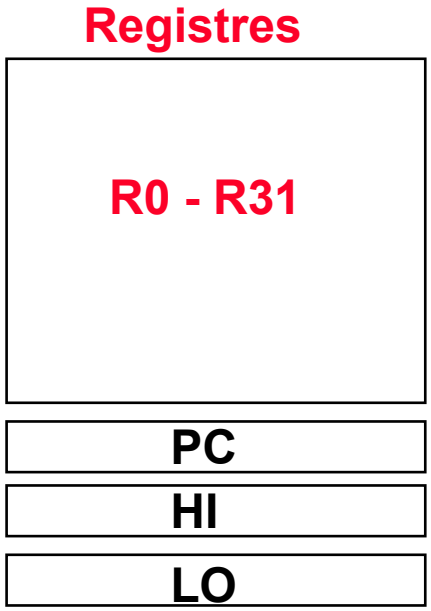
Architecture des ordinateurs
Instructions MIPS – chemin de données

Instructions MIPS R3000

- Instructions
  - Load/Store
  - Arith - Logique
  - Jump - Branch
  - Floating Point
  - Memory Management
  - Special



- Notation des registres : $0, $1, ..., $31
  notation mnémotechnique : $zero ($0), $t0 ($8), etc.

Exemple d'instructions

```
add $t0, $t1, $t2    # $t0=$t1+$t2
sub $t0, $t1, $t2    # $t0=$t1-$t2

lw $t1, a_addr       # $t1=Mem[a_addr]
sw $s1, a_addr       # Mem[a_addr]=$t1
```

Plusieurs modes d'adressage

- ex load

```
la   $a0, addr   # load address addr into $a0
li    $a0, 12    # load immediate $a0 = 12
lb $a0, c($s1) # load byte $a0 = Mem[$s1+c]
lh $a0, c($s1) # load half word
lw $a0, c($s1) # load word
move $s0, $s1   # $s0 = $s1
```

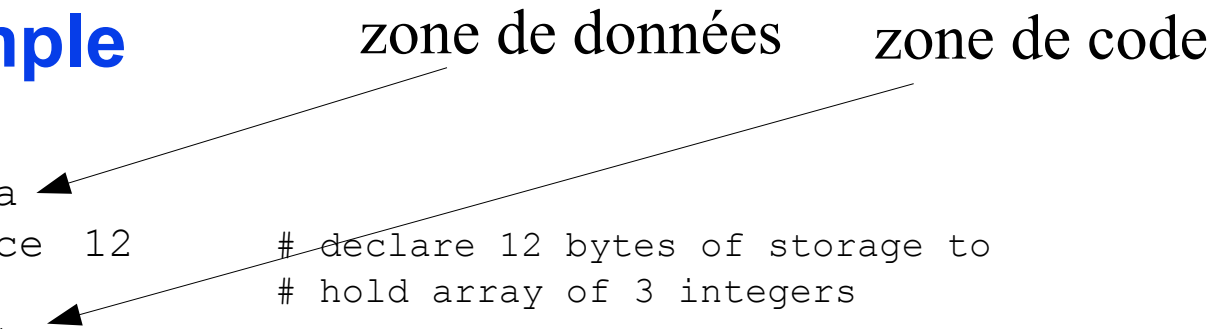
Structures de contrôle

- instruction de saut : jump label
- instruction de branchement : branch

  if cond then goto label

`beqz $s0, label`	if $s0==0 goto label
`bnez $s0, label`	if $s0!=0 goto label
`bge $s0, $s1, label`	if $s0>=$s1 goto label
`ble $s0, $s1, label`	if $s0<=$s1 goto label
`blt $s0, $s1, label`	if $s0<$s1 goto label
`beq $s0, $s1, label`	if $s0==$s1 goto label
`bgez $s0, $s1, label`	if $s0>=0 goto label

Exemple




```
        .data
array1: .space  12      # declare 12 bytes of storage to
                        # hold array of 3 integers
        .text
main:   la $t0, array1 #  load base address of array into register $t0
        li $t1, 5      #  $t1 = 5   ("load immediate")
        sw $t1, ($t0)  #  first array element set to 5; indirect addressing
        li $t1, 13     #   $t1 = 13
        sw $t1, 4($t0) #  second array element set to 13
        li $t1, -7     #   $t1 = -7
        sw $t1, 8($t0) #  third array element set to -7
        done
```

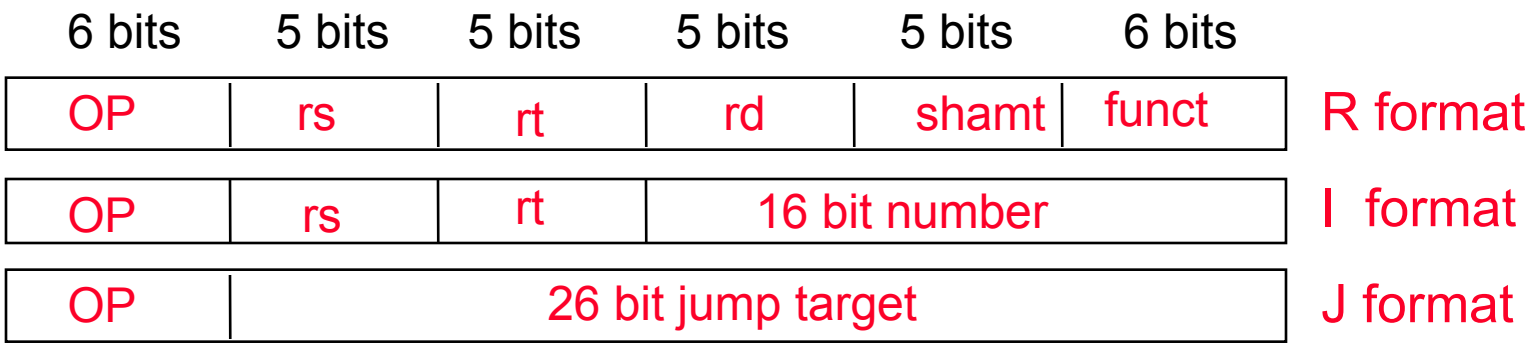
Exemple de programme (2)

```
for (i=0;i<n;i++) a[i]=b[i]+10;
```

```
        xor $2,$2,$2    # zero out index register (i)
        lw $3,n         # load iteration limit
        sll $3,$3,2     # multiply by 4 (words)
        li $4,a         # get address of a (assume < 2^16)
        li $5,b         # get address of b (assume < 2^16)
loop:   add $6,$5,$2    # compute address of b[i]
        lw $7,0($6)     # load b[i]
        addi $7,$7,10   # compute b[i]=b[i]+10
        add $6,$4,$2    # compute address of a[i]
        sw $7,0($6)     # store into a[i]
        addi $2,$2,4    # increment i
        blt $2,$3,loop # loop if post-test succeeds
```

Formats instructions

- 3 formats d'instruction tous sur 32 bits



Exemple de codage

ADD $2, $3, $4

- R-type instruction Arith/Log/Shift/Comparaison
- Opcode : 0 (UAL reg), rd=2, rs=3, rt=4, func=100000
- 000000 00011 00100 00010 00000 100000
- codage hexa : 0x00641020

ADDI $2, $3, 12

- I-type instruction A/L/S/C
- Opcode : 001000, rs=3, rt=2, imm=12
- 001000 00011 00010 0000000000001100
- codage hexa : 0x2062000C

Exemple de codage

BEQ $3, $4, 4

- I-type conditional branch instruction
- Opcode : 000100, rs=00011, rt=00100, imm=4 (skips next 4 instructions)
- 000100 00011 00100 0000000000000100

SW $2, 128($3)

- I-type memory address instruction
- Opcode : 101011, rs=00011, rt=00010, imm=0000000010000000
- 101011 00011 00010 0000000010000000

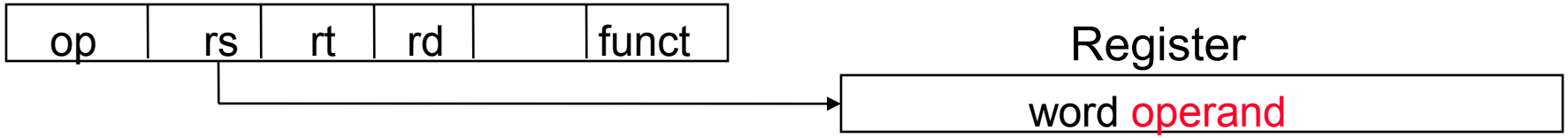
J 128

- J-type pseudodirect jump instruction
- Opcode : 000010, 26-bit pseudodirect address is 128/4 = 32
- 000010 00000000000000000000100000

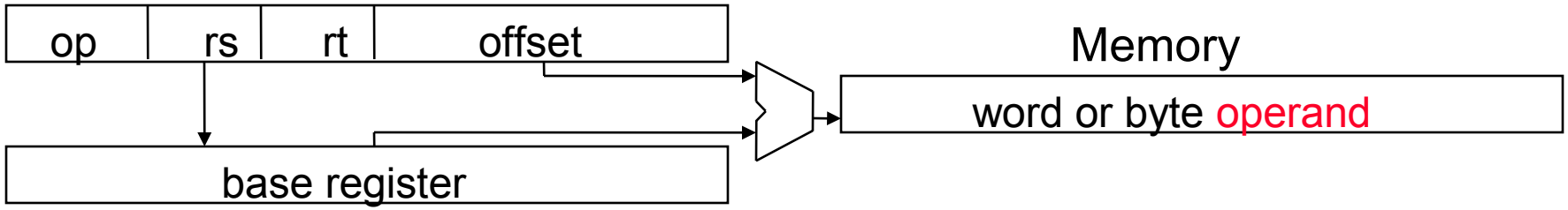
Mode d'adressage

Modes d'adressage MIPS

- **Registre** – opérande dans un registre

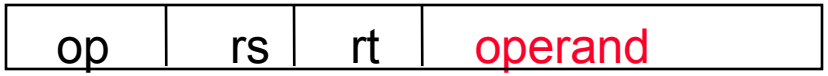


- **Basé** (déplacement) – l'adresse (en mémoire) de l'opérande = contenu registre + offset (16 bits) contenu dans l'instruction



- **Immédiat** – opérande sur 16 bits dans l'instruction

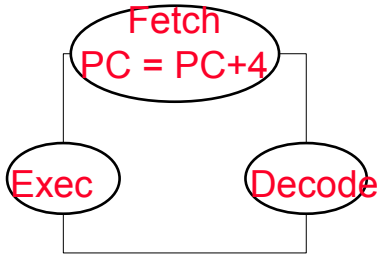
3. Immediate addressing



Construction du chemin de données

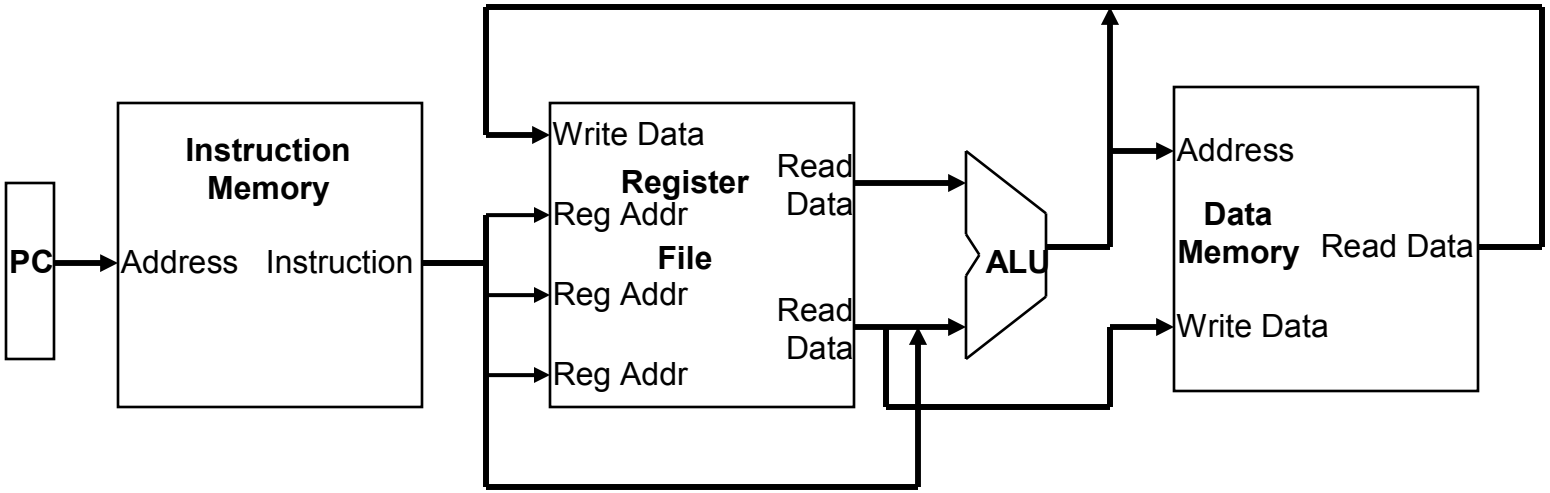
Fonctionnalité du chemin de données

- Chemin de données simplifié pour les instructions :
  - chargement/rangement : **`lw, sw`**
  - arith. et logique : **`add, addu, sub, subu, and, or, xor, nor, slt, sltu`**
  - arith. et logique immédiate : **`addi, addiu, andi, ori, xori, slti, sltiu`**
  - de contrôle de flux : **`beq, j`**
- Schéma d'exécution :
  - Fetch : lecture et mise à jour de registre d'adresse RAM de l'instruction courante
  - Decode : décodage de l'instruction (et lecture des registres)
  - Execute : exécution de l'instruction



Vue simplifiée

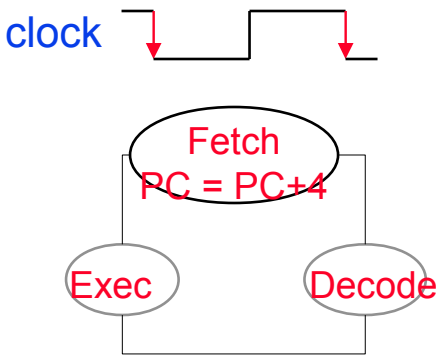
- Deux types d'unités :
  - Combinatoires (UAL, logique de contrôle)
  - Séquentielles : bancs de registres, mémoires

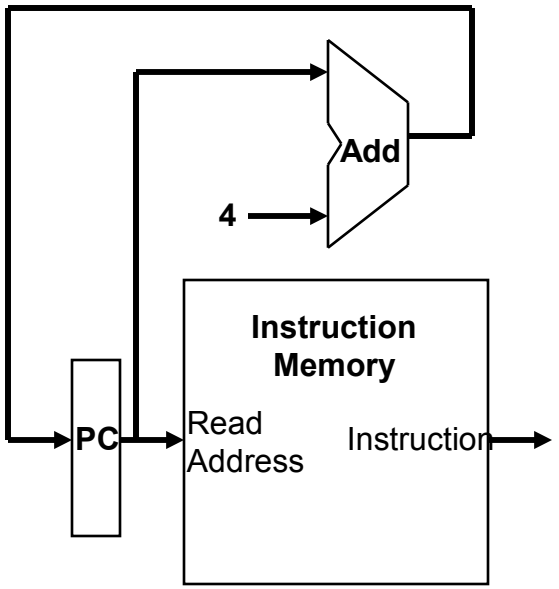


- Hyp : opération traitée en 1 seul cycle
- Mémoire instructions – mémoire de données (modèle Harvard)

Recherche d'instruction (*Fetch*)

- Deux étapes
  - Lire l'instruction dans la mémoire
  - Mettre à jour le registre PC



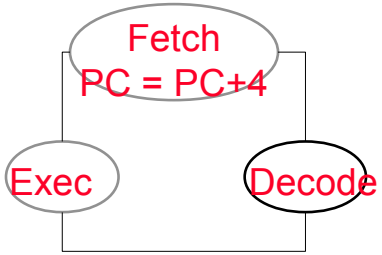


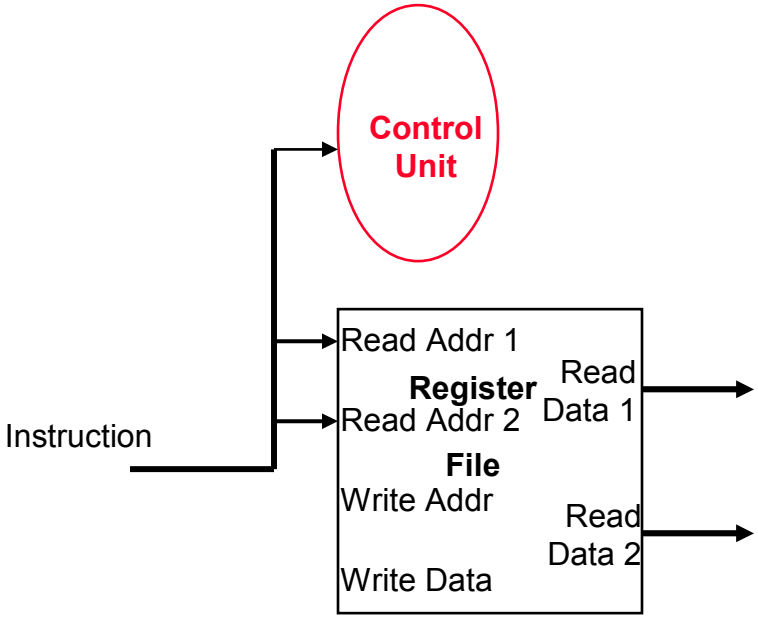
PC est mis à jour à chaque top horloge (pas besoin de signal d'écriture)

La lecture est une action « combinatoire » (pas besoin de signal de contrôle)

Décodage (*Decode*)

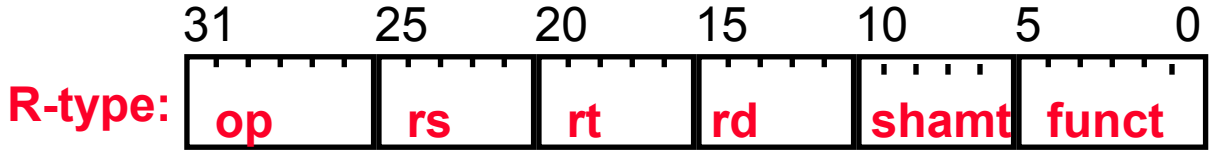
- Décoder consiste à envoyer l'opcode à l'unité de contrôle et les signaux de lecture de registres (signaux d'adresse de registres contenu dans l'instruction).



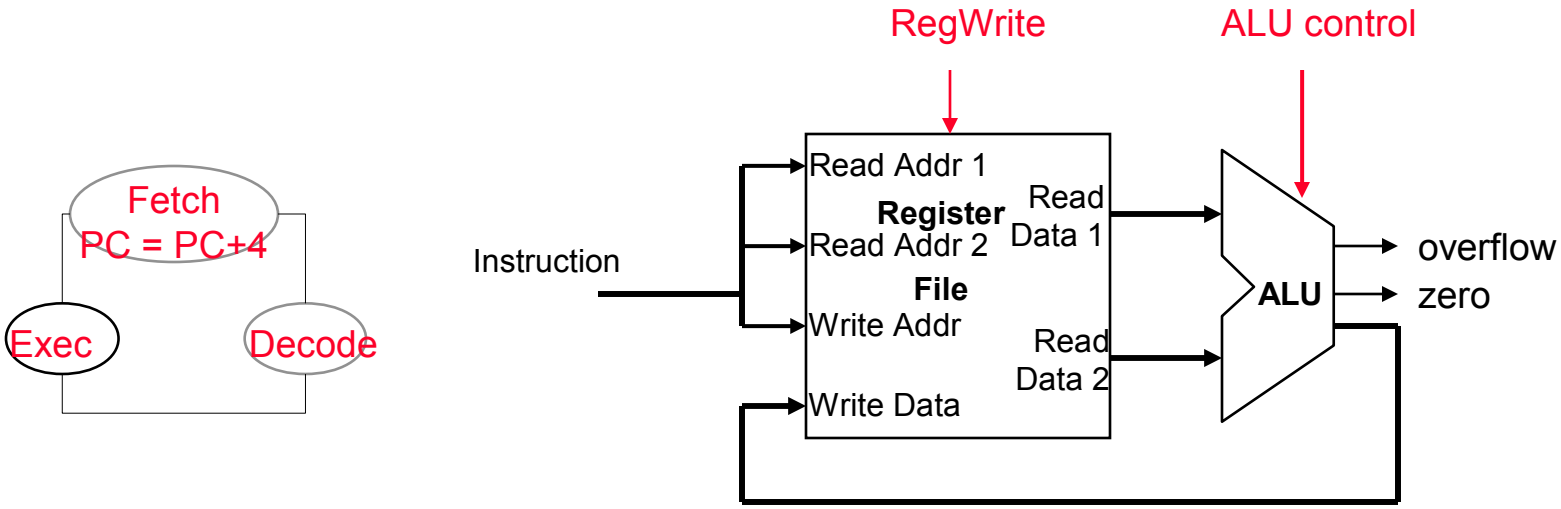


Exécution (*Execute*) Instruction de type R

- **`add, sub, slt, and, or`**

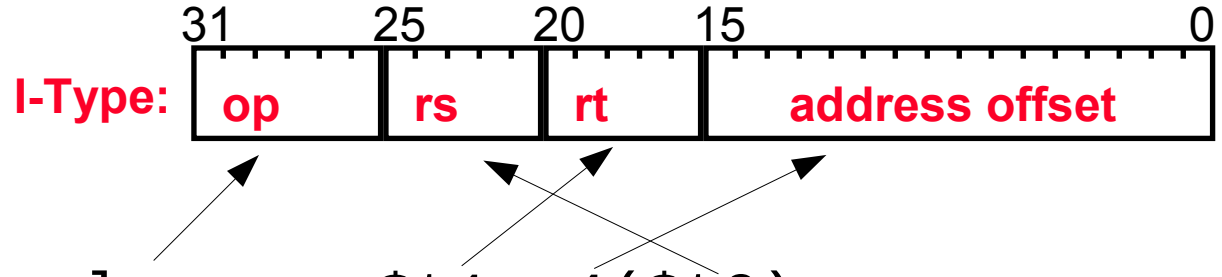


exécuter l'op (op et funct) sur les opérandes rs et rt
stocker le résultat dans rd



Banc de registre pas mis à jour à chaque instruction (e.g. **`sw`**), signal spécifique (écriture sur front descendant sinon conflit avec la lecture)

Exécution : Instructions Load et Store




```
Ex : lw      $t1, 4($t0)
     sw      $t0, 12($sp)
```

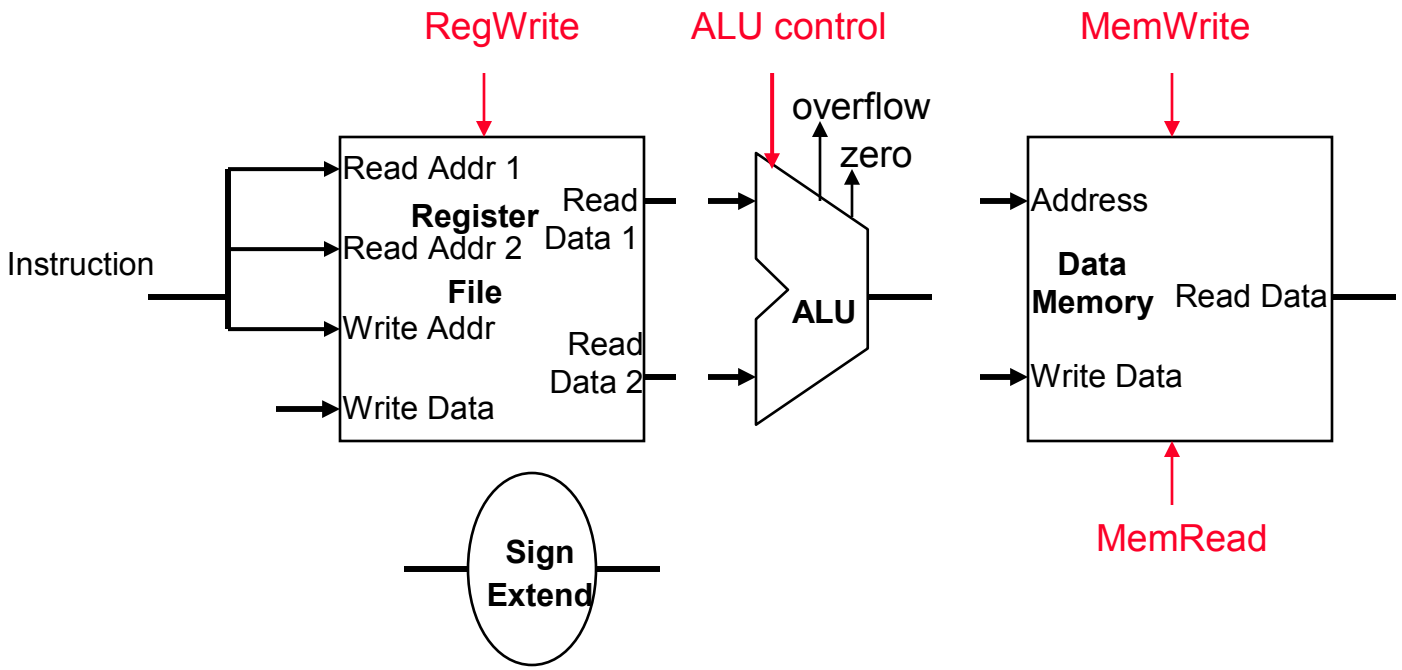
additionne le contenu d'un registre (rs) à une constante 16 bits :

- registre de base (rs) lu après décodage
- extension de la constante à 32 bits

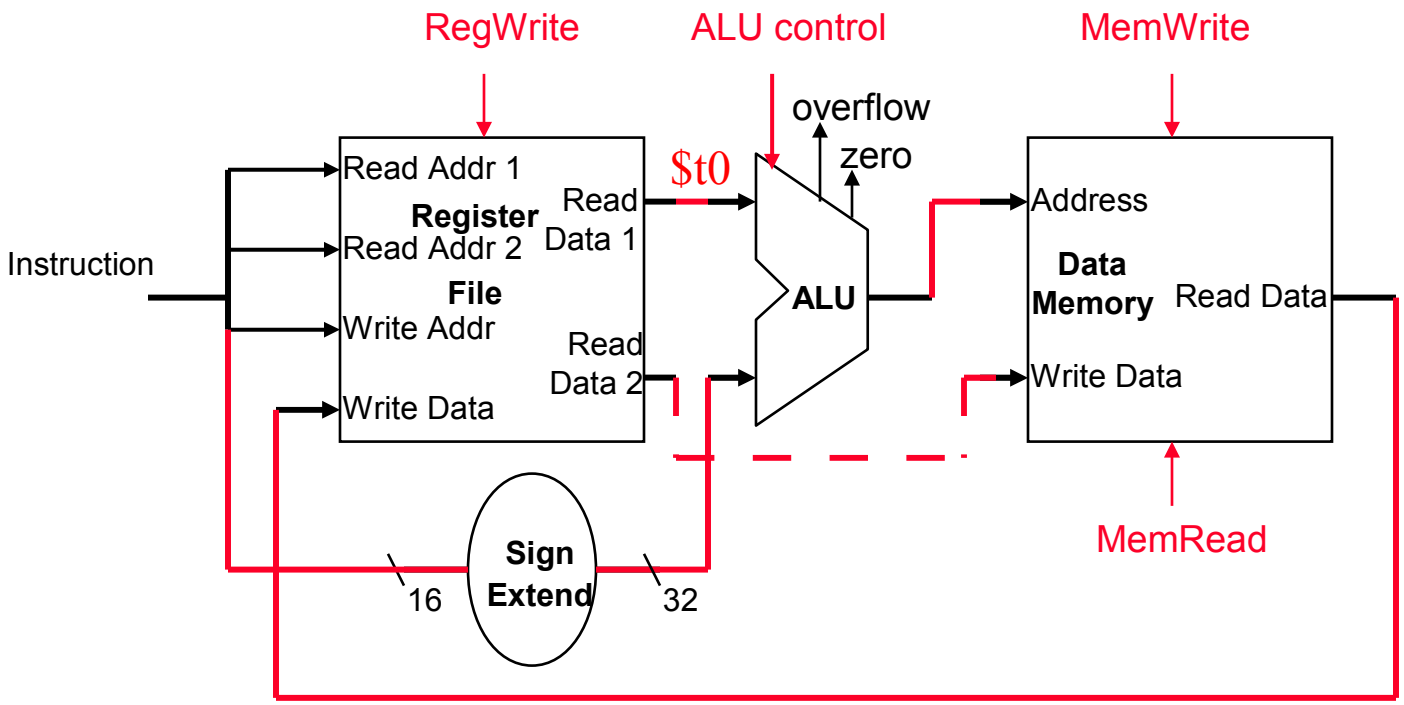
store : lecture dans le banc de registres, écriture en mémoire

load : lecture mémoire, écriture dans le banc de registres

Instructions Load et Store



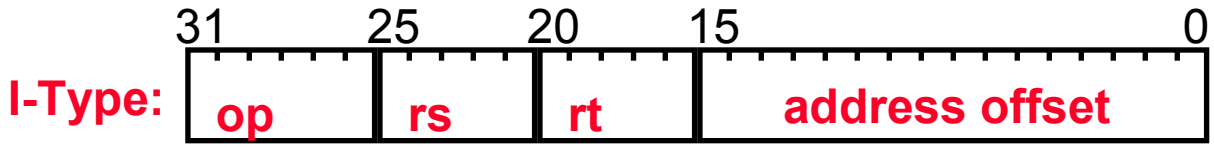
Instructions Load et Store




```
lw  $t1, 4($t0)
```

```
Read data 1 : @ $t0
4 sur 16 bits -> 4 sur 32 bits
ALU control : addition
MemRead = 1
Write addr = @ $t1
RegWrite = 1
```

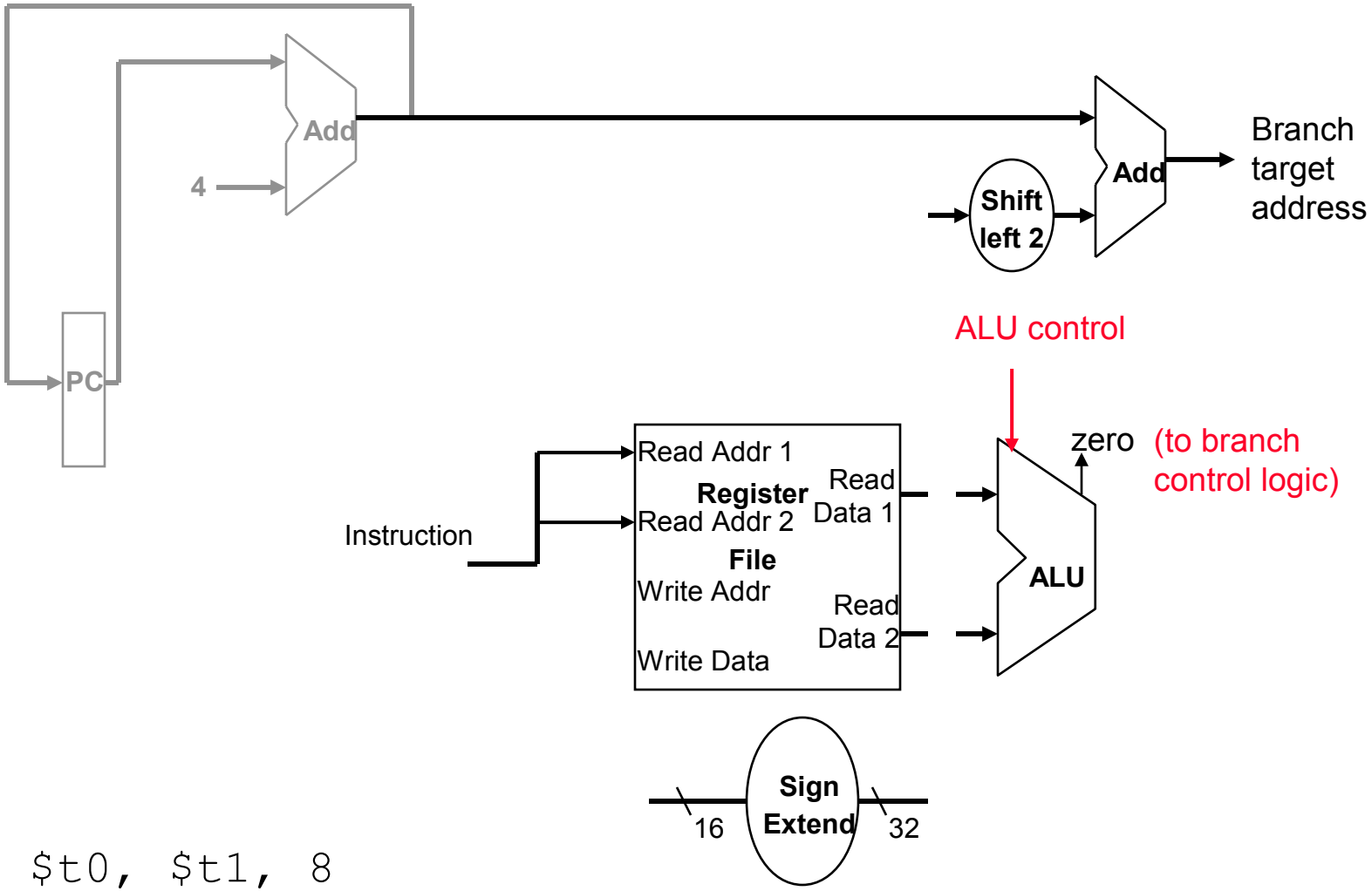
Instruction Branch



compare les registres rs et rt (test flag **`zero`**)

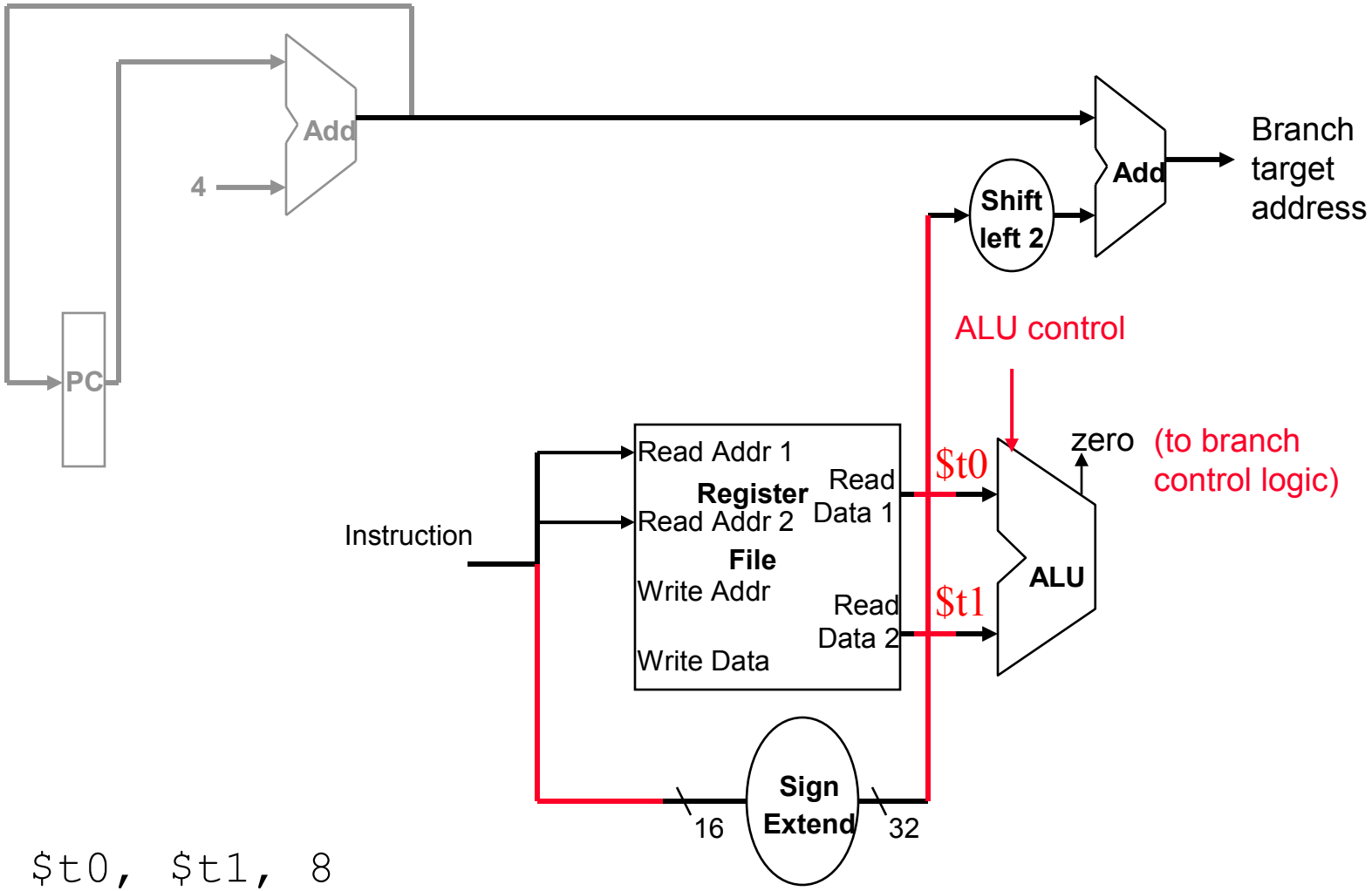
calcule de l'adresse destination (relative à PC) :
(PC+4) + extension(offset) << 2

Instruction Branch



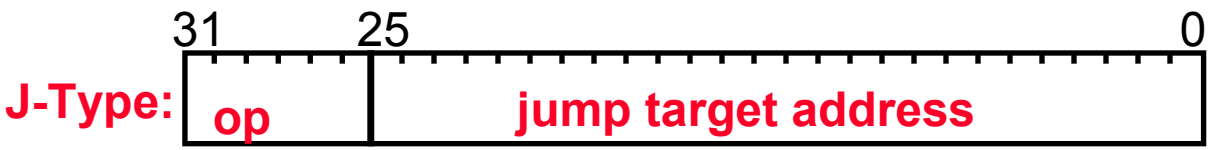

```
beq $t0, $t1, 8
(if ($t0-$t1 == 0) PC += 8)
```

Instruction Branch

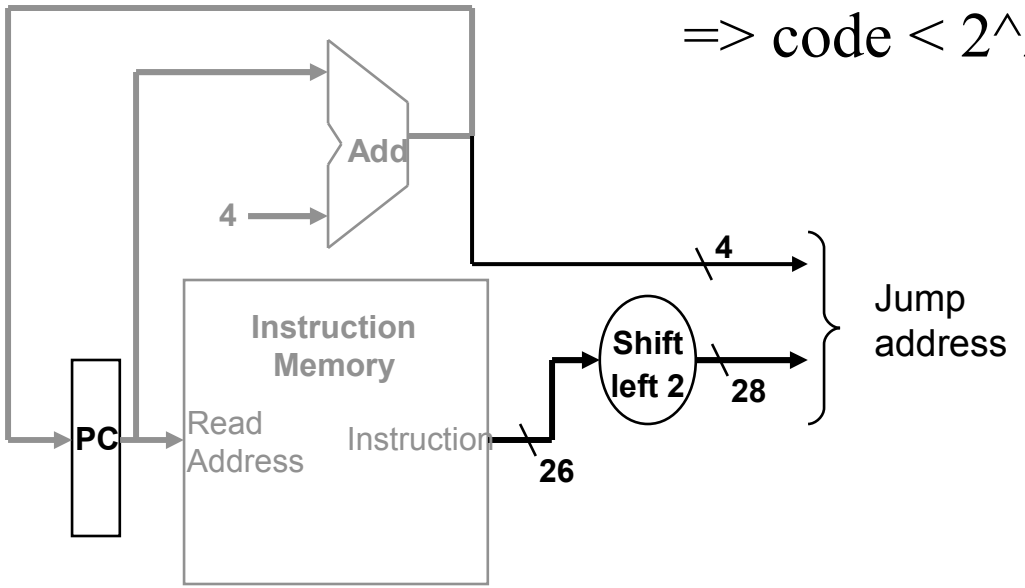



```
beq $t0, $t1, 8
(if ($t0-$t1 == 0) PC += 8)
```

Instruction Jump



récupération des 4 bits
de poids forts de PC+4
=> code < 2^28



Assemblage des composants

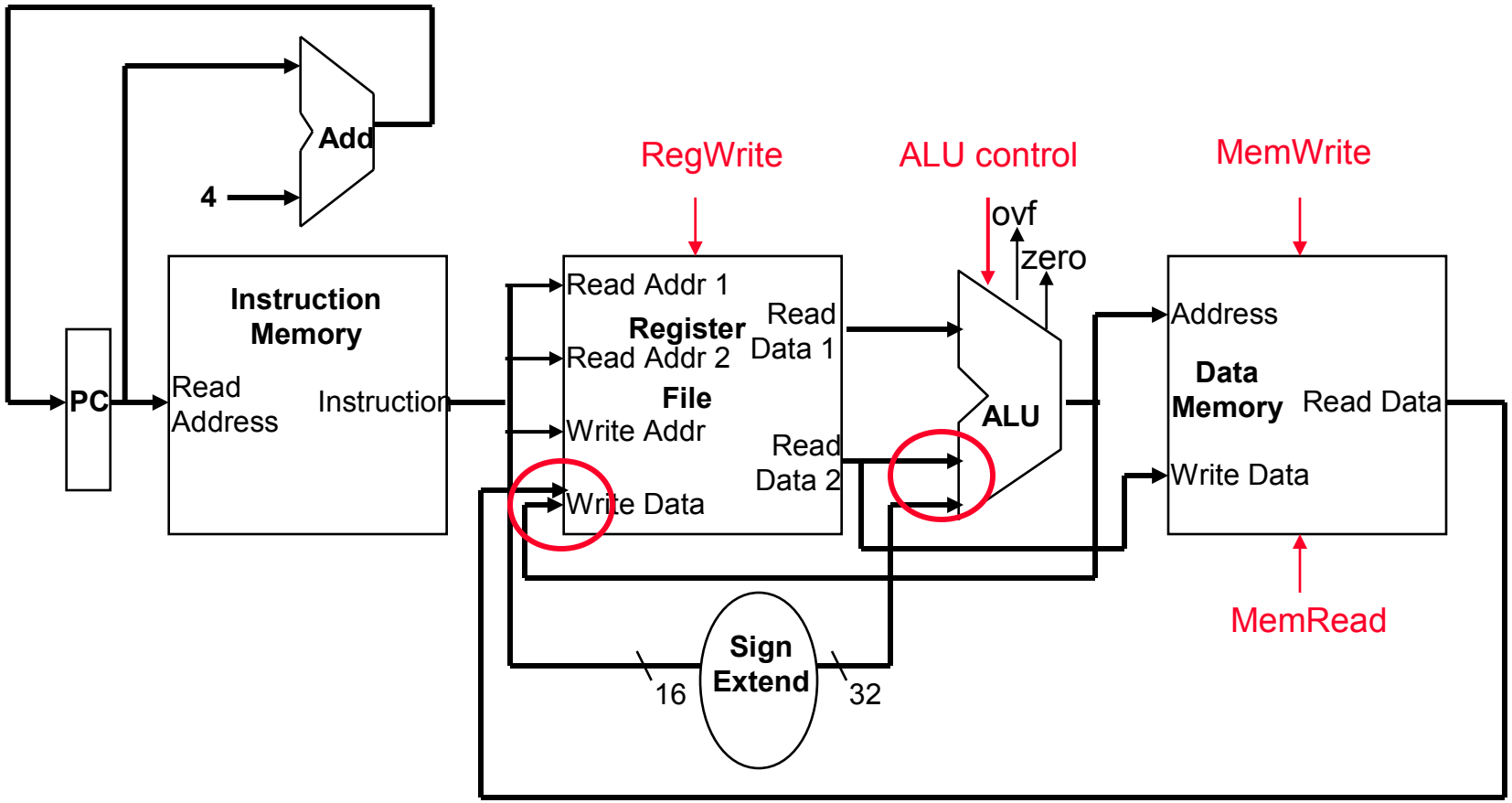
- ajout des lignes de contrôle
- Fetch, decode et execute en un seul cycle :

  chaque ressource utilisée une seule fois (ex : séparation des mémoires)

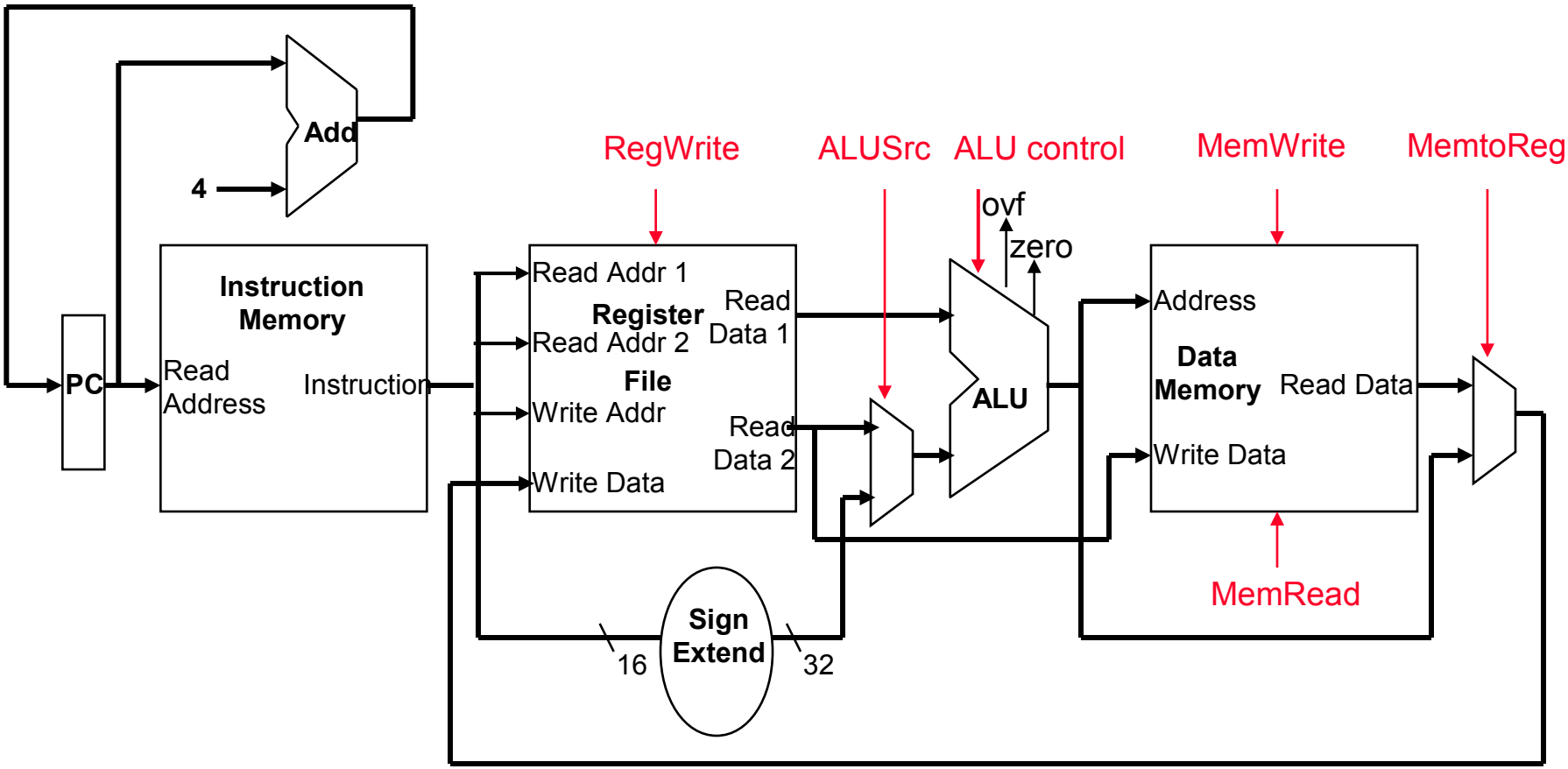
  multiplexeur

- temps de cycle réglé sur l'instruction la plus longue

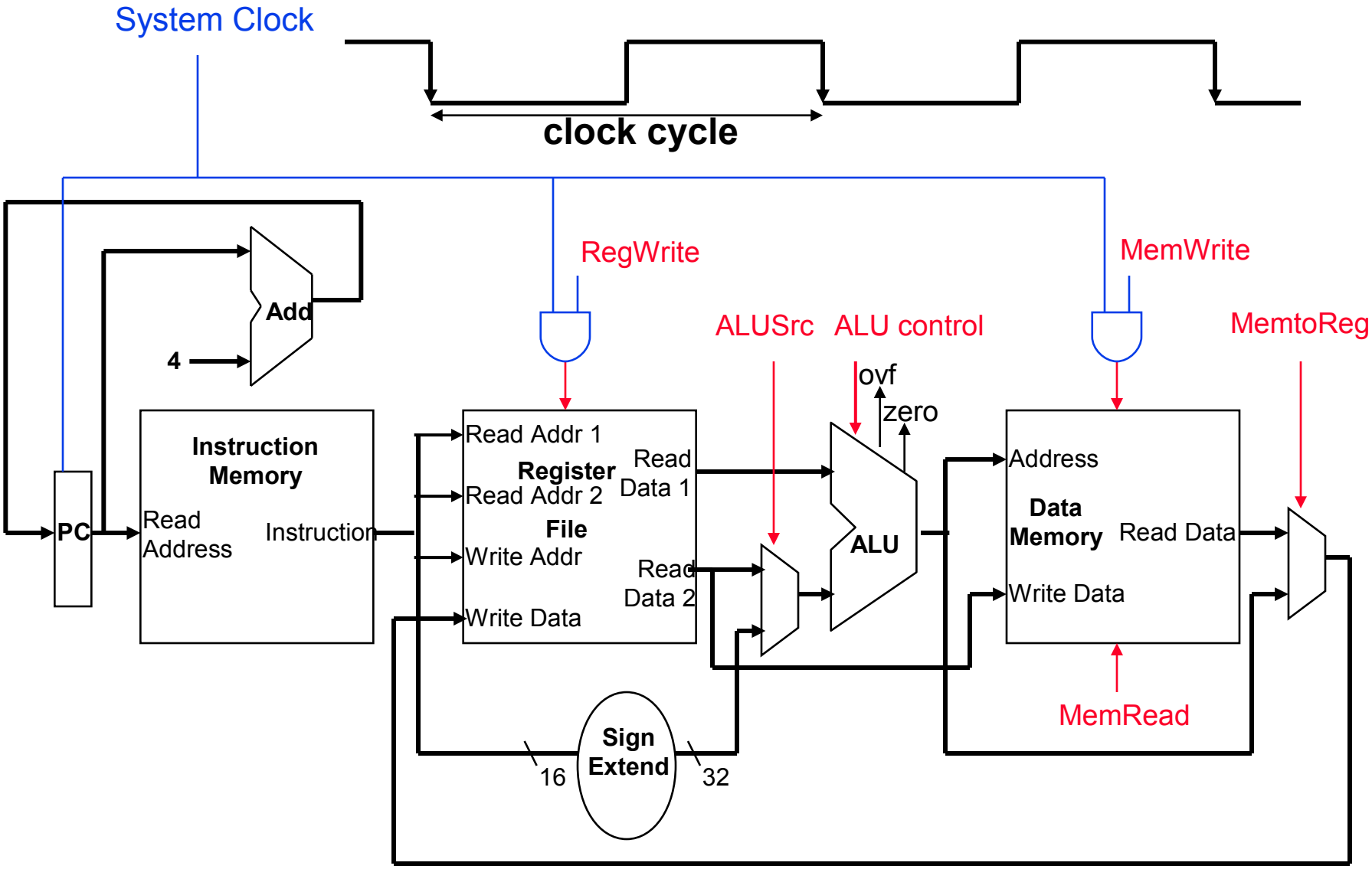
Fetch, Decode, Execute



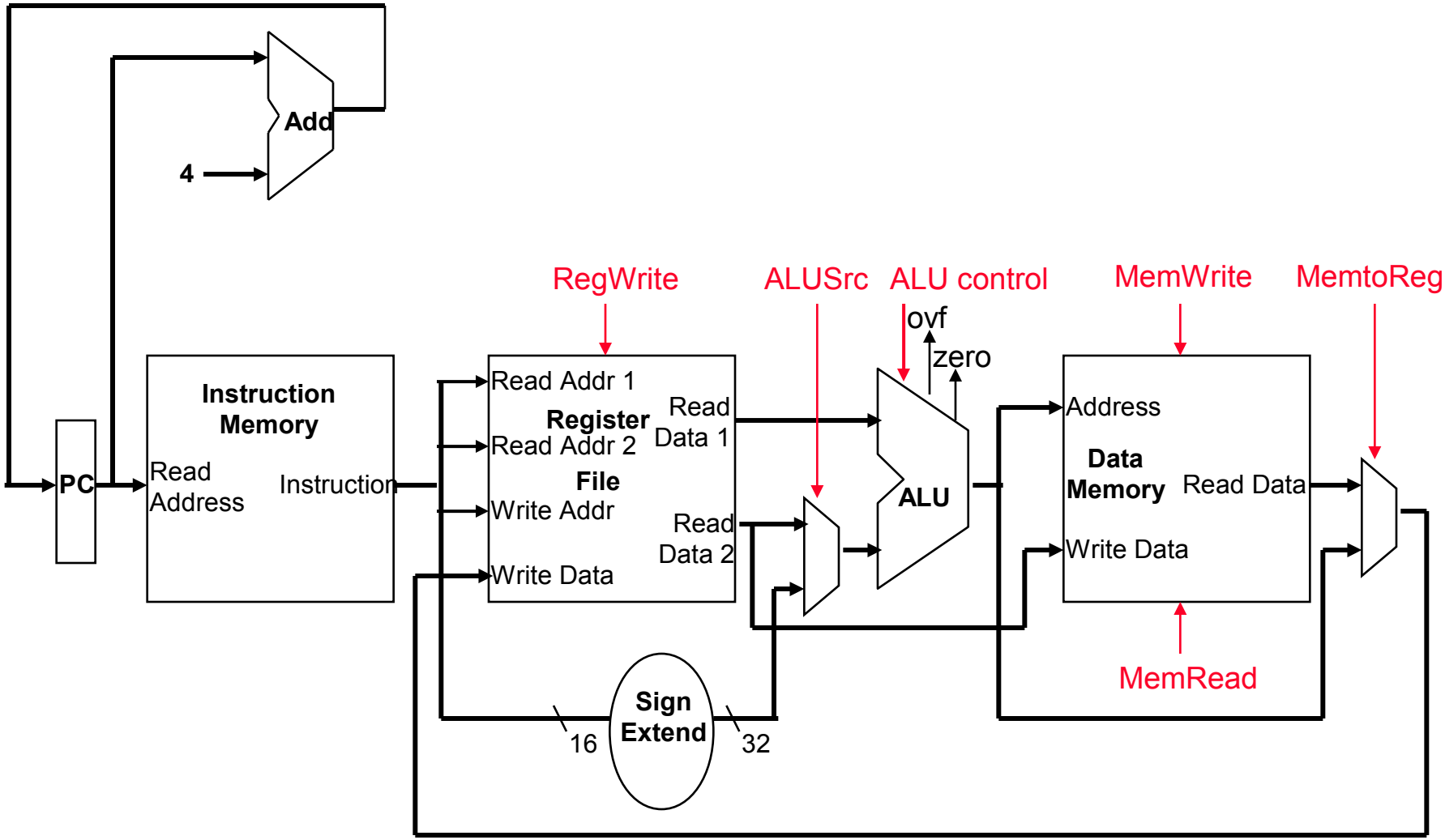
Ajout de mutlipexeurs



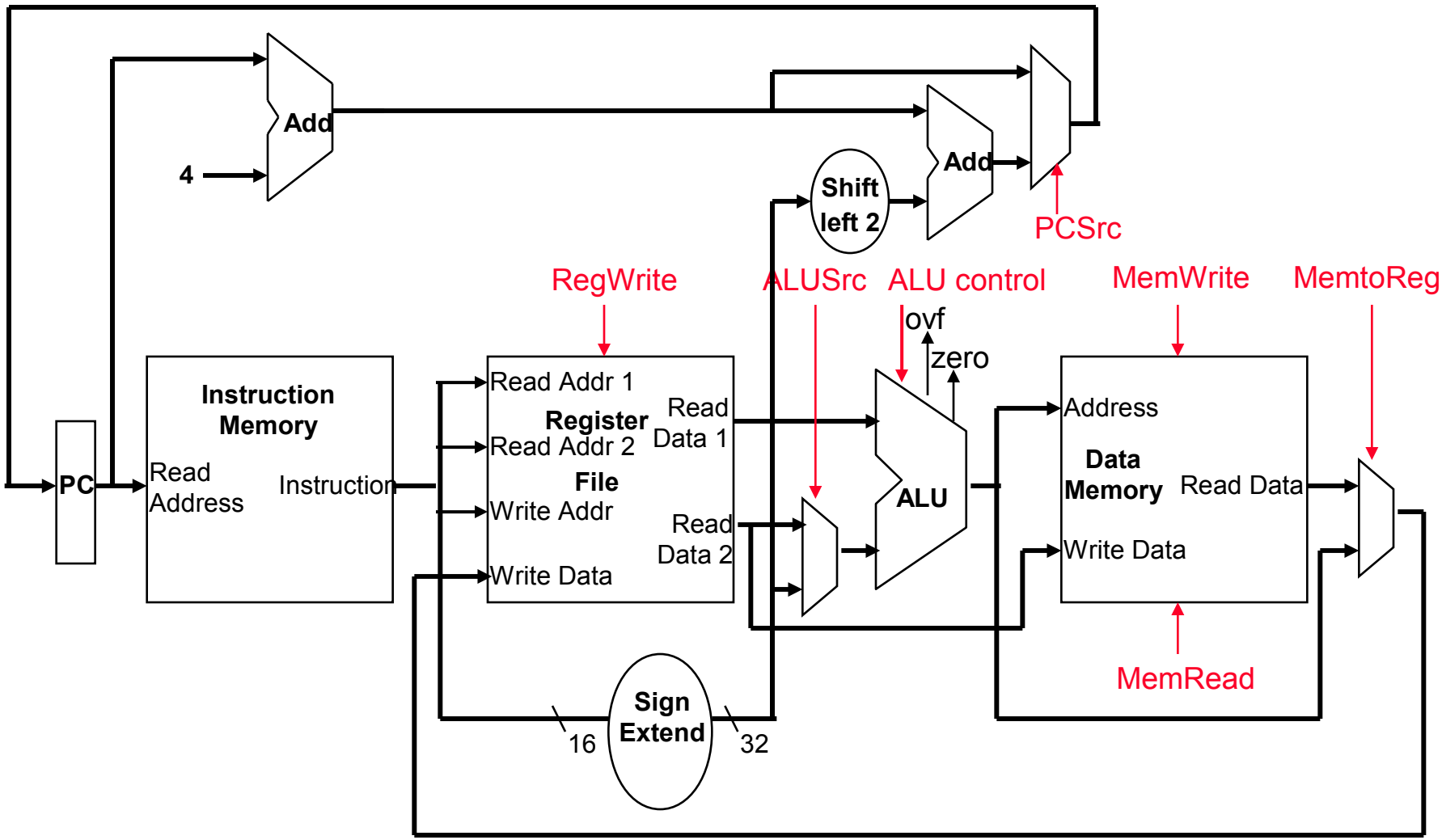
Ajout de l'horloge



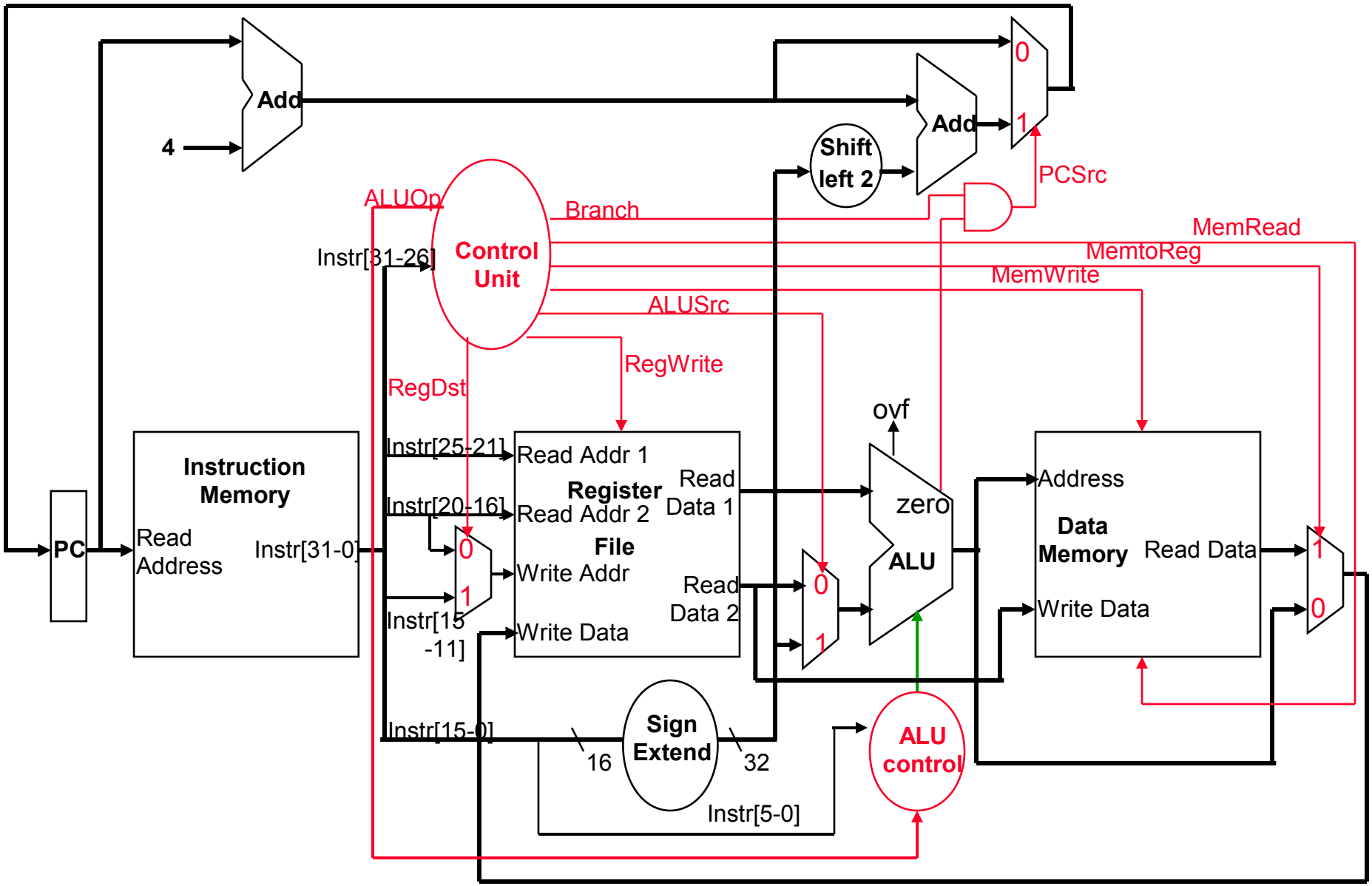
Ajout de l'unité de branchement



Ajout de l'unité de branchement



Ajout de l'unité de contrôle



ALU control

Opérations UAL

- Code op de l'UAL

Entrées op UAL	Fonction
0000	and
0001	or
0010	xor
0011	nor
0110	add
1110	subtract
1111	set on less than

Circuit : ALU control

- entrées : funct (dans le code op) + bits ALUOp (de l'unité de contrôle)
- sorties : ALUcontrol bits

Instr op	funct	ALUOp	action	ALUcontrol
lw	xxxxxx	00		
sw	xxxxxx	00		
beq	xxxxxx	01		
add	100000	10	add	0110
sub	100010	10	subtract	1110
and	100100	10	and	0000
or	100101	10	or	0001
xor	100110	10	xor	0010
nor	100111	10	nor	0011
slt	101010	10	slt	1111

Circuit : ALU control

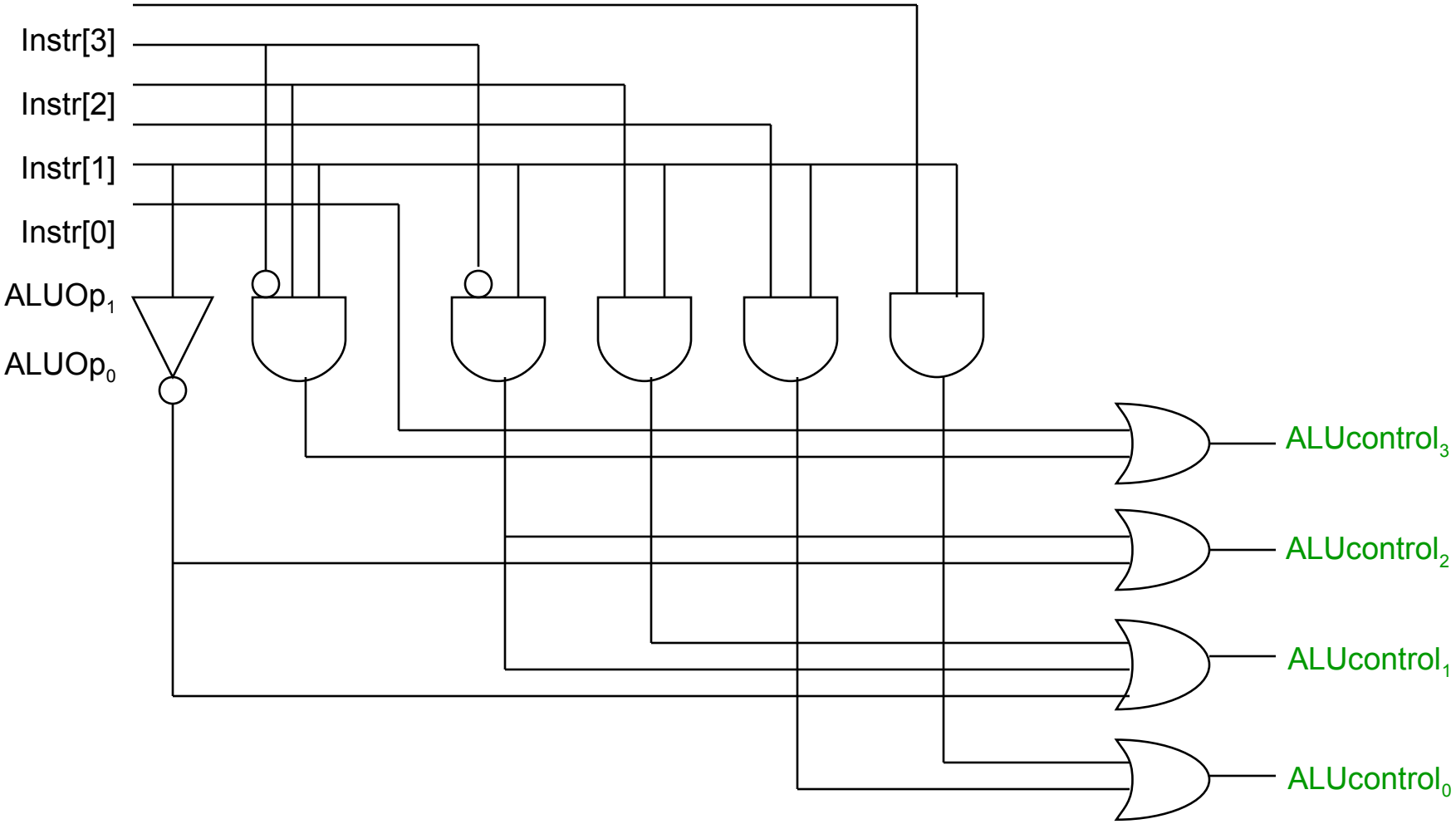
- entrées : funct (dans le code op) + bits ALUOp (de l'unité de contrôle)
- sorties : ALUcontrol bits

Instr op	funct	ALUOp	action	ALUcontrol
lw	xxxxxx	00	add	0110
sw	xxxxxx	00	add	0110
beq	xxxxxx	01	subtract	1110
add	100000	10	add	0110
subt	100010	10	subtract	1110
and	100100	10	and	0000
or	100101	10	or	0001
xor	100110	10	xor	0010
nor	100111	10	nor	0011
slt	101010	10	slt	1111

ALU Control : table de vérité

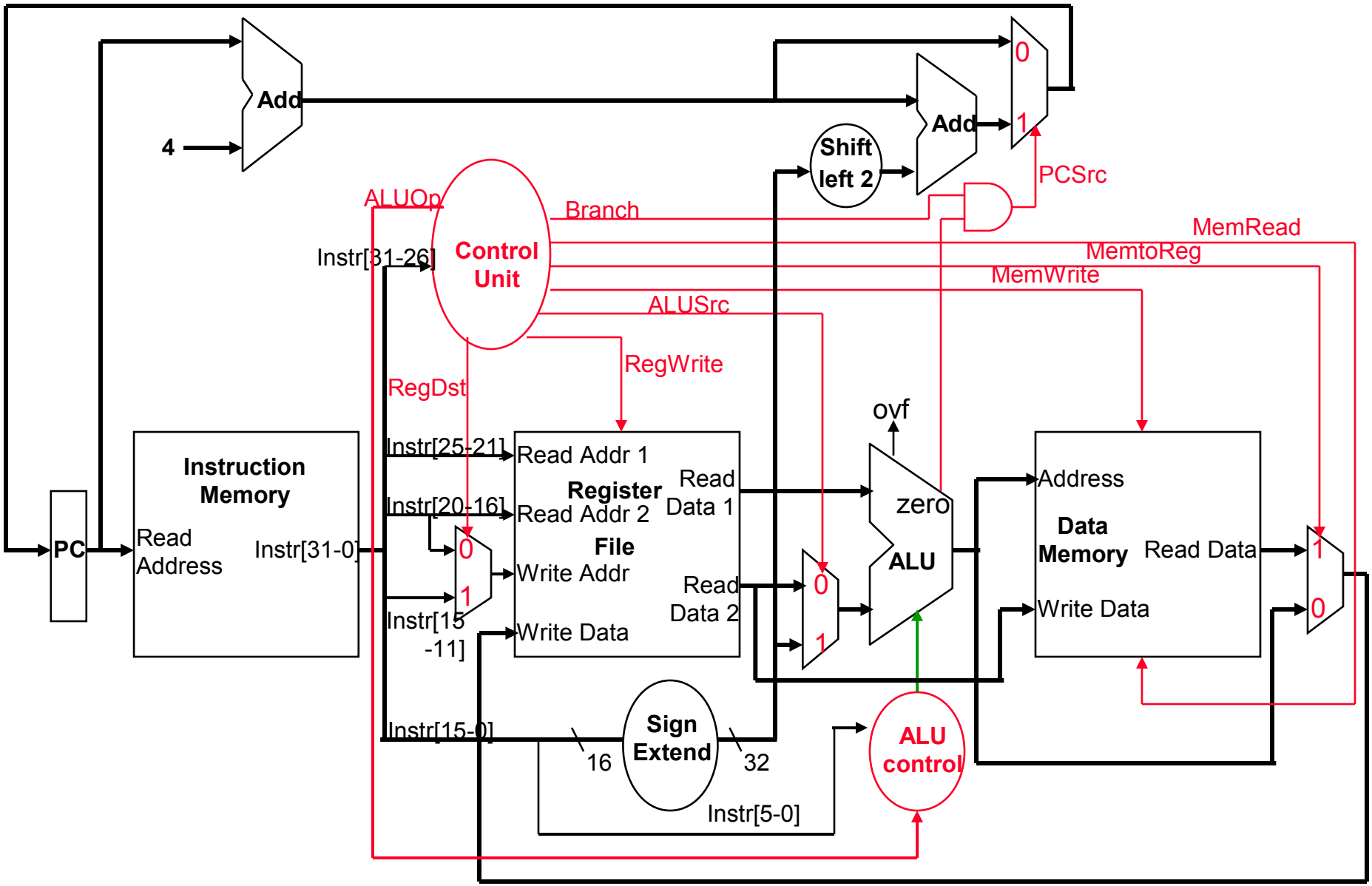
F5	F4	F3	F2	F1	F0	ALU Op_1	ALU Op_0	ALU $control_3$	ALU $control_2$	ALU $control_1$	ALU $control_0$
X	X	X	X	X	X	0	0	0	1	1	0
X	X	X	X	X	X	0	1	1	1	1	0
X	X	0	0	0	0	1	0	0	1	1	0
X	X	0	0	1	0	1	0	1	1	1	0
X	X	0	1	0	0	1	0	0	0	0	0
X	X	0	1	0	1	1	0	0	0	0	1
X	X	0	1	1	0	1	0	0	0	1	0
X	X	0	1	1	1	1	0	0	0	1	1
X	X	1	0	1	0	1	0	1	1	1	1

ALU Control : circuit combinatoire

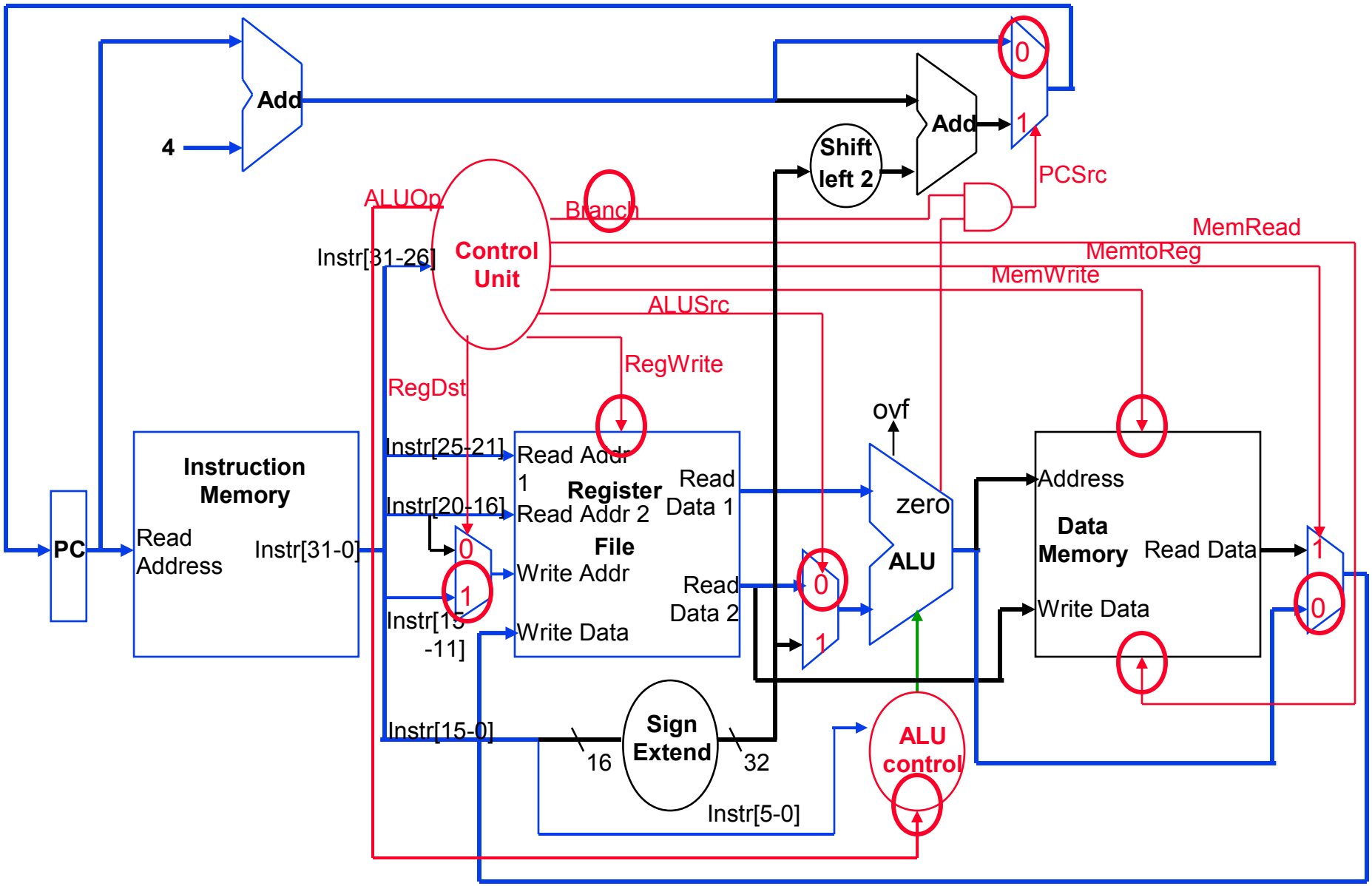


Conception de l'unité de contrôle

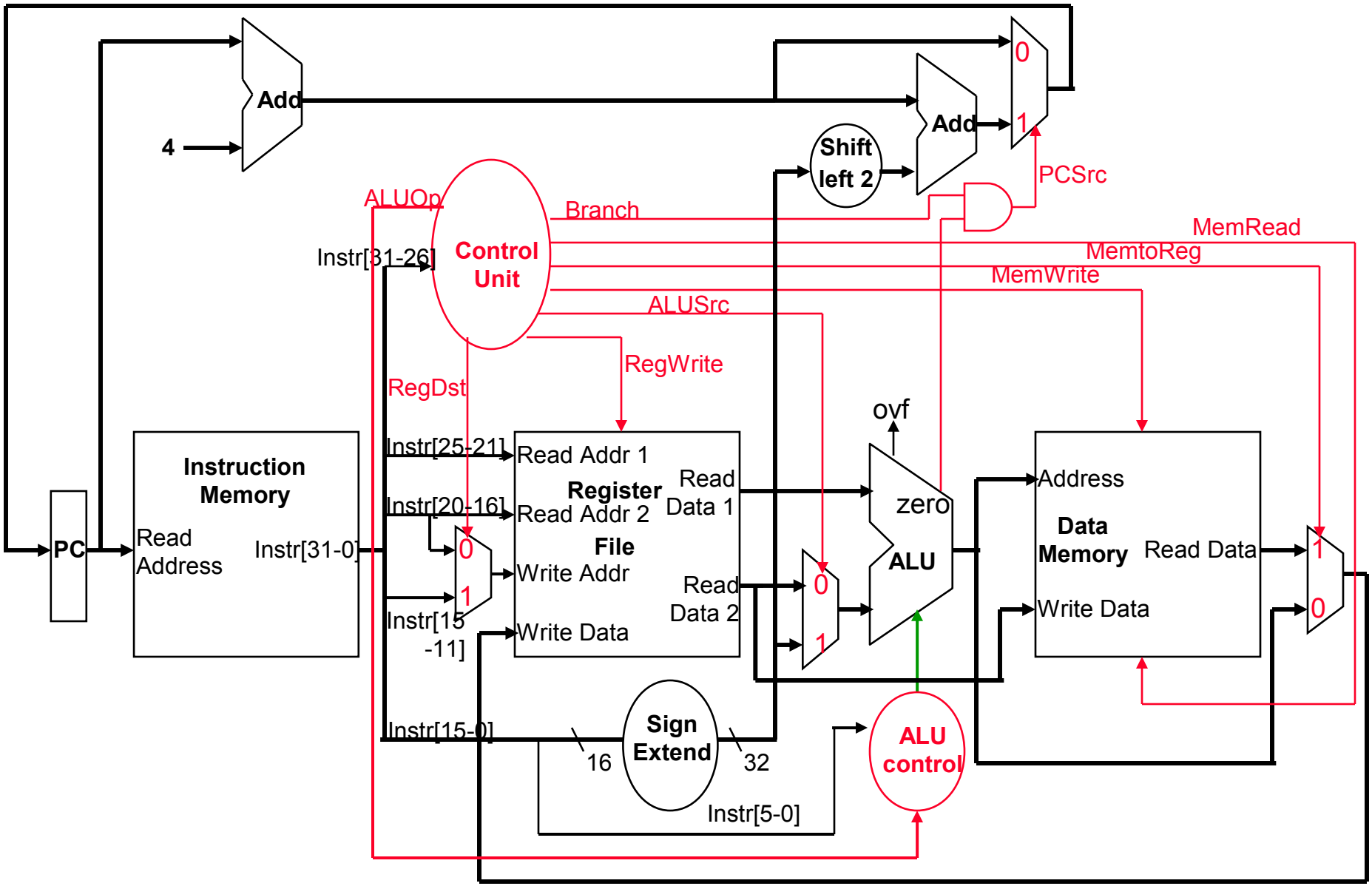
R-type Instruction Data/Control Flow



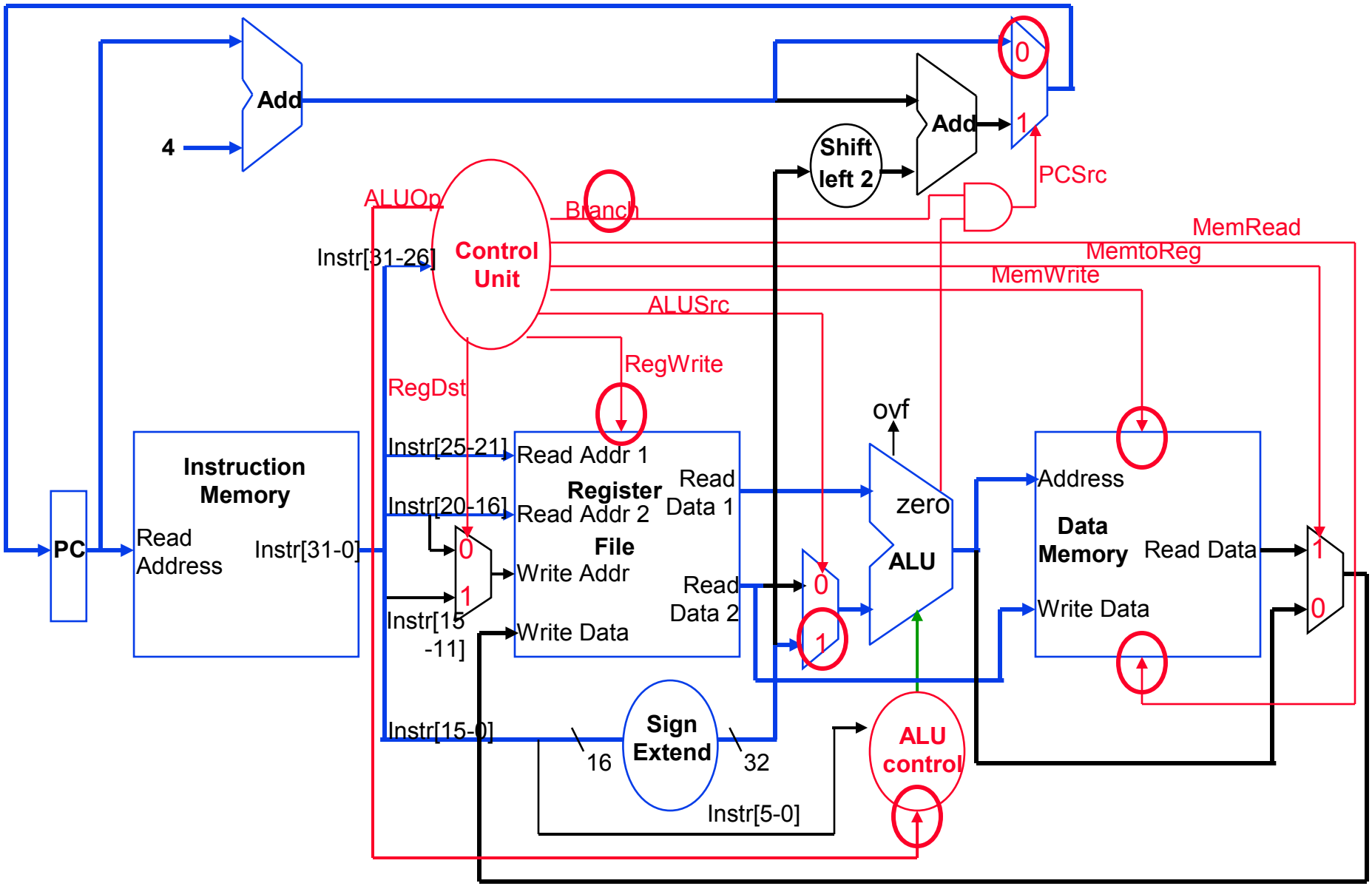
R-type Instruction Data/Control Flow



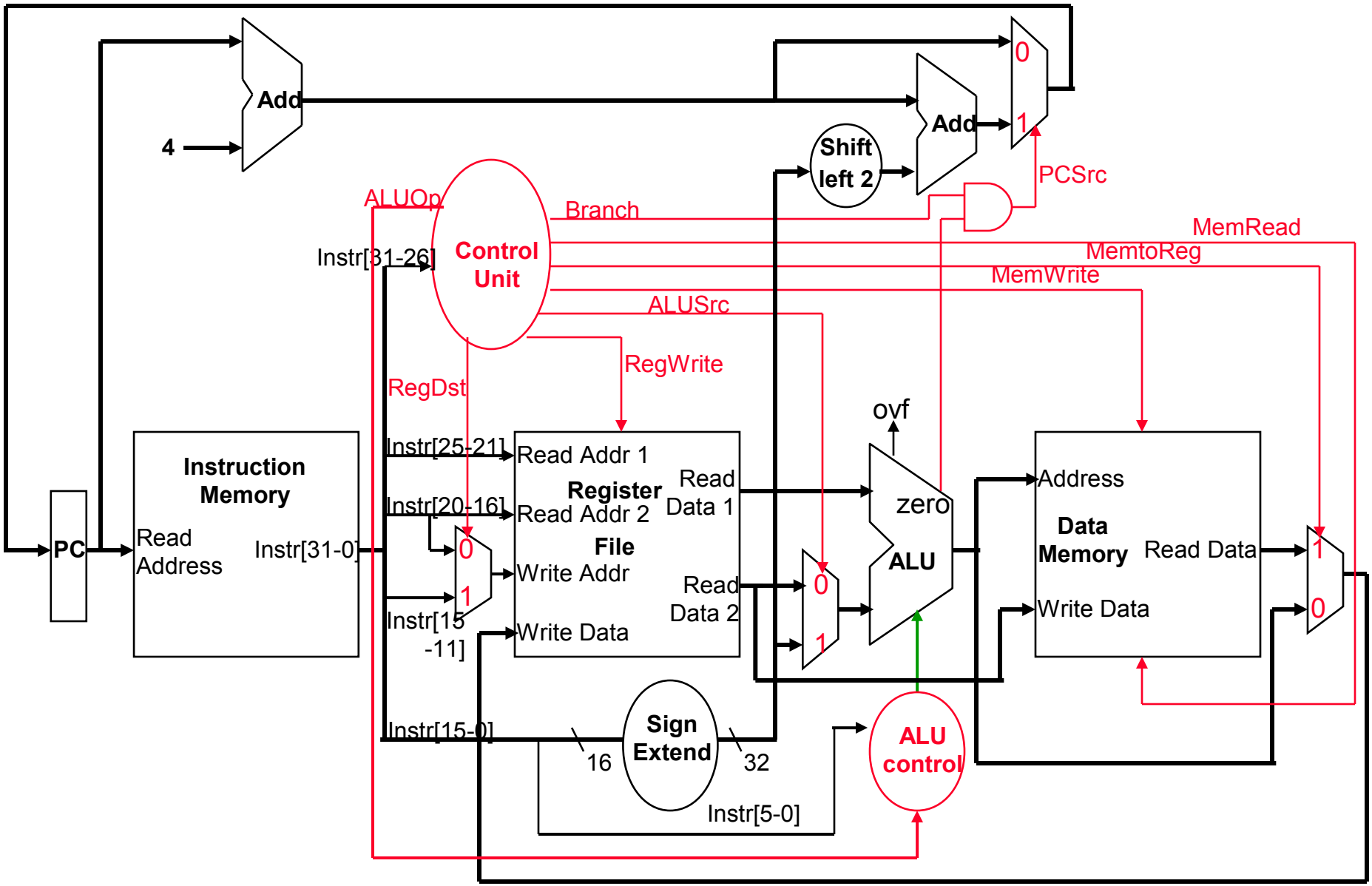
Store Word Instruction Data/Control Flow



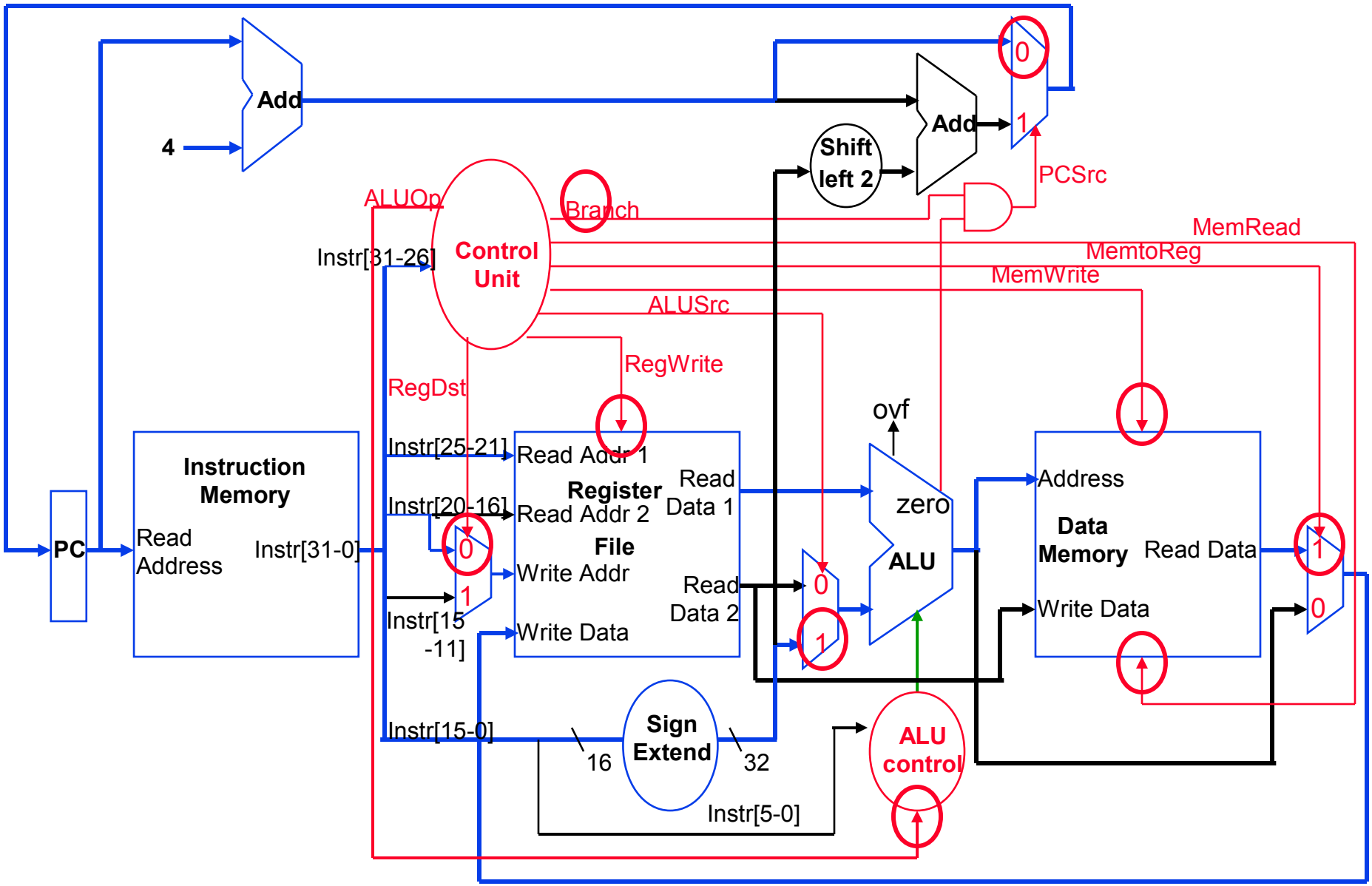
Store Word Instruction Data/Control Flow



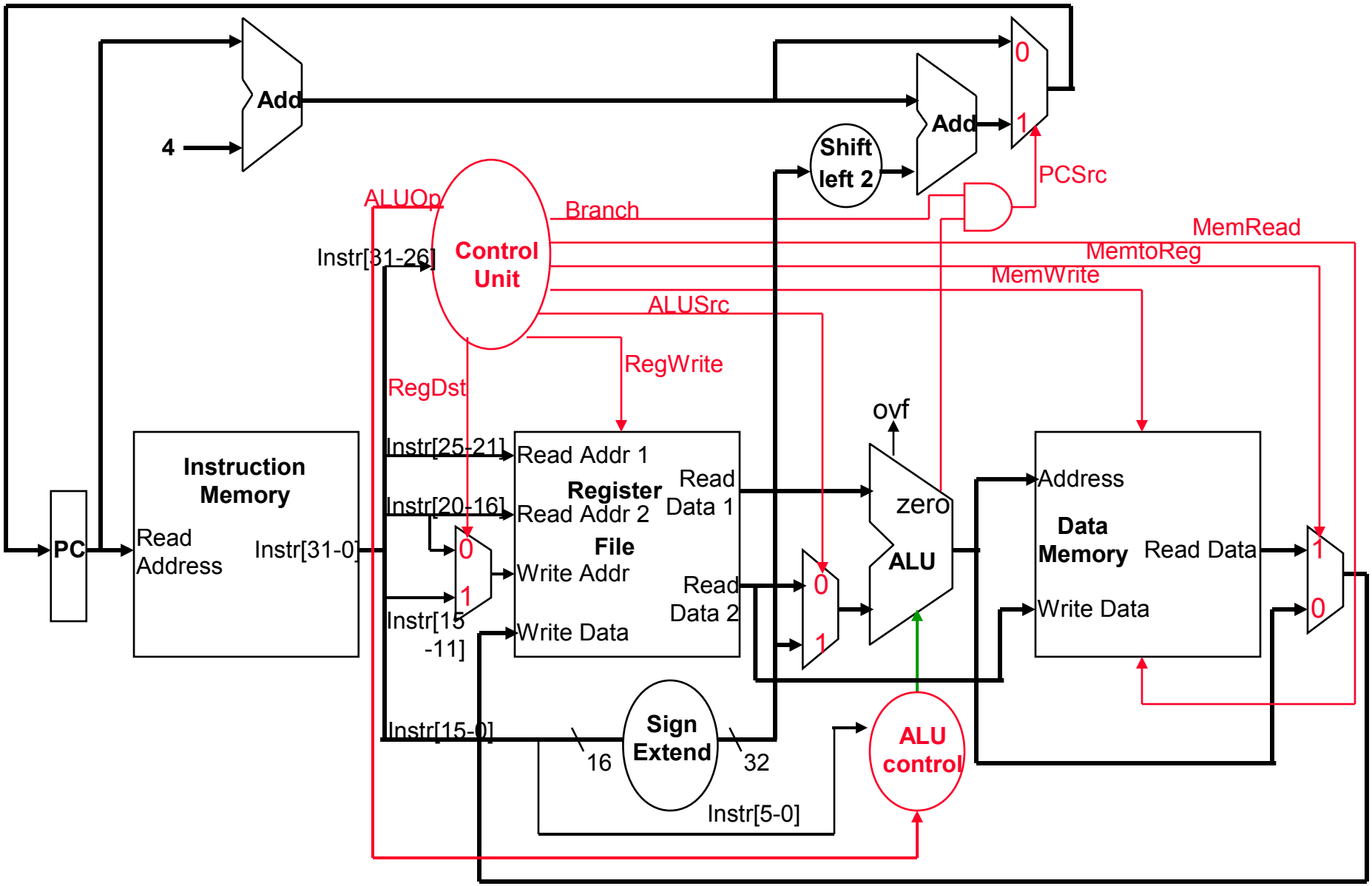
Load Word Instruction Data/Control Flow



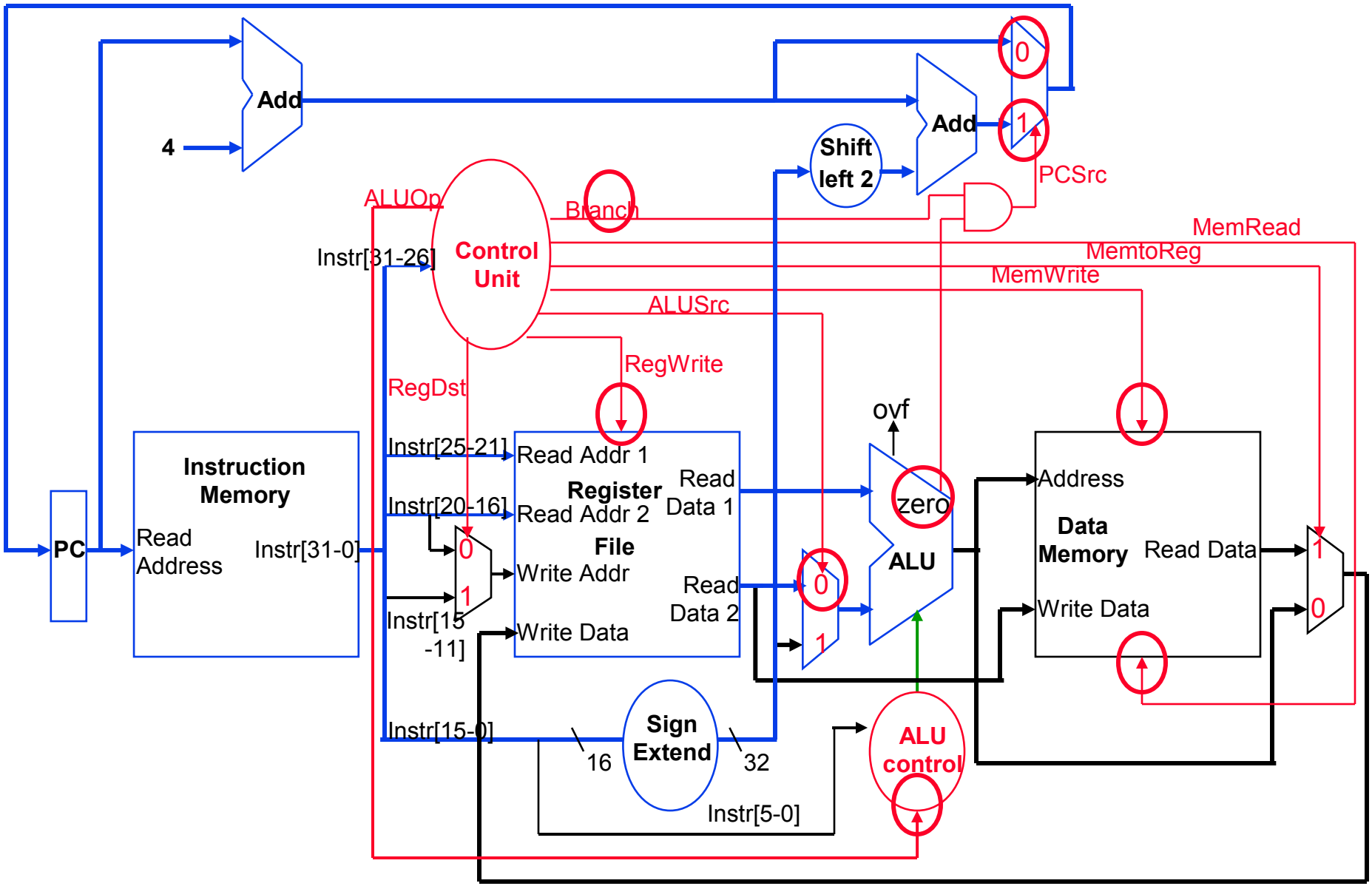
Load Word Instruction Data/Control Flow



Branch Instruction Data/Control Flow



Branch Instruction Data/Control Flow



Unité de contrôle

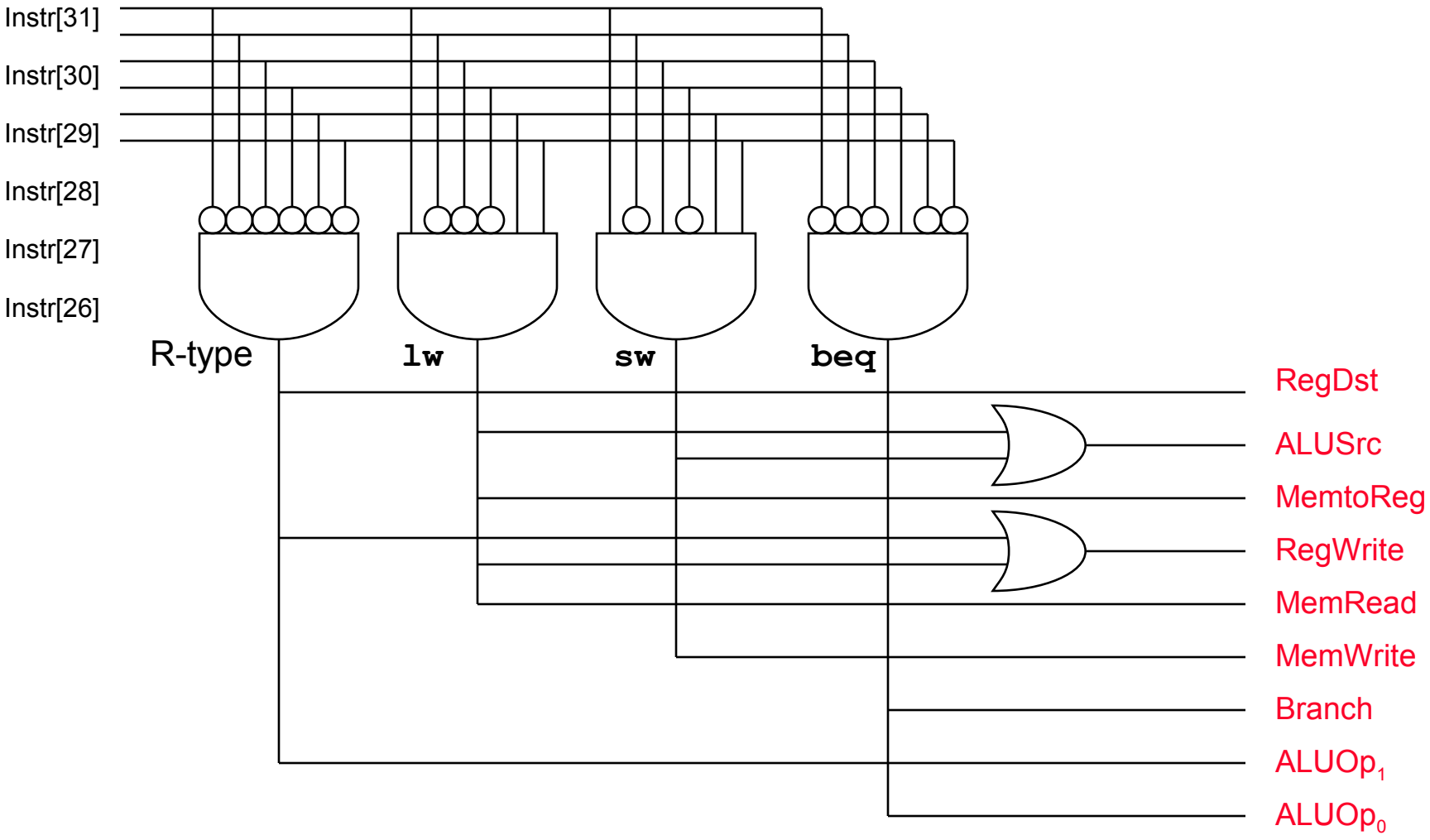
Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000								
lw 100011								
sw 101011								
beq 000100								

- entrée : uniquement code op de l'instruction
- sorties : les différents signaux pour les circuits

Unité de contrôle : table de vérité

Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp
R-type 000000	1	0	0	1	0	0	0	10
lw 100011	0	1	1	1	1	0	0	00
sw 101011	X	1	X	0	0	1	0	00
beq 000100	X	0	X	0	0	0	1	01

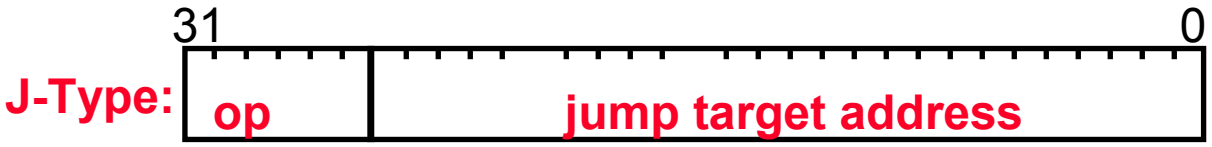
Unité de contrôle : table de vérité

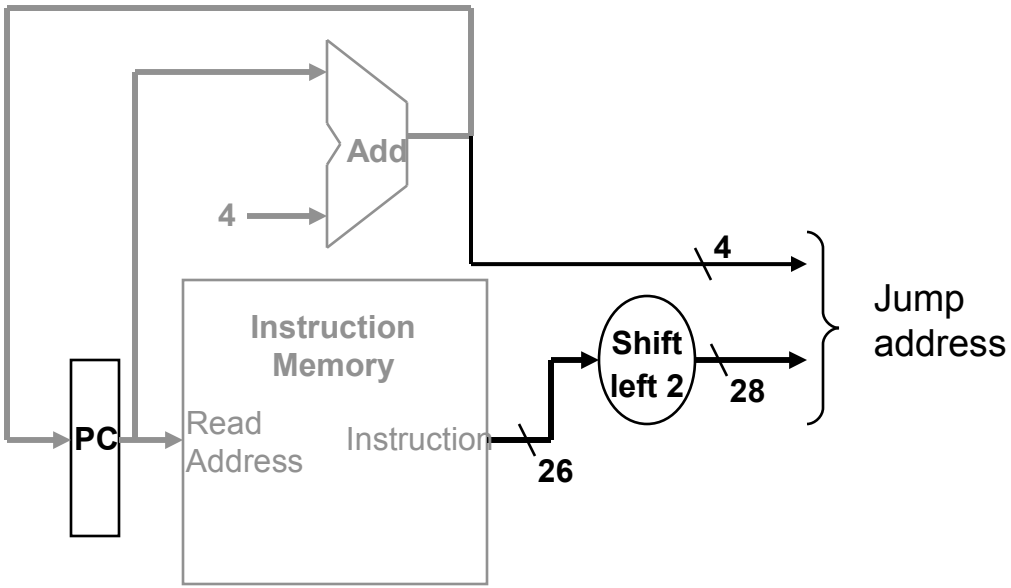


Ajoutons les instructions de saut

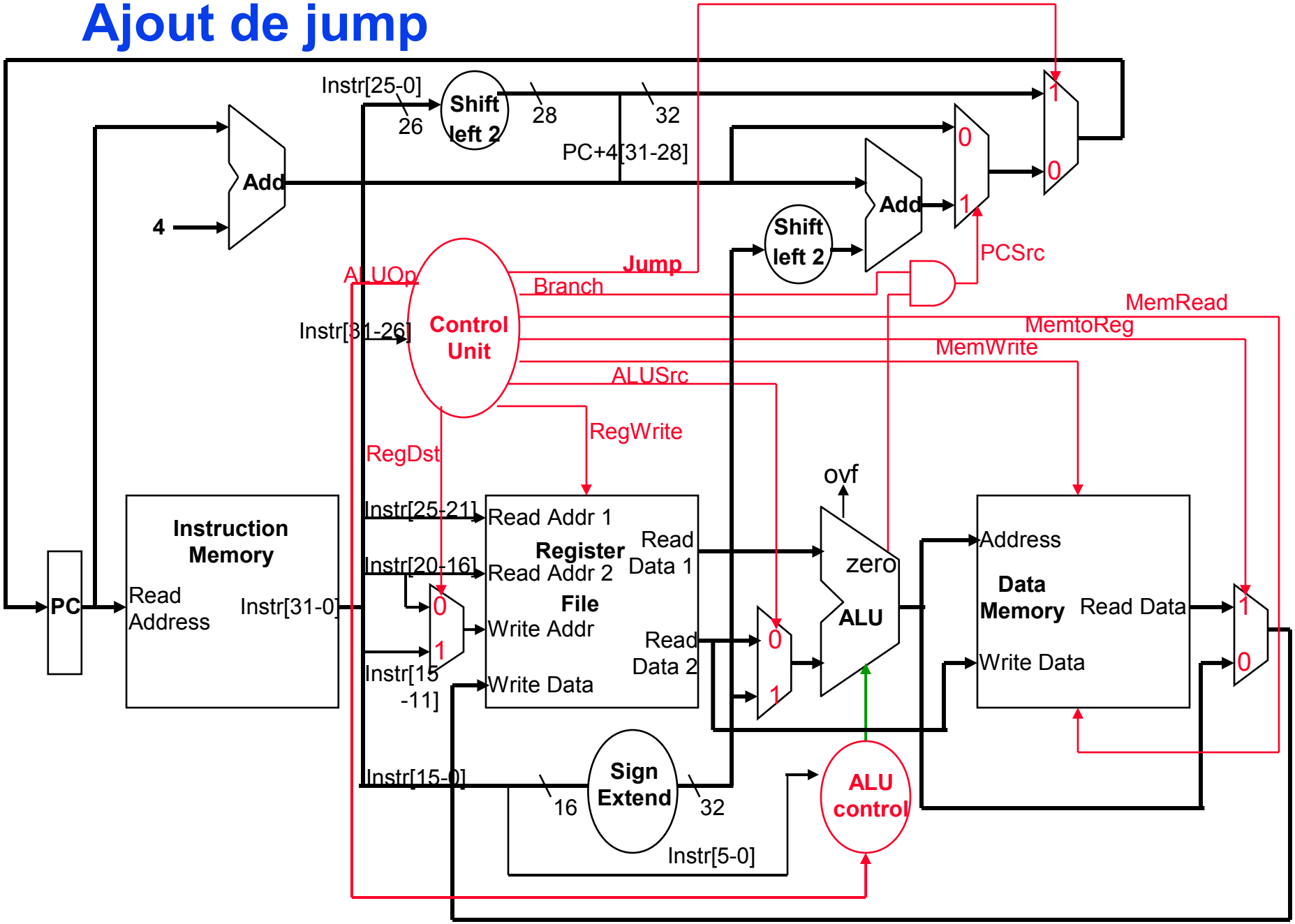
- Jump :

  remplace les 28 bits de poids faible de PC par les 26 bits indiqués dans l'adresse décalée de 2 bits à gauche

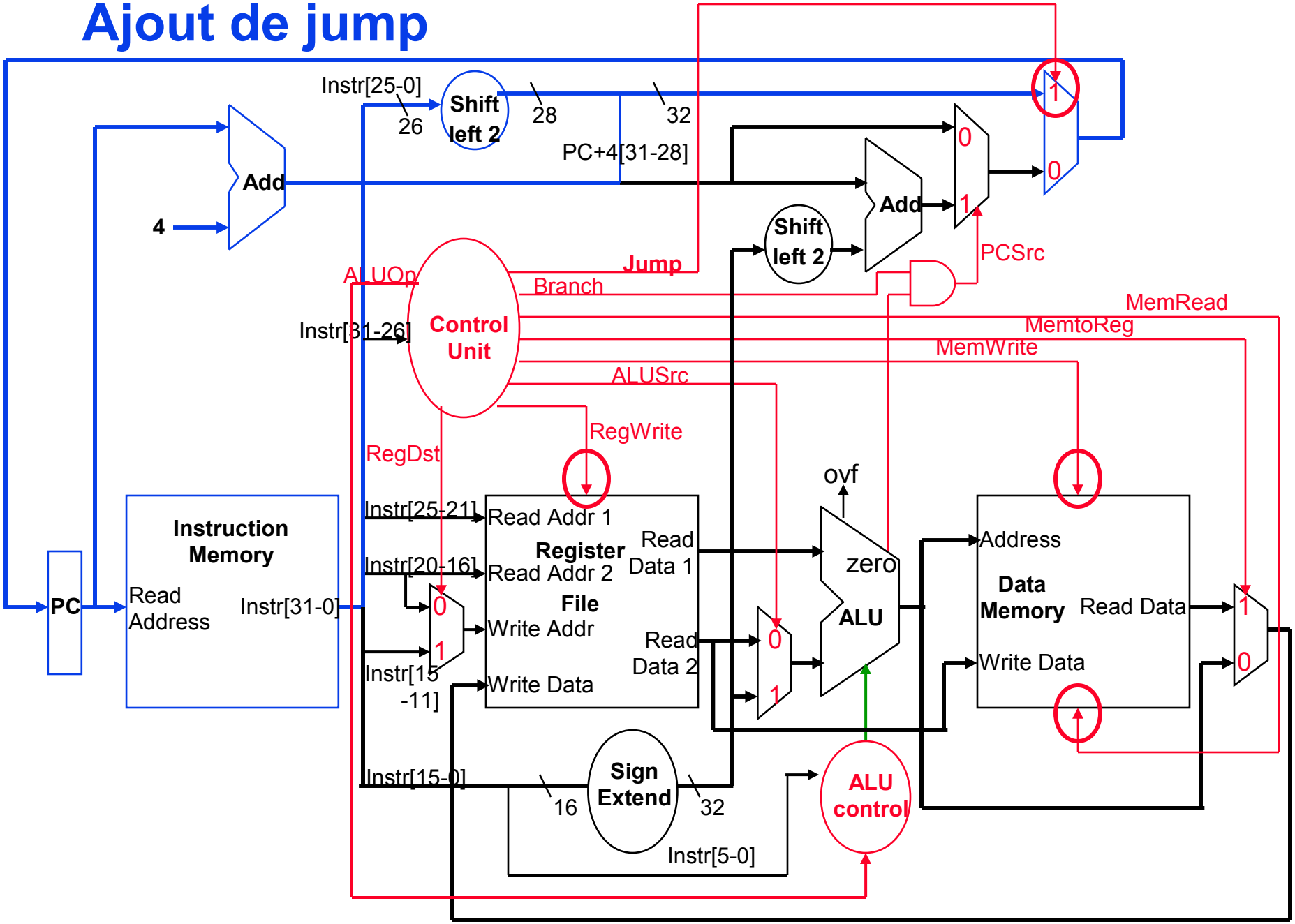




Ajout de jump



Ajout de jump



Unité de contrôle

Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp	Jump
R-type 000000	1	0	0	1	0	0	0	10	0
lw 100011	0	1	1	1	1	0	0	00	0
sw 101011	X	1	X	0	0	1	0	00	0
beq 000100	X	0	X	0	0	0	1	01	0
j 000010	X	X	X	0	0	0	X	X	1

Utilisation du chemin de données

Les instructions n'utilisent pas toutes les mêmes parties du chemin de données :

- format R / branch : pas d'accès à la mémoire de données
- store : pas d'écriture de registre
- toutes les instructions ne stabilisent pas leurs circuits utiles dans le même temps => gain de temps grâce à une exécution multicycles (prochain chapitre)