Chapitre 3 - Les instructions machine

3.1 - Format des instructions

3.1 - Format des instructions

Langage machine

Langage directement interprétable/exécutable par le processeur.
Chaque instruction est un mot binaire (sur 8 bits, 16 bits, 32 bits, 64 bits) :

- stockage en mémoire
- notion de format de l'instruction
- deux parties :
  - code opération (opcode)
  - indications complémentaires (opérandes, numéro de registres, valeurs)
- possibilité d'utiliser plusieurs mots (opérandes longues)

Chaque processeur a son propre langage machine.

Langage machine : exemple

adresse	instruction
0x2A68	0x08FA
0x2A6A	0x2028
0x2A6C	0x00F0
0x2A6E	0x07A1
0x2A70	0x00F0
0x2A72	0x18ED
0x2A74	0x1101

3.1 - Format des instructions

Jeu d'instructions

Ensemble des opérations élémentaires d'un processeur.

Permet de constituer les programmes en langage machine.

CISC/RISC

Il existe principalement deux approches pour définir le jeu d'instruction d'un processeur (et donc pour définir un processeur) :

- RISC : reduced instruction set computer
- CISC : complex instruction set computer

3.1 - Format des instructions

RISC

- nombre limité d'instructions
- instructions simples
- format réduit et fixé
- programmes "volumineux"
- processeur "simple"
- optimisation et pipeline plus aisés

CISC

- jeu d'instructions de grande taille
- instructions simples et complexes
- format variable (éventuellement plusieurs mots)
- programmes plus "compact"
- processeur "complexe"
- un peu plus proche des langages de haut niveau
- exécution des instructions peut être longue

3.2 - Langage d'assemblage

Assembleur

Langage de programmation.

Version (humainement) lisible du langage machine.

Utilise des mnémoniques plutôt que des codes opérations.

Par extension, un assembleur est un programme qui traduit "mot à mot" du code assembleur (textuel) en code machine.

Exemple : assembleur MIPS

Assembleur des processeurs MIPS (SGI jusqu'en 2006, systèmes embarqués type routeurs ou PSP, box ADSL, processeurs actuellement développés en Chine et Russie basés sur architecture MIPS).

Exemple typique d'assembleur RISC.

Processeur MIPS 32 bits

- instructions 32 bits : opcode 6 bits (64 opérations)
- 32 registres (utilisables) de 32 bits
- RAM adressable de 2^{32} octets
- i.e. 2^{30} mots de 32 bits (adresses multiples de 4)
- bus d'adresse de 32 bits
- bus de données de 32 bits

Processeur MIPS 32 bits

Registres

Numéro	Désignation assembleur	Rôle
0	`$zero`	constante nulle
1	`$at`	réservé (assembler temporary)
2-3	`$v0,$v1`	résultat évaluation (values)
4-7	`$a0,...,$a3`	arguments
8-15	`$t0,...,$t7`	résultats temporaires
16-23	`$s0,...,$s7`	résultats sauvegardés (cf appels)
24-25	`$t8,$t9`	résultats temporaires
26-27	`$k0,$k1`	réservé interruptions
28	`$gp`	pointeur global (milieu mémoire)
29	`$sp`	pointeur de pile
30	`$s8,$fp`	pointeur de bloc
31	`$ra`	adresse de retour

Assembleur MIPS : addition/soustraction

Opérations à trois opérandes.

Instructions de type :

```
add $x,$y,$z
sub $x,$y,$z
```

correspondant à $x = y + z$ ou $x = y - z$, où x, y, z désignent les contenus de `$x`, `$y`, `$z`.

Exemples

```
add $t0,$s4,$s7
add $s1,$t0,$s2
sub $s3,$s1,$s5
```

Remarque : les opérations logiques `and`, `or`, `xor` fonctionnent sur le même principe.

Assembleur MIPS : addition/soustraction avec valeur immédiate

Opérations à trois opérandes.

Instructions de type :

```
addi $x,$y,val
subi $x,$y,val
```

correspondant à $x = y + val$ ou $x = y - val$, où x et y désignent les contenus de `$x`, `$y`.

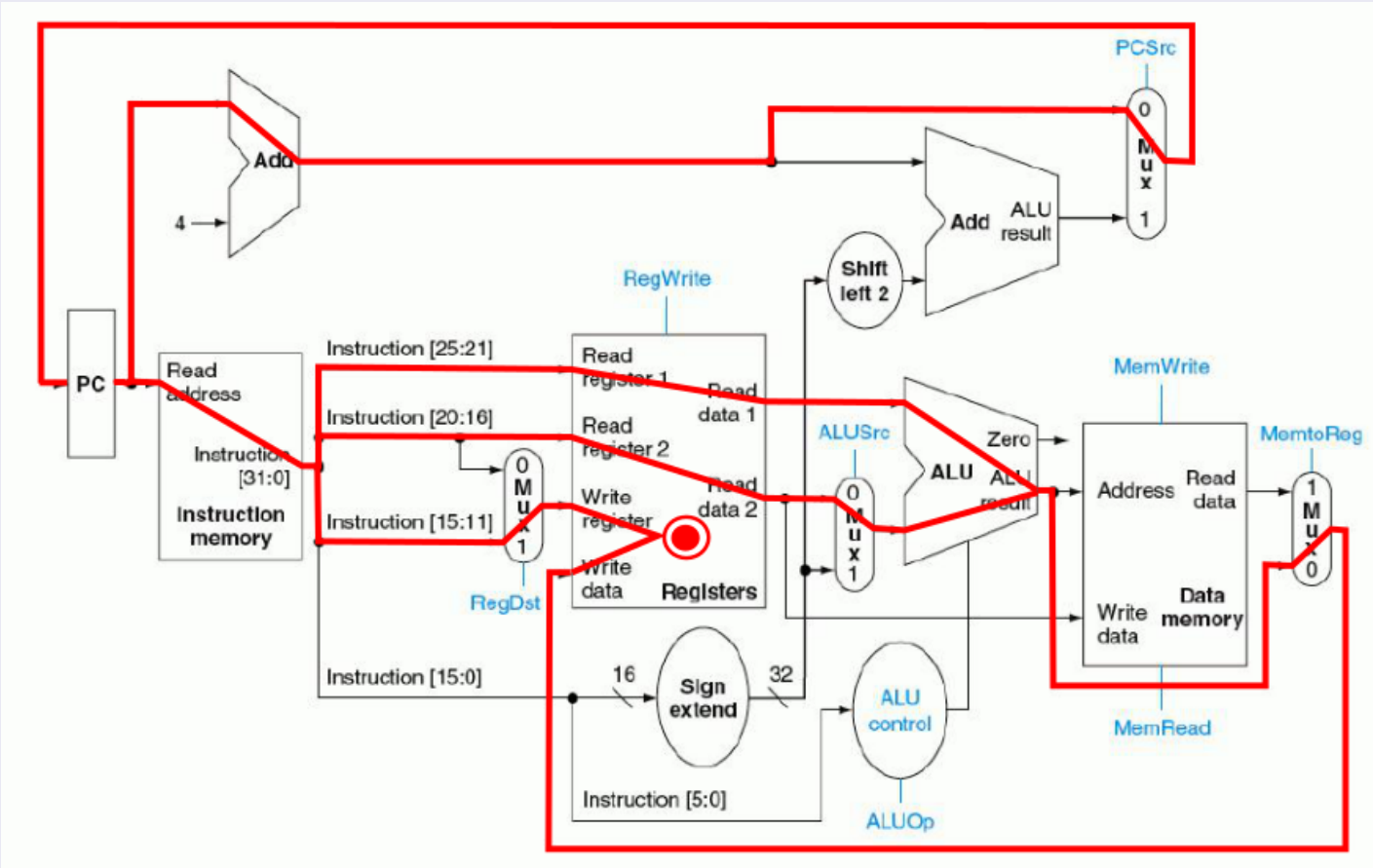
Exemples

```
addi $t0,$s4,-1234
addi $s1,$t0,0x541
subi $s3,$s1,6
```

Remarque : les opérations logiques `andi`, `ori`, `xori` fonctionnent sur le même principe.

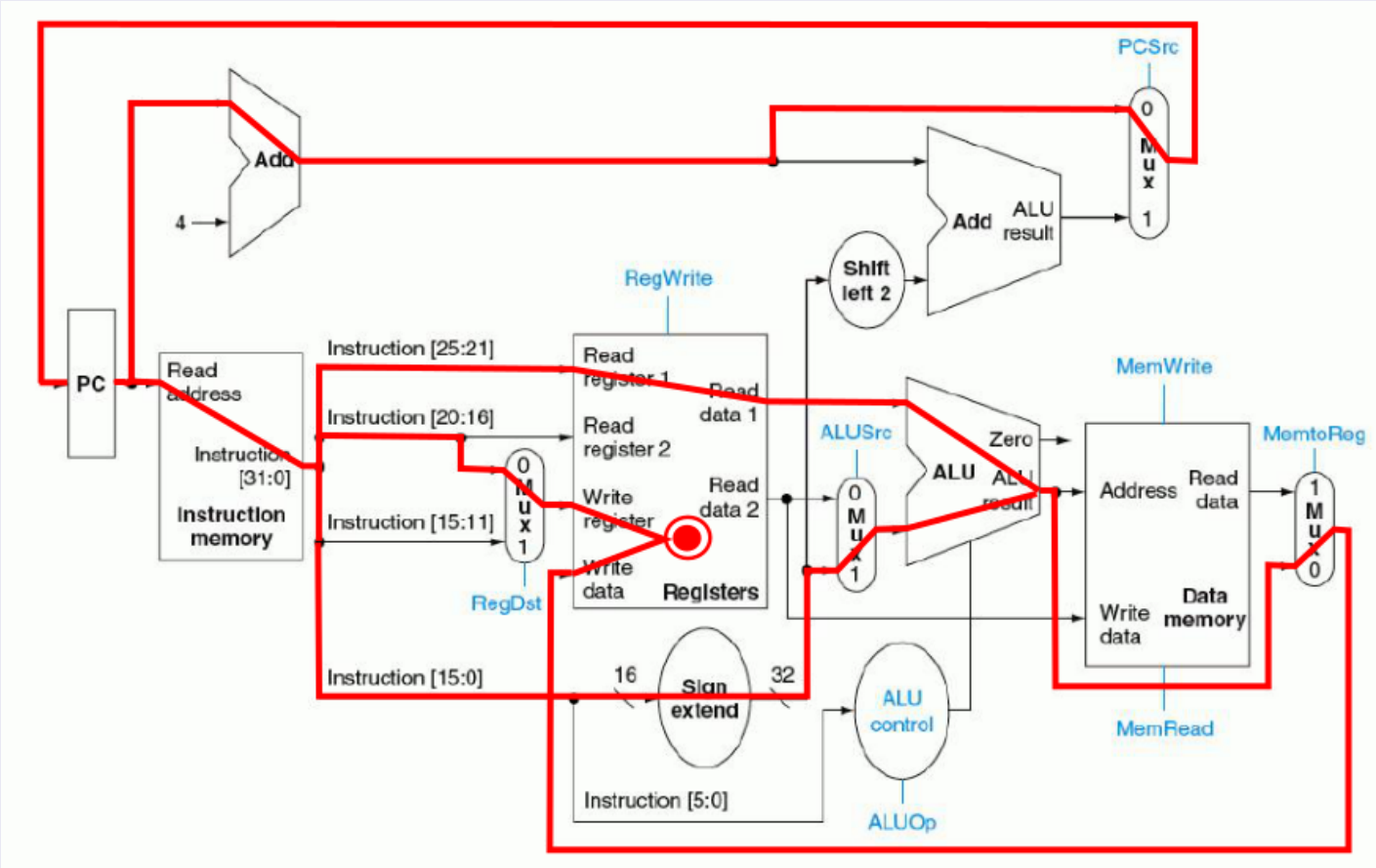
Assembleur MIPS : addition

Rappel :



3.2 - Langage d'assemblage

Assembleur MIPS : addition avec valeur immédiate



Assembleur MIPS : multiplication

Opération à deux opérandes.

Instructions de type :

```
mult $x,$y
```

correspondant à [Hi,Lo]$= x \times y$, où Hi et Lo sont deux registres spéciaux (accessibles via autres instructions).

Exemple

```
mult $s4,$s7
mfhi $s1
mflo $s3
```

(récupère Hi dans s_1 et Lo dans s_3)
correspond à $(s_1, s_3) = s_4 \times s_7$

Assembleur MIPS : division

Opération à deux opérandes.

Instructions de type :

```
div $x,$y
```

correspondant à [Hi,Lo]= $[x\%y, x/y]$.

Exemple

```
div $s4,$s7
mfhi $s1
mflo $s3
```

correspond à $s_1 = s_4 \% s_7$ et $s_3 = s_4 / s_7$

Assembleur MIPS : échange de registres

Opération à deux opérandes.

Instructions de type :

```
move $x,$y
```

correspondant à $x = y$.

En fait, pseudo-instruction, car compilée en `add $x,$zero,$y`

Assembleur MIPS : affectation de valeur

Opération à deux opérandes (load immediate). Adressage immédiat.
Instructions de type :

```
li $x,val
```

correspondant à x =val, où val est une valeur sur 32 bits.

En fait, pseudo-instruction, car compilée en :

```
lui $x,(val & 0xFFFF0000)>>16
ori $x,$x,val & 0x0000FFFF
```

i.e.

- initialisation des 16 bits de poids fort de x avec ceux de val (les autres sont mis à zéro)
- OU logique entre x et les 16 bits de poids faible de val (autres bits inchangés)

Assembleur MIPS : lecture mémoire

Opération à deux opérandes (load word). Adressage (pseudo-)indexé.

Instructions de type :

```
lw $x,val($y)
```

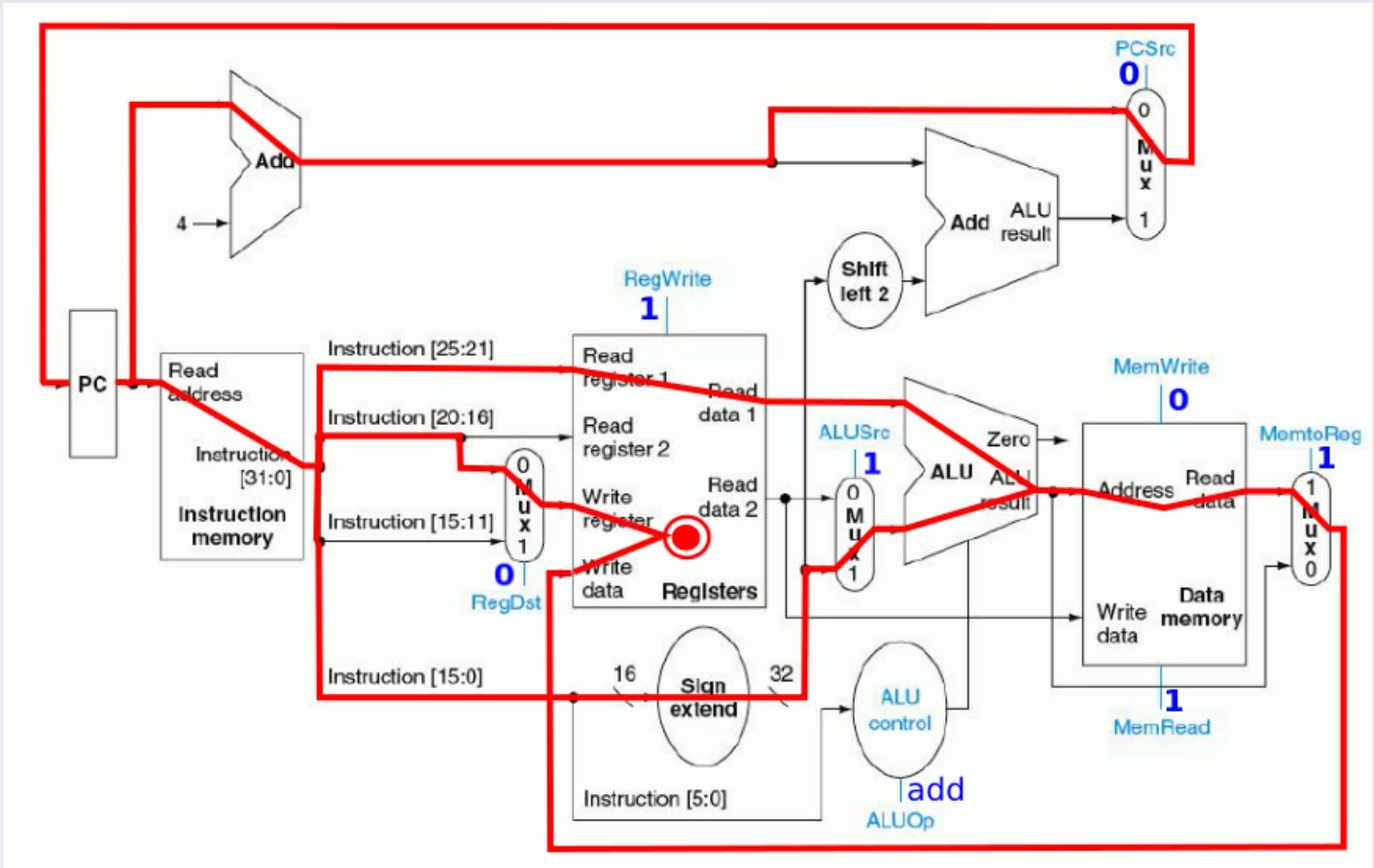
correspondant à $x = \text{MEM}[y + \text{val}]$.

Exemples

```
lw $ra,-36($sp)
lw $s5,0x4F0($s5)
```

Assembleur MIPS : lecture mémoire

Chemin de données MIPS impliquant la mémoire :



Assembleur MIPS : écriture mémoire

Opération à deux opérandes (store word). Adressage (pseudo-)indexé.

Instructions de type :

```
sw $x,val($y)
```

correspondant à MEM[y+val]= x.

Exemples

```
lw $s4,0($sp)
lw $s0,0x4F0($s7)
```

Assembleur MIPS : branchement conditionnel

Opérations à trois opérandes (branch not equal/branch equal).
Adressage relatif.

Instructions de type :

`bne $x,$y,val`

si $x \neq y$ *alors prochaine instruction est à l'adresse* $PC+4+val*4$

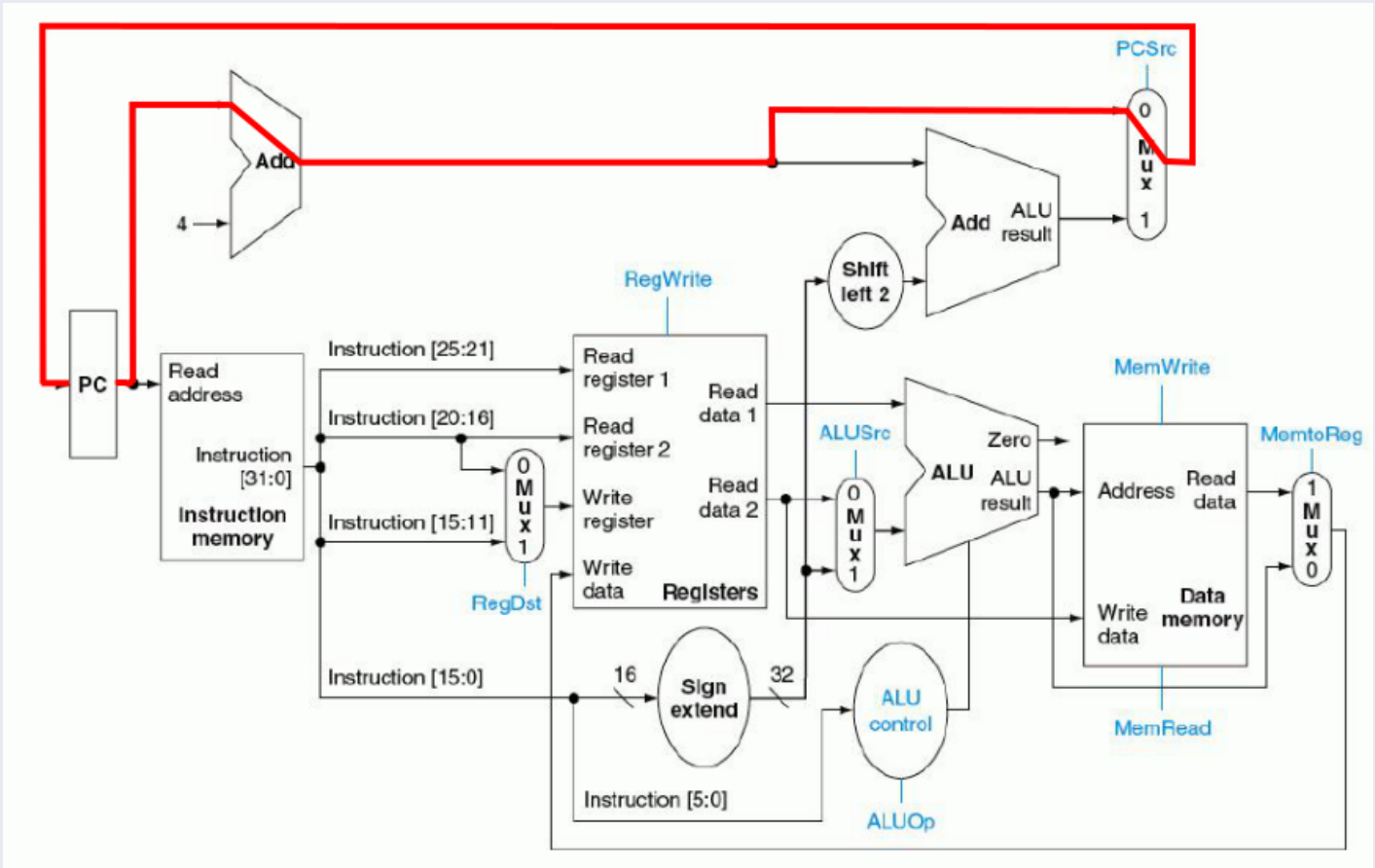
`beq $x,$y,val`

si $x = y$ *alors prochaine instruction est à l'adresse* $PC+4+val*4$

Assembleur MIPS : branchement conditionnel

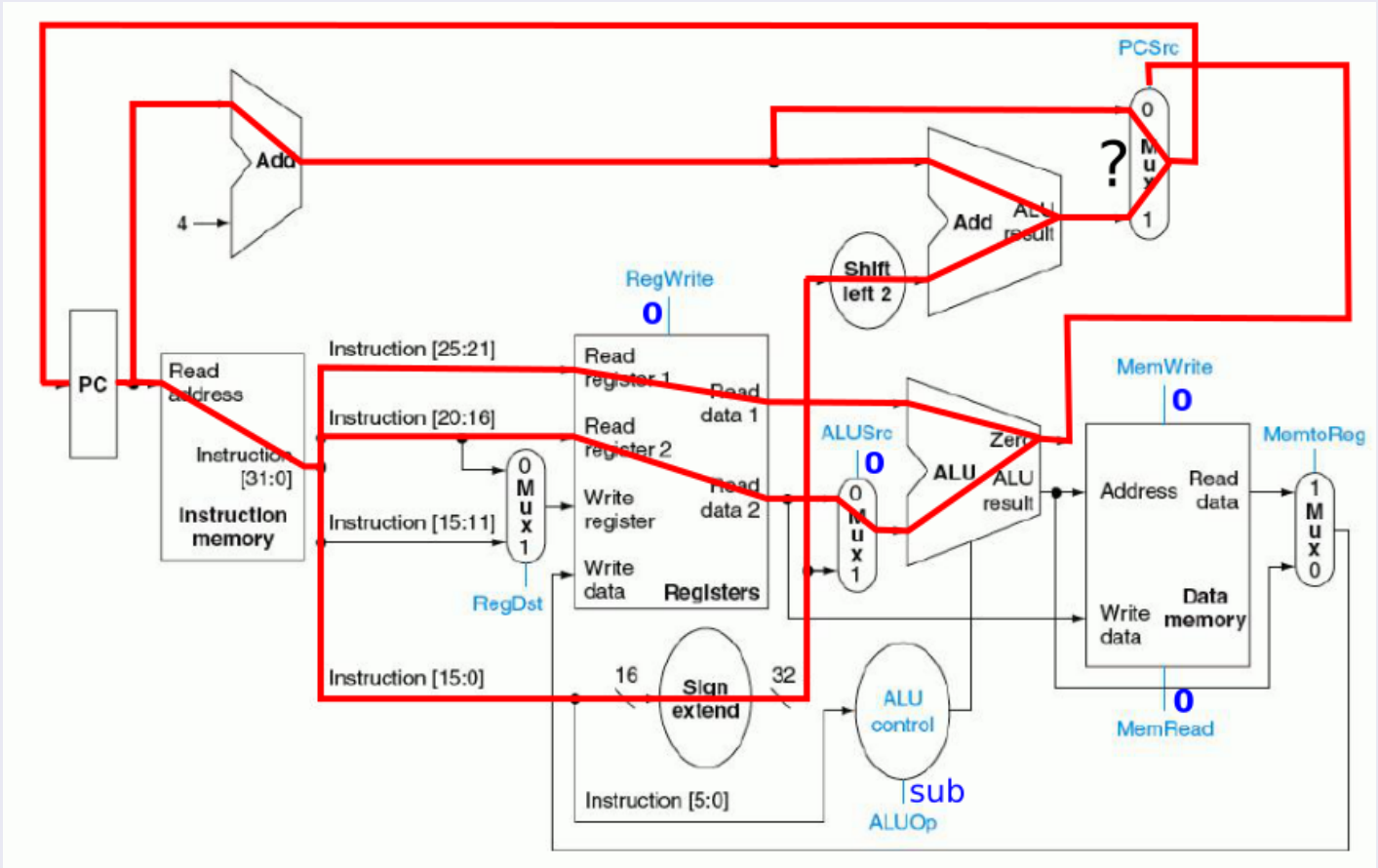
Pourquoi PC+4+val∗4 ?

Rappel :



Assembleur MIPS : branchement conditionnel

Chemin de données avec branchement :



Facilité de l'assembleur : utilisation d'étiquettes. Le compilateur détermine les adresses des instructions et les décalages induits.

Exemple de branchement conditionnel

```
        beq $s1,$s6,Suite
        sub $s1,$s1,$s6
Suite : mult $s2,$s1
```

correspond à

si $s_1 \neq s_6$ *alors* $s_1 = s_1 - s_6$ *fsi*

Hi,Lo= $s_2 * s_1$

Assembleur MIPS : branchement inconditionnel

Opérations à une seule opérande (jump). Adressage absolu (ou registre). Possibilité d'utiliser des étiquettes.

Instructions de type :

`j val`

prochaine instruction à l'adresse val∗4

`jr $ra`

prochaine instruction à l'adresse contenue dans le registre indiqué

`jal val`

prochaine instruction à l'adresse val∗4, et `$ra`$=PC+4$.

Exemple de branchement inconditionnel

```
        beq $s1,$s6,Suite
        sub $s1,$s1,$s6
        j Fin
Suite : add $s1,$s6,$s2
Fin :   add $s2,$s2,$s1
```

correspond à

si $s_1 \neq s_6$ *alors* $s_1 = s_1 - s_6$
sinon $s_1 = s_6 + s_2$ *fsi*
$s_2 = s_2 + s_1$

Autre exemple

```
        li $t2,0
        li $t3,1
Cond :  beq $t1,$zero,Fin
        add $t2,$t1,$t2
        sub $t1,$t1,$t3
        j Cond
Fin :
```

correspond à

$t_2 = 0$

tant que $(t_1 \neq 0)$ *faire*

$t_2 = t_2 + t_1$

$t_1 = t_1 - t_3$

ftq

Exemple d'appel de sous-programme

```
        ...
        jal Proc
        add $s1,$a1,$s2
        ...
        ...
Proc :  ...
        ...
        sub $t1,$t1,$t3
        jr $ra
```

Gestion des appels de sous-programmes

Un seul registre `$ra` : pas d'appels imbriqués a priori, mais ...

- Les registres (pas tous) sont sauvegardés en mémoire, dans une pile.
- Après le retour, les valeurs des registres sont restaurées à partir de la mémoire.
- Utilisation des registres `$a0, ..., $a3` (non sauvegardés) pour effectuer le passage de paramètres lors de l'appel, et les résultats retournés ensuite.

Pile d'appels

- Principe : un sous-programme doit rendre la pile dans l'état où elle était au moment de son appel.
- La sauvegarde des registres peut être effectuée par le programme appelant ou par le sous-programme appelé, mais ce choix doit être fixé et identique pour tous les appels.
- Pile : action d'empiler (au sommet) une nouvelle valeur, action de dépiler.
- Ces actions sont réalisés grâce aux instructions `lw`,`sw` et grâce au registre `$sp`.

Exemple (sauvegardes effectuées par le programme appelant)

```
...
sw $s0,0($sp)
sw $s1,-4($sp)
sw $ra,-8($sp) # sauvegarde adresse retour
...
addi $sp,$sp,-12 # ajustement sommet pile
jal Sub
...
lw $ra,4($sp) # restauration adresse retour
lw $s1,8($sp)
lw $s0,12($sp)
addi $sp,$sp,12 # ajustement sommet pile
...
jr $ra
...
```

Autres instructions

cf par exemple

`http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html`

Format des instructions MIPS

Trois formats (chaque opcode associé à un format).

- Format I : opcode (6 bits), num. registre source (5 bits), num. registre destination (5 bits), valeur/adresse (16 bits)
- adresse sur 16 bits : adresse = PC + valeur$*$4
- Format J : opcode (6 bits), adresse (26 bits)
- adresse sur 26 bits : on rajoute en poids fort les 4 bits de poids fort de PC, et 00 en poids faible (adresses multiples de 4)
- Format R : opcode (6 bits), num. registre source 1 (5 bits), num. registre source 2 (5 bits), num. registre destination (5 bits), nombre de décalages (5 bits), identificateur fonction (6 bits)

Format des instructions MIPS

- Format I :

```
lui $s1,100 # [15,0,1,100]
lw $s1,100($s2) # [35,2,1,100]
beq $s1,$s2,100 # [4,1,2,100]
```

- Format J :

```
j 1000 # [2,1000]
```

- Format R :

```
add $s1,$s2,$s3 # [0,2,3,1,0,32]
jr $ra # [0,31,0,0,0,8]
```

Exemple de programme complet.

Elements notables

- commentaires
- .data : allocation mémoire de données
- .data : étiquettes (sortes de noms de variables) désignent les adresses des zones allouées
- .text : programme
- syscall : appels système (affichage, sortie, etc.)

3.2 - Langage d'assemblage

```
.data
        data:   .word     0 : 256
        # storage for 16x16 matrix of words
.text
        jal     by_rows         # no register to save
        li      $v0, 10         # system service 10 is exit
        syscall
by_rows: li     $t0, 16         # number of rows
        li      $t1, 16         # number of columns
        move    $s0, $zero      # row counter
        move    $s1, $zero      # column counter
        move    $t2, $zero      # value to be stored
#  Each loop iteration will store computed value
#  from incremented $t2 value into next matrix element
loop:   mult    $s0, $t1
        mflo    $s2
        add     $s2, $s2, $s1   # array index
        sll     $s2, $s2, 2     # shift left (logical)
                                # 2 bits for byte offset
```

```
# registers to save: ra,t0,t1,t2,s0,s1,s2
sw      $ra, 0($sp)
sw      $t0, -4($sp)
sw      $t1, -8($sp)
sw      $t2, -12($sp)
sw      $s0, -16($sp)
sw      $s1, -20($sp)
sw      $s2, -24($sp)
addi    $sp, $sp, -28
move    $a0, $t2
jal     compute
lw      $s2, 4($sp)
lw      $s1, 8($sp)
lw      $s0, 12($sp)
lw      $t2, 16($sp)
lw      $t1, 20($sp)
lw      $t0, 24($sp)
lw      $ra, 28($sp)
addi    $sp, $sp, 28
```

```
        sw      $v0, data($s2) # store result
        addi    $t2, $t2, 1
        addi    $s1, $s1, 1    # next column
        bne     $s1, $t1, loop # not at end of row: loop back
        move    $s1, $zero     # reset column counter
        addi    $s0, $s0, 1    # next row
        bne     $s0, $t0, loop # not at end of matrix: loop back
        jr      $ra
compute: mult   $a0, $a0
        mflo    $t0
        add     $t1, $t0, $a0
        addi    $v0, $t1, -3
        jr      $ra
```

Equivaut à :

```
int main()  {
     by_rows();
}
void by_rows() {
     int size = 16;
     int[size][size] data;
     int value = 0;
     for (int row = 0; col < size; row++) {
        for (int col = 0; col < size; col++) }
           data[row][col] = compute(value);
           value++;
        }
     }
}
int compute(int v) {
     return v*v + v - 3;
}
```

3.2 - Langage d'assemblage

/Users/girau/Desktop/HomeBUL/CoursUHP/CoursUHP_airbg/2018_2019/Archi/mips1.asm - MARS 4.4

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Edit Execute

mips1.asm

```
1  .data
2          data:  .word     0 : 256
3          # storage for 16x16 matrix of words
4  .text
5          jal    by_rows        # no register to save
6          li     $v0, 10        # system service 10 is exit
7          syscall
8  by_rows: li     $t0, 16        # number of rows
9          li     $t1, 16        # number of columns
10         move   $s0, $zero     # row counter
11         move   $s1, $zero     # column counter
12         move   $t2, $zero     # value to be stored
13 #  Each loop iteration will store computed value
14 #  from incremented $t2 value into next matrix element
15 loop:    mult   $s0, $t1
16         mflo   $s2
17         add    $s2, $s2, $s1  # array index
18         sll    $s2, $s2, 2    # shift left (logical)
19                                # 2 bits for byte offset
20         # registers to save: ra,t0,t1,t2,s0,s1,s2
21         sw     $ra, 0($sp)
22         sw     $t0, -4($sp)
23         sw     $t1, -8($sp)
24         sw     $t2, -12($sp)
25         sw     $s0, -16($sp)
26         sw     $s1, -20($sp)
27         sw     $s2, -24($sp)
28         addi   $sp, $sp, -28
29         move   $a0, $t2
30         jal    compute
31         lw     $s2, 4($sp)
32         lw     $s1, 8($sp)
33         lw     $s0, 12($sp)
34         lw     $t2, 16($sp)
35         lw     $t1, 20($sp)
36         lw     $t0, 24($sp)
37         lw     $ra, 28($sp)
38         addi   $sp, $sp, 28
39         sw     $v0, data($s2) # store result
40         addi   $t2, $t2, 1
41         addi   $s1, $s1, 1    # next column
42         bne    $s1, $t1, loop # not at end of row: loop back
43         move   $s1, $zero     # reset column counter
44         addi   $s0, $s0, 1    # next row
45         bne    $s0, $t0, loop # not at end of matrix: loop back
46         jr     $ra
47 compute: mult   $a0, $a0
48         mflo   $t0
49         add    $t1, $t0, $a0
50         addi   $v0, $t1, -3
51         jr     $ra
```

Registers Coproc 1 Coproc 0

Name	Number	Value
$zero	0	0
$at	1	268502012
$v0	2	10
$v1	3	0
$a0	4	255
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	16
$t1	9	16
$t2	10	256
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	16
$s1	17	0
$s2	18	1020
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	4194308
pc		4194316
hi		0
lo		65025

Chapitre 3 - Les instructions machines
3.2 - Langage d'assemblage