Ch. 6
Micro-architecture

B. Quoitin
(bruno.quoitin@umons.ac.be)

Table des Matières

Objectifs

Du jeu d'instructions à la micro-architecture

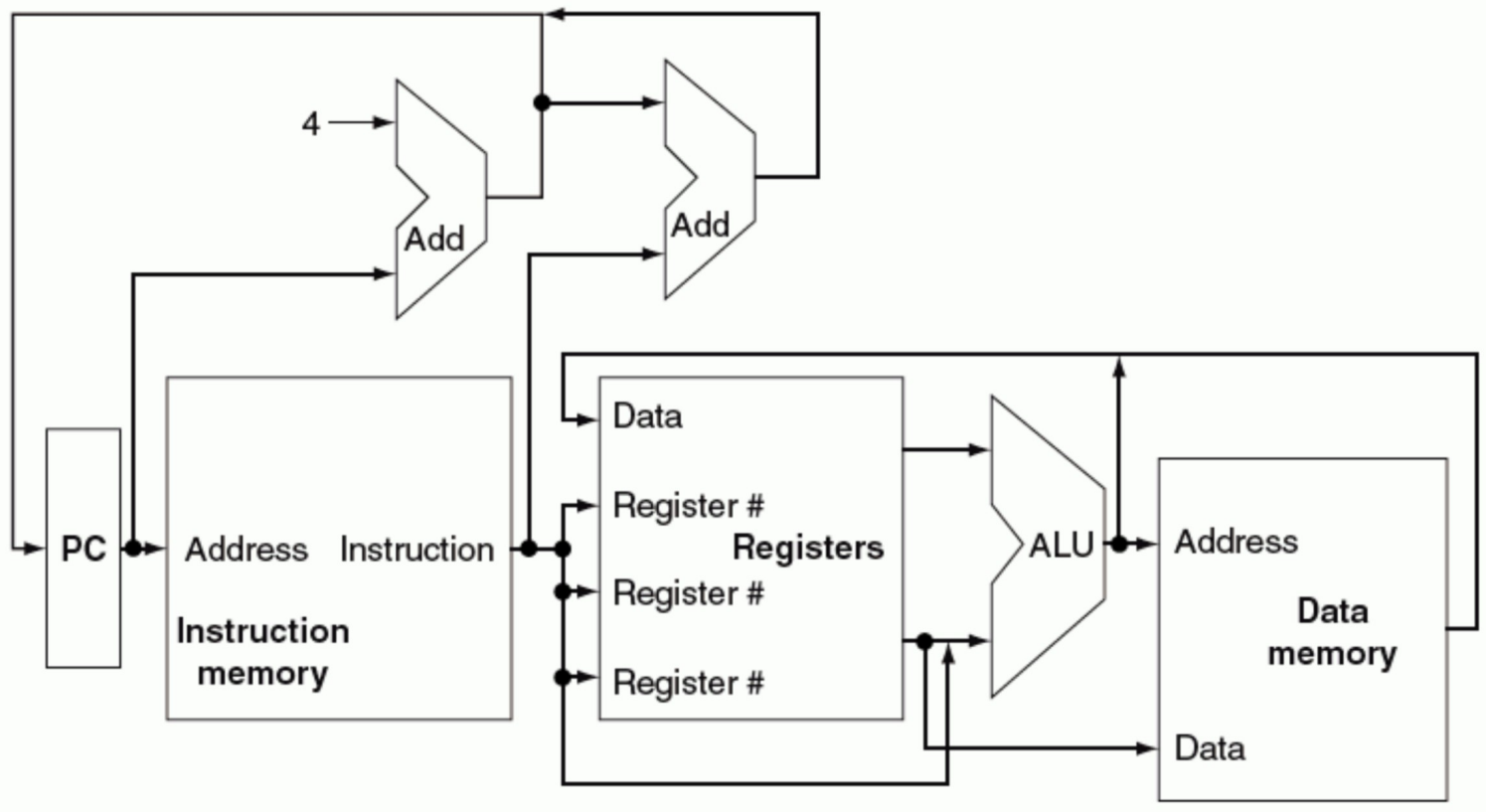
- Construire pas-à-pas l'**implémentation d'un processeur** en utilisant les blocs logiques du Chapitre 4.
- Renforcer la compréhension du langage machine et l'encodage des instructions sous forme de codes binaires.

Implémentation simplifiée

- Les instructions s'exécutent en **1 cycle d'horloge** (processeur mono-cycle).
- Support d'un **nombre limité d'instructions** : ADD, AND, OR, ADDI, ANDI, ORI, LUI, LW, SW, J, BEQ (toutes les catégories vues au Chapitre 5 sont couvertes).
- Instructions et données situées dans **deux mémoires distinctes**.

Architecture MIPS

- Ce chapitre considère une version simplifiée du chemin de données de l'architecture MIPS : toutes les instructions **s'exécutent en un cycle d'horloge** (processeur mono-cycle).



Source: Patterson & Hennessy, Computer Organization and Design: The Hardware/Software Interface, 3rd edition, 2005

Table des Matières

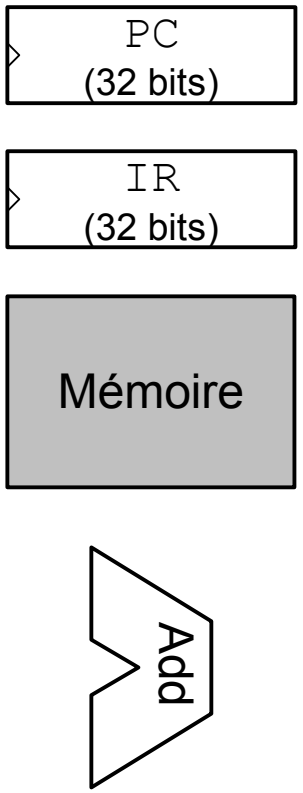
Cycle d'instruction

Implémentation

- L'implémentation de l'*instruction fetch*(1) et du passage à l'instruction suivante nécessite

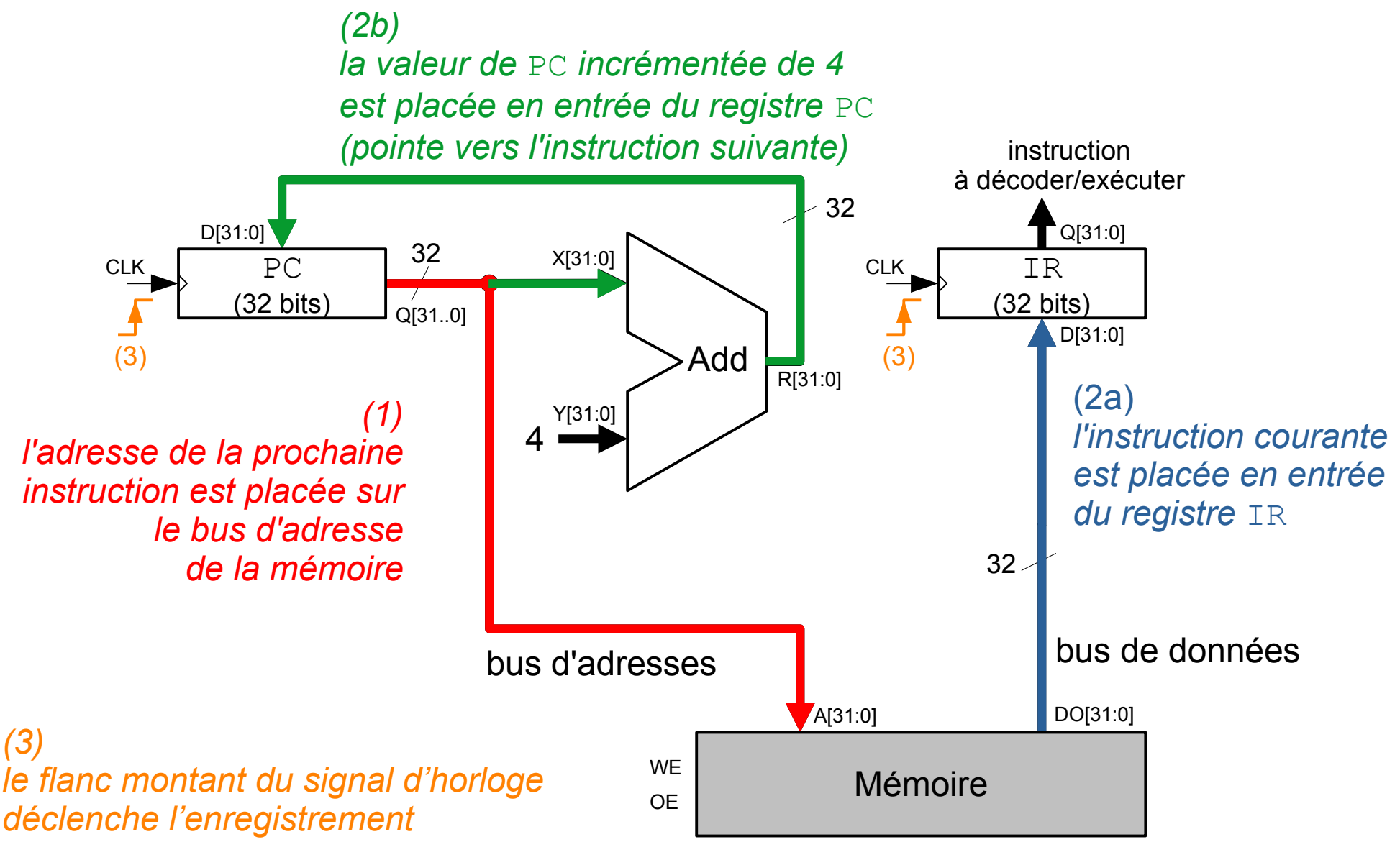
- **Logique séquentielle**
  - le registre pointeur d'instruction (`PC`) donnant l'adresse de l'instruction à obtenir.
  - le registre d'instruction (`IR`) destiné à contenir l'instruction lue.
  - une mémoire contenant les instructions.
    Cette mémoire peut être en lecture seule (ROM).

- **Logique combinatoire**
  - un additionneur pour passer à l'instruction suivante
    (`PC` ← `PC`+4)



(1) *fetch* = « aller chercher »

Cycle d'instruction



Cycle d'instruction

Implémentation

- Dans l'ouvrage de référence, des hypothèses simplificatrices sont prises pour l'implémentation du processeur :
  - Les mémoires d'instructions et de données sont séparées.
  - La mémoire d'instructions est en lecture seule.

- Pour cette raison
  - Le code d'instruction est directement disponible en sortie de la mémoire → il n'est pas nécessaire de le copier dans un registre d'instruction (`IR`).
  - Un seul flanc montant de l'horloge suffit à faire fonctionner le cycle complet d'instruction (processeur mono-cycle).

Cycle d'instruction

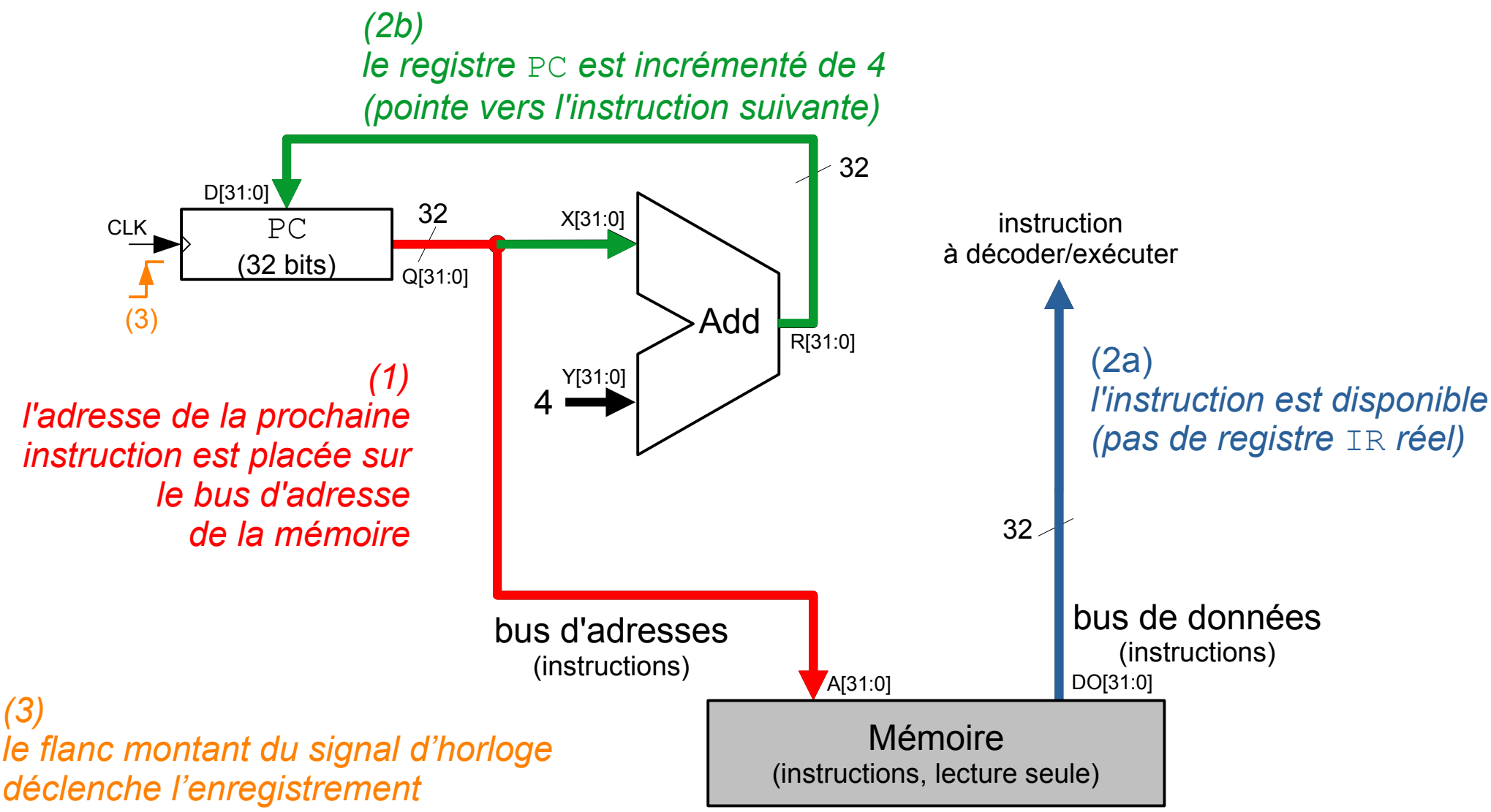
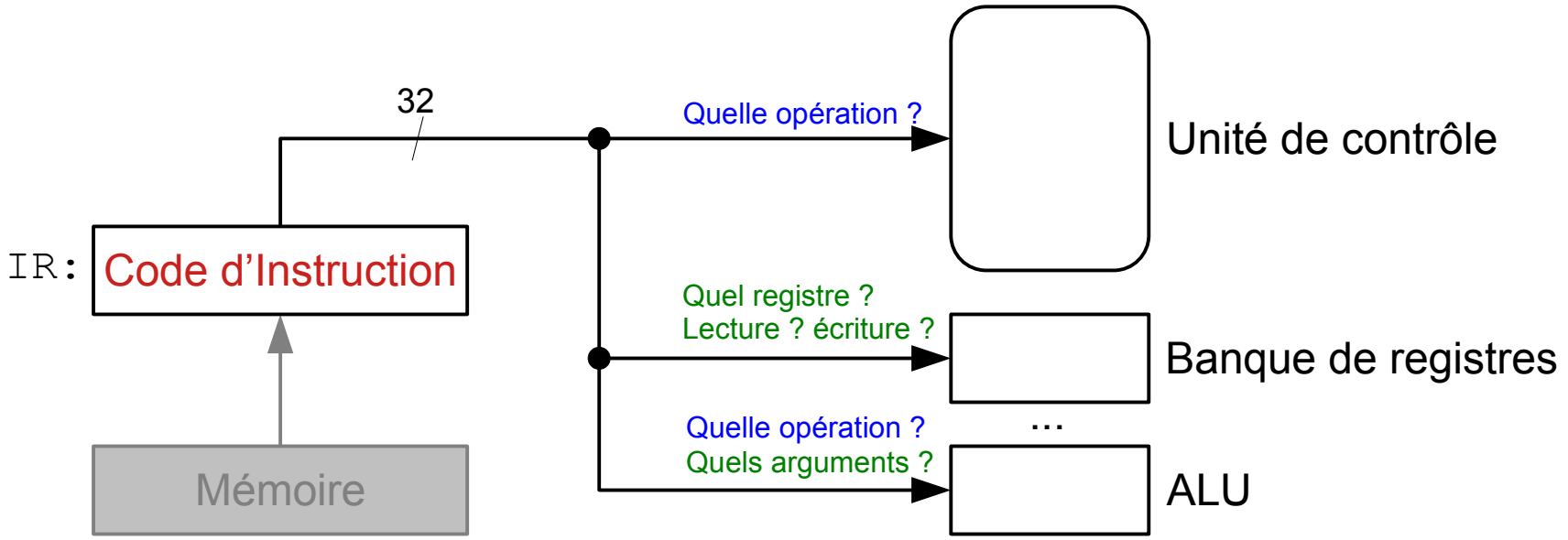


Table des Matières

Décodage d'instruction

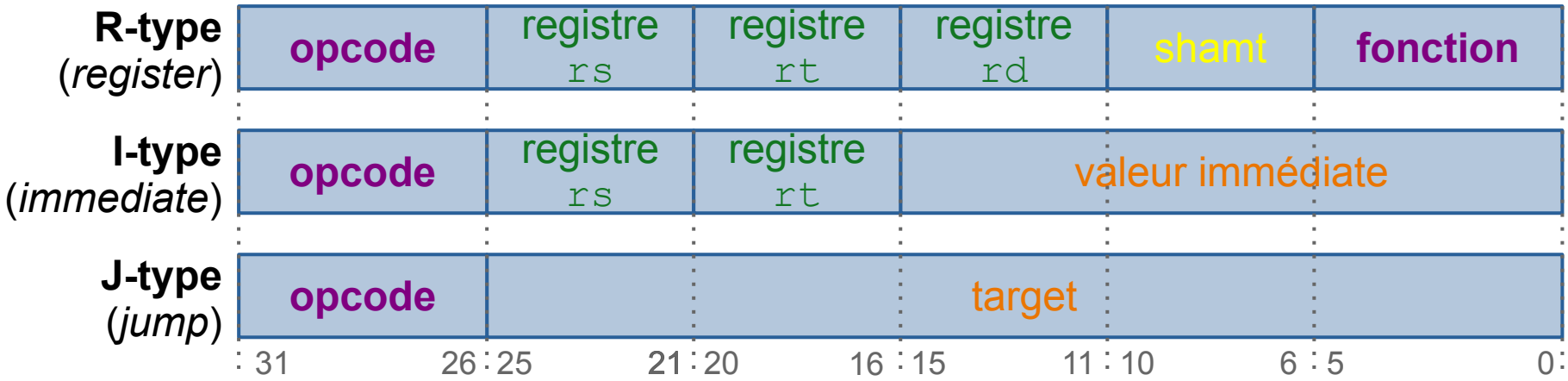
Décodage d'instruction

- Le code d'instruction lu lors de l'*instruction fetch* est utilisé
  - par l'unité de contrôle du processeur et éventuellement par l'ALU pour indiquer **quelle opération doit être effectuée**.
  - par différents éléments du processeur (ALU, banque de registres) et par la mémoire pour indiquer **sur quoi l'opération doit être effectuée**.



Décodage d'instruction

Rappel : MIPS repose sur 3 formats principaux d'instructions



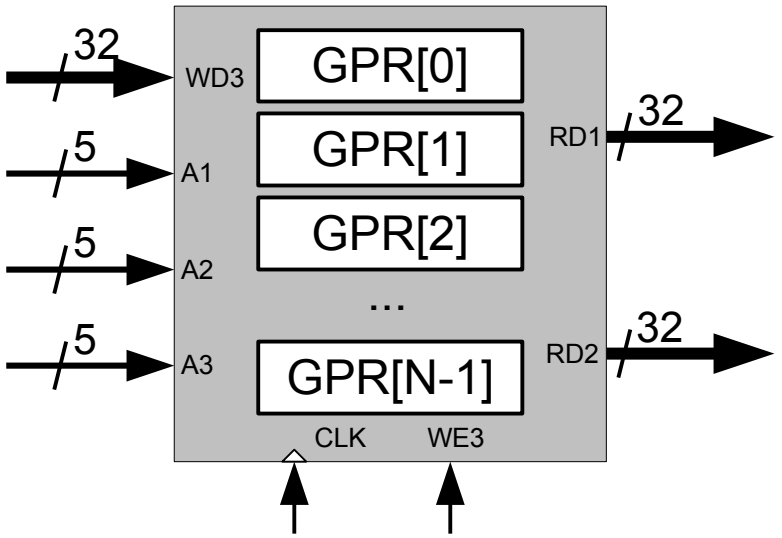
- L'opcode est à une position commune dans tous les codes d'instruction.
- Une fois l'opcode identifié,
  - le type et le format de l'instruction peuvent être déterminés et les différents champs utilisés.
  - l'instruction à exécuter est également identifiée (pour les instructions de type R, il est nécessaire d'utiliser également le champ fonction).

Table des Matières

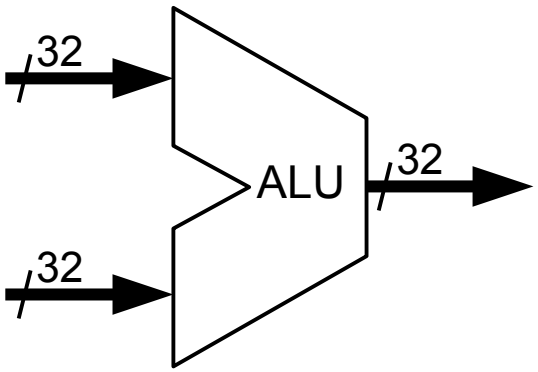
Opérations arithmétiques et logiques

Instructions de type R

- Afin d'exécuter les instructions arithmétiques et logiques de type R, les éléments suivants sont nécessaires.
- **Banque de 32 registres GPR de 32 bits**.
  - 3 adresses : A1, A2 et A3
  - 2 registres lus : RD1 et RD2
  - 1 registre écrit : WD3
  - signaux de contrôle : CLK et WE3



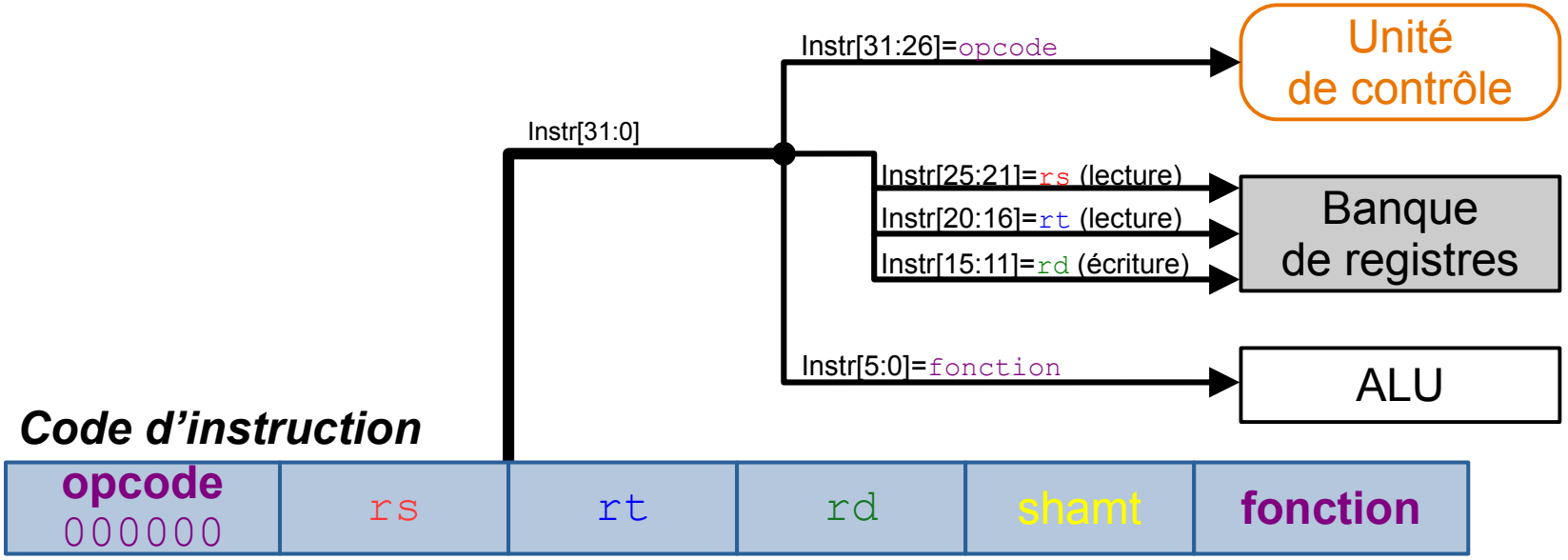
- **Unité arithmétique et logique (ALU)**.
  - opérations sur 32 bits
  - 2 entrées, 1 sortie



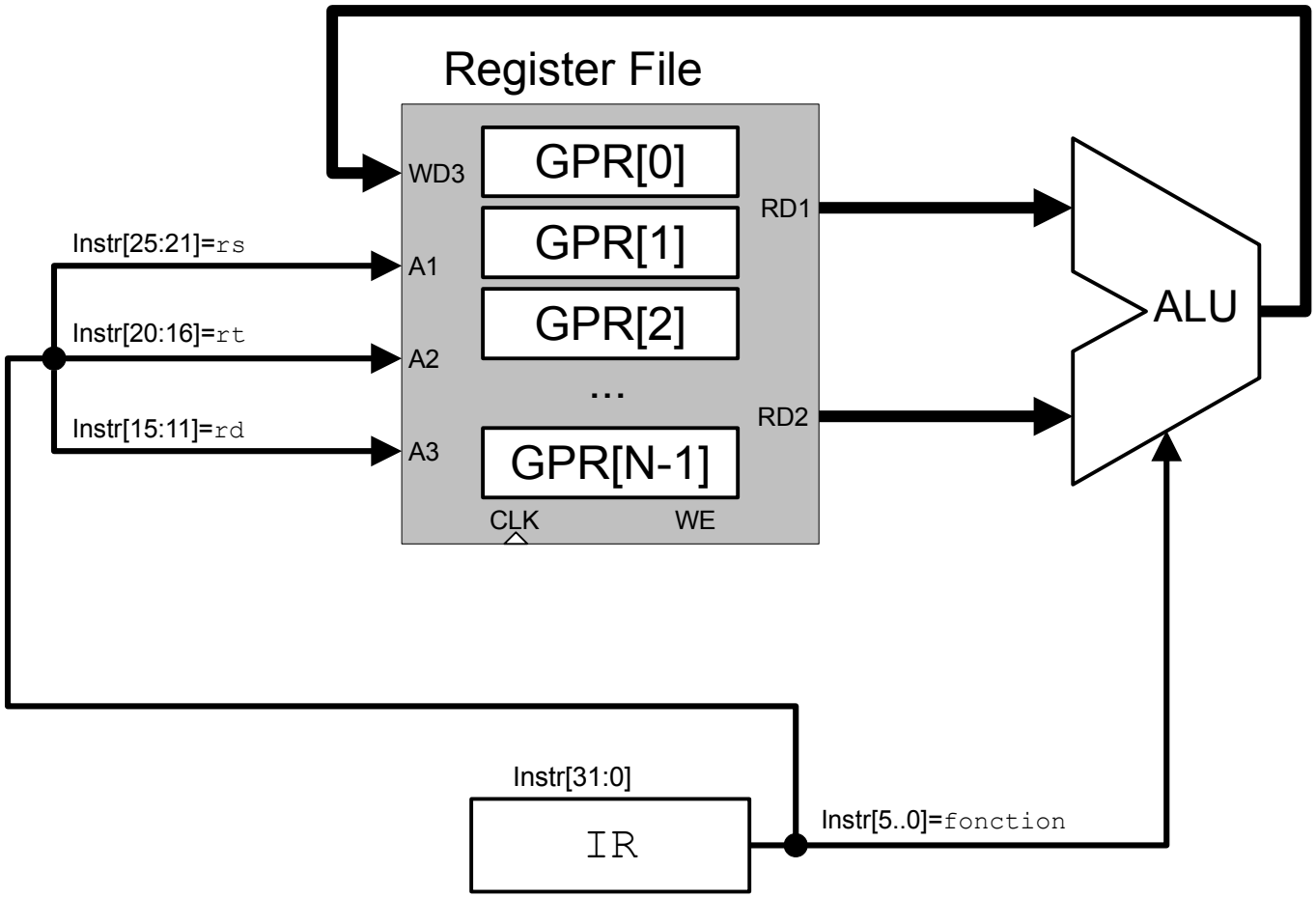
Opérations arithmétiques et logiques

Instructions de type R

- Les différentes parties du code d'instruction sont utilisées au sein du processeur pour contrôler les différents éléments internes du processeur.
  - l'`opcode` (=0) identifie l'instruction comme étant de **type R**
  - les champs `rs`, `rt` et `rd` désignent les adresses des registres qui doivent être lus et/ou écrits
  - le champ `fonction` indique à l'ALU quelle opération doit être effectuée

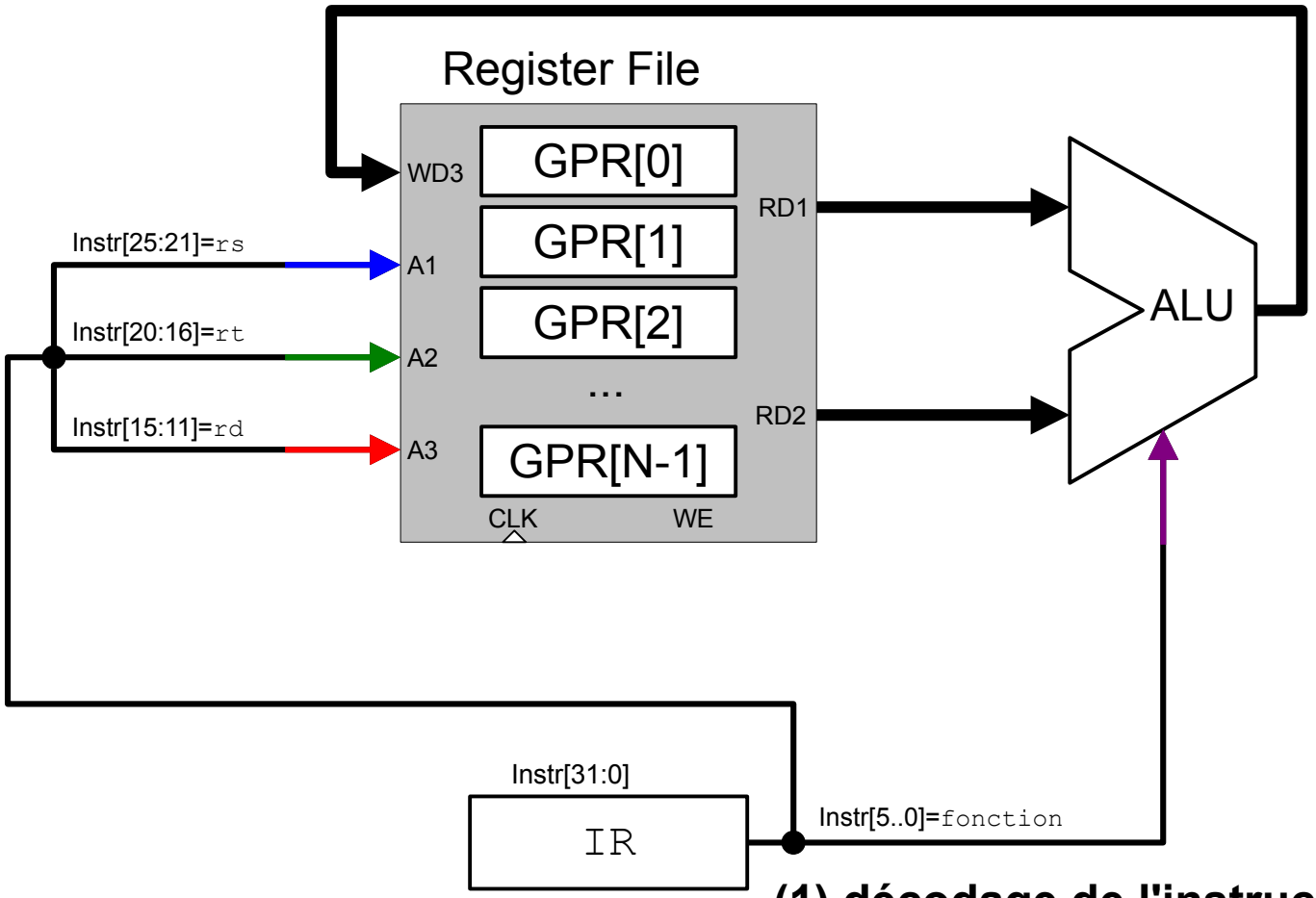


Opérations arithmétiques et logiques



Opérations arithmétiques et logiques

```
add rd, rs, rt
GPR[rd] ← GPR[rs] + GPR[rt]
```



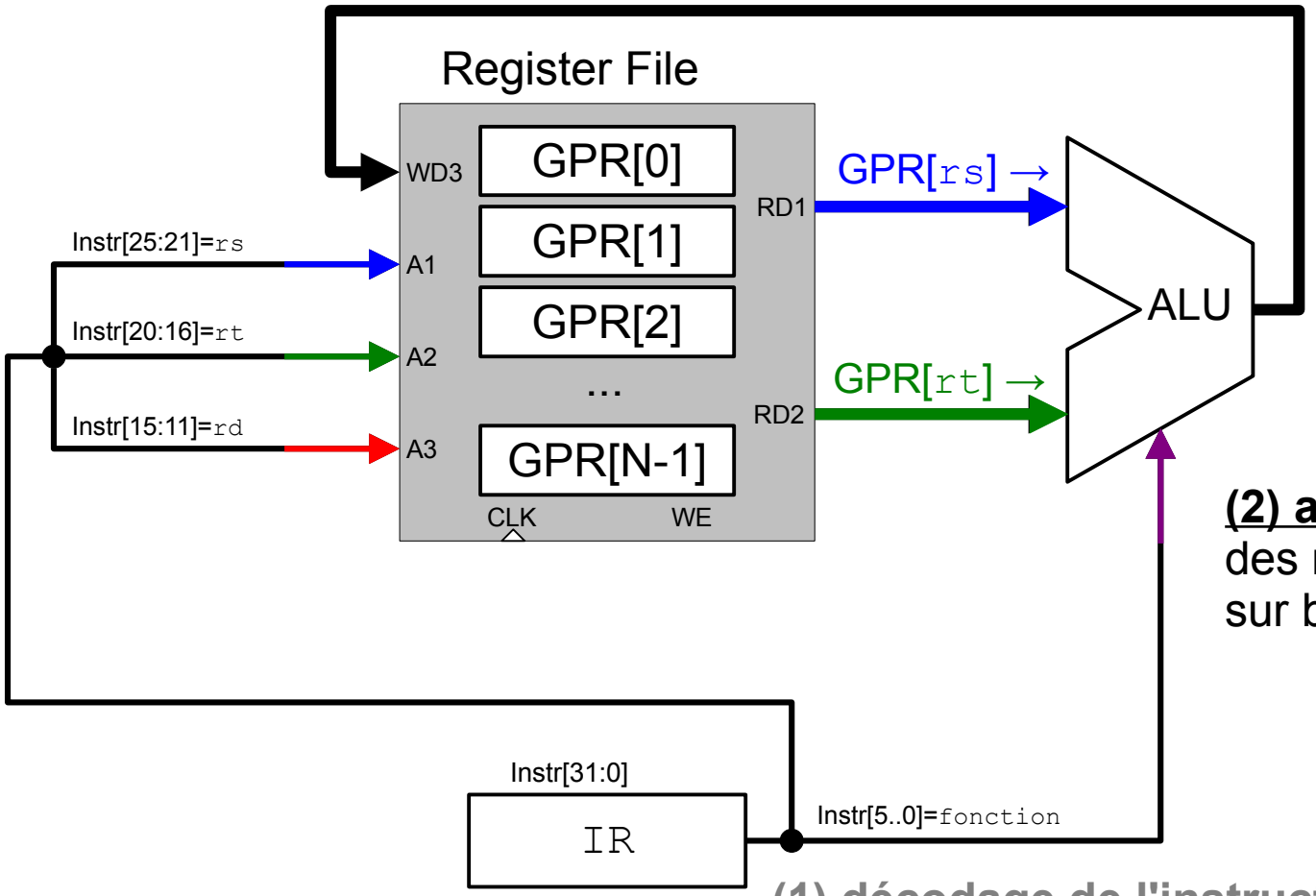
(1) <u>décodage de l'instruction</u>

→ index des registres `rs`, `rt`, `rd`

→ code d'opération pour l'ALU (`funct`)

Opérations arithmétiques et logiques

add rd, rs, rt
GPR[rd] ← GPR[rs] + GPR[rt]



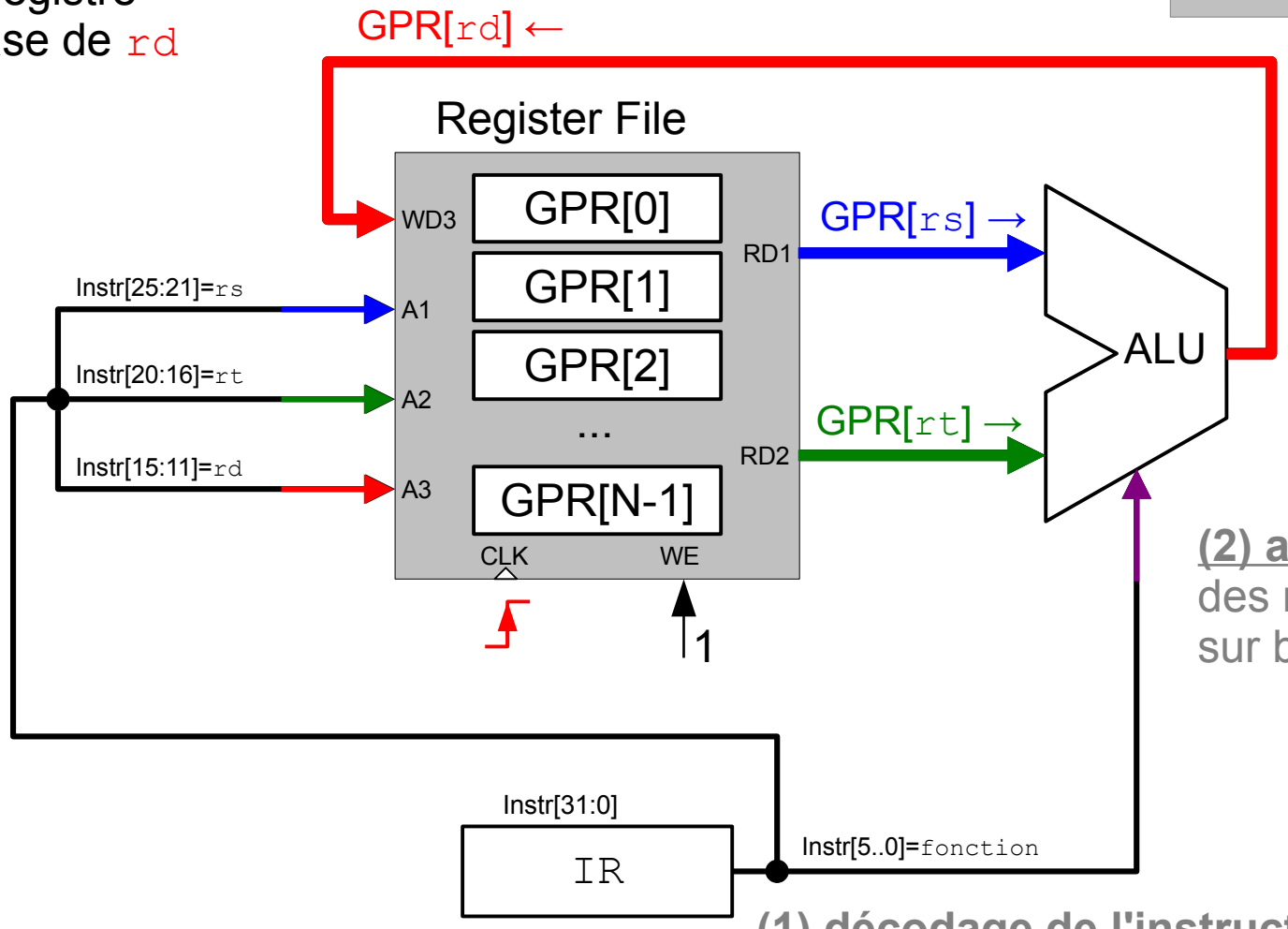
(2) addition par l'ALU
des registres sélectionnés
sur base de rs et rt

(1) décodage de l'instruction
→ index des registres rs, rt, rd
→ code d'opération pour l'ALU (funct)

Opérations arithmétiques et logiques

add rd, rs, rt
GPR[rd] ← GPR[rs] + GPR[rt]

(3) écriture du résultat
dans registre
sur base de rd



(2) addition par l'ALU
des registres sélectionnés
sur base de rs et rt

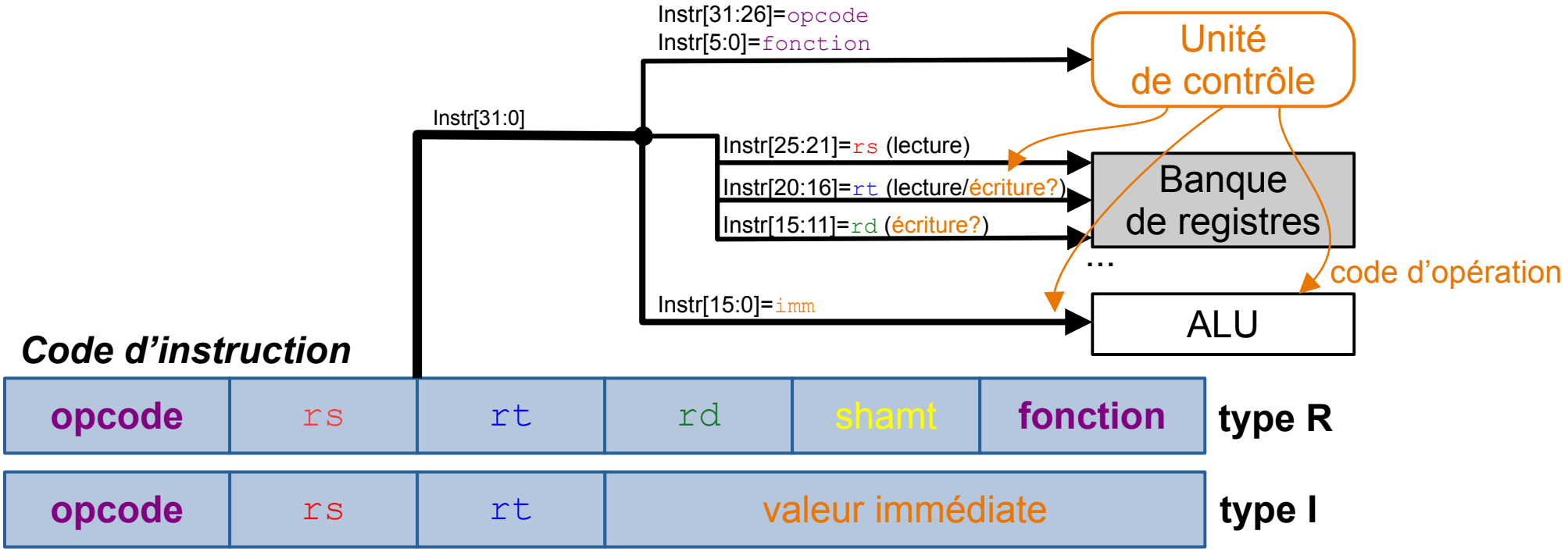
(1) décodage de l'instruction
→ index des registres rs, rt, rd
→ code d'opération pour l'ALU (funct)

Table des Matières

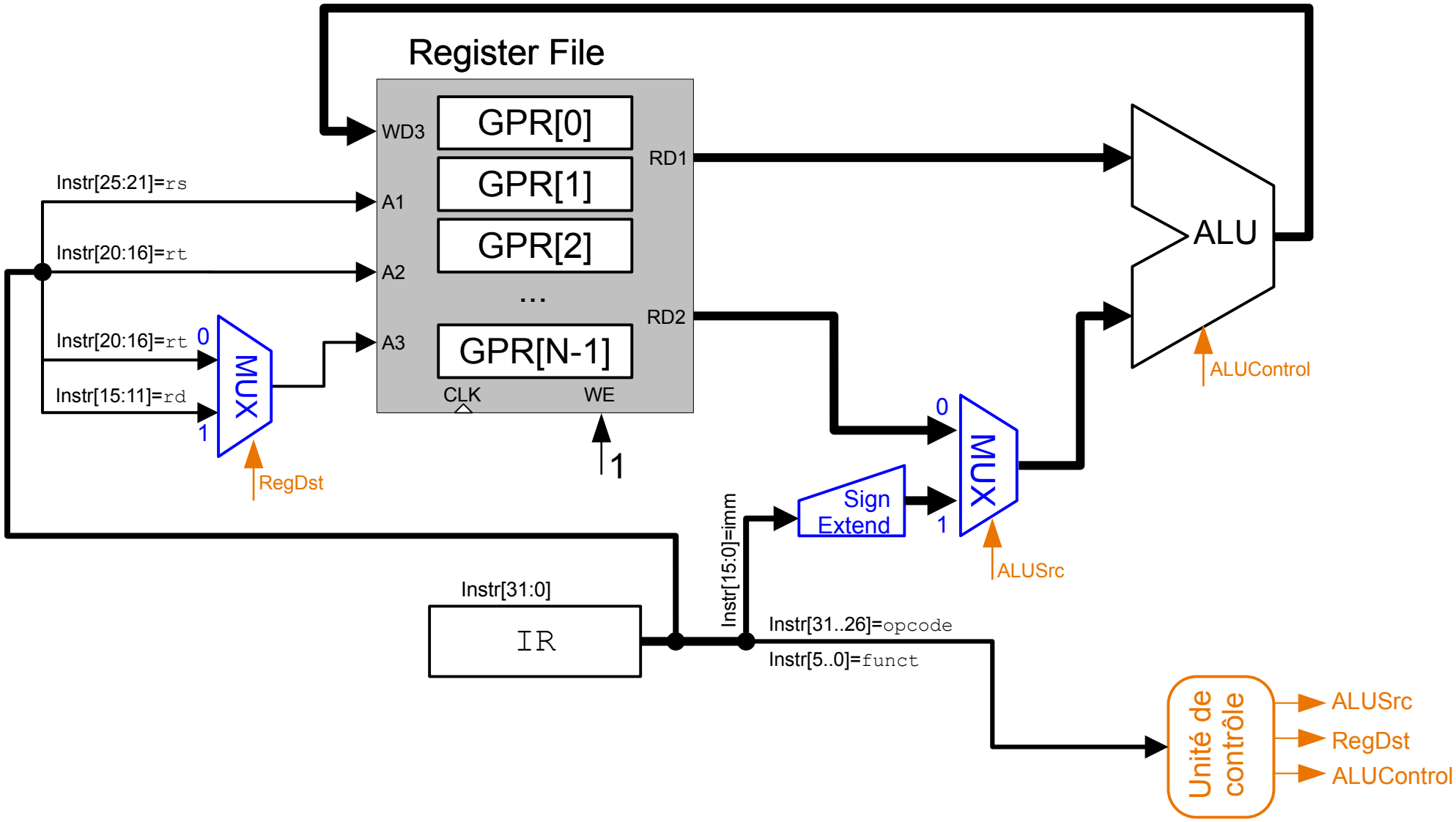
Valeurs immédiates

Opérations avec valeur immédiate

- Pour supporter deux formats d'instructions arithmétiques et logiques, le processeur se base sur l'opcode.
- L'unité de contrôle détermine sur base des champs opcode et fonction
  - le code d'opération de l'ALU
  - des signaux de contrôle supplémentaires



Valeurs immédiates

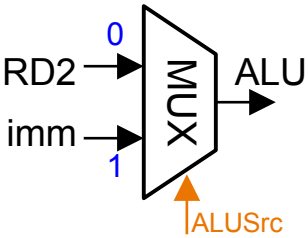


Valeurs immédiates

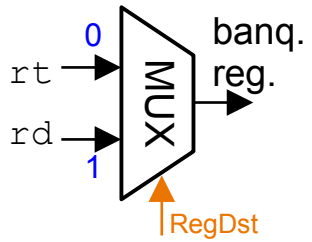
Modification du processeur

- Le support des instructions arithmétiques et logiques de type I nécessite quelques aménagements, tous en logique combinatoire.

- **Source de l'ALU** (ALUSrc) : seconde entrée de l'ALU provient de
  - Type R : sortie RD2 banque de registres (ALUSrc=0)
  - Type I : valeur immédiate (ALUSrc=1)



- **Destination** (RegDst) : le registre destination est désigné par
  - Type R : `rd` (RegDst=1)
  - Type I : `rt` (RegDst=0)

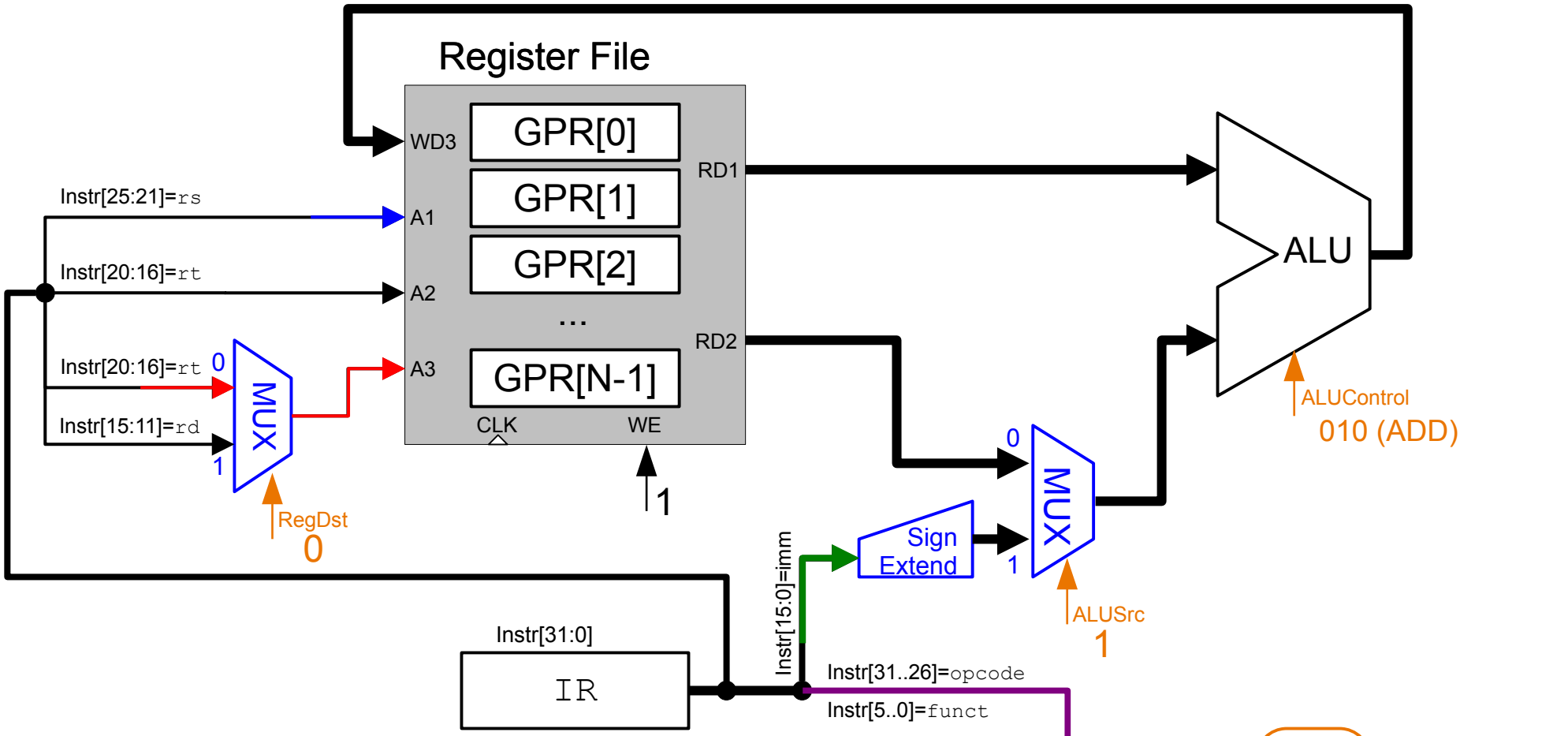


- **Extension signée** : la valeur immédiate extraite de l'instruction (16 bits) doit pouvoir être étendue à 32 bits.

- **Fonction de l'ALU** (ALUControl) : le code d'opération de l'ALU dépend de l'`opcode` et du code `funct`. L'unité de contrôle effectue la sélection.

Valeurs immédiates

addi rt, rs, imm
GPR[rt] ← GPR[rs] + s-ext(imm)



(1) <u>décodage de l'instruction</u>

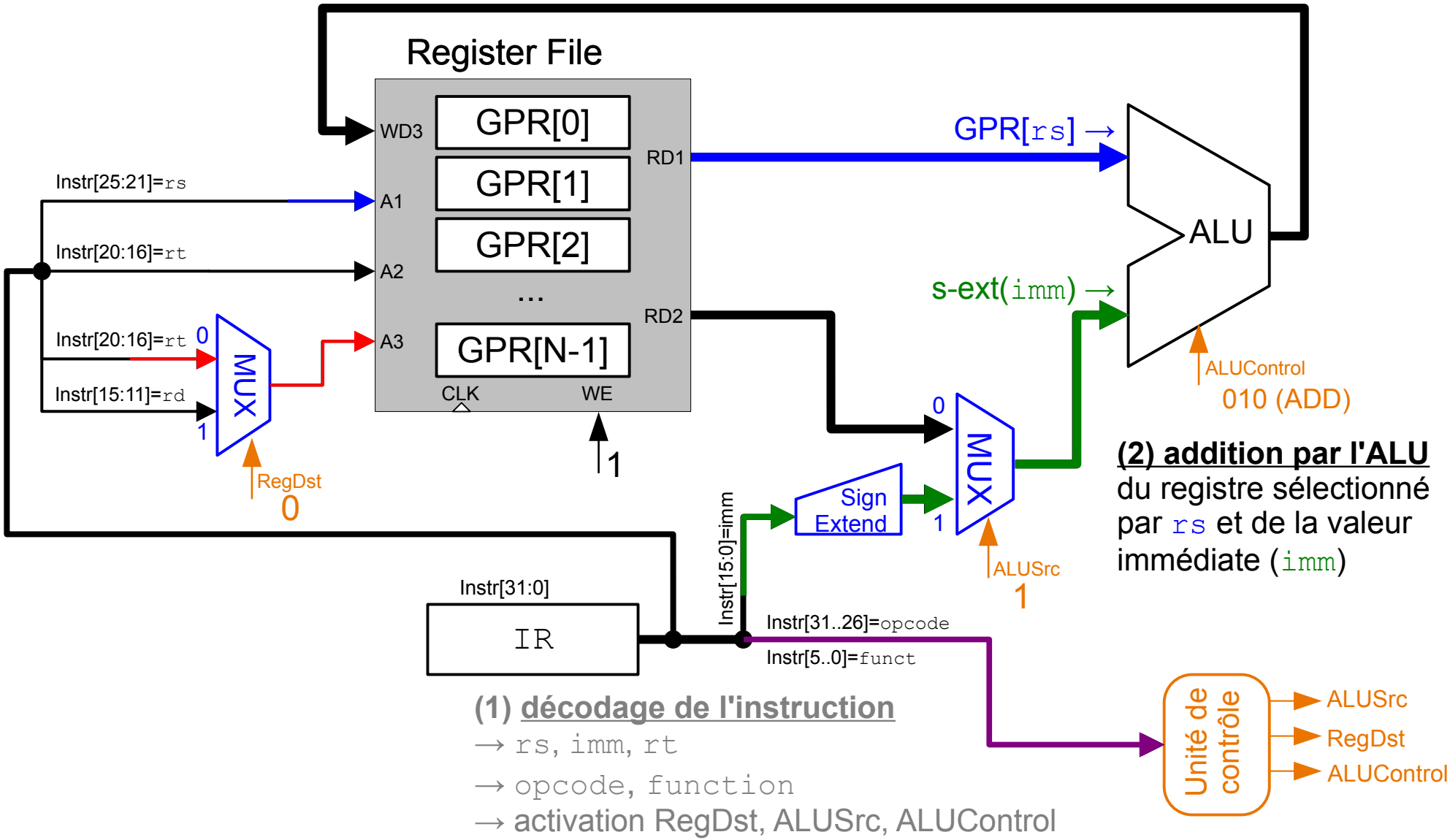
→ rs, imm, rt

→ opcode, function

→ activation RegDst, ALUSrc, ALUControl

Valeurs immédiates

addi rt, rs, imm
GPR[rt] ← GPR[rs] + s-ext(imm)



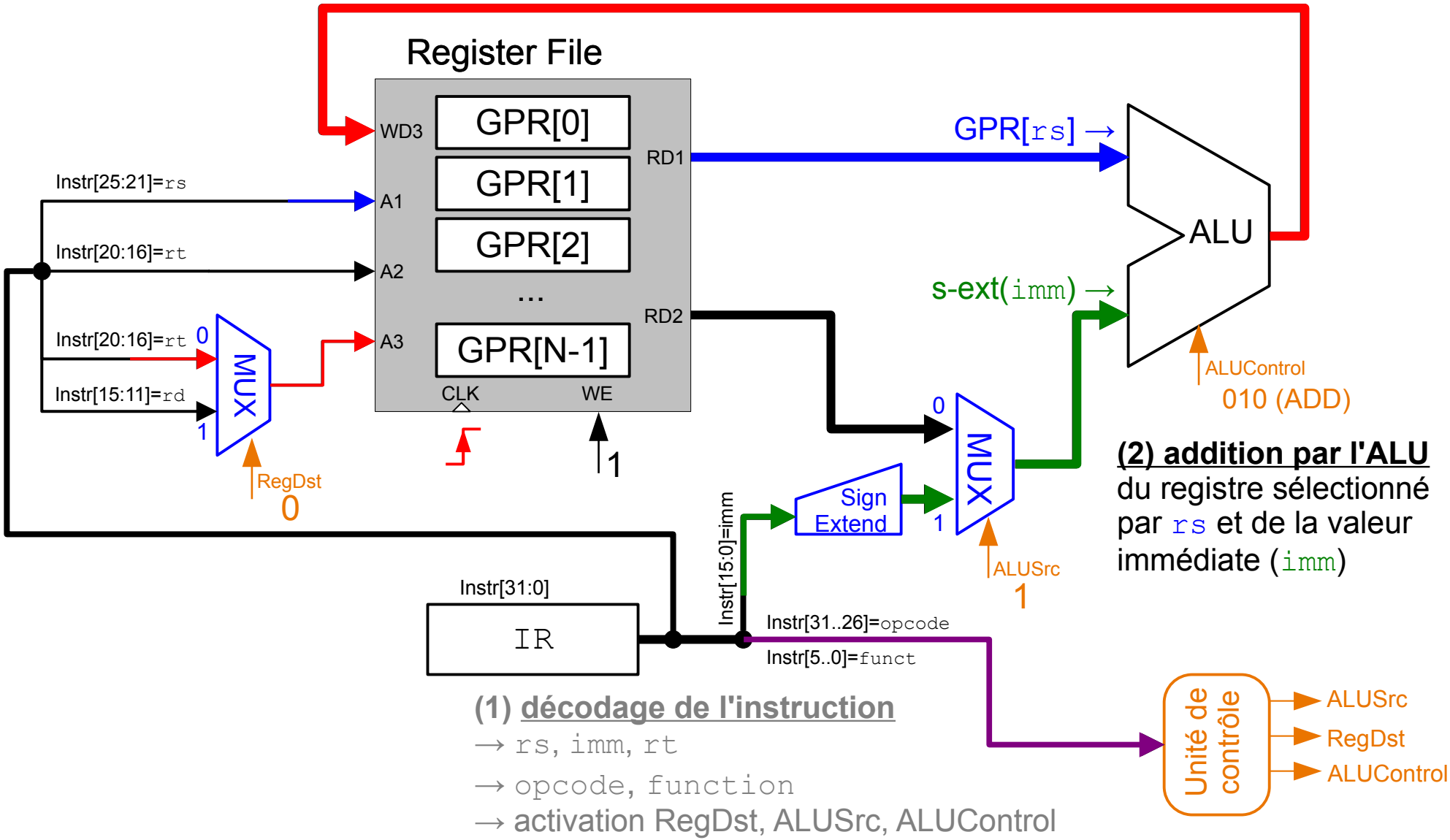
(2) addition par l'ALU
du registre sélectionné par rs et de la valeur immédiate (imm)

(1) décodage de l'instruction
→ rs, imm, rt
→ opcode, function
→ activation RegDst, ALUSrc, ALUControl

Valeurs immédiates

addi rt, rs, imm
GPR[rt] ← GPR[rs] + s-ext(imm)

(3) écriture du résultat
dans registre
sur base de rt



(2) addition par l'ALU
du registre sélectionné par rs et de la valeur immédiate (imm)

(1) décodage de l'instruction
→ rs, imm, rt
→ opcode, function
→ activation RegDst, ALUSrc, ALUControl

Valeurs immédiates

Unité de contrôle

- L'unité de contrôle pilote les signaux ALUControl, ALUSrc et RegDst en fonction des champs opcode et function du code d'instruction.

	Instruction	Opcode	Function	ALUControl	ALUSrc	RegDst
Instructions de **type R** (opcode = 0)	SLL	000000	000000	000 (SLL)	0	1
	SRL	000000	000010	001 (SRL)	0	1
	ADD	000000	100000	010 (ADD)	0	1
	SUB	000000	100010	011 (SUB)	0	1
	AND	000000	100100	100 (AND)	0	1
	OR	000000	100101	101 (OR)	0	1
		000000	*autre*	*invalid instruction*		
Instructions de **type I** (opcode ≠ 0)	ADDI	001000	X	010 (ADD)	1	0
	ANDI	001100	X	100 (AND)	1	0
	ORI	001101	X	101 (OR)	1	0
	LUI	001111	X	110 (LUI)	1	0
		autre	*invalid instruction*			

Table des Matières

Load-Store

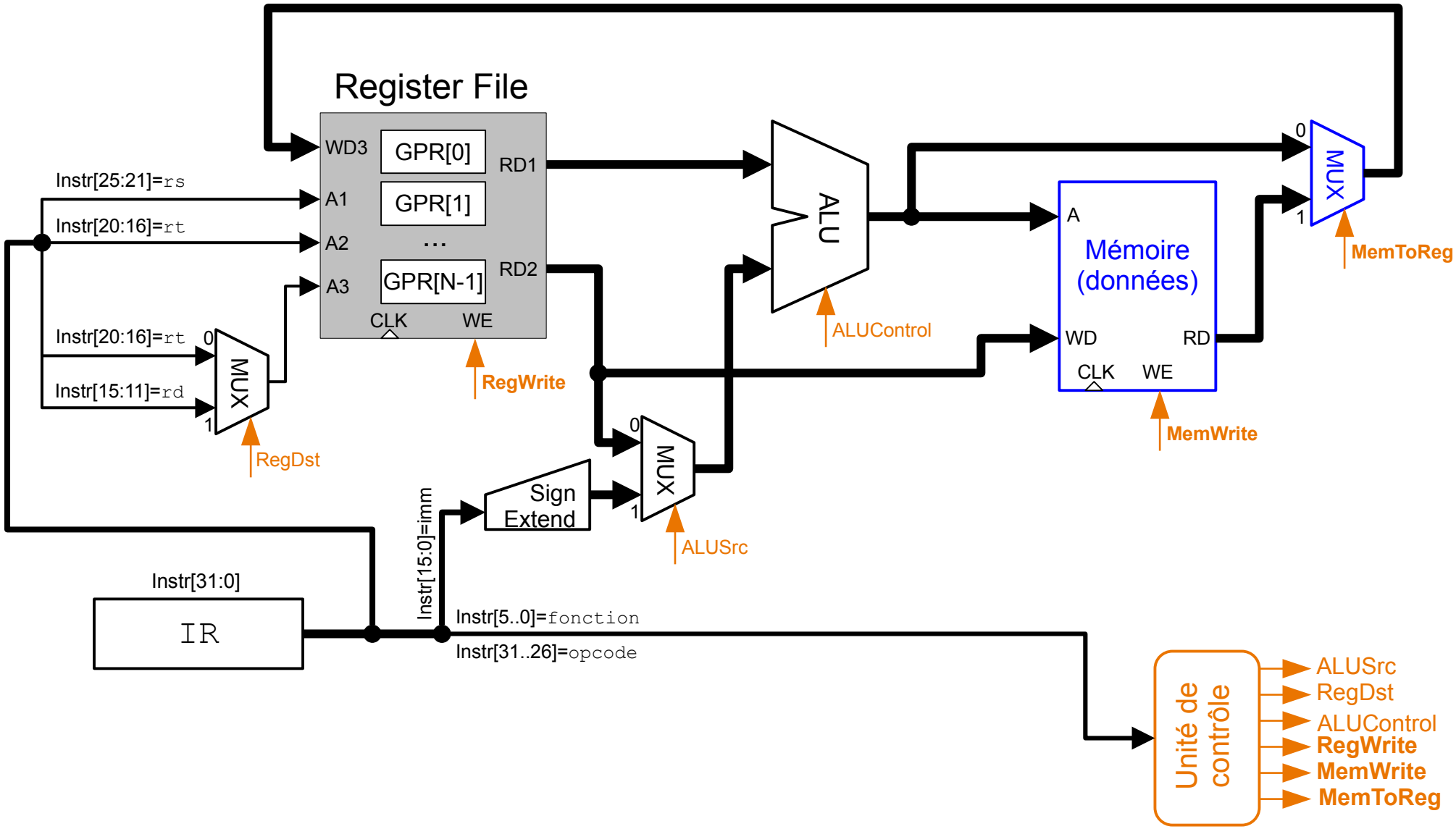
Modification du processeur

- Afin d'implémenter le support des instructions *load* (**`lw`**) et *store* (**`sw`**) dans le processeur, on y ajoute une **mémoire de données** qui peut être lue et écrite. Il est également nécessaire d'ajouter les éléments suivants en logique combinatoire et de mettre à jour l'unité de contrôle.
- **Calcul d'adresse** (indirect indexé) : ré-utilisation de l'ALU pour effectuer l'addition du *registre de base* et de l'*offset* (valeur immédiate).

$$v = \text{memory}[\text{GPR}[b] + \textit{offset}]$$

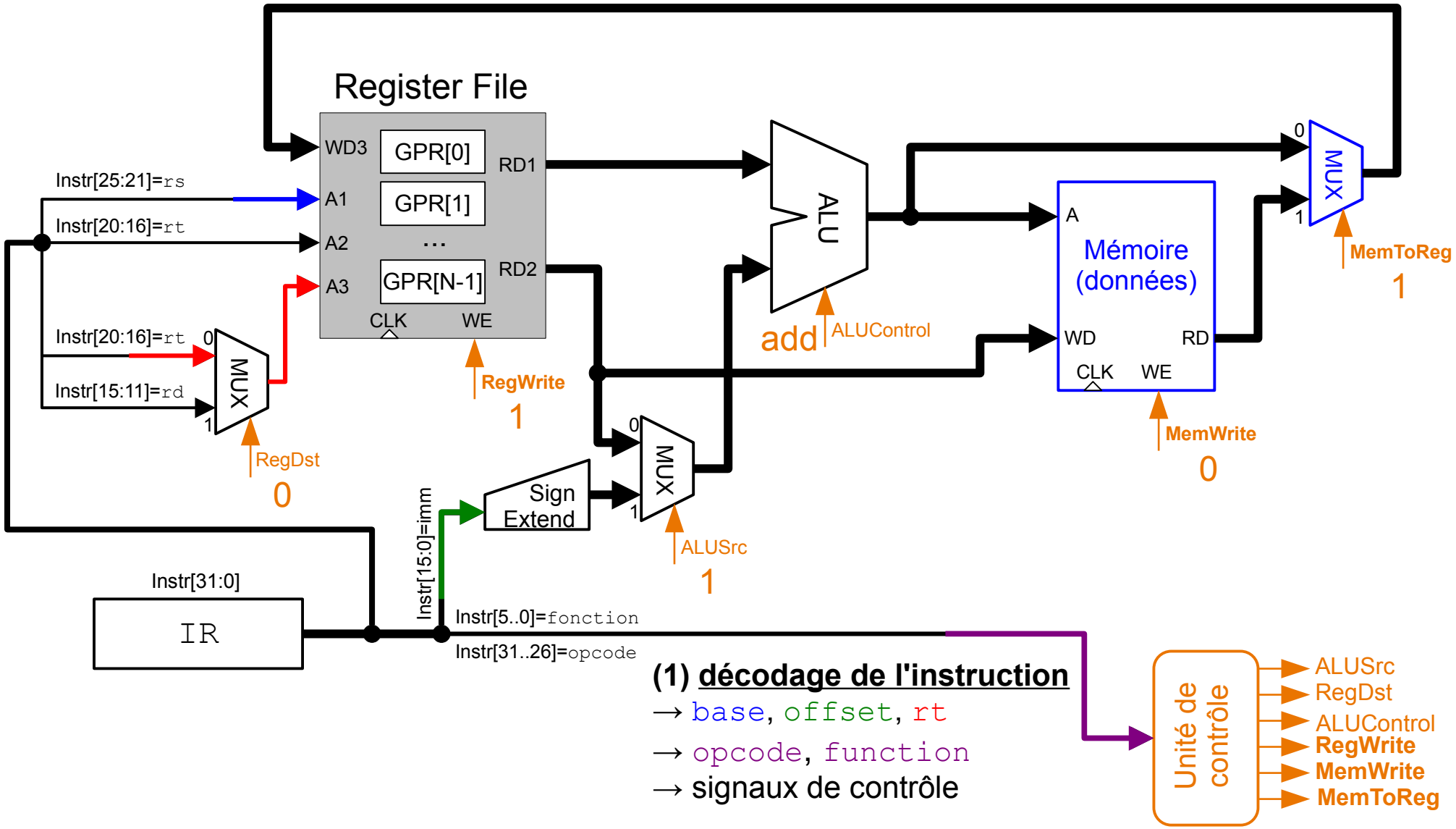
- **Source à écrire dans un registre** (MemToReg) : Multiplexeur 2:1 qui permet de sélectionner si ce qui est enregistré dans un registre provient de l'ALU ou de la mémoire.
  - résultat de l'ALU (MemToReg = 0)
  - mémoire (MemToReg = 1).
- **Sens de l'écriture** : des signaux déterminés par l'unité de contrôle indiquent si l'écriture se fait vers un registre (RegWrite = 1) ou vers la mémoire (MemWrite = 1).

Load-Store


Register File
WD3
A1
A2
A3
GPR[0]
GPR[1]
...
GPR[N-1]
RD1
RD2
CLK
WE
RegWrite
Instr[25:21]=rs
Instr[20:16]=rt
Instr[20:16]=rt
Instr[15:11]=rd
MUX
0
1
RegDst
ALU
ALUControl
A
Mémoire
(données)
WD
RD
CLK
WE
MemWrite
MUX
0
1
MemToReg
Sign
Extend
Instr[15:0]=imm
MUX
0
1
ALUSrc
Instr[31:0]
IR
Instr[5..0]=fonction
Instr[31..26]=opcode
Unité de
contrôle
ALUSrc
RegDst
ALUControl
RegWrite
MemWrite
MemToReg

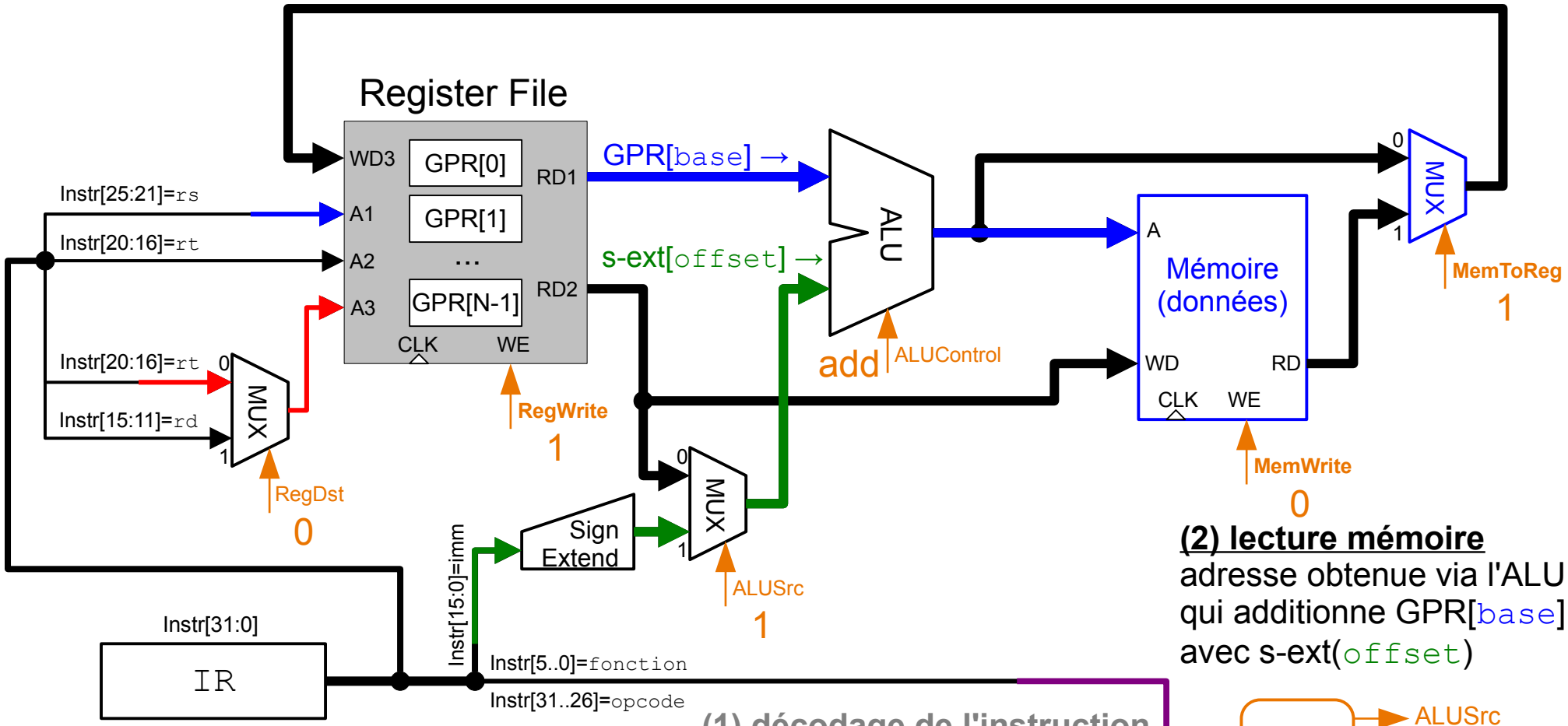
Load

lw rt,offset(base)
GPR[rt] ← memory[GPR[base] + s-ext(offset)]



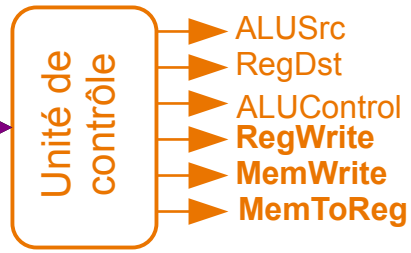
Load

lw rt,offset(base)
GPR[rt] ← memory[GPR[base] + s-ext(offset)]



(2) lecture mémoire
adresse obtenue via l'ALU qui additionne GPR[base] avec s-ext(offset)

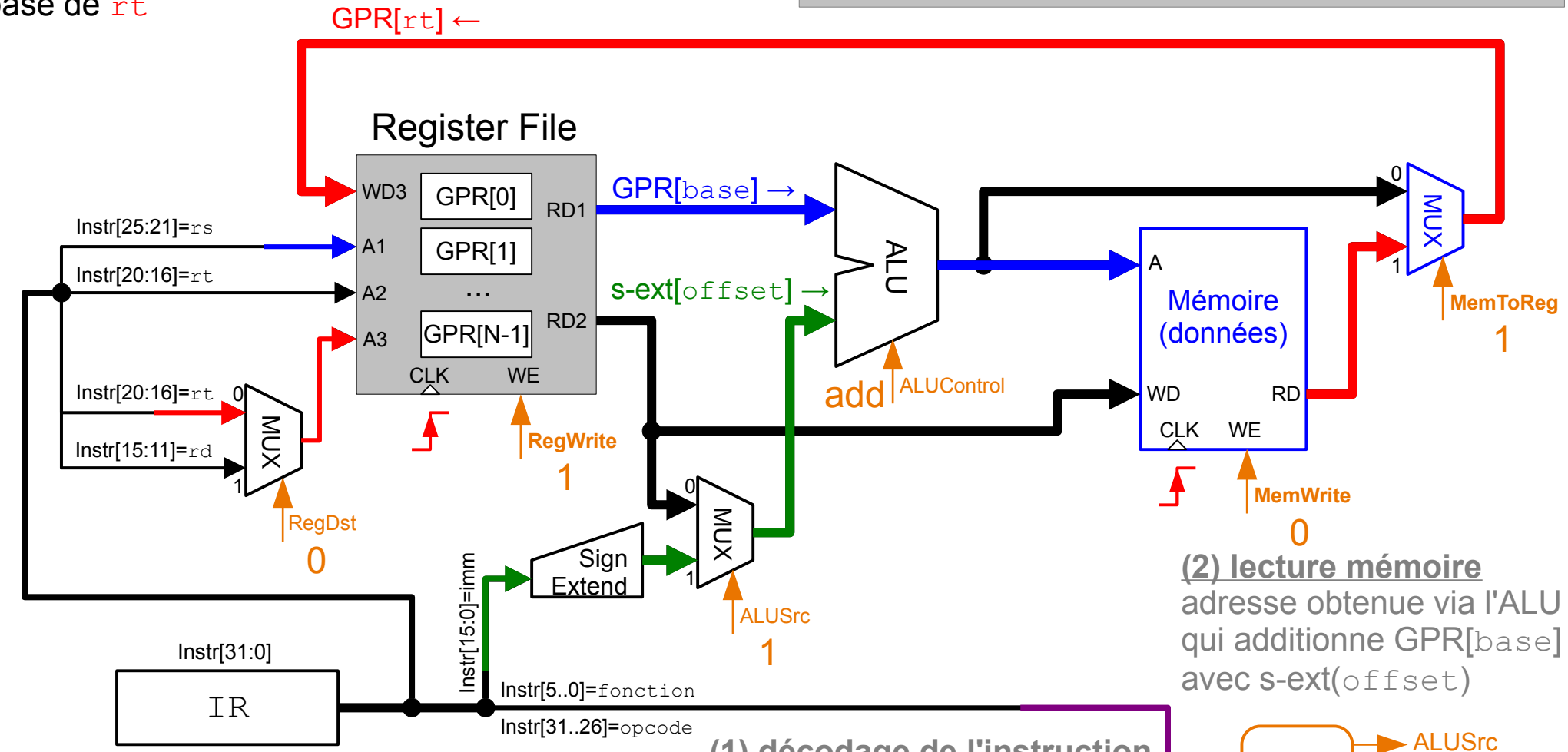
(1) décodage de l'instruction
→ base, offset, rt
→ opcode, function
→ signaux de contrôle



Load

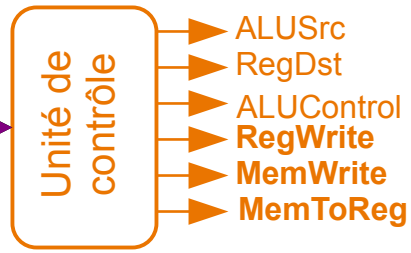
(3) écriture registre
sélectionné sur
base de `rt`

```
lw rt,offset(base)
GPR[rt] ← memory[GPR[base] + s-ext(offset)]
```



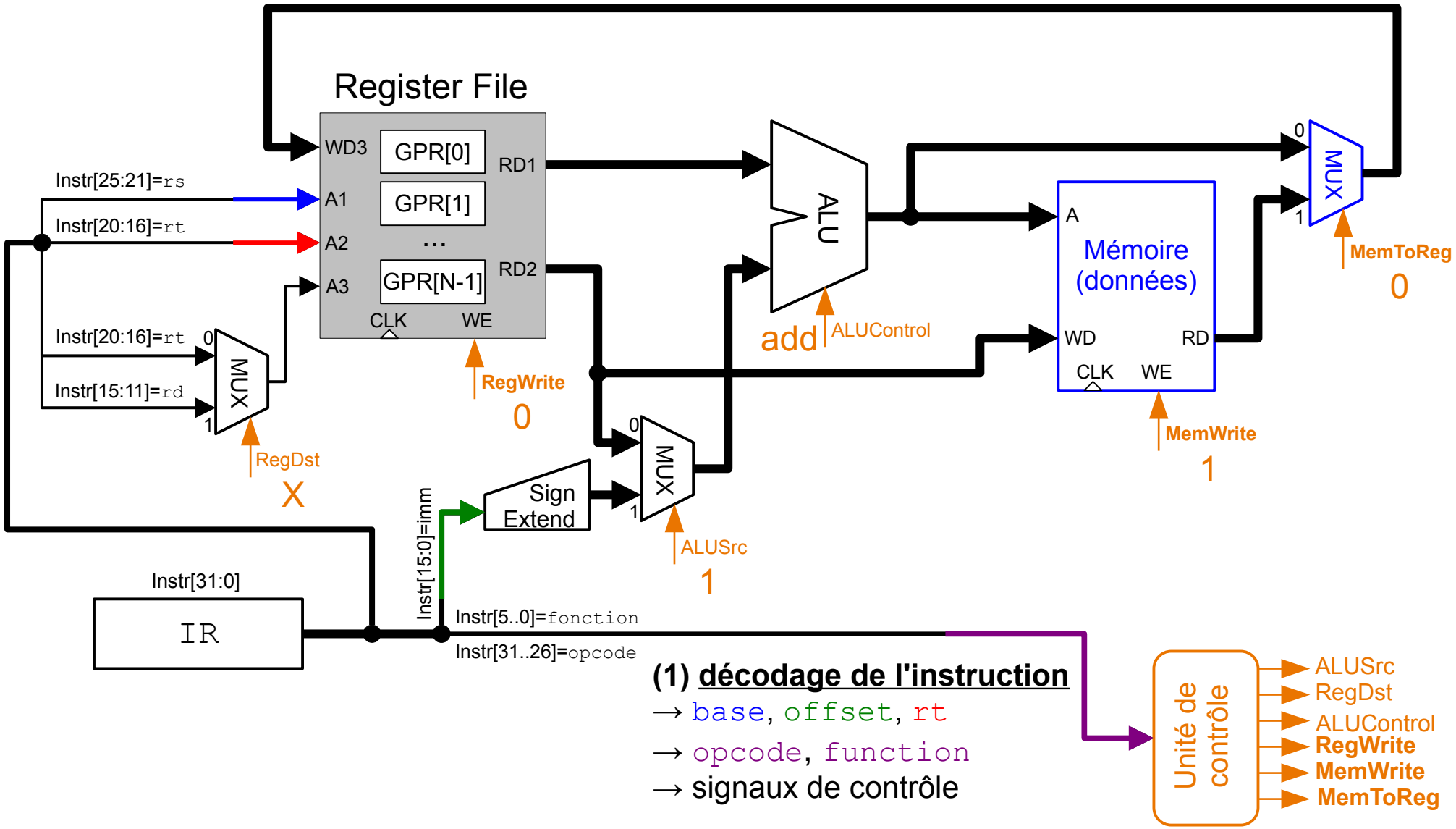
(2) lecture mémoire
adresse obtenue via l'ALU
qui additionne GPR[`base`]
avec s-ext(`offset`)

(1) décodage de l'instruction
→ `base`, `offset`, `rt`
→ `opcode`, `function`
→ signaux de contrôle



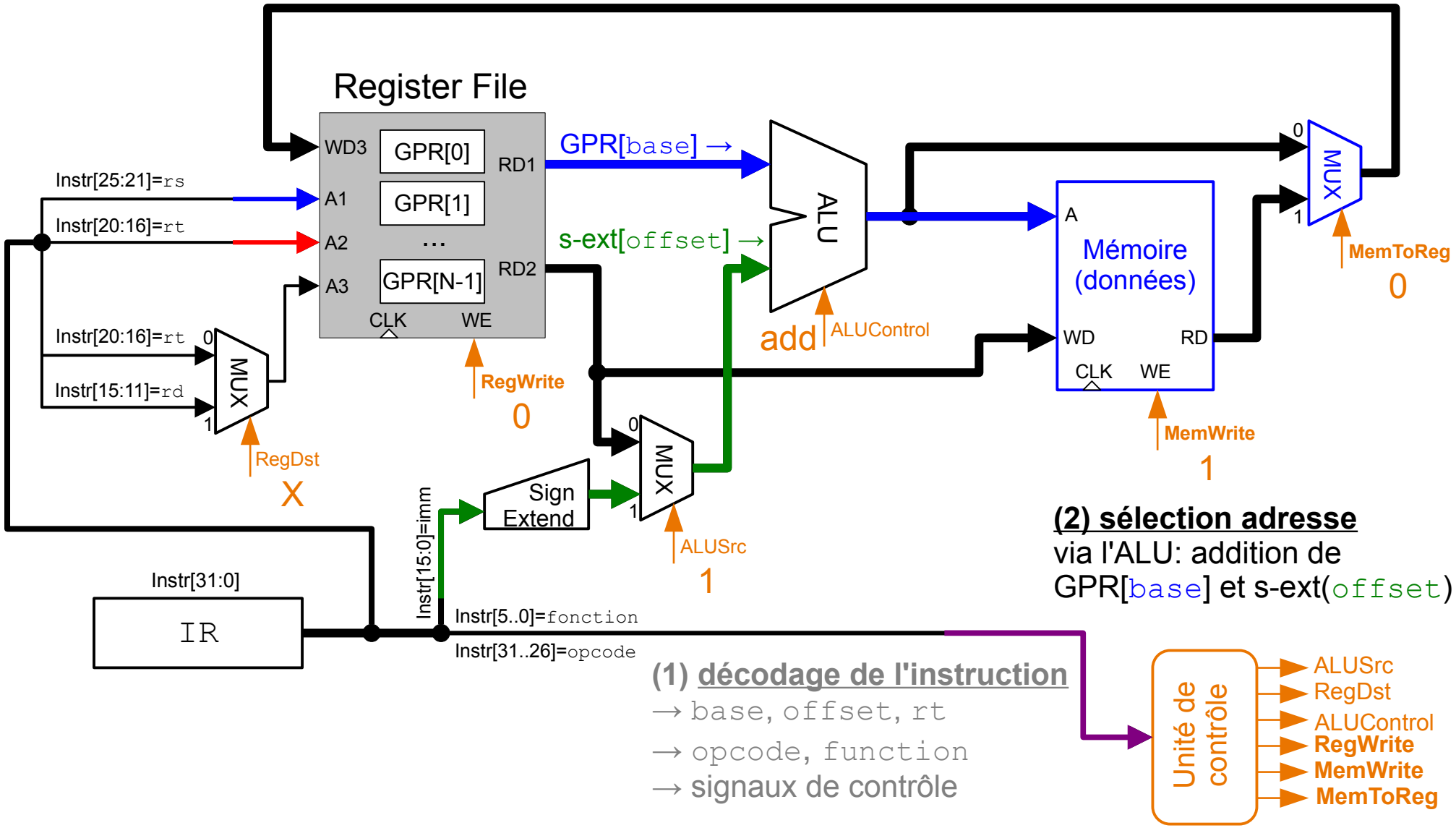
Store

sw rt,offset(base)
memory[GPR[base] + s-ext(offset)] ← GPR[rt]



Store

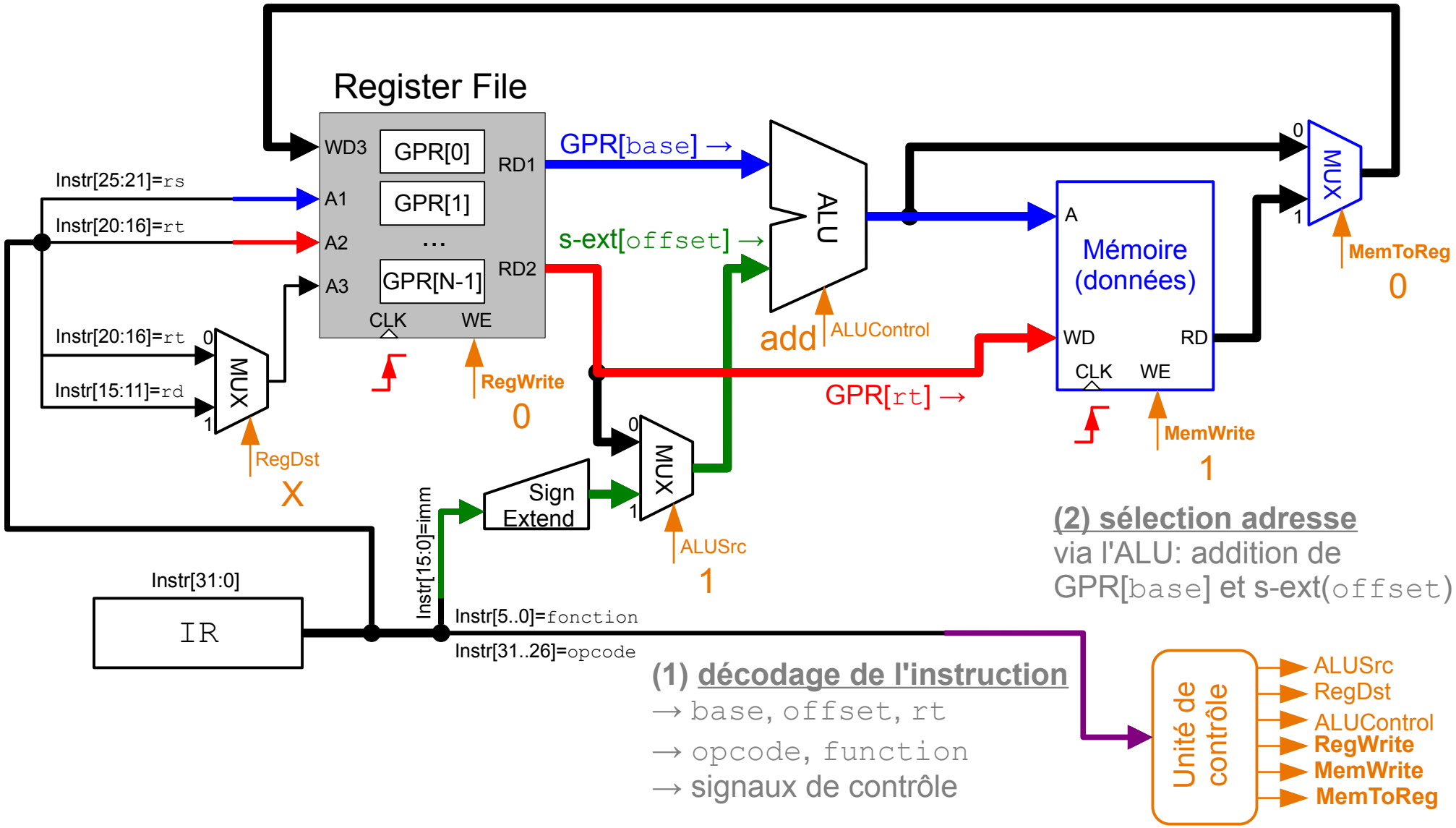
sw rt,offset(base)
memory[GPR[base] + s-ext(offset)] ← GPR[rt]



Store

(3) écriture mémoire
du contenu du registre sélectionné par `rt`

```
sw rt,offset(base)
memory[GPR[base] + s-ext(offset)] ← GPR[rt]
```



Unité de contrôle

- L'unité de contrôle pilote les signaux supplémentaires RegWrite, ALMemWrite et MemToReg.

Instr.	Opcode	Function	ALUControl	ALUSrc	RegDst	RegWrite	MemWrite	MemToReg
SLL	000000	000000	000 (SLL)	0	1	1	0	0
SRL	000000	000010	001 (SRL)	0	1	1	0	0
ADD	000000	100000	010 (ADD)	0	1	1	0	0
SUB	000000	100010	011 (SUB)	0	1	1	0	0
AND	000000	100100	100 (AND)	0	1	1	0	0
OR	000000	100101	101 (OR)	0	1	1	0	0
	000000	*autre*	*invalid instruction*					
ADDI	001000	X	010 (ADD)	1	0	1	0	0
ANDI	001100	X	100 (AND)	1	0	1	0	0
ORI	001101	X	101 (OR)	1	0	1	0	0
LUI	001111	X	110 (LUI)	1	0	1	0	0
LW	100011	X	010 (ADD)	1	0	1	0	1
SW	101011	X	010 (ADD)	1	X	0	1	0
	autre	*invalid instruction*						

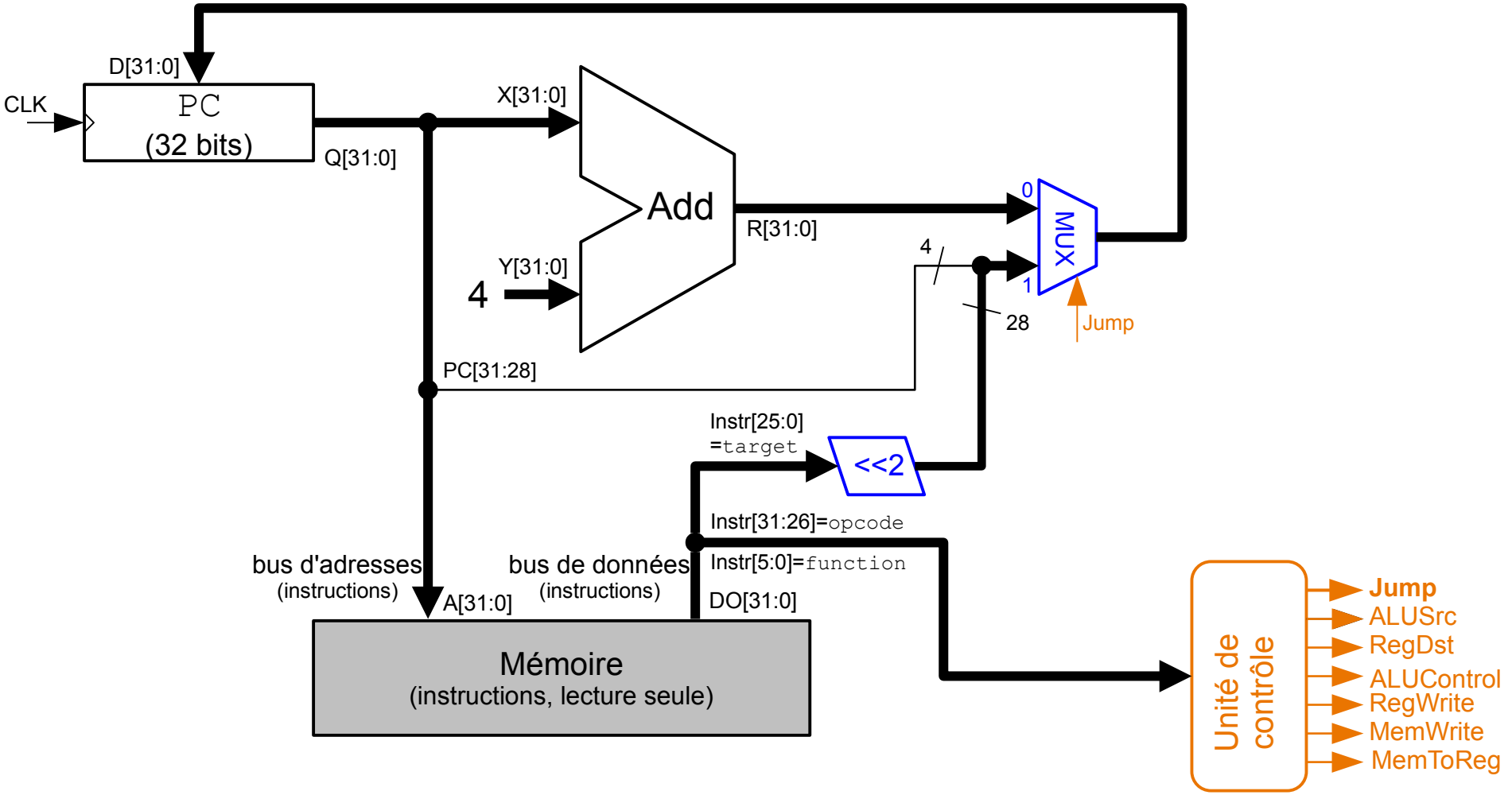
Table des Matières

Branchements inconditionnels

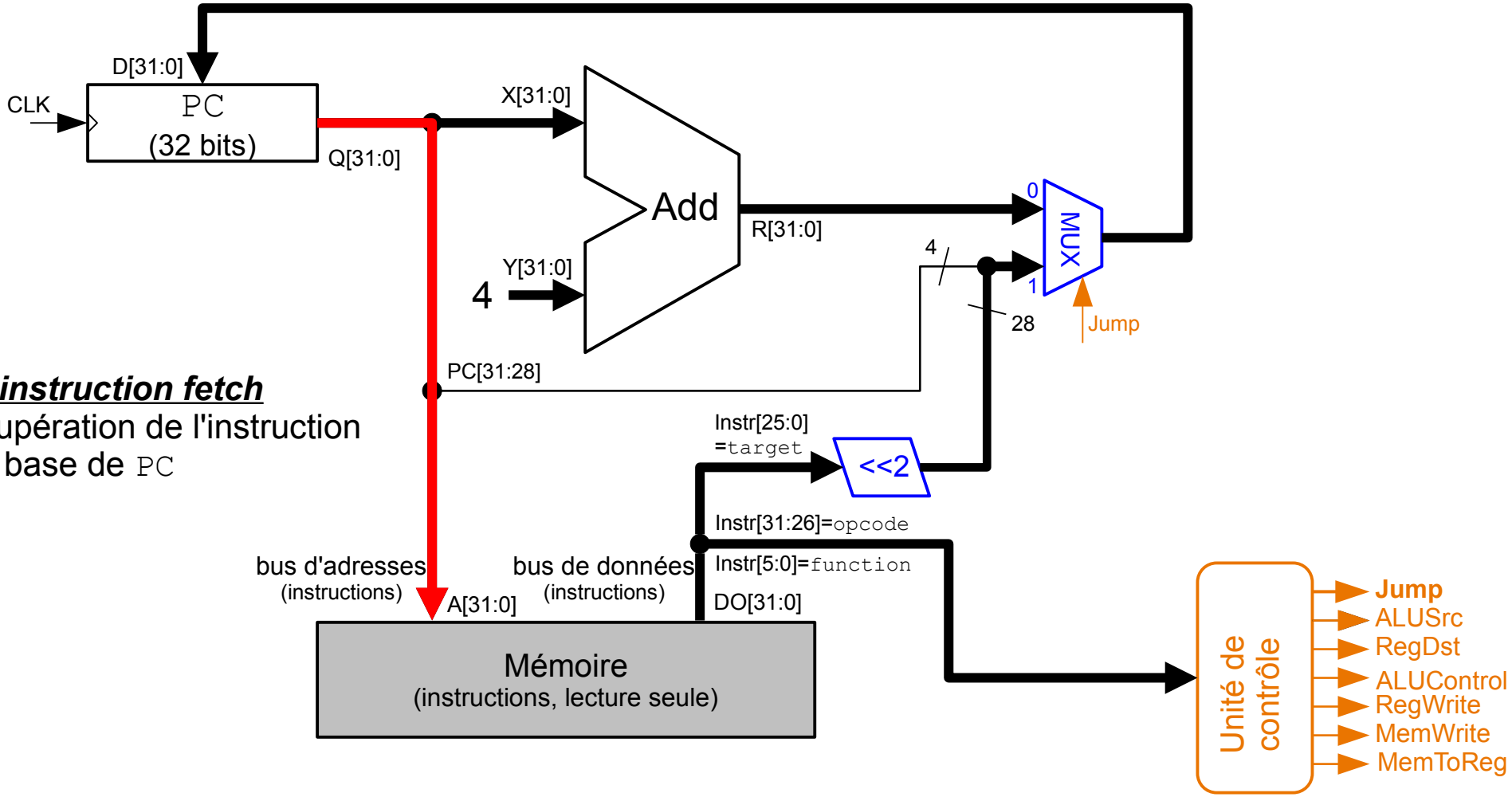
Modification du processeur

- Pour implémenter le support de l'instruction *Jump* (`J`), il est nécessaire de pouvoir placer dans le registre `PC` une valeur autre que `PC`+4.
- A cette fin, des blocs en logique combinatoire doivent être ajoutés et l'unité de contrôle doit être mise à jour.
- **Signal Jump** : un multiplexeur 2:1 permettant de sélectionner entre `PC`+4 ou l'adresse déterminée par l'instruction de branchement.
  - `PC`+4 (Jump = 0)
  - adresse de branchement (Jump = 1)
- **Décalage de target** : un bloc permettant de décaler de 2 bits vers la gauche la valeur du champ `target` obtenue de l'instruction.
- **Instruction de type J** : identifier format **type J** sur base de l'opcode.

Branchements inconditionnels

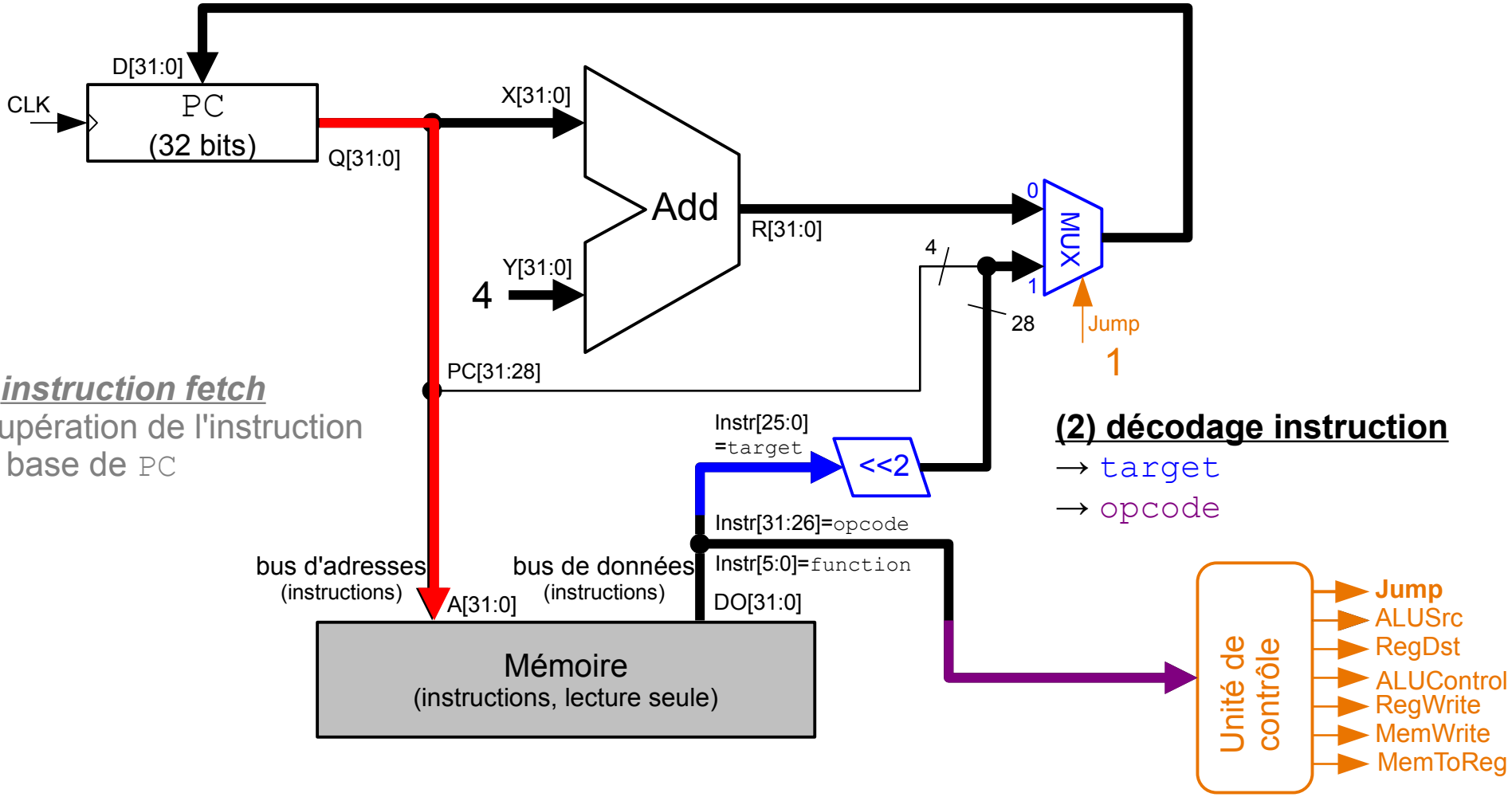


Branchements inconditionnels



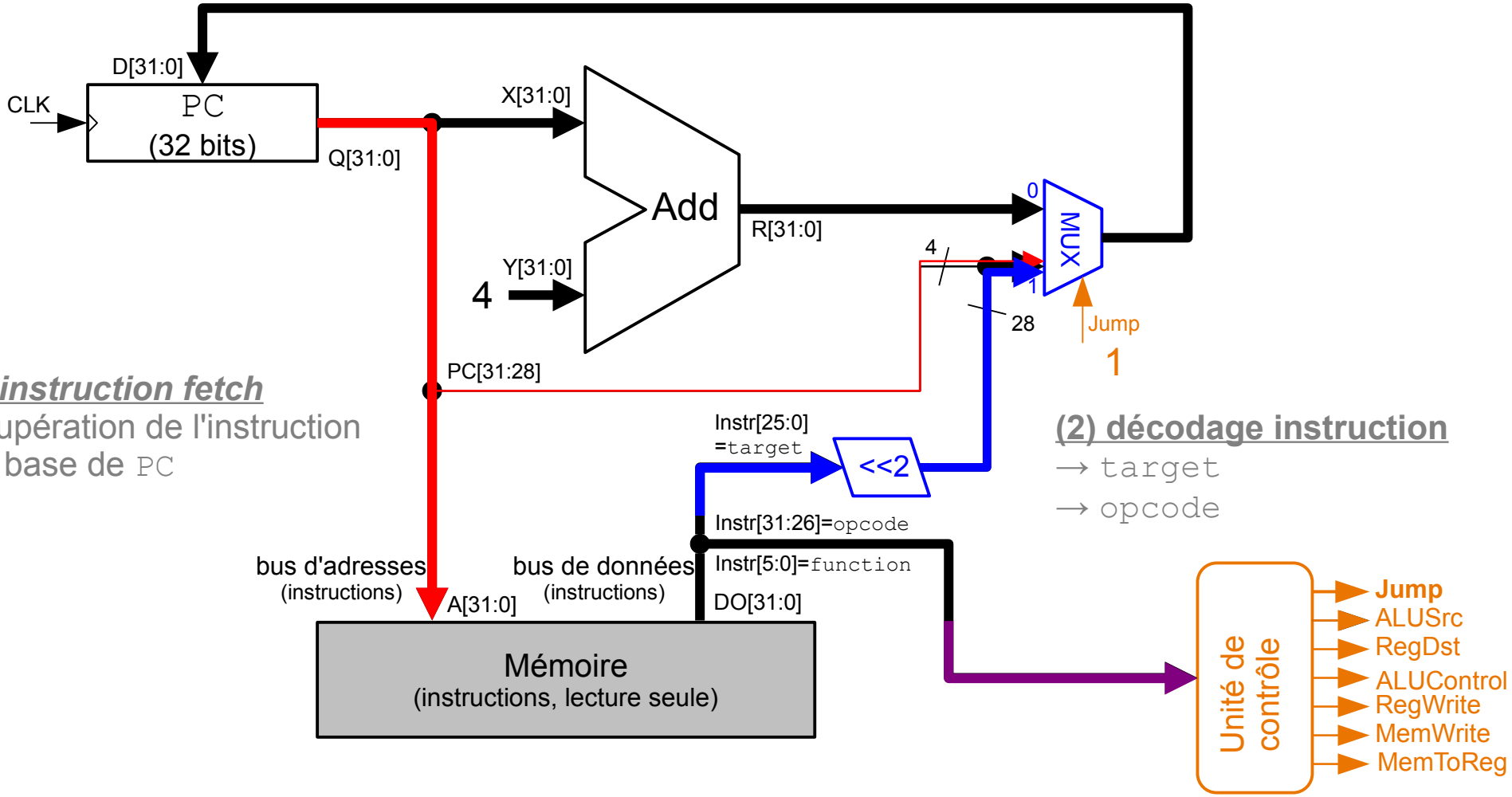
(1) *instruction fetch*
récupération de l'instruction sur base de `PC`

Branchements inconditionnels



Branchements inconditionnels

(3) calcul de l'adresse
sur base de $PC_{31..28}$
et `target` décalé de 2 vers la gauche



(1) *instruction fetch*
récupération de l'instruction sur base de `PC`

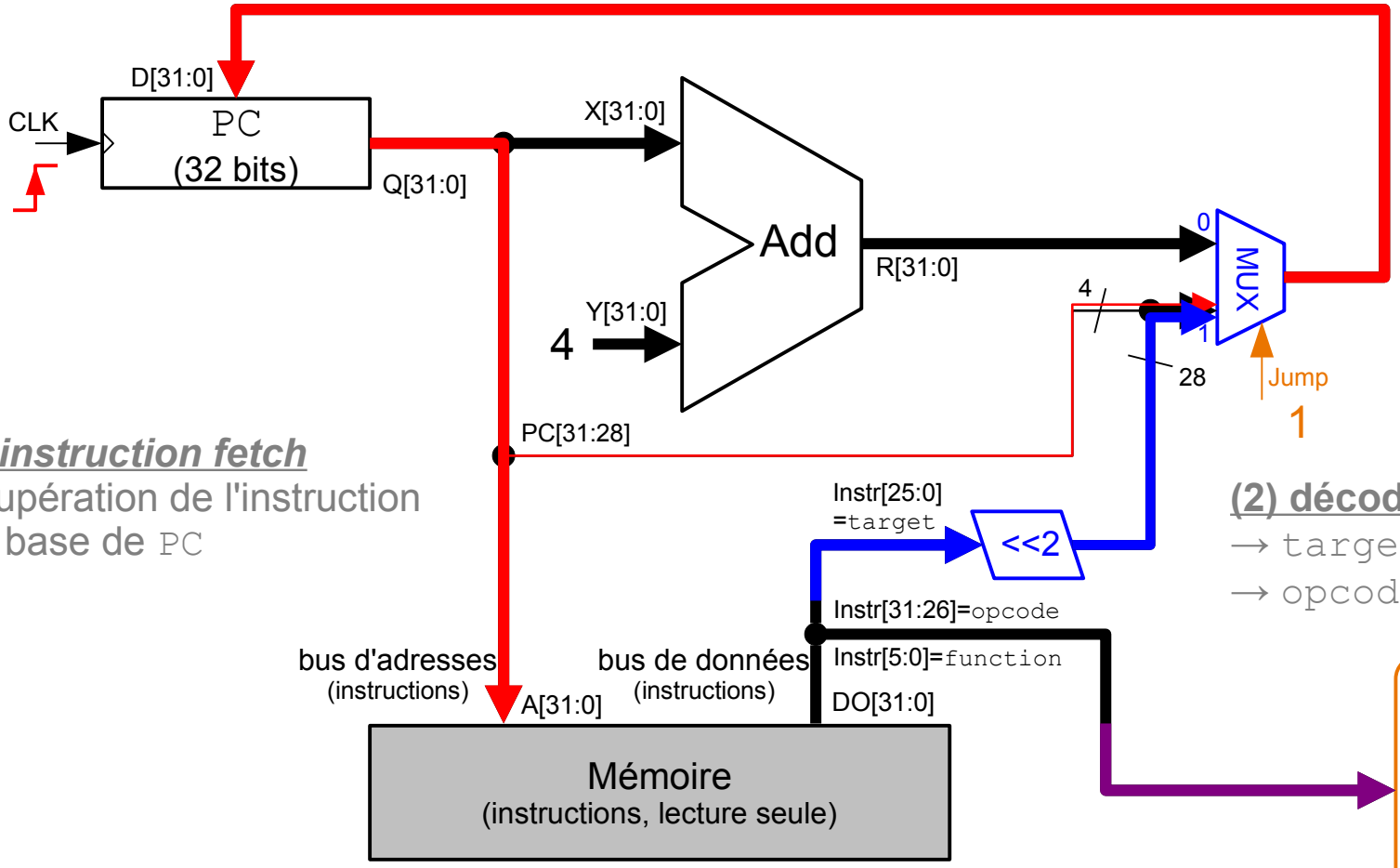
(2) décodage instruction
→ `target`
→ `opcode`

Branchements inconditionnels

(3) calcul de l'adresse
sur base de $PC_{31..28}$
et `target` décalé de 2 vers la gauche

(4) écriture de PC

$\leftarrow PC_{31..28} \,||\, target \,||\, 0_2$



(1) *instruction fetch*
récupération de l'instruction sur base de `PC`

(2) décodage instruction
→ `target`
→ `opcode`

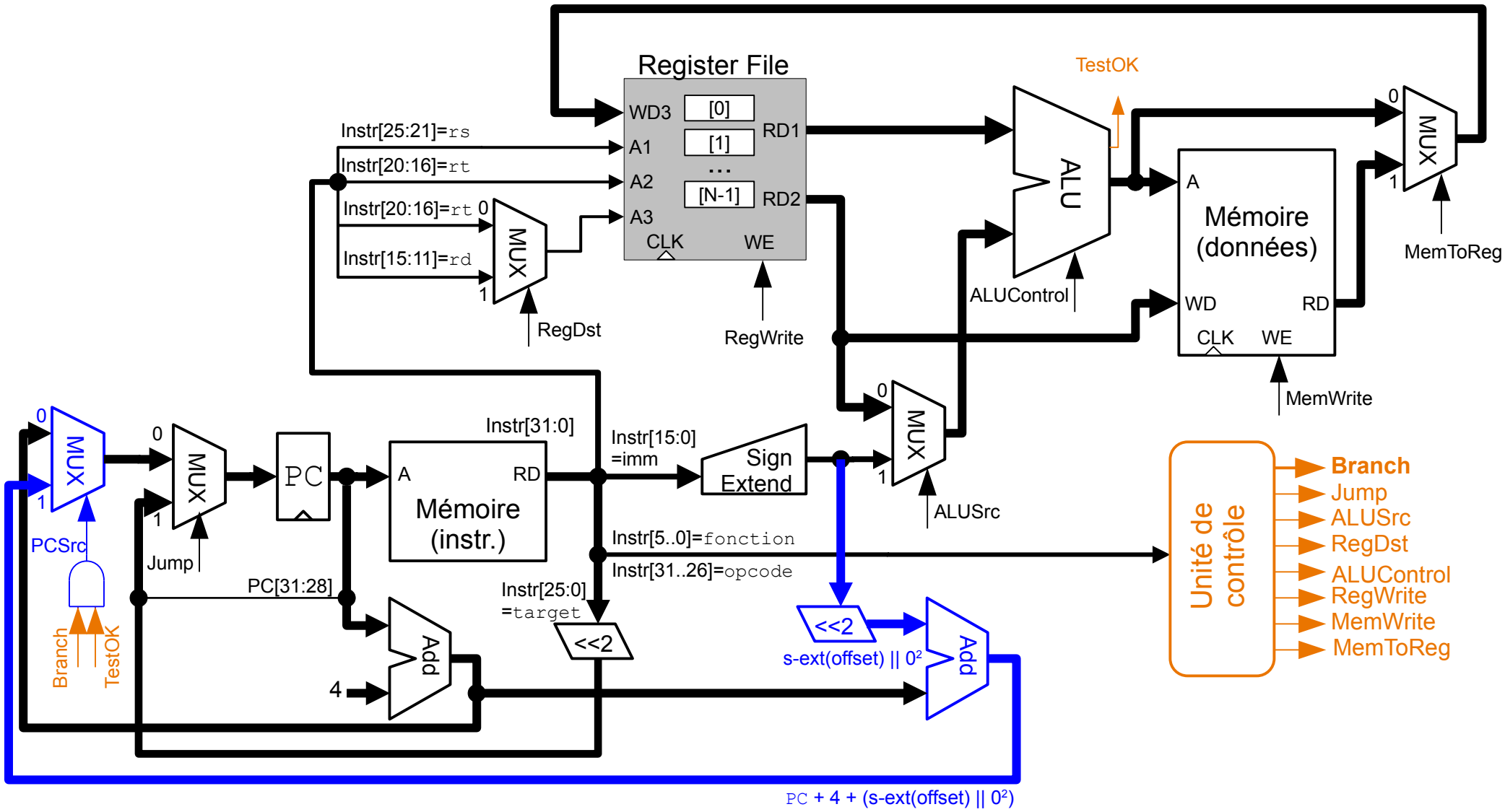
Table des Matières

Branchements conditionnels

Modification du processeur

- Afin d'implémenter les branchements conditionnels, il faut
- **Effectuer des comparaisons**
  - Les comparaisons sont effectuées par l'ALU.
  - Un signal TestOK est activé lorsque la condition du test est vraie.
- **Déterminer la nouvelle valeur du registre PC**. Des blocs de logique combinatoire sont nécessaires:
  - Un additionneur prenant `PC`+4 et l'offset décalé et étendu.
  - Un multiplexeur 2:1 qui sélectionne la source de la nouvelle valeur de PC sur base d'un signal PCSrc (0:PC+4, 1:PC+offset).
  - Un signal fourni par l'unité de contrôle détermine s'il s'agit d'une opération de branchement Branch (1:branchement).

Branchements conditionnels



Note: cette architecture ne supporte pas l'instruction `JR`. Que faudrait-il ajouter/modifier à cet effet ?

Table des Matières

Conclusion

- **Micro-architecture** = implémentation d'une architecture.
- Micro-architecture pas unique (différents fabricants peuvent fournir des implémentations différentes d'une même architecture).

- Dans notre architecture, toutes les instructions s'exécutent en **un cycle**.
  - La durée d'un cycle est contrainte par le chemin critique de la plus longue instruction (typiquement *load* / *store*). La fréquence maximale dépend de cette durée. Les instructions plus courtes (p.ex. opérations arithmétiques) s'exécutent plus lentement que nécessaire).
  - Deux mémoires séparées sont utilisées pour les instructions et les données.

- Architectures modernes implémentent un **pipeline d'instructions**.
  - L'exécution d'une instruction est découpée en plusieurs étapes. A un moment donné, plusieurs instructions peuvent être exécutées en parallèle, chacune étant à une étape différente.

Références

Computer Organization and Design, 4th Edition, D. Patterson and J. Hennessy, Morgan-Kauffman, 2009

Digital Design and Computer Architecture, 2nd Edition, D. Harris and S. Harris, Morgan-Kauffman, 2013