



Jeu d'instructions et modes d'adressage MIPS

Vincent Risch, mai 2008, révision mai 2014

I.U.T., Aix-Marseille Université

MIPS : Introduction



- Architecture (0,3) typique d'une machine RISC

MIPS : Introduction



- Architecture (0,3) typique d'une machine RISC
- Développé fin des années 90, à partir des travaux de John L. Hennessy

MIPS : Introduction



- Architecture (0,3) typique d'une machine RISC
- Développé fin des années 90, à partir des travaux de John L. Hennessy
- Implémentations actuelles : MIPS32 et MIPS64

32 registres

Registres généraux :

Nom	Numéro de registre	Usage
\$zero	0	Dédié à la constante 0
\$v0-\$v1	2-3	Dédié à l'évaluation des expressions
\$a0-\$a3	4-7	Stockage des arguments lors des appels de fonctions
\$t0-\$t7	8-15	Registres temporaires
\$s0-\$s7	16-23	Registres sauvegardés
\$t8-\$t9	24-25	Registres temporaires supplémentaires
\$gp	28	Pointeur global
\$sp	29	Pointeur de pile
\$fp	30	Pointeur de "frame"
\$ra	31	Pointeur d'adresse retour

32 registres

Registres généraux :

Nom	Numéro de registre	Usage
\$zero	0	Dédié à la constante 0
\$v0-\$v1	2-3	Dédié à l'évaluation des expressions
\$a0-\$a3	4-7	Stockage des arguments lors des appels de fonctions
\$t0-\$t7	8-15	Registres temporaires
\$s0-\$s7	16-23	Registres sauvegardés
\$t8-\$t9	24-25	Registres temporaires supplémentaires
\$gp	28	Pointeur global
\$sp	29	Pointeur de pile
\$fp	30	Pointeur de "frame"
\$ra	31	Pointeur d'adresse retour

Registres réservés :

Nom	Numéro de registre	Usage
\$at	1	Réservé à l'assembleur
\$k0-\$k1	26-27	Réservé au système d'exploitation

Formats d'instruction



Trois formats d'instruction :

- Registre (**type R**) → instructions UAL

Formats d'instruction



Trois formats d'instruction :

- Registre (*type R*) → instructions UAL
- Immédiat (*type I*) → instructions UAL

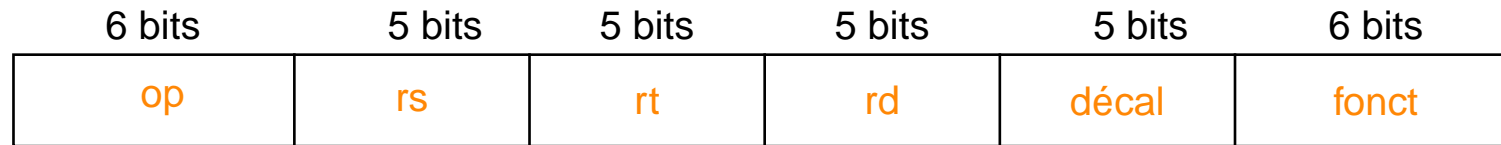
Formats d'instruction



Trois formats d'instruction :

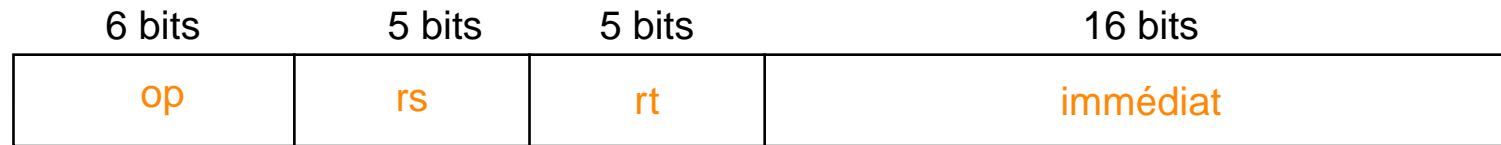
- ❑ Registre (**type R**) → instructions UAL
- ❑ Immédiat (**type I**) → instructions UAL
- ❑ Saut (**type J**) → ruptures de séquence

Format R



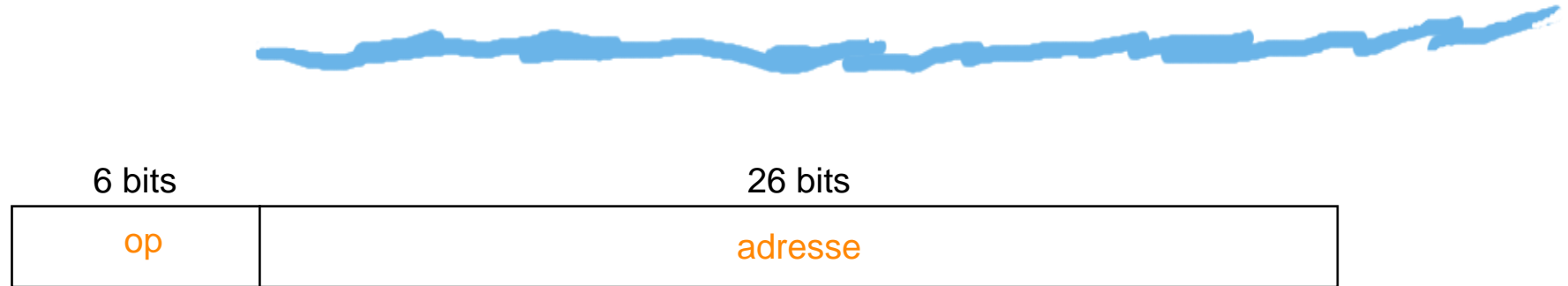
1. **op** : code opération
2. **rs** : registre–opérande source
3. **rt** : registre–opérande source
4. **rd** : registre–opérande destination
5. **décal** : décalage
6. **fonction** : sélectionne la variante de l'opération

Format I



1. **op** : code opération
2. **rs** : registre–opérande source ou destination
3. **rt** : registre–opérande source ou destination
4. **immédiate** : valeur immédiate

Format J



1. **op** : code opération
2. **adresse** : adresse de saut

Addition



1. mémorique : `add`
2. exemple : `add $s1, $s2, $s3`
3. signification : $\$s1 = \$s2 + \$s3$

Addition immédiate



1. mémorique : `addi`
2. exemple : `addi $s1, $s2, 100`
3. signification : $\$s1 = \$s2 + 100$

Soustraction

1. mémorique : `sub`
2. exemple : `sub $s1, $s2, $s3`
3. signification : $\$s1 = \$s2 - \$s3$

Opérations logiques

1. mémorique : `and`

2. exemple : `and $s1, $s2, $s3`

3. signification : $\$s1 = \$s2 \ \& \ \$s3$

1. mémorique : `or`

2. exemple : `or $s1, $s2, $s3`

3. signification : $\$s1 = \$s2 \ | \ \$s3$

1. mémorique : `nor`

2. exemple : `nor $s1, $s2, $s3`

3. signification : $\$s1 = \sim (\$s2 \ | \ \$s3)$

Décalages

1. mémorique : `sll`
2. exemple : `sll $s1, $s2, 10`
3. signification : $\$s1 = \$s2 \ll 10$
1. mémorique : `srl`
2. exemple : `srl $s1, $s2, 10`
3. signification : $\$s1 = \$s2 \gg 10$

Le champs décalage contient la valeur de décalage

Chargement

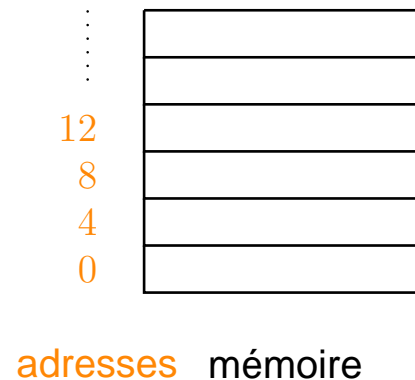
1. mémorique : `lw`
2. exemple : `lw $s1, 100($s2)`
3. signification : $\$s1 = \text{Mem}[\$s2 + 100]$

Rangement

1. mémorique : `sw`
2. exemple : `sw $s1, 100($s2)`
3. signification : $\text{Mem}[\$s2 + 100] = \$s1$

Offset mémoire

L'unité d'adressage-mémoire est l'octet. Puisque instructions et données occupent 32 bits, tous les mots-mémoire sont *alignés* sur des adresses multiples de quatre :



Exemple

Soit l'instruction C : $A[12] = h + A[8];$
avec h associé au registre $\$s2$, et l'adresse de base de A
rangée dans le registre $\$s3$.

Exemple

Soit l'instruction C : $A[12] = h + A[8]$;
avec h associé au registre $\$s2$, et l'adresse de base de A
rangée dans le registre $\$s3$.

```
lw $t0, 32($s3)    # t0 <- A[8]
add $t0, $s2, $t0  # t0 <- h + A[8]
sw $t0, 48($s3)    # A[12] <- h + A[8]
```

Comparaison

1. mémorique : `slt` → *Set on Less Than*
2. exemple : `slt $s1, $s2, $s3`
3. signification : `if ($s2 < $s3) $s1 = 1; else $s1 = 0;`

Comparaison immédiate



1. mémorique : `slti`
2. exemple : `slti $s1, $s2, 100`
3. signification : `if ($s2 < 100) $s1 = 1; else $s1 = 0;`

Branchement conditionnel (=)



1. mémorique : `beq` → *Branch on EQual*
2. exemple : `beq $s1, $s2, Etiquette`
3. signification : `if ($s1 == $s2) goto Etiquette;`

Branchement conditionnel (!=)



1. mémorique : `bne` → *Branch on Not Equal*
2. exemple : `bne $s1, $s2, Etiquette`
3. signification : `if ($s1 != $s2) goto Etiquette;`

Branchement inconditionnel



1. mémorique : `j` → *Jump*
2. exemple : `j Etiquette`
3. signification : `goto Etiquette;`

Exemple

Soit l'instruction C : `while (save[i] == k) i+=1;`
avec `i` et `k` associés au registres `$s3` et `$s5`, et l'adresse de base de `save` rangée dans le registre `$s6`.

Exemple

Soit l'instruction C : `while (save[i] == k) i+=1;`
avec `i` et `k` associés au registres `$s3` et `$s5`, et l'adresse
de base de `save` rangée dans le registre `$s6`.

```
Loop:  sll $t1, $s3, 2  # t1 <- 4*i
```

Exemple

Soit l'instruction C : `while (save[i] == k) i+=1;`
avec `i` et `k` associés au registres `$s3` et `$s5`, et l'adresse
de base de `save` rangée dans le registre `$s6`.

```
Loop:  sll $t1, $s3, 2      # t1 <- 4*i
        add $t1, $t1, $s6  # t1 <- adresse de save[i]
```

Exemple

Soit l'instruction C : `while (save[i] == k) i+=1;`
avec `i` et `k` associés au registres `$s3` et `$s5`, et l'adresse de base de `save` rangée dans le registre `$s6`.

```
Loop:  sll $t1, $s3, 2      # t1 <- 4*i
        add $t1, $t1, $s6  # t1 <- adresse de save[i]
        lw $t0, 0($t1)    # t0 <- save[i]
```

Exemple

Soit l'instruction C : `while (save[i] == k) i+=1;`
avec `i` et `k` associés au registres `$s3` et `$s5`, et l'adresse de base de `save` rangée dans le registre `$s6`.

```
Loop:  sll $t1, $s3, 2      # t1 <- 4*i
        add $t1, $t1, $s6  # t1 <- adresse de save[i]
        lw $t0, 0($t1)    # t0 <- save[i]
        bne $t0, $s5, Exit # goto Exit if save[i]!=k
```

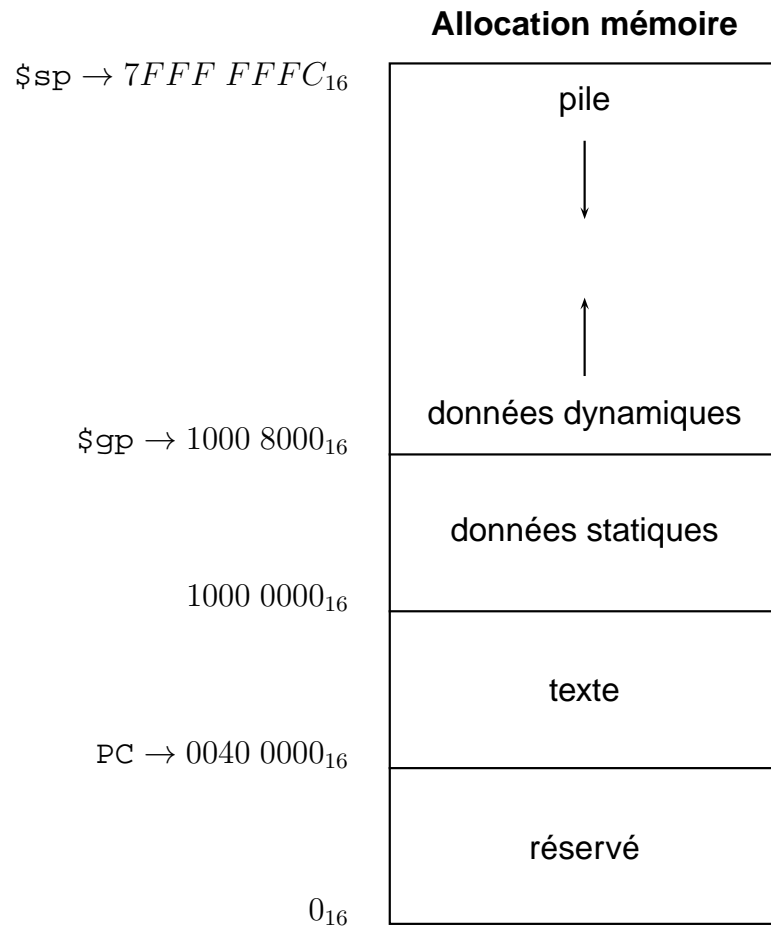
```
Exit:                                     # sortie de boucle
```


Exemple

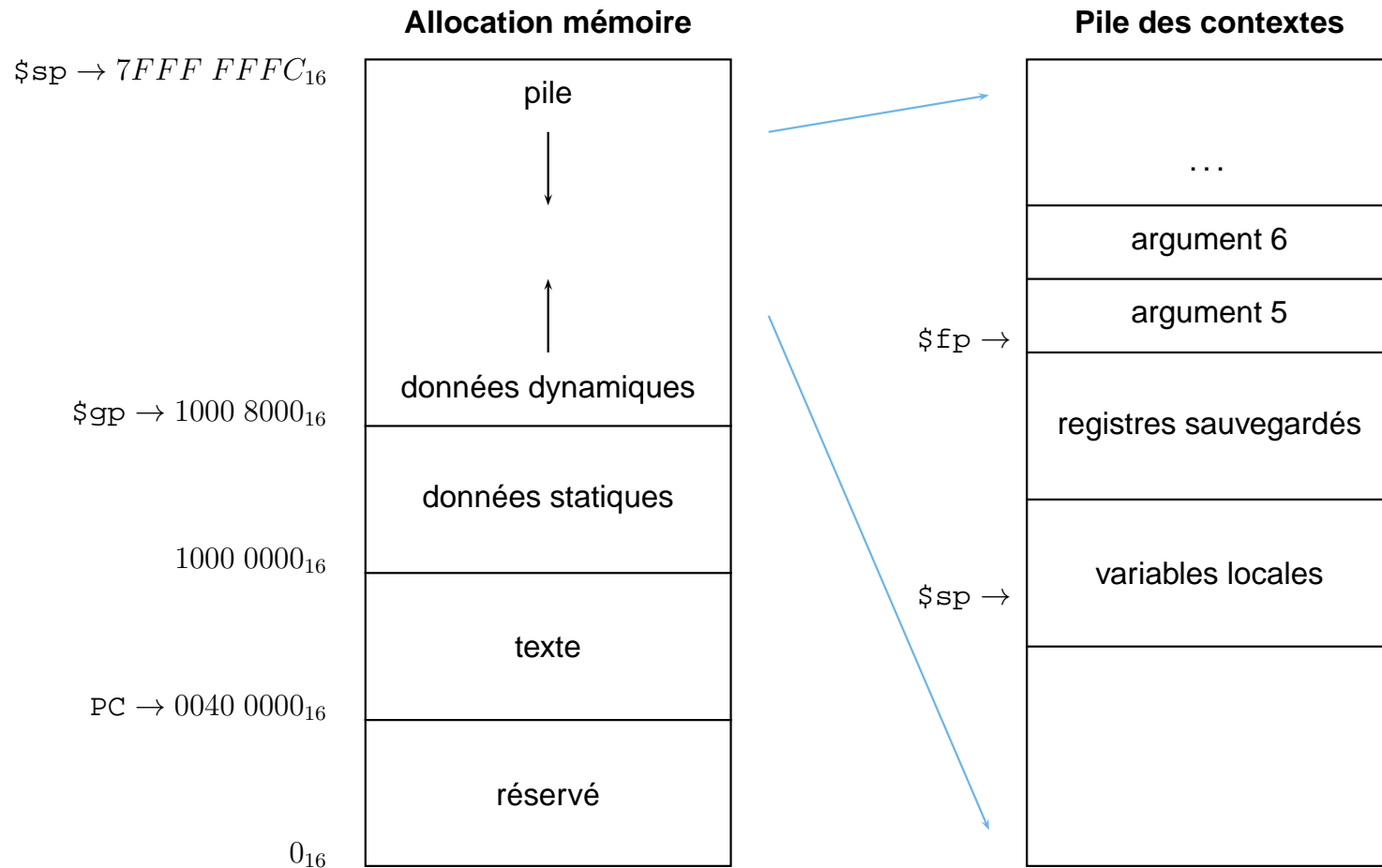
Soit l'instruction C : `while (save[i] == k) i+=1;`
avec `i` et `k` associés au registres `$s3` et `$s5`, et l'adresse de base de `save` rangée dans le registre `$s6`.

```
Loop:  sll $t1, $s3, 2      # t1 <- 4*i
        add $t1, $t1, $s6  # t1 <- adresse de save[i]
        lw $t0, 0($t1)    # t0 <- save[i]
        bne $t0, $s5, Exit # goto Exit if save[i]!=k
        addi $s3, $s3, 1  # i += 1
        j Loop           # goto Loop
Exit:                                     # sortie de boucle
```

Organisation de la mémoire



Organisation de la mémoire



Modes d'adressage MIPS



- **Immédiat** : l'opérande est une constante contenue dans l'instruction

Modes d'adressage MIPS



- **Immédiat** : l'opérande est une constante contenue dans l'instruction
- **Registre** : l'opérande est un registre

Modes d'adressage MIPS



- **Immédiat** : l'opérande est une constante contenue dans l'instruction
- **Registre** : l'opérande est un registre
- **Indexé** : l'opérande est à une adresse de base à laquelle est ajouté le contenu d'un registre comme déplacement

Modes d'adressage MIPS



- **Immédiat** : l'opérande est une constante contenue dans l'instruction
- **Registre** : l'opérande est un registre
- **Indexé** : l'opérande est à une adresse de base à laquelle est ajouté le contenu d'un registre comme déplacement
- **Indexé sur le PC** : l'opérande est à une adresse à laquelle est ajouté le contenu du compteur de programme comme déplacement

Modes d'adressage MIPS



- **Immédiat** : l'opérande est une constante contenue dans l'instruction
- **Registre** : l'opérande est un registre
- **Indexé** : l'opérande est à une adresse de base à laquelle est ajouté le contenu d'un registre comme déplacement
- **Indexé sur le PC** : l'opérande est à une adresse à laquelle est ajouté le contenu du compteur de programme comme déplacement
- **Pseudo-direct** : saut obtenu par concaténation des 26 bits du champs adresse d'une instruction j et des quatre bits de poids fort du PC

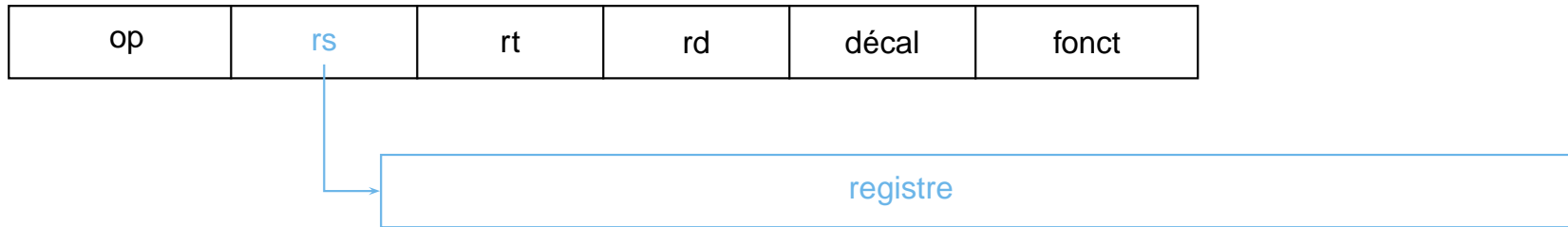
Adressage immédiat

L'opérande est une constante contenue dans l'instruction



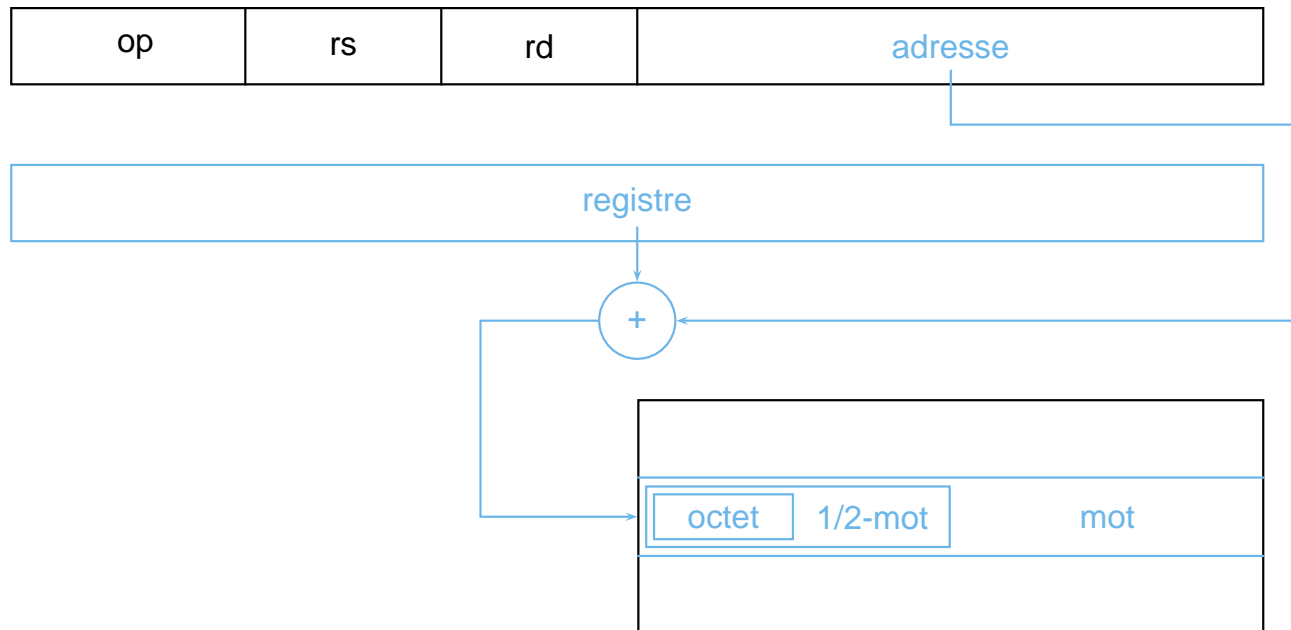
Adressage registre

L'opérande est un registre



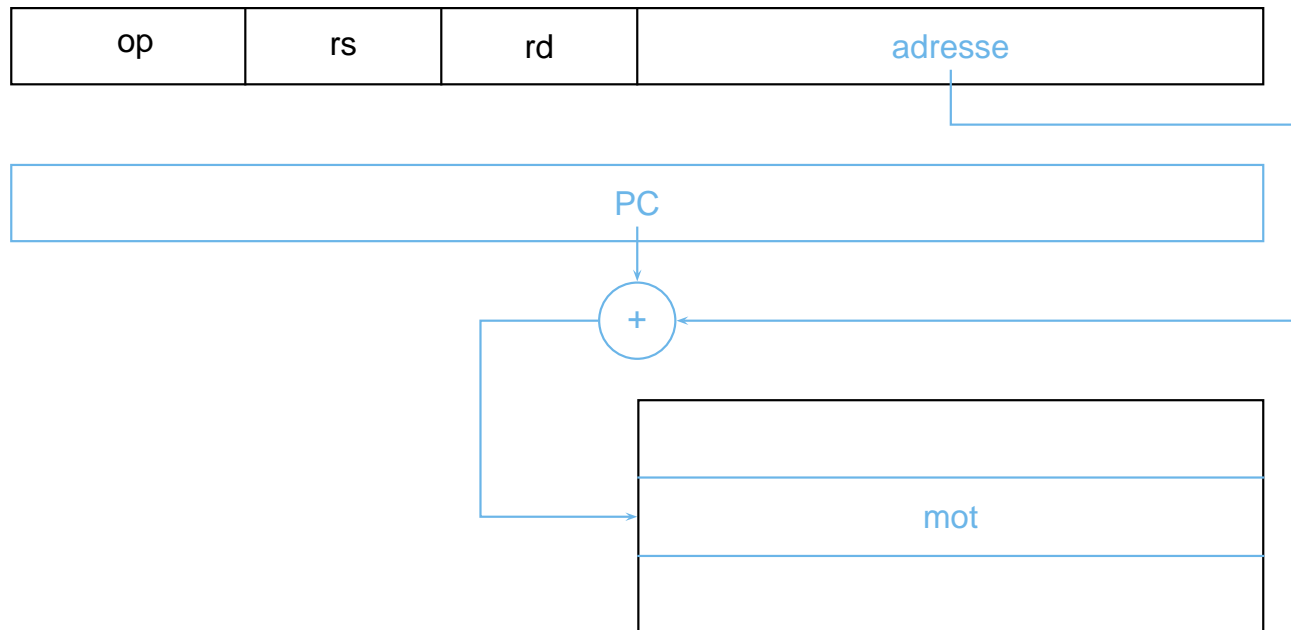
Adressage indexé

L'opérande est à une adresse de base à laquelle est ajouté le contenu d'un registre comme déplacement



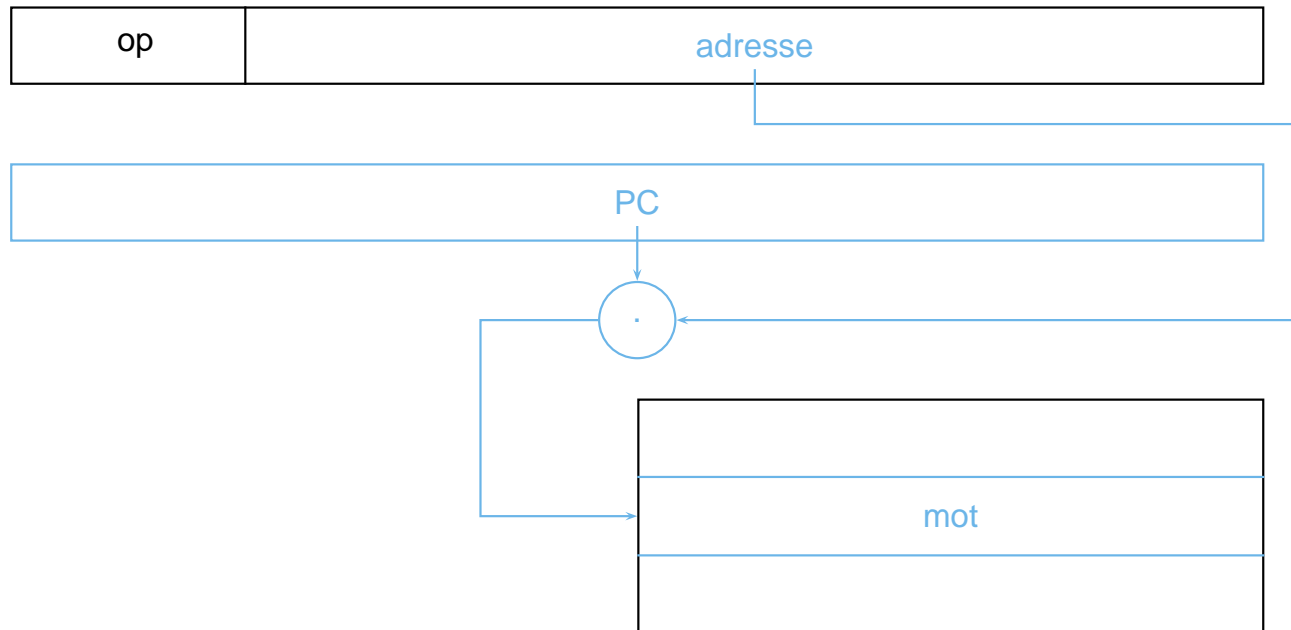
Adressage indexé sur le PC

L'opérande est à une adresse à laquelle est ajouté le contenu du compteur de programme comme déplacement



Adressage pseudo-direct

Saut obtenu par concaténation des 26 bits du champs adresse d'une instruction j et des quatre bits de poids fort du PC



Appels de fonction



Six étapes :

- la fonction appelante place les paramètres–données dans une zone accessible par la fonction appelée

Appels de fonction



Six étapes :

- la fonction appelante place les paramètres–données dans une zone accessible par la fonction appelée
- elle transfère le contrôle à la fonction appelée

Appels de fonction



Six étapes :

- la fonction appelante place les paramètres–données dans une zone accessible par la fonction appelée
- elle transfère le contrôle à la fonction appelée
- la fonction appelée acquiert les ressources mémoire nécessaires

Appels de fonction



Six étapes :

- ❑ la fonction appelante place les paramètres–données dans une zone accessible par la fonction appelée
- ❑ elle transfère le contrôle à la fonction appelée
- ❑ la fonction appelée acquiert les ressources mémoire nécessaires
- ❑ le corps de la fonction appelée est exécuté

Appels de fonction



Six étapes :

- ❑ la fonction appelante place les paramètres–données dans une zone accessible par la fonction appelée
- ❑ elle transfère le contrôle à la fonction appelée
- ❑ la fonction appelée acquiert les ressources mémoire nécessaires
- ❑ le corps de la fonction appelée est exécuté
- ❑ la fonction appelée place les paramètres–résultats dans une zone accessible par la fonction appelante

Appels de fonction



Six étapes :

- ❑ la fonction appelante place les paramètres–données dans une zone accessible par la fonction appelée
- ❑ elle transfère le contrôle à la fonction appelée
- ❑ la fonction appelée acquiert les ressources mémoire nécessaires
- ❑ le corps de la fonction appelée est exécuté
- ❑ la fonction appelée place les paramètres–résultats dans une zone accessible par la fonction appelante
- ❑ elle rend le contrôle au point d’appel de la fonction appelante

Registres dédiés aux appels de fonction

1. `$a0–$a3` : quatre registres dédiés au passage de paramètres
2. `$v0–$v1` : deux registres dédiés au retour de résultats
3. `$ra` : registre-adresse de retour au point d'appel, utilisé par l'instruction `jr`
4. une instruction dédiée à l'appel de fonction : `jal`
5. le registre de gestion de pile `$sp` pour le sauvegarde et la restauration de variables du programme appelant

Appel de fonction



1. mémorique : `jal` → *Jump And Link*
2. exemple : `jal Etiquette`
3. signification : `$ra = PC + 4; goto Etiquette;`

Retour de fonction



1. mémorique : `jr` → *Jump Register*
2. exemple : `jr $ra`
3. signification : `goto $ra;`

Modèle général appelant-appelé



```
appelant:  ...           # ...  
           jal appelé   # $ra=PC+4; goto appelé;  
           ...           # adresse retour : PC+4
```

```
appelé:   ...           ...  
           jr $ra       retour à PC+4
```

Modèle général appelant-appelé

```
appelant:  ...           # ...
           jal appelé  # $ra=PC+4; goto appelé;
           ...         # adresse retour : PC+4

appelé:   ...           ...
           jr $ra      retour à PC+4
```


Pile et registres supplémentaires



1. Tout registre de l'appelant utilisé par la fonction appelée doit être sauvegardé à l'entrée, et restauré au sortir par celle-ci.
2. La structure de donnée idéale est la **pile** : les valeurs des registres sont empilées à l'entrée de la fonction, et dépilées à la sortie.
3. MIPS utilise le registre dédié `$sp` (*Stack Pointer*)
4. Pour des raisons "historiques", la pile croît des adresses hautes vers les adresses basses.

fonction_exemple/C

```
int fonction_exemple (int g, int h,  
                      int i, int j)  
{  
    int f;  
    f = (g+h)-(i+j);  
    return f;  
}
```

avec g, h, i, j associées aux registres $\$a0, \$a1, \$a2, \$a3$, f associée au registre $\$s0$, la valeur de retour associée au registre $\$v0$, et l'adresse de retour au registre $\$ra$

Corps de la fonction `_exemple/MIPS`



`f = (g+h) - (i+j);`

va être compilé en :

```
add $t0, $a0, $a1 # t0 <- g+h
add $t1, $a2, $a3 # t1 <- i+j
sub $s0, $t0, $t1 # f = $t0 - $t1
```

→ il faut préalablement sauvegarder les contenus des registres `$s0`, `$t0`, `$t1`

Sauvegarde des registres (“PUSH”)



```
addi $sp, $sp, -12 # alloue l'espace dans la pile
sw $t1, 8($sp)    # sauve $t1 dans la pile
sw $t0, 4($sp)    # sauve $t0 dans la pile
sw $s0, 0($sp)    # sauve $s0 dans la pile
```

Passage des paramètres retours



```
add $v0, $s0, $zero # renvoie f
```

Transfert du registre \$v0 vers \$s0

Restauration des registres(“POP”)



```
lw $s0, 0($sp)      # restaure $s0 pour l'appelant
lw $t0, 4($sp)      # restaure $t0 pour l'appelant
lw $t1, 8($sp)      # restaure $t1 pour l'appelant
addi $sp, $sp, 12   # rend l'espace alloué dans la pile
```

f_exemple

```
f_exemple:  addi $sp, $sp, -12    # alloue l'espace dans la pile
            sw $t1, 8($sp)    # sauve $t1 dans la pile
            sw $t0, 4($sp)    # sauve $t0 dans la pile
            sw $s0, 0($sp)    # sauve $s0 dans la pile
            add $t0, $a0, $a1  # t0 <- g+h
            add $t1, $a2, $a3  # t1 <- i+j
            sub $s0, $t0, $t1  # f = $t0 - $t1
            add $v0, $s0, $zero # renvoie f
            lw $s0, 0($sp)    # restaure $s0 pour l'appelant
            lw $t0, 4($sp)    # restaure $t0 pour l'appelant
            lw $t1, 8($sp)    # restaure $t1 pour l'appelant
            addi $sp, $sp, 12  # rend l'espace dans la pile
            jr $ra            # retour à l'appelant
```

Conventions de programmation



Registres :

non sauvegardés	à sauvegarder

Conventions de programmation



Registres :

non sauvegardés	à sauvegarder
\$v0 – \$v1 \$a0 – \$a3 \$t0 – \$t9	

Conventions de programmation



Registres :

non sauvegardés	à sauvegarder
\$v0 – \$v1	\$s0 – \$s7
\$a0 – \$a3	\$gp
\$t0 – \$t9	\$sp
	\$fp
	\$ra

f_exemple (suite)

```
f_exemple:  addi $sp, $sp, -12    # alloue l'espace dans la pile
            sw $t1, 8($sp)      # sauve $t1 dans la pile
            sw $t0, 4($sp)      # sauve $t0 dans la pile
            sw $s0, 0($sp)      # sauve $s0 dans la pile
            add $t0, $a0, $a1    # t0 <- g+h
            add $t1, $a2, $a3    # t1 <- i+j
            sub $s0, $t0, $t1    # f = $t0 - $t1
            add $v0, $s0, $zero  # renvoie f
            lw $s0, 0($sp)      # restaure $s0 pour l'appelant
            lw $t0, 4($sp)      # restaure $t0 pour l'appelant
            lw $t1, 8($sp)      # restaure $t1 pour l'appelant
            addi $sp, $sp, 12    # rend l'espace dans la pile
            jr $ra              # retour à l'appelant
```

f_exemple (suite)

```
f_exemple:  addi $sp, $sp, -12    # alloue l'espace dans la pile
            sw $t1, 8($sp)     # sauve $t1 dans la pile
            sw $t0, 4($sp)     # sauve $t0 dans la pile
            sw $s0, 0($sp)     # sauve $s0 dans la pile
            add $t0, $a0, $a1   # t0 <- g+h
            add $t1, $a2, $a3   # t1 <- i+j
            sub $s0, $t0, $t1   # f = $t0 - $t1
            add $v0, $s0, $zero # renvoie f
            lw $s0, 0($sp)     # restaure $s0 pour l'appelant
            lw $t0, 4($sp)     # restaure $t0 pour l'appelant
            lw $t1, 8($sp)     # restaure $t1 pour l'appelant
            addi $sp, $sp, 12   # rend l'espace dans la pile
            jr $ra             # retour à l'appelant
```

f_exemple (suite)

```
f_exemple:  addi $sp, $sp, -4      # alloue l'espace dans la pile

            sw $s0, 0($sp)        # sauve $s0 dans la pile
            add $t0, $a0, $a1     # t0 <- g+h
            add $t1, $a2, $a3     # t1 <- i+j
            sub $s0, $t0, $t1     # f = $t0 - $t1
            add $v0, $s0, $zero   # renvoie f
            lw $s0, 0($sp)       # restaure $s0 pour l'appelant

            addi $sp, $sp, 4      # rend l'espace dans la pile
            jr $ra                # retour à l'appelant
```

Deux cas à considérer



Si la fonction appelée est

- terminale :

Deux cas à considérer



Si la fonction appelée est

- terminale :
 - △ empiler `$s0`, `$s1` ... utilisés

Deux cas à considérer



Si la fonction appelée est

- terminale :
 - △ empiler `$s0`, `$s1` ... utilisés
- non-terminale :

Deux cas à considérer



Si la fonction appelée est

- terminale :
 - △ empiler `$s0`, `$s1` ... utilisés
- non-terminale :
 - △ empiler `$ra`

Deux cas à considérer



Si la fonction appelée est

- terminale :
 - △ empiler `$s0`, `$s1` ... utilisés
- non-terminale :
 - △ empiler `$ra`
 - △ empiler `$a0`, `$a1`, ... utilisés

Deux cas à considérer



Si la fonction appelée est

- terminale :
 - △ empiler `$s0`, `$s1` ... utilisés
- non-terminale :
 - △ empiler `$ra`
 - △ empiler `$a0`, `$a1`, ... utilisés
 - △ empiler `$s0`, `$s1` ... utilisés

Deux cas à considérer

Si la fonction appelée est

- terminale :
 - △ empiler `$s0`, `$s1` ... utilisés
- non-terminale :
 - △ empiler `$ra`
 - △ empiler `$a0`, `$a1`, ... utilisés
 - △ empiler `$s0`, `$s1` ... utilisés

Exemple important de fonction non-terminale : les fonctions récursives...

fonction fact/C



```
int fact (int n)
{
    if (n<1) return 1;
    else return (n * fact(n-1));
}
```

Le paramètre **n** est placé dans le registre \$a0, la valeur de retour de la fonction dans le registre \$v0.

fact

```
fact:    addi $sp, $sp, -8      # alloue l'espace dans la pile
        sw $ra, 4($sp)        # empile l'adresse de retour
        sw $a0, 0($sp)        # empile l'argument n
        slti $t0, $a0, 1      # test pour n < 1
        beq $t0, $zero, recurs # if (n>=1) goto recurs;
        addi $v0, $zero, 1     # return 1;
        lw $a0, 0($sp)        # dépile l'argument n
        lw $ra, 4($sp)        # dépile l'adresse de retour
        addi $sp, $sp, 8      # pop des deux valeurs sauvées
        jr $ra                # retour à l'appelant
recurs:  addi $a0, $a0, -1     # n>=1: l'argument reçoit n-1
        jal fact              # appel récursif
        lw $a0, 0($sp)        # dépile l'argument n
        lw $ra, 4($sp)        # dépile l'adresse de retour
        addi $sp, $sp, 8      # rend l'espace dans la pile
        mul $v0, $a0, $v0     # return n*fact(n-1)
        jr $ra                # retour à la fonction appelante
```

fact

```
fact:    addi $sp, $sp, -8      # alloue l'espace dans la pile
         sw $ra, 4($sp)        # empile l'adresse de retour
         sw $a0, 0($sp)        # empile l'argument n
         slti $t0, $a0, 1      # test pour n < 1
         beq $t0, $zero, recurs # if (n>=1) goto recurs;
         addi $v0, $zero, 1     # return 1;
         lw $a0, 0($sp)         # dépile l'argument n
         lw $ra, 4($sp)         # dépile l'adresse de retour
         addi $sp, $sp, 8      # pop des deux valeurs sauvées
         jr $ra                # retour à l'appelant
recurs:  addi $a0, $a0, -1      # n>=1: l'argument reçoit n-1
         jal fact              # appel récursif
         lw $a0, 0($sp)        # dépile l'argument n
         lw $ra, 4($sp)        # dépile l'adresse de retour
         addi $sp, $sp, 8      # rend l'espace dans la pile
         mul $v0, $a0, $v0     # return n*fact(n-1)
         jr $ra                # retour à la fonction appelante
```

fact

```
fact:    addi $sp, $sp, -8      # alloue l'espace dans la pile
        sw $ra, 4($sp)        # empile l'adresse de retour
        sw $a0, 0($sp)        # empile l'argument n
        slti $t0, $a0, 1      # test pour n < 1
        beq $t0, $zero, recurs # if (n>=1) goto recurs;
        addi $v0, $zero, 1     # return 1;

        addi $sp, $sp, 8      # pop des deux valeurs sauvées
        jr $ra                # retour à l'appelant
recurs:  addi $a0, $a0, -1     # n>=1: l'argument reçoit n-1
        jal fact              # appel récursif
        lw $a0, 0($sp)        # dépile l'argument n
        lw $ra, 4($sp)        # dépile l'adresse de retour
        addi $sp, $sp, 8      # rend l'espace dans la pile
        mul $v0, $a0, $v0     # return n*fact(n-1)
        jr $ra                # retour à la fonction appelante
```


Compilateur et éditeur de liens



- Compilateur : transforme le programme source en programme en langage d'assemblage utilisant des emplacements mémoire symboliques.

Compilateur et éditeur de liens



- ❑ Compilateur : transforme le programme source en programme en langage d'assemblage utilisant des emplacements mémoire symboliques.
- ❑ Editeur de liens : résoud les références symboliques sur des programmes compilés séparément.

Compilateurs optimisants



Un compilateur peut réaliser trois types d'optimisation :

- *Optimisations locales*. Réalisées à l'intérieur d'un bloc.

Compilateurs optimisants



Un compilateur peut réaliser trois types d'optimisation :

- *Optimisations locales*. Réalisées à l'intérieur d'un bloc.
- *Optimisations globales*. Réalisées sur plusieurs blocs.

Compilateurs optimisants



Un compilateur peut réaliser trois types d'optimisation :

- *Optimisations locales*. Réalisées à l'intérieur d'un bloc.
- *Optimisations globales*. Réalisées sur plusieurs blocs.
- *Allocation des registres*. Association des variables aux registres en fonction des différentes portions de code.

Compilateurs optimisants



Un compilateur peut réaliser trois types d'optimisation :

- ❑ *Optimisations locales*. Réalisées à l'intérieur d'un bloc.
- ❑ *Optimisations globales*. Réalisées sur plusieurs blocs.
- ❑ *Allocation des registres*. Association des variables aux registres en fonction des différentes portions de code.

Beaucoup d'optimisations sont réalisées à la fois localement et globalement, par exemple l'élimination de sous-expressions communes, la propagation des constantes, la propagation par copie...

Elimination de sous-expressions communes

Le compilateur détecte plusieurs occurrences d'une même expression, et utilise la première pour éliminer les suivantes.

Exemple : $T[i] += 3;$

```
# T[i]+3;
li R100, T
lw R101, i
sll R102, R101, 2
add R103, R100, R102
lw R104, 0(R103)
add R105, R104, 3
# T[i]=
li R106, T
lw R107, i
sll R108, R107, 2
add R109, R106, R108
sw R105, 0(R109)
```

Elimination de sous-expressions communes

Le compilateur détecte plusieurs occurrences d'une même expression, et utilise la première pour éliminer les suivantes.

Exemple : $T[i] += 3;$

```
# T[i]+3;
li R100, T
lw R101, i
sll R102, R101, 2
add R103, R100, R102
lw R104, 0(R103)
add R105, R104, 3
# T[i]=
li R106, T
lw R107, i
sll R108, R107, 2
add R109, R106, R108
sw R105, 0(R109)
```


Elimination de sous-expressions communes

Le compilateur détecte plusieurs occurrences d'une même expression, et utilise la première pour éliminer les suivantes.

Exemple : $T[i] += 3;$

```
# T[i]+3;
li R100, T
lw R101, i
sll R102, R101, 2
add R103, R100, R102
lw R104, 0(R103)
add R105, R104, 3
# T[i]=
li R106, T
lw R107, i
sll R108, R107, 2
add R109, R106, R108
sw R105, 0(R109)
```

```
# T[i]+3;
li R100, T
lw R101, i
sll R102, R101, 2
add R103, R100, R102
lw R104, 0(R103)
add R105, R104, 3
sw R105, 0(R103)
```

Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3
# T[i]=
li R106, T           # adresse de base du tableau T
lw R107, i           # valeur de l'incrément i
sll R108, R107, 2    # alignement sur 4 octets
add R109, R106, R108 # valeur du déplacement
sw R105, 0(R109)     # T[i]=T[i]+3;
```

Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3
# T[i]=
li R106, T           # adresse de base du tableau T
lw R107, i           # valeur de l'incrément i
sll R108, R107, 2    # alignement sur 4 octets
add R109, R106, R108 # valeur du déplacement
sw R105, 0(R109)     # T[i]=T[i]+3;
```

Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3
# T[i]=
li R100, T           # adresse de base du tableau T
lw R107, i           # valeur de l'incrément i
sll R108, R107, 2    # alignement sur 4 octets
add R109, R100, R108 # valeur du déplacement
sw R105, 0(R109)     # T[i]=T[i]+3;
```

Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3
# T[i]=
li R100, T           # adresse de base du tableau T
lw R107, i           # valeur de l'incrément i
sll R108, R107, 2    # alignement sur 4 octets
add R109, R100, R108 # valeur du déplacement
sw R105, 0(R109)     # T[i]=T[i]+3;
```

Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3
# T[i]=
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R108, R101, 2    # alignement sur 4 octets
add R109, R100, R108 # valeur du déplacement
sw R105, 0(R109)     # T[i]=T[i]+3;
```

Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3
# T[i]=
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R108, R101, 2    # alignement sur 4 octets
add R109, R100, R108 # valeur du déplacement
sw R105, 0(R109)     # T[i]=T[i]+3;
```

Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3
# T[i]=
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R109, R100, R102 # valeur du déplacement
sw R105, 0(R109)     # T[i]=T[i]+3;
```


Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3
# T[i]=
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R109, R100, R102 # valeur du déplacement
sw R105, 0(R109)     # T[i]=T[i]+3;
```

Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3
# T[i]=
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
sw R105, 0(R103)     # T[i]=T[i]+3;
```

Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3
# T[i]=
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
sw R105, 0(R103)     # T[i]=T[i]+3;
```

Compilation optimisée de $T[i] += 3;$

```
# T[i]+3;
li R100, T           # adresse de base du tableau T
lw R101, i           # valeur de l'incrément i
sll R102, R101, 2    # alignement sur 4 octets
add R103, R100, R102 # valeur du déplacement
lw R104, 0(R103)     # R104 <- adresse de T[i]
add R105, R104, 3    # R105 <- T[i]+3

sw R105, 0(R103)     # T[i]=T[i]+3;
```

Directives d'assemblage



Dit à l'assembleur comment interpréter ce qui suit en mémoire. Par exemple :

- `.ascii <chaîne>` : ce qui suit la directive est une chaîne de caractères
- `.asciiz <chaîne>` : ce qui suit la directive est une chaîne terminée par le caractère `\0`
- `.byte <b1, ..., bn>` : range `b1, ..., bn` dans n octets successifs
- `.data` : ce qui suit la directive est placé dans le segment de données

Directives d'assemblage



- **glob** <étiquette> : ce qui suit la directive est une étiquette globale et peut être référencée à partir d'autres fichiers
- **text** : ce qui suit la directive est placé dans le segment Texte
- **word** <w1, ..., wn> : range w1, ..., wn dans n mots successifs

Exemple de directive

```
        .data
tableau: .word      0x00000001,
                   0x00000002,
                   0x00000003,
                   0x00000004,
                   0x00000005

        .text
main:   la $t0, tableau
        ...
```

Exemple de directive

```
        .data
tableau: .word      0x00000001,
                   0x00000002,
                   0x00000003,
                   0x00000004,
                   0x00000005

        .text
main:   la $t0, tableau
        ...
```

- portion de programme référençant un tableau d'entiers dans le segment de données
- `la` est une pseudo-instruction permettant le chargement d'une adresse

Entrées-sorties : `syscall`

Toutes les entrées–sorties sont prises en charge par la routine système `syscall`

Opération	<code>\$v0 =</code>	Arguments	Valeur retour
<code>print_int</code>	1	<code>\$a0 = entier à afficher</code>	
<code>print_string</code>	4	<code>\$a0 = chaîne à afficher</code>	
<code>read_int</code>	5		<code>\$v0 = entier lu</code>
<code>read_string</code>	8	<code>\$a0 = buffer, \$a0 = longueur</code>	
<code>exit</code>	10		
<code>print_char</code>	11	<code>\$a0 = car. à afficher</code>	<code>\$a0 = car. lu</code>
<code>read_char</code>	12		<code>\$a0 = car. lu</code>
<code>exit2</code>	17	<code>\$a0 = résultat</code>	

Exemple : *syscall*

```
    .data
str:  .asciiz      "Bonjour"
    .text
main: ori $v0, $zero, 4   # $v0 <- 4
      la $a0, str        # $a0 <- str
      syscall           # affiche "Bonjour"
      ...
      ori $v0, $zero, 10 # $v0 <- 10
      syscall           # appel système de
                        # terminaison de programme
```

Exemple : `syscall`

```
        .data
str:    .asciiz      "Bonjour"
        .text
main:   ori $v0, $zero, 4    # $v0 <- 4
        la $a0, str        # $a0 <- str
        syscall           # affiche "Bonjour"
        ...
        ori $v0, $zero, 10  # $v0 <- 10
        syscall           # appel système de
                          # terminaison de programme
```

- l'étiquette `str` : référence l'adresse de la chaîne de caractères en mémoire
- `$a0` est l'adresse de la chaîne, passée comme argument à `syscall`