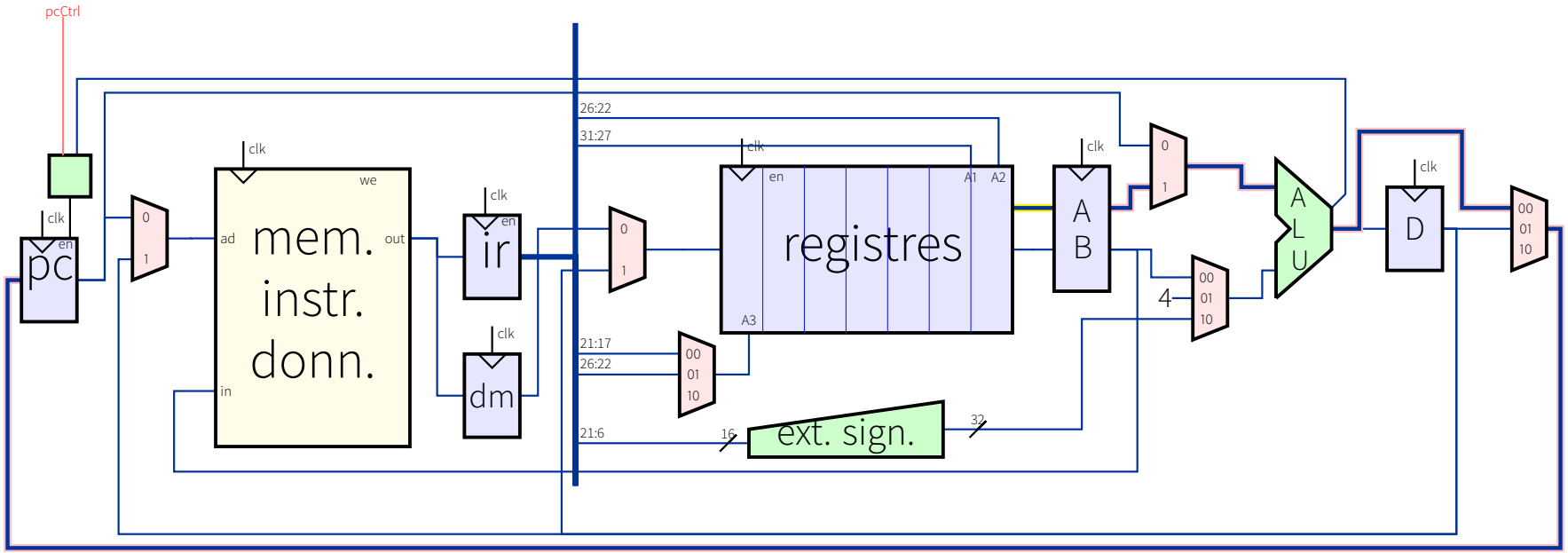
Branchements (cont.)

On peut ensuite effectuer le branchement.
On utilise l'ALU pour faire la comparaison entre les registres **A** et **B** et valider (ou non) l'autorisation d'écriture de **pc**
Simultanément, on amène le contenu du registre D vers l'entrée du registre **pc**.



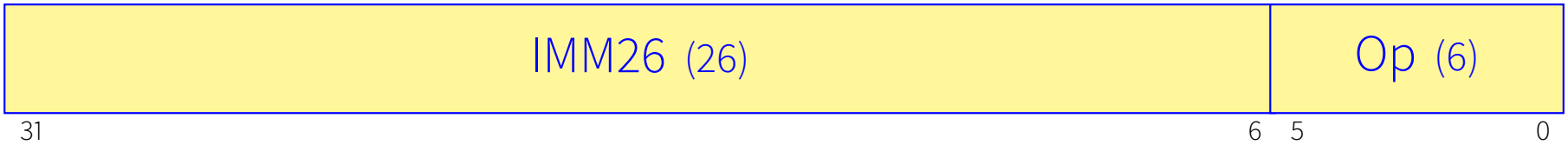
Sauts

Les instructions de sauts existent sous deux formes.
Dans la version registre **`jmp`**, il faut copier le contenu du registre **`rA`** dans PC.
Peut se faire en phase 3 sans modification architecturale.
En phase 2 (**`decode`**), rA est copié vers A
En phase 3, on le transfère vers pc



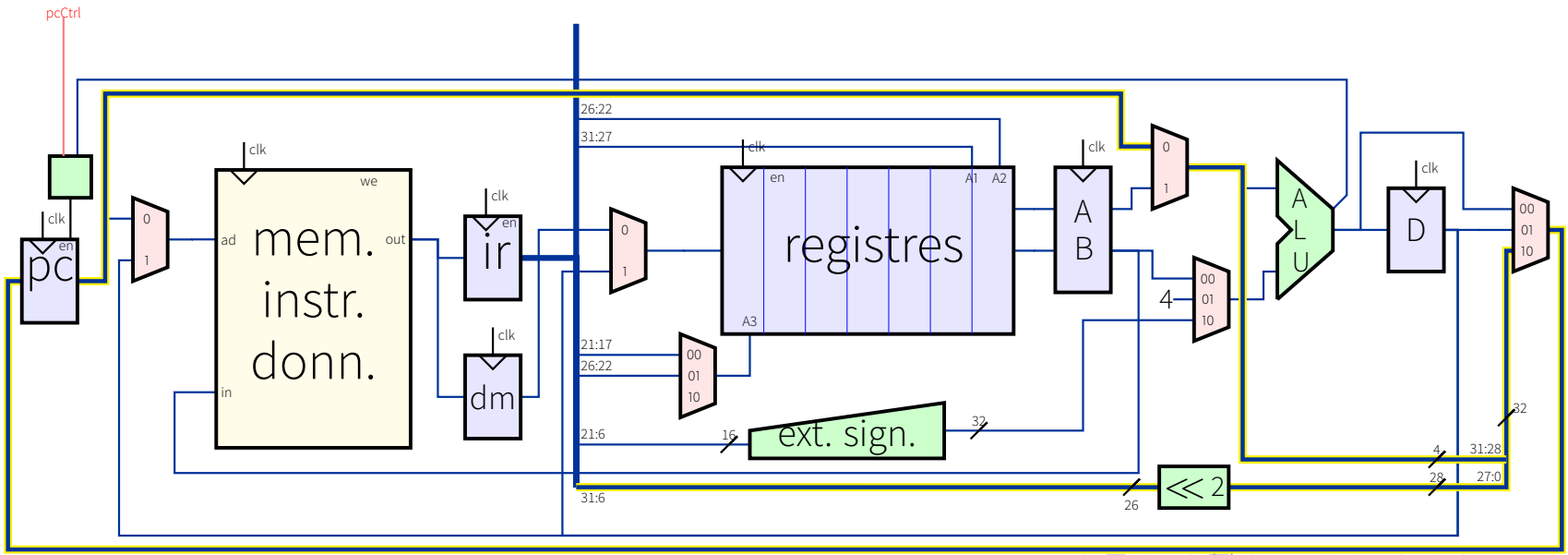
Sauts (cont.)

Dans la version avec immédiat **`jmpi`**, utilisation d'un immédiat sur 26 bits.



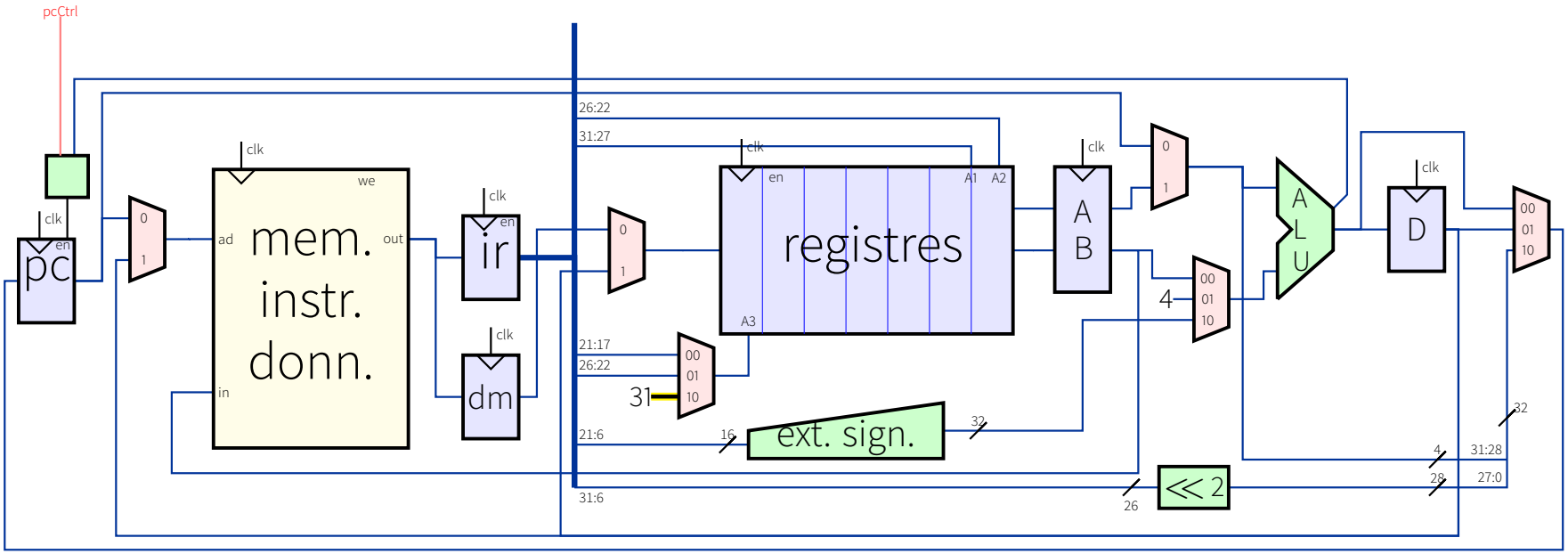
Les 26 de poids fort de l'instruction forment un immédiat, qui doit être :

- multiplié par 4
- complété en poids fort par les 4 bits de poids forts de **`pc`**
- copié dans **`pc`**



Appels de procédure

Similaires aux sauts, mais il faut pouvoir réécrire `pc` dans le registre `ra` (`r31`)
Ajout de la génération d'une adresse à 31 dans la sélection du registre d'écriture.[1]

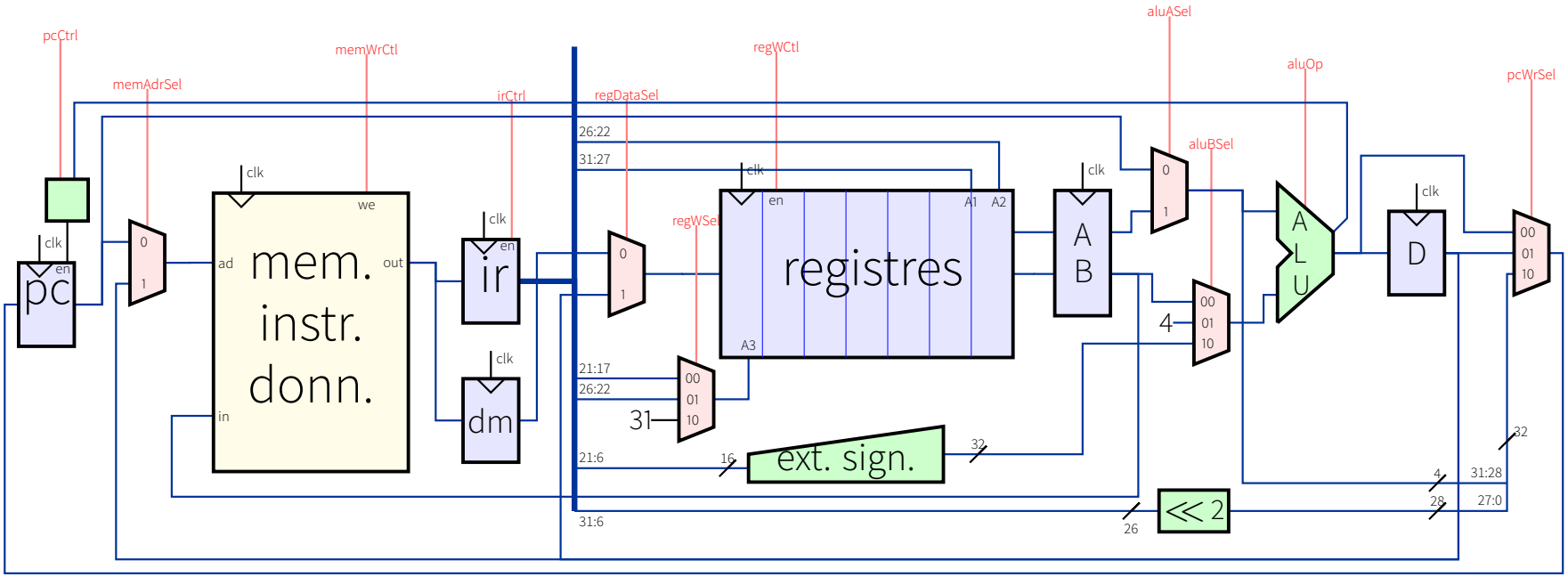


[1] La mise en oeuvre sera étudiée en TD

Contrôle du processeur

Nécessité de générer des commandes pour :

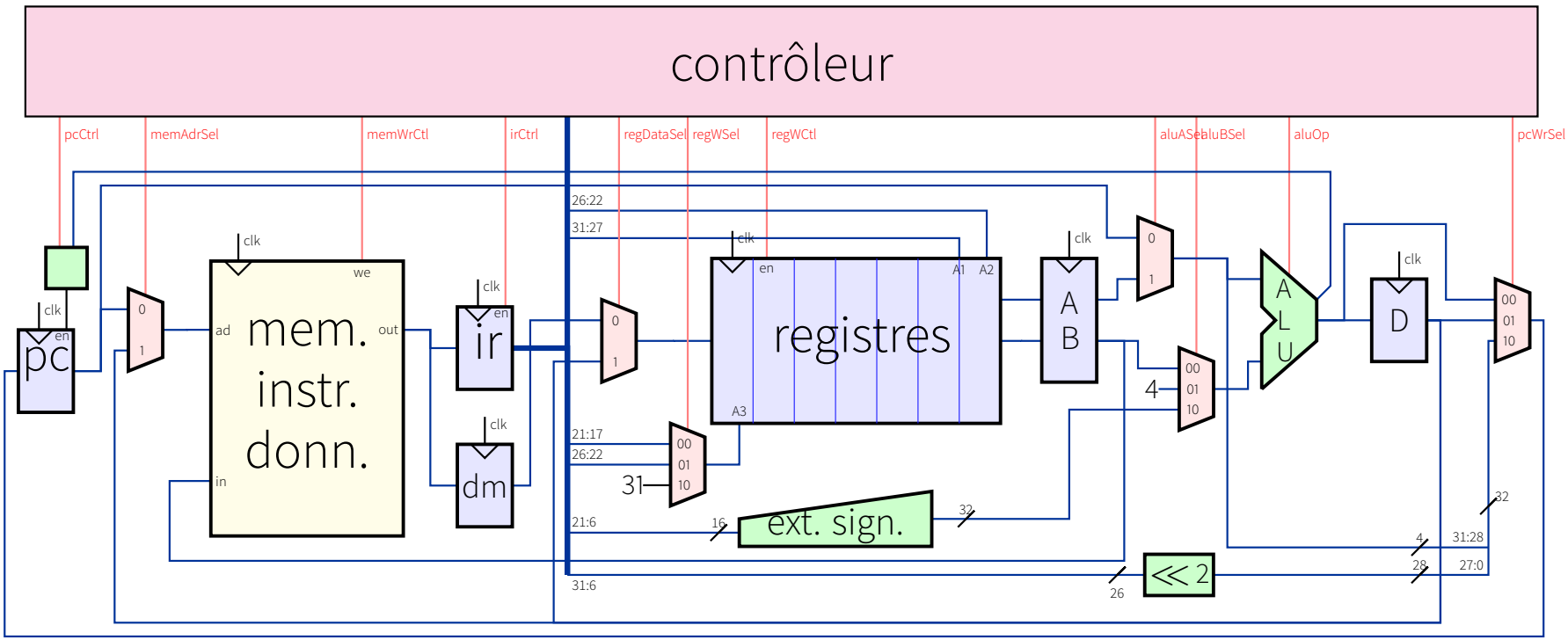
- registres avec contrôle d'écriture (*enable*)
- multiplexeurs
- unité arithmétique et logique



Contrôle du processeur (cont.)

Se fait au moyen d'un séquenceur (ou *contrôleur*) entrée : registre d'instruction, sorties : signaux de contrôle de l'architecture :

- **`memAdrSel`** sélection adresse mémoire et **`memWrCtl`** contrôle écriture
- **`irCtrl`** écriture registre instruction
- **`regWCtl`** contrôle écriture du banc de registres, **`regDataSel`** sélection source de données et **`regWSel`** sélection du registre écrit
- **`aluOp`** opération ALU, **`aluASel`** et **`aluBSel`** sélection des opérandes
- **`pcCtrl`** contrôle écriture de pc, **`pcWrSel`** sélection source

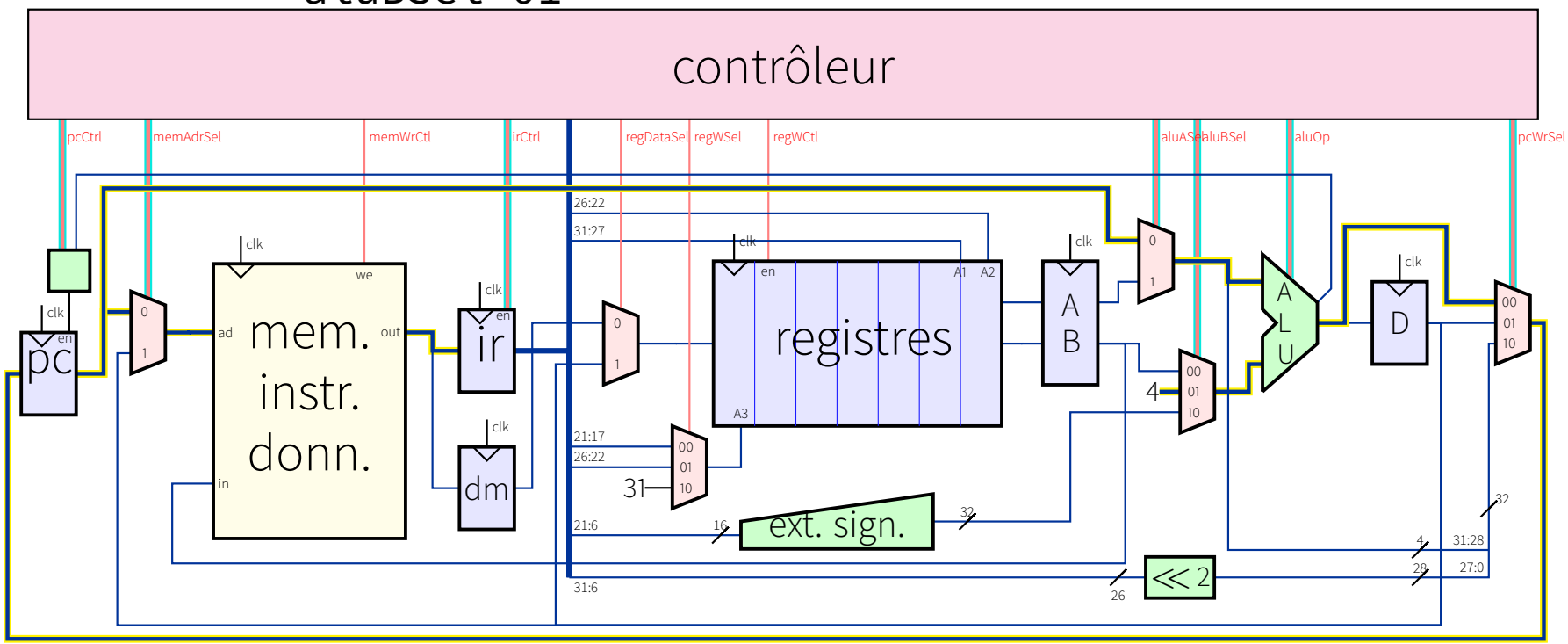


Contrôle du processeur (cont.)

Premier état : acquisition de l'instruction *fetch*

Correspond à
```
pcWrSel=00
pcCtrl=wr
memAdrSel=0
irCtrl
aluOp=add
aluASel=0
aluBSel=01
```

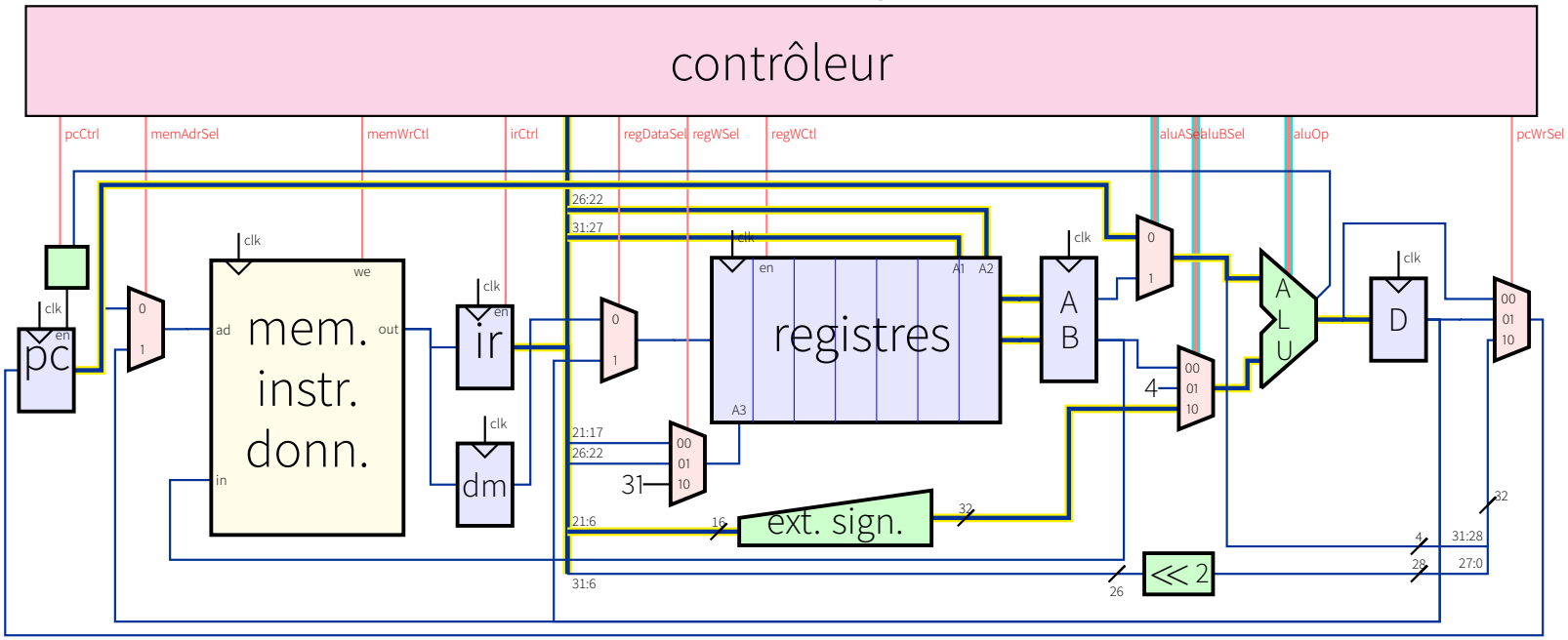
fetch:
```
ir <- mem[pc]
pc <- pc + 4
```



Contrôle du processeur (cont.)

Deuxième état : décodage de l'instruction *decode*
identique pour toutes les instructions
Ecriture dans les registres **A** et **B** systématique (aucun contrôle).
Le calcul `D <- pc + imm16` est nécessaire pour les branchements.
Correspond à `aluOp=add`
`aluASel=0`
`aluBSel=10`

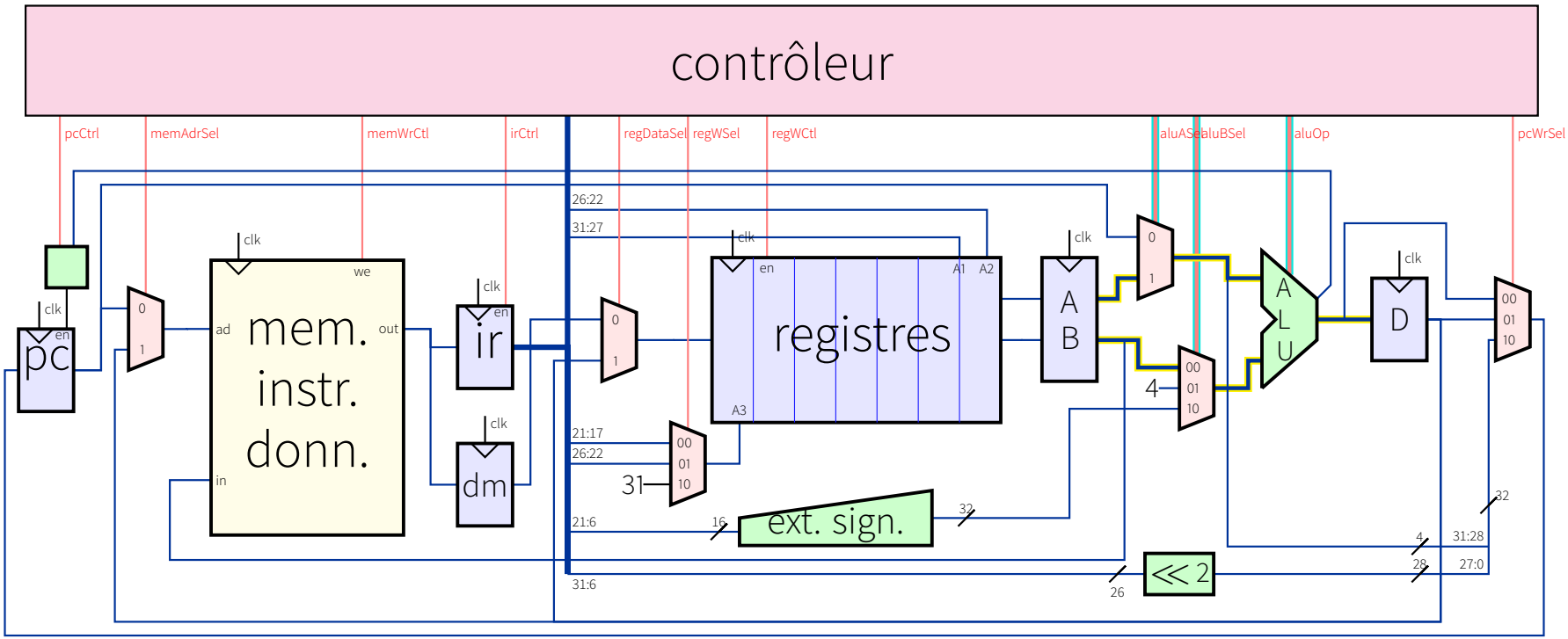
decode:
`A <- rA`
`B <- rB`
`D <- pc + imm16`



Contrôle du processeur (cont.)

Les autres états dépendent des types d'instruction.
Exemple : addition entre registres (et toutes des instructions arithmétiques et logiques de type R). *add*
Correspond à `aluOp=add`
`aluASel=1`
`aluBSel=00`

add-execute:
`D <- A + B`

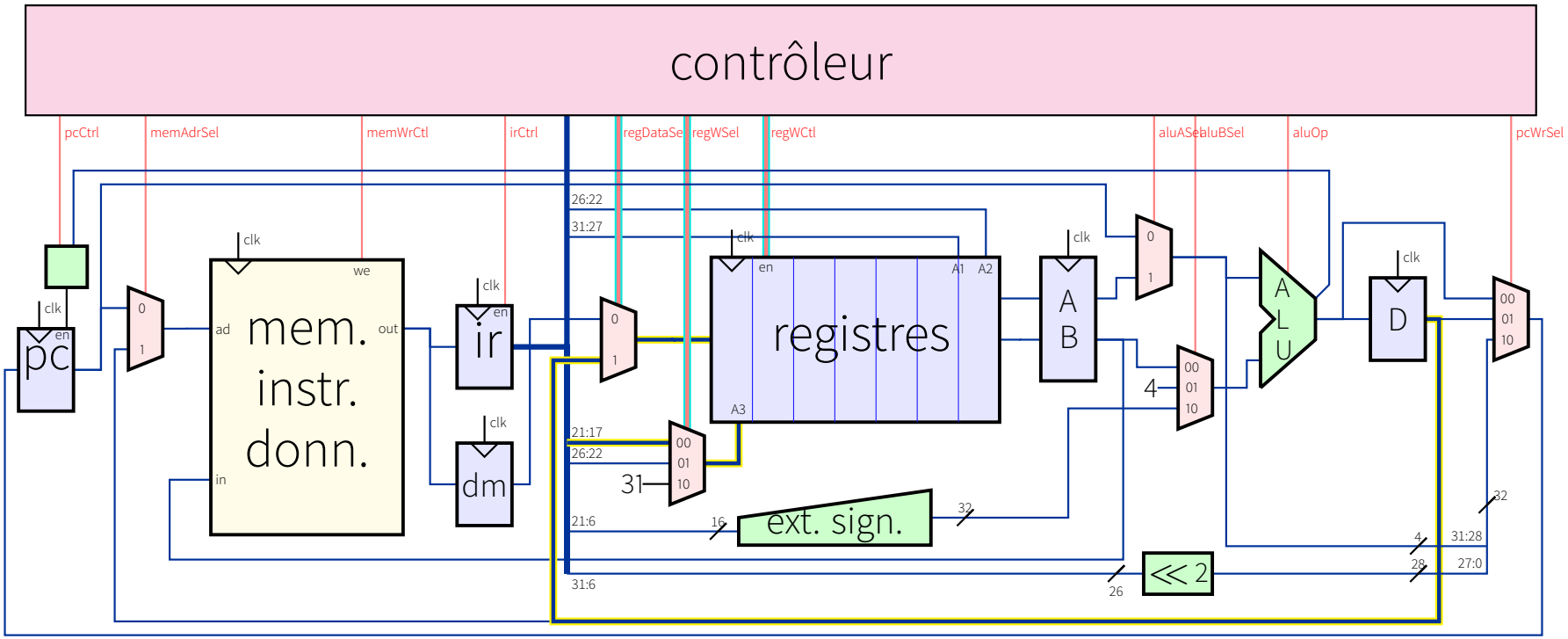


Contrôle du processeur (cont.)

La dernière phase de l'exécution d'une addition est une réécriture du résultat dans le registre `rC`

Correspond à `regWCtl`
`regDataSel=1`
`regWSel=00`

add-rr:
`rC <- D`



Contrôle du processeur (cont.)

On peut donc décrire l'instruction `add` par

1. `fetch`
2. `decode`
3. `D <- A + B`
4. `rC <- D`

On peut faire de même pour toutes les instructions :
`addi` (et toutes instruction arithmétiques et logiques avec un immédiat)

1. `fetch`
2. `decode`
3. `D <- A + imm`$_{16}$
4. `rB <- D`

Contrôle du processeur (cont.)

Instructions d'accès mémoire

`load`

1. `fetch`
2. `decode`
3. `D <- A + imm16`
4. `dm <- mem[D]`
5. `rB <- dm`

`store`

1. `fetch`
2. `decode`
3. `D <- A + imm16`
4. `mem[D] <- B`

Contrôle du processeur (cont.)

Instructions de contrôle de flot

`br`	`jmp`	`jmpi`
1 `fetch`	1 `fetch`	1 `fetch`
2 `decode`	2 `decode`	2 `decode`
3 `if(pcComp==1)` `pc <- A+imm`$_{16}$	3 `pc <- rA`	3 `pc<-pc`$_{31:28}$`:(imm`$_{26}$`<<2)`

NB : Le signal de contrôle d'écriture de `pc` doit être à 1 :

- dans la phase `fetch`
- dans la phase 3
  - pour un `jmp` ou un `jmpi`
  - pour un `br` si la comparaison est vraie

Le contrôleur génère 2 bits : `wrpc` et `br` et l'écriture de `pc` est vraie si (`wrpc` || (`br` && `pcComp`))

Contrôle du processeur (cont.)

On peut en déduire l'automate de contrôle