Architecture des ordinateurs

Licence Informatique - Université de Provence

Jean-Marc Talbot

jtalbot@cmi.univ-mrs.fr

Langage d’assemblage

Processeur et Programmation

D'un point de vue de la programmation, le processeur offre

- un certain **jeu d'instructions** qu'il sait exécuter.
- un certain nombre de **registres** :
  - utilisable/modifiable directement par le programme : registres de travail - pointeur de segment

    registres vu par le jeu d'instructions

  - modifiable indirectement par le programme : compteur ordinal - pointeur de pile - registre d'instruction - registre d'états

    registres manipulés implicitement par le jeu d'instructions

- un certain nombre de manière d'accéder à la mémoire : **modes d'adressage**

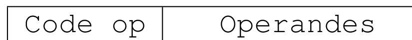
Langage machine (I)

Le **langage machine** est le langage directement interprétable par le processeur.

Le langage est défini par un ensemble d'**instructions** que le processeur exécute directement

Chaque instruction correspond à un nombre (codé selon le cas sur un octet, un mot de 16 bits, ... : le **format de l'instruction**) et se décompose en

- une partie codant l'opération à exécuter appelé **opcode** ou **code opération**
- une partie pour les opérandes



Langage machine (II)

Un programme en langage machine est une suite de mots codant opérations et opérandes

Chaque processeur possède son propre langage machine.

Jeu d'instructions

Le **jeu d'instructions** est l'ensemble des opérations élémentaires qu'un processeur peut accomplir.

Le type de jeu d'instructions d'un processeur détermine son architecture.

Deux types d'architectures

- RISC (Reduced Instruction Set Computer)

  PowerPC, MIPS, Sparc
- CISC (Complex Instruction Set Computer)

  Pentium

RISC/CISC (I)

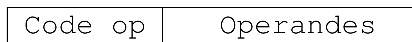
- RISC :
  - jeu d'instructions de taille limitée
  - instructions simples
  - format des instructions petit et fixé
  - modes d'adressage réduits
- CISC :
  - jeu d'instructions de taille importante
  - instructions pouvant être complexes
  - format d'instructions variables (de 1 à 5 mots)
  - modes d'adressages complexes.

RISC/CISC (II)

- CISC
  - ⊕ programmation de plus haut niveau
  - ⊕ programmation plus compacte (écriture plus rapide et plus élégante des applications)
  - ⊕ moins d'occupation en mémoire et à l'exécution
  - ⊖ complexifie le processeur
  - ⊖ taille des instructions élevée et variable : pas de structure fixe
  - ⊖ exécution des instructions : complexe et peu performante.
- RISC
  - ⊕ instructions de format standard
  - ⊕ traitement plus efficace
  - ⊕ possibilité de pipeline plus efficace
  - ⊖ programmes plus volumineux
  - ⊖ compilation plus compliquée

Modes d'adressage (I)

Les instructions du langage machine manipulent des données. Selon où ces données se trouvent, on parle de différents **modes d'adressage**.



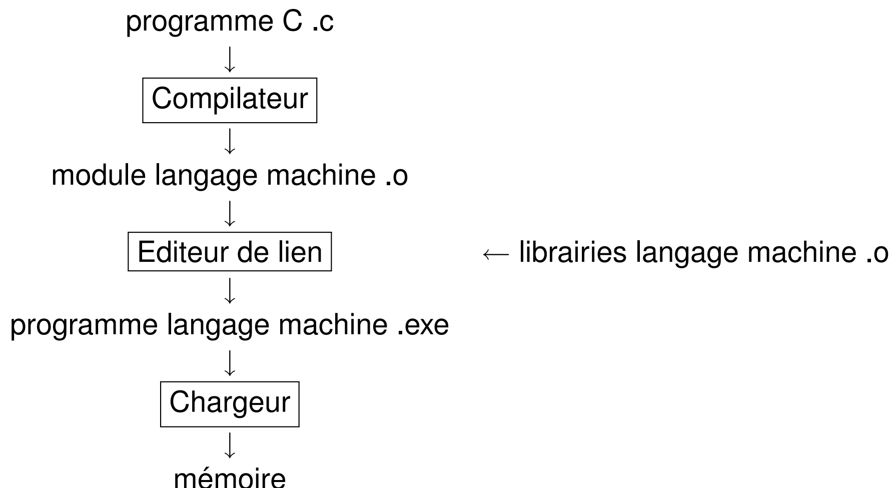
Comment interpréter `Operandes` pour trouver les données de l'instruction `Code op` ?

- *adressage implicite* : l'instruction opère sur une donnée qui se trouve à un emplacement précis et déterminé.
  - manipulation du registre d'états (Pentium)
- *adressage par registres* : `Operandes` contient le(s) numéro(s) du (des) registre(s) où se trouvent les données manipulées par l'instruction.

Modes d'adressage (II)

- *adressage direct (ou direct restreint)* : `Operandes` est l'adresse (ou un fragment de l'adresse) où se trouve la donnée en mémoire.
- *adressage relatif* : `Operandes` contient un déplacement relatif par rapport à une adresse qui se trouve dans un registre précis (par exemple, le compteur ordinal *PC*).
- *adressage indirect* : `Operandes` contient le numéro d'un registre dont le contenu est l'adresse où se trouve la donnée en mémoire.
- *adressage (indirect) indexé* : `Operandes` contient le numéro d'un registre contenant une adresse a. La donnée est en mémoire à l'adresse $a + i$, où i est le contenu d'un autre registre dans `Operandes` ou d'un registre spécifique, appellé *registre d'index*
- *adressage immédiat* : `Operandes` est la valeur utilisée par l'instruction

D’un programme de haut niveau à son exécution



Cycle d'exécution d'une instruction

1. **Récupérer (en mémoire) l'instruction à exécuter** :
   $RI \leftarrow Mémoire[PC]$
   L'instruction à exécuter est présente en mémoire à l'adresse contenue dans le compteur de programme *PC* et est placé dans le registre d'instruction *RI*.

2. **Le compteur de programme est incrémenté :** $PC \leftarrow PC + 4$
   Par défaut, la prochaine instruction à exécuter est la suivante en mémoire (sauf si l'instruction est un saut)

3. **L'instruction est décodée :** On identifie les opérations qui vont devoir être réalisées pour exécuter l'instruction

4. **L'instruction est exécutée :** elle peut modifier les registres (opérations arithmétiques - lecture en mémoire), la mémoire (écriture), le registre PC (instructions de saut)

Assembleur

Le **langage assembleur** ou assembleur est le langage de programmation.

C'est une version lisible par un humain du langage machine, obtenu en remplaçant les valeurs entières du langage machine par des **mnémoniques** (instruction du langage assembleur).

Pour un même langage machine, il peut exister différents langages assembleur : variation sur la syntaxe.

assembleur : programme qui transforme du langage assembleur en langage machine.

Assembleur MIPS

Assembleur du processeur MIPS R2000 (processeur de type RISC)

processeur MIPS :

- NEC, SGI, Sony PSP, PS2

Assembleur proche des autres assembleurs RISC

Processeur MIPS

Processeur 32 bits constitué de

- 32 registres de 32 bits
- une mémoire vive adressable de 2^{32} octets
- un compteur de programmes *PC* (Program Counter) sur 32 bits
- un registre d'instruction *RI* sur 32 bits

- le programme est stocké en mémoire
- l'adresse de l'instruction en cours d'exécution est stockée dans le registre *PC*
- l'instruction en cours d'exécution est stockée dans le registre *RI*

NB : une instruction est codée sur 32 bits.

Mémoire

Mémoire de 2^{32} octets = 2^{30} mots de 32 bits

Les mots mémoires sont adressés par des adresses qui sont des multiples de 4

- bus d'adresses de 32 bits
- bus de données de 8 bits

Registres MIPS

Les 32 registres du processeur MIPS sont :

Nom	Numéro	Description
$zero	0	constante 0
$at	1	réservé à l'assembleur
$v0,$v1	2-3	résultats d'évaluation
$a0,..,$a3	4-7	arguments de procédure
$t0,..,$t7	8-15	valeurs temporaires
$s0,..,$s7	16-23	sauvegardes
$t8,$t9	24-25	temporaires
$k0,$k1	26-27	réservé pour les interruptions
$gp	28	pointeur global
$sp	29	pointeur de pile
$fp	30	pointeur de bloc
$ra	31	adresse de retour

Arithmétique (I)

Code C	Assembleur
`A = B + C`	`add $s0, $s1, $s2`

Toutes les opérandes se trouvent dans des registres.

Le choix des registres est déterminé par le compilateur : ici, A$\mapsto$\$s0, B$\mapsto$\$s1, C$\mapsto$\$s2.

Le résultat est placé dans \$s0, ma première opérande

Arithmétique (II)

Code C	Assembleur

`A = B + C + D`
`E = F - A`
	`add $t0, $s1, $s2`
	`add $s0, $t0, $s3`
	`sub $s4, $s5, $s0`

Ici, A$\mapsto$\$s0, B$\mapsto$\$s1, C$\mapsto$\$s2, D$\mapsto$\$s3, E$\mapsto$\$s4, F$\mapsto$\$s5.

Toutes les opérations arithmétiques ont **trois opérandes**

Nécessaire car une instruction doit se coder sur un nombre borné de bits.

Pseudo-instruction `move`

Comment traduire `A=B` ?

Sachant que A$\mapsto$\$s0, B$\mapsto$\$s1 et que le registre \$0 vaut toujours 0 on peut écrire :

```
add $s0, $0, $s1
```

Il vaut mieux utiliser l'instruction `move` :

Code C	Assembleur
`A = B`	`move $s0, $s1`

`move` est une pseudo-instruction : sa traduction en langage machine est celle de `add $s0, $0, $s1`.

Pseudo-instruction `li`

- `li r, imm` (load immediate) charge la valeur `imm` (sur 32 bits) dans le registre `r`.

est assemblé comme

```
lui r, immh
ori r, imml
```

où

- `immh`,`imml` sont respectivement les 16 bits de poids fort et de poids faible de `imm`
- `ori r, imml` : réalise un "ou logique" entre les 16 bits de `imml` (étendus à 32 bits en mettant ceux de poids fort à 0) et le contenu de `r`, le résultat étant placé dans `r`

Lecture-Ecriture dans la mémoire principale

Les deux instructions `lw` (load word = lecture) et `sw` (store word = écriture) permettent les échanges entre la mémoire centrale et les registres.

syntaxe

`lw $2, 10($3)` copie dans le registre $2 la valeur située dans la mémoire principale à l'adresse *m* obtenue en ajoutant 10 au nombre stocké dans la registre $3.

`sw $2, 15($1)` copie la valeur présente dans le registre $2 dans dans la mémoire principale à l'adresse *m* obtenue en ajoutant 15 au nombre stocké dans la registre $1.

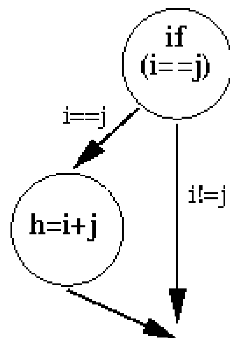
Branchements conditionnels (I)

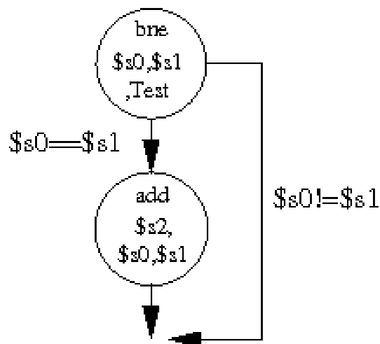
Syntaxe

`bne $t0, $t1, Label`
Si la valeur contenue dans le registre $t0 n'est pas égale à celle stockée dans le registre $1 alors la prochaine instruction à exécuter est celle placée après l'étiquette `Label`

`beq $t0, $t1, Label`
Si la valeur contenue dans le registre $t0 est égale à celle stockée dans le registre $1 alors la prochaine instruction à exécuter est celle placée après l'étiquette `Label`

Branchements conditionnels (II)





Code C	Assembleur

`if (i==j) h =i+j;`	`bne $s0, $s1, Test`
	`add $s2, $s0, $s1`
	`Test :`

Ici, i↦$s0, j↦$s1, h↦$s2.

Branchements inconditionnels (I)

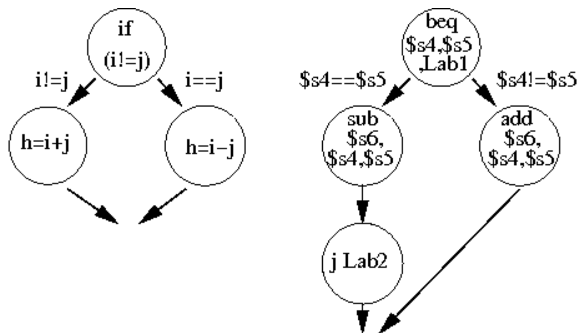
Syntaxe

`j Label`
La prochaine instruction à exécuter est celle placée après l'étiquette `Label` : $PC \leftarrow$ `Label`.

`jr registre`
La prochaine instruction à exécuter est celle à l'adresse se trouvant dans le registre `registre` : $PC \leftarrow$ `registre`.

`jal Label`
La prochaine instruction à exécuter est celle placée après l'étiquette `Label` et l'adresse de l'instruction suivant l'instruction courante (adresse de retour est stockée dans $ra : $\$ra \leftarrow PC + 4$, $PC \leftarrow$ `Label`.

Branchements inconditionnels (II)

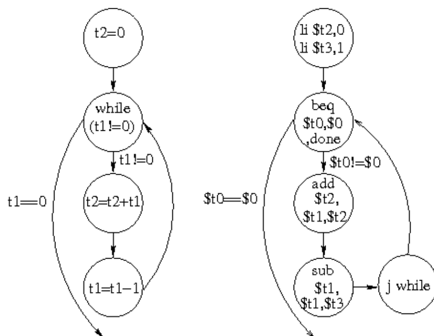


Code C	Assembleur

`if (i !=j) h =i+j`
`else h =i-j`
	`beq $s4, $s5, Lab1`
	`add $s6, $s4, $s5`
	`j Lab2`
	`Lab1 :sub $s6, $s4, $s5`
	`Lab2 :`

Ici, i$\mapsto$\$s4, j$\mapsto$\$s5, h$\mapsto$\$s6.

Branchements inconditionnels (III)

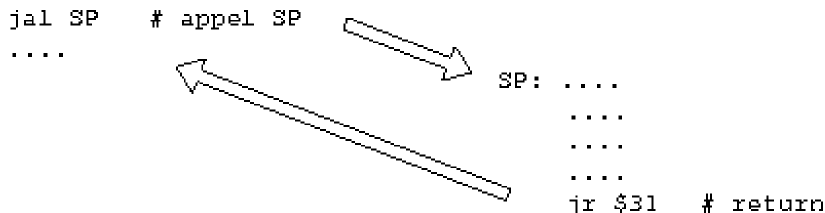


Code C	Assembleur

t2=0
while (t1 != 0){
t2 = t2 + t1
t1=t1-1 }
	li $t2, 0
	li $t3, 1
	while :beq $t1, $0, done
	add $t2, $t1, $t2
	sub $t1, $t1, $t3
	j while
	done :

Appel de sous-programmes (I)

L'instruction `jal SP` permet d'exécuter le sous-programme de label `SP`, la sauvegarde de l'adresse de retour étant réalisée par cette instruction (dans le registre \$31).



Cependant,

- Le sous-programme peut affecter les valeurs contenues dans les registres au moment de l'appel : pas de notion de variables locales et de portée/masquage de variables.
- La sauvegarde de l'adresse de retour dans un registre ne permet pas l'enchainement des appels à des sous-programmes, encore moins des sous-programmes récursifs

Appel de sous-programmes (II)

Solution :

- Sauvegarder la valeur des registres (en mémoire) de l'appelant et restaurer ces valeurs à l'issue de l'appel
- Sauvegarder l'adresses de retour du programme appelant en mémoire

On sauvegarde les (une partie des) registres en mémoire dans **une pile**.

Les registres $a0-$a3 sont ceux qui ne sont pas sauvegardés car ils contiennent lors de l'appel la valeur des paramètres effectifs et au retour les valeurs retournés par le sous-programme.

Appel de sous-programmes : pile

Une pile est une mémoire qui se manipule via deux opérations :

- push : empiler un élément (le contenu d'un registre) au sommet de la pile
- pop : dépiler un élement (et le récuperer dans un registre)

Ces deux instructions n'existent pas en assembleur MIPS, mais elles peuvent être “simulées”

- en utilisant les intructions `sw` et `lw`
- en stockant l'adresse du sommet de pile dans le registre \$sp (le pointeur de pile)

Traditionnellement, la pile croît vers les adresses les moins élevées.

Appel de sous-programmes : politique de gestion de la pile

Deux politiques de sauvegarde des registres :

- *sauvergarde par l'appelant* : le programme appelant sauvegarde tous les registres sur la pile (avant l'appel).
- *sauvergarde par l'appelé* : le programme appelant suppose que tous les registres seront préservés par le sous-programme appelé.

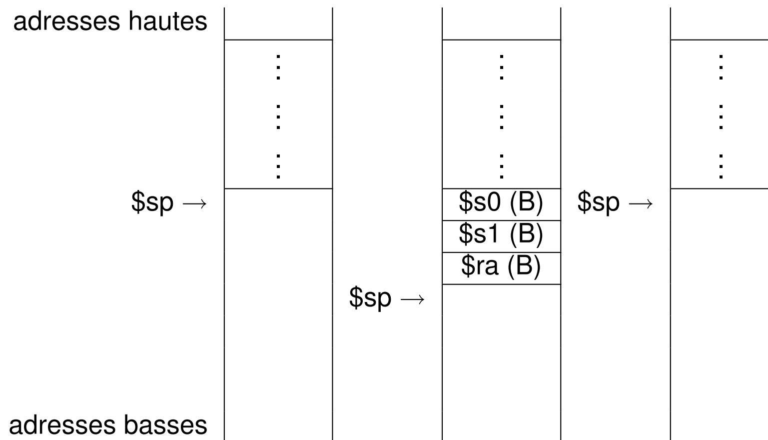
Quelque soit la politique utilisée,

un sous-programme doit rendre la pile intacte

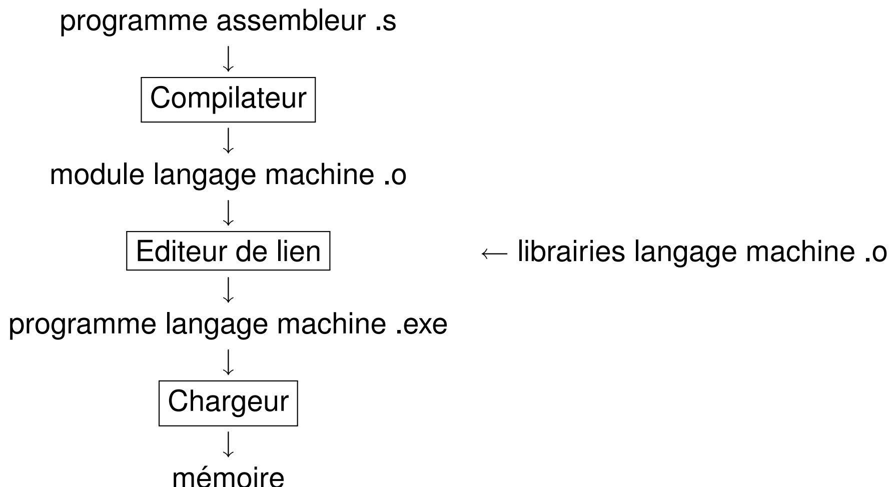
Appel de sous-programmes : exemple

```
    ...
B   ...               debut de B
    ...
    sw $s0,0($sp)     sauvegarde de $s0
    sw $s1,-4($sp)    sauvegarde de $s1
    sw $ra,-8($sp)    sauvegarde de l'adresse de retour de B
    li $t0,12
    sub $sp,$sp,12    ajustement du sommet de pile
    jal C             appel du sous-programme C
    lw $ra,4($sp)     restauration de l'adresse de retour de B
    lw $s1,8($sp)     restauration de $s1
    sw $s0,12($sp)    sauvegarde de $s0
    li $t0,12
    add $sp,$sp,12    ajustement du sommet de pile
    ...
    jr $ra
    ...               fin de B
```

Appel de sous-programmes : exemple



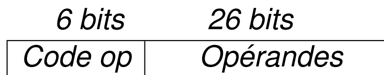
De l’assembleur à l’exécution



Codage des instructions assembleur

Format d'instructions MIPS (I)

Rappel : les instructions du langage machine MIPS sont codées sur **32 bits**



$2^6 = 64$ opérateurs possibles

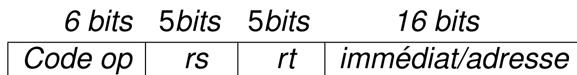
Trois formats d'instructions :

- Instructions de type immédiat (Format I)
- Instructions de type saut (Format J)
- Instructions de type registre (Format R)

Les 6 bits du Code op détermine le format de l'instruction.

Format d'instructions I (I)

Format I :



6 bits	5bits	5bits	16 bits
Code op	*rs*	*rt*	*immédiat/adresse*

- *rs* : registre source
- *rt* : registre cible / condition de branchement
- *immédiat/adresse* : opérande immédiate ou déplacement d'adresse

Format d'instructions I (II)

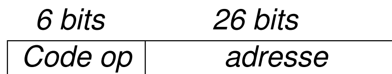
Format I :

	op	*rs*	*rt*	*16 bits*
`lui $1, 100`	15	0	1	100
`lw $1, 100($2)`	35	2	1	100
`sw $1, 100($2)`	43	2	1	100
`beq $1, $2 ,100`	4	1	2	100
`bne $1, $2 ,100`	5	1	2	100

- `lui r, imm` (load upper immediate) utilise les 16 bits de `imm` pour initialiser les 16 bits de poids fort du registre `r`, les 16 bits de poids faible étant mis à 0.

Format d'instructions J

Format J



	op	*adresse 26 bits*
`j 1000`	2	1000
`jal 1000`	3	1000

Codage des adresses

Les adresses dans les instructions ne sont **pas sur 32 bits** !

- Pour les instructions de type I : 16 bits

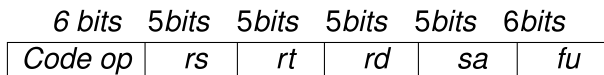
  $\Longrightarrow$ Adresse = PC + signé(16 bits) * 4 adressage relatif

- Pour les instructions de type J : 26 bits

  $\Longrightarrow$ On obtient l'adresse d'un mot mémoire (de 32 bits) en ajoutant devant les 26 bits les 4 bits de poids fort de PC (Il faut multiplier par 4 pour l'adresse d'un octet)

Format d'instructions R (I)

Format R :



6 bits	5bits	5bits	5bits	5bits	6bits
Code op	*rs*	*rt*	*rd*	*sa*	*fu*

- *rs* : registre source 1
- *rt* : registre source 2
- *rd* : registre destination
- *sa* : nombre de décalage à effectuer (shift amount)
- *fu* : identificateur de la fonction

Format d'instructions R (II)

Format R :

	op	*rs*	*rt*	*rd*	*sa*	*fu*
`add $1,$2,$3`	0	2	3	1	0	32
`sub $1,$2,$3`	0	2	3	1	0	34
`slt $1,$2,$3`	0	2	3	1	0	42
`jr $31`	0	31	0	0	0	8

- `sub $1,$2,$3` : soustrait `$3` de `$2` et place le résultat dans `$1`.
- `slt $1,$2,$3` (set less than) : met `$1` à 1 si `$2` est inférieur à `$3` et à 0 sinon.

Exemple (I)

Code C	Assembleur MIPS

`while (tab[i] == k)`
`i = i+j;`
	`Loop : mul $9, $19, $10`
	`lw $8 , Tstart($9)`
	`bne $8 , $21, Exit`
	`add $19, $19, $20`
	`j Loop`
	`Exit :`

avec `i` ↦ `$19`, `j` ↦ `$20`, `k` ↦ `$21` et `$10` est initialisé à 4.

Exemple (II)

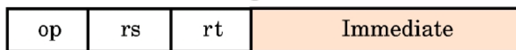
Programme chargé à l'adresse 80000 et `Tstart` vaut 1000

Adresse	Contenu					
80000	28	19	10	9	0	2
80004	35	9	8	1000		
80008	5	8	21	2		
80012	0	19	20	19	0	32
80016	5	20000				
80020	....					

car

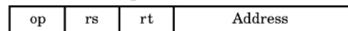
- $\overbrace{80008 + 4}^{PC} + 2 * 4$
- $20000 * 4 = 80000$

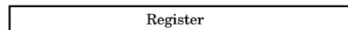
Mode d'adressages (I)

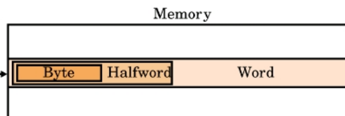



```
addi $t1, $t2, 4
```

```
addi $t1, $t2, 4
```

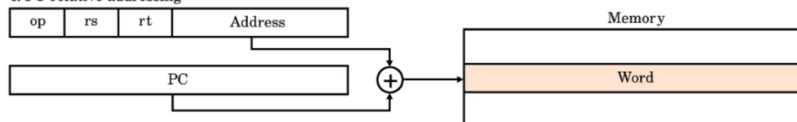




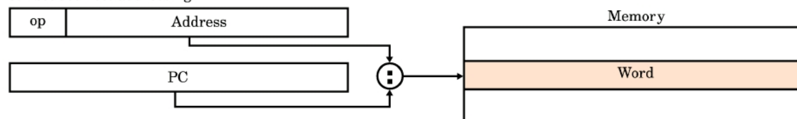



```
lw $1, 100($2)
```

Mode d'adressages (II)



est utilisé pour les branchements : `bne`, `beq`, etc.



est utilisé pour les sauts : `j`, `jal`.