Processeur MIPS R3000.

Langage d'assemblage

Version 1.4

(septembre 2001)

A) INTRODUCTION

Ce document décrit le langage d'assemblage du processeur MIPS R3000, ainsi que différentes conventions relatives à l'écriture des programmes en langage d'assemblage.

Un document séparé décrit l'architecture externe du processeur, c'est-à-dire les registres visibles du logiciel, les règles d'adressage de la mémoire, le codage des instructions machine, et les mécanismes de traitement des interruptions et des exceptions.

On présente ici successivement l'organisation de la mémoire, les principales règles syntaxiques du langage, les instructions et les macro-instructions, les directives acceptées par l'assembleur, les quelques appels système disponibles, ainsi que conventions imposées pour les appels de fonctions et la gestion de la pile.

Les programmes assembleur source qui respectent les règles définies dans le présent document peuvent être assemblés par l'assembleur MIPS de l'environnement GNU pour générer du code exécutable. Ils sont également acceptés par le simulateur du MIPSR3000 utilisé en TP qui permet de visualiser le comportement du processeur instruction par instruction.

B) ORGANISATION DE LA MÉMOIRE

Rappelons que le but d'un programme X écrit en langage d'assemblage est de fournir à un programme particulier (appelé $<<$ assembleur $>>$) les directives nécessaires pour générer le code binaire représentant les instructions et les données qui devront être chargées en mémoire pour permettre au programme X de s'exécuter sur du matériel.

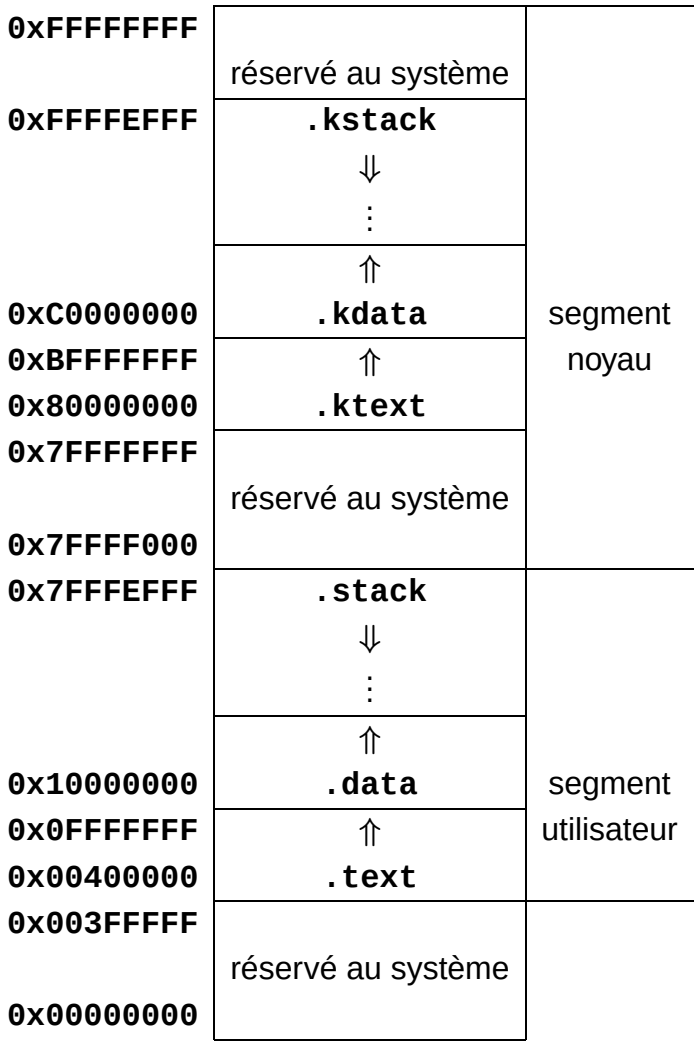
Dans l'architecture MIPS R3000, l'espace adressable est divisé en deux segments : le segment utilisateur, et le segment noyau.

Un programme utilisateur utilise généralement trois sous-segments (appelés sections) dans le segment utilisateur :

– la section **`text`** contient le code exécutable en mode utilisateur. Elle est implantée conventionnellement à l'adresse **`0x00400000`**. Sa taille est fixe et calculée lors de l'assemblage. La principale tâche de l'assembleur consiste à générer le code binaire correspondant au programme source décrit en langage d'assemblage, qui sera chargé dans cette section ;
– la section **`data`** contient les données globales manipulées par le programme utilisateur. Elle est implantée conventionnellement à l'adresse **`0x10000000`**. Sa taille est fixe et calculée lors de l'assemblage. Les valeurs contenue dans cette section peuvent être initialisées grace a des directives contenues dans le programme source en langage d'assemblage ;
– la section **`stack`** contient la pile d'exécution du programme. Sa taille varie au cours de l'exécution. Elle est implantée conventionnellement à l'adresse **`0x7FFFEFFF`**. Contrairement aux sections **`data`** et **`text`**, la pile s'étend vers les adresses décroissantes.

Deux autres sections sont définies dans le segment noyau :

– la section **`ktext`** contient le code exécutable en mode noyau. Elle est implantée conventionnellement à l'adresse **`0x80000000`**. Sa taille est fixe et calculée lors de l'assemblage ;
– la section **`kdata`** contient les données globales manipulées par le système d'exploitation en mode noyau. Elle est implantée conventionnellement à l'adresse **`0xC0000000`**. Sa taille est fixe et calculée lors de l'assemblage ;
– la section **`kstack`** contient la pile d'exécution du programme. Sa taille varie au cours de l'exécution. Elle est implantée conventionnellement à l'adresse **`0xFFFFEFFF`**. Contrairement aux sections **`data`** et **`text`**, la pile s'étend vers les adresses décroissantes.


0xFFFFFFFF
réservé au système
0xFFFFEFFF
.kstack
⇓
⋮
⇑
0xC0000000
.kdata
segment
noyau
0xBFFFFFFF
⇑
0x80000000
.ktext
0x7FFFFFFF
réservé au système
0x7FFF000
0x7FFEFFF
.stack
⇓
⋮
⇑
0x10000000
.data
segment
utilisateur
0x0FFFFFFF
⇑
0x00400000
.text
0x003FFFFF
réservé au système
0x00000000

C) RÈGLES SYNTAXIQUES

1. **les noms de fichiers** :
   Les noms des fichiers contenant un programme source en langage d'assemblage doivent être suffixé par << **.s** >>. Exemple : `monprogramme.s`
2. **les commentaires** :
   ils commencent par un **#** ou un **;** et s'achèvent à la fin de la ligne courante. Exemple :

```
################################################
# Source Assembleur MIPS de la fonction memcpy
################################################
....        ; sauve la valeur copiée dans la mémoire
```

3. **les entiers** :
   une valeur entière décimale est notée **250**, une valeur entière octale est notée **0372** (préfixée par un zéro), et une valeur entière héxadécimale est notée **0xFA** (préfixée par zéro suivi de x). En hexadécimal, les lettres de **A** à **F** peuvent être écrites en majuscule ou en minuscule.
4. **les chaînes de caractères** :
   elles sont simplement entre guillemets, et peuvent contenir les caractères d'échappement du langage C.
   Exemple : `"Oh la jolie chaîne avec retour à la ligne\n"`
5. **les labels** :
   ce sont des mnémoniques correspondant à des adresses. Ces adresses peuvent être soit des adresses de variables,soit des adresses de saut. Ce sont des chaînes de caractères qui commencent par une lettre, majuscule ou minuscule, un **$**, un _, ou un **.**. Ensuite, un nombre quelconque de ces mêmes caractères auquels on ajoute les chiffres sont utilisés. Pour la déclaration, le label doit être suffixé par << **:** >>. Exemple : `$LC12:`
   Pour y référer, on supprime le << **:** >>.
   Exemple :

```
message:
       .asciiz  "Ceci est une chaîne de caractères...\n"

       .text
__start:
```

```
la      $4, message ; adresse de la chaine dans $4
ori     $2, $0, 4       ; code du 'print_string' dans $2
syscall
```

Attention : sont illégaux les labels qui ont le même nom qu'un mnémonique de l'assembleur ou qu'un nom de registre.

6. **les immédiats** :
   ce sont les opérandes contenus dans l'instruction. Ce sont des constantes. Ce sont soit des entiers, soit des labels. Ces constantes doivent respecter une taille maximum qui est fonction de l'instruction qui l'utilise : 16 ou 26 bits.
7. **les registres** :
   le processeur MIPS possède 32 registres accessibles au programmeur. Chaque registre est connu par son numéro, qui varie entre 0 et 31, et est préfixé par un **$**. Par exemple, le registre 31 sera noté **$31** dans l'assembleur.
   En dehors du registre **$0**, tous les registres sont identiques du point de vue de la machine. Lorsque le registre **$0** est utilisé comme registre source dans une instruction, la valeur lue est toujours 0, et utiliser ce registre comme registre destination dans une instruction ne modifie pas sa valeur.
   Afin de normaliser et de simplifier l'écriture du logiciel, des conventions d'utilisation des registres sont définies. Ces conventions sont particulièrement nécessaires lors de l'utilisation des fonctions.

$0	vaut zéro en lecture, non modifié par écriture
$1	réservé à l'assembleur. Ne doit pas être employé dans les programmes utilisateur
$2	valeur de retour des fonctions
$3, $4	argument des *syscall*
$5, ..., $26	registres de travail à sauver
$27, ..., $28	registres réservés aux procédures noyau. Ils ne doivent pas être employés dans les programmes utilisateur
$29	pointeur de pile
$30	pointeur sur les variables globales
$31	adresse de retour d'appel de fonction

8. **les arguments** :
   si une instruction nécessite plusieurs arguments, comme par exemple l'addition entre deux registres, ces arguments sont séparés par des virgules. Dans une instruction assembleur, on aura en général comme argument en premier le registre

dans lequel est mis le résultat de l'opération, puis ensuite le premier registre source, puis enfin le second registre source ou une constante.

Exemple : `add $3, $2, $1`

9. **l'adressage mémoire** :
   le MIPS ne possède qu'un unique mode d'adressage : l'adressage indirect registre avec déplacement. Ainsi l'accès à une case mémoire à partir de l'adresse présente dans un registre se note par le déplacement, c.-à-d. un entier comme défini précédemment, suivi du registre entre parenthèses.

   Exemples : `11($12)`, `013($12)` `0xB($12)`.

   Ces trois exemples indiquent la case mémoire ayant comme adresse le contenu du registre **$12** plus 11.

   S'il n'y a pas d'entier devant la parenthèse ouvrante, le déplacement est nul.

   En revanche, il n'est pas possible d'écrire des sauts à des adresses absolues ou relatives *constantes*, comme par exemple un `j 0x400000` ou un `bnez $3, -12`. Il faut nécessairement utiliser des labels.

D) INSTRUCTIONS

Dans ce qui suit, le registre noté **`$rr`** est le registre destination, c.-à-d. qui reçoit le résultat de l'opération, les registres notés **`$ri`** et **`$rj`** sont les registres source qui contiennent les valeurs sur lesquelles s'effectue l'opération. Notons qu'un registre source peut être le registre destination d'une même instruction assembleur. Un opérande immédiat sera noté **`imm`**, et sa taille sera spécifié dans la description de l'instruction. Les instructions de saut prennent comme argument une étiquette, où *label*, qui est utilisée pour calculer l'adresse de saut. Toutes les instructions modifient un registre non accessible du logiciel, le *program counter*. De même, le résultat d'une multiplication ou d'une division est mis dans deux registres spéciaux, **`$hi`** pour les poids forts, et **`$lo`** pour les poids faibles.

Ceci nous amème à introduire quelques notations :

$=$	test d'égalité
$+$	addtition entière en complément à deux
$-$	soustraction entière en complément à deux
$\times$	multiplication entière en complément à deux
$\div$	division entière en complément à deux
mod	reste de la division entière en complément à deux
and	opérateur **et** bit-à-bit
or	opérateur **ou** bit-à-bit
nor	opérateur **non-ou** bit-à-bit
xor	opérateur **ou-exclusif** bit-à-bit
mem$[a]$	contenu de la mémoire à l'adresse a
$\leftarrow$	assignation
$\Rightarrow$	implication
$\|\|$	concaténation de chaînes de bits
x^n	réplication du bit x dans une chaîne de n bits. Notons que x est un unique bit
$x_{p...q}$	sélection des bits p à q de la chaîne de bits x

Certains opérateurs n'étant pas évidents, nous donnons ici quelques exemples.

```
15 14 13 12 11 10 9  8  7  6  5  4  3  2  1  0
```

Posons **`0 0 0 1 1 0 1 1 0 1 0 0 1 0 0 0`** la chaîne de bit x, qui a une longueur de 16 bits, le bit le plus à droite étant le bit de poids faible et de numéro zéro, et le bit le plus à gauche étant le bit de poids fort et de numéro 15. $x_{6...3}$ est la chaîne **`1001`**. x_{15}^{16} crée une chaîne de 16 bits de long dupliquant le bit 15 de x, zéro dans le cas présent. $x_{15}^{16} \,||\, x_{15...0}$ est la valeur 32 bits avec extension de signe d'un immédiat en complément à deux de 16 bits.

add

action

Addition registre registre signée

syntaxe

```
add $rr, $ri, $rj
```

description

Les contenus des registres **$ri** et **$rj** sont ajoutés pour former un résultat sur 32 bits qui est placé dans le registre **$rr**.

opération

$rr \leftarrow ri + rj$

exception

génération d'une exception si dépassement de capacité.

addi

action

Addition registre immédiat signée

syntaxe

```
addi $rr, $ri, imm
```

description

La valeur immédiate sur 16 bits subit une extension de signe, et est ajoutée au contenu du registre **$ri** pour former un résultat sur 32 bits qui est placé dans le registre **$rr**.

opération

$rr \leftarrow (imm_{15}^{16} \parallel imm_{15...0}) + ri$

exception

génération d'une exception si dépassement de capacité.

addiu

action

Addition registre immédiat non-signée

syntaxe

```
addiu $rr, $ri, imm
```

description

La valeur immédiate sur 16 bits subit une extension de signe, et est ajoutée au contenu du registre **$ri** pour former un résultat sur 32 bits qui est placé dans le registre **$rr**.

opération

$rr \leftarrow (imm_{15}^{16} \parallel imm_{15\ldots0}) + ri$

addu

action

Addition registre registre non-signée

syntaxe

```
addu $rr, $ri, $rj
```

description

Les contenus des registres **$ri** et **$rj** sont ajoutés pour former un résultat sur 32 bits qui est placé dans le registre **$rr**.

opération

$rr \leftarrow ri + rj$

and

action

Et bit-à-bit registre registre

syntaxe

```
and $rr, $ri, $rj
```

description

Un **et** bit-à-bit est effectué entre les contenus des registres **$ri** et **$rj**. Le résultat est placé dans le registre **$rr**.

opération

$rr \leftarrow ri$ and rj

andi

action

Et bit-à-bit registre immédiat

syntaxe

```
andi $rr, $ri, imm
```

description

La valeur immédiate sur 16 bits subit une extension de zéros. Un **et** bit-à-bit est effectué entre cette valeur étendue et le contenu du registre **$ri** pour former un résultat placé dans le registre **$rr**.

opération

$rr \leftarrow (0^{16} \parallel imm)$ and ri

beq

action

Branchement si registre égal registre

syntaxe

```
beq $ri, $rj, label
```

description

Les contenus des registres **`$ri`** et **`$rj`** sont comparés. S'ils sont égaux, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$ri = rj \Rightarrow pc \leftarrow pc + 4 + addr$

bgez

action

Branchement si registre supérieur ou égal à zéro

syntaxe

```
bgez $ri, label
```

description

Si le contenu du registre **`$ri`** est supérieur ou égal à zéro le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$ri \geq 0 \Rightarrow pc \leftarrow pc + 4 + addr$

bgezal

action

Branchement à une fonction si registre supérieur ou égal à zéro

syntaxe

```
bgezal $ri, label
```

description

Inconditionnellement, l'adresse de l'instruction suivant le **`bgezal`** est sauvée dans le registre **`$31`**. Si le contenu du registre **`$ri`** est supérieur ou égal à zéro le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$\$31 \leftarrow pc + 4$

$ri \geq 0 \Rightarrow pc \leftarrow pc + 4 + addr$

bgtz

action

Branchement si registre strictement supérieur à zéro

syntaxe

```
bgtz $ri, label
```

description

Si le contenu du registre **$ri** est strictement supérieur à zéro le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$ri > 0 \Rightarrow pc \leftarrow pc + 4 + addr$

blez

action

Branchement si registre inférieur ou égal à zéro

syntaxe

```
blez $ri, label
```

description

Si le contenu du registre **$ri** est inférieur ou égal à zéro le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$ri \leq 0 \Rightarrow pc \leftarrow pc + 4 + addr$

bltz

action

Branchement si registre strictement inférieur à zéro

syntaxe

```
bltz $ri, label
```

description

Si le contenu du registre **$ri** est strictement inférieur à zéro le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$ri < 0 \Rightarrow pc \leftarrow pc + 4 + addr$

bltzal

action

Branchement à une fonction si registre inférieur ou égal à zéro

syntaxe

```
bltzal $ri, label
```

description

Inconditionnellement, l'adresse de l'instruction suivant le **`bgezal`** est sauvée dans le registre **`$31`**. Si le contenu du registre **`$ri`** est strictement inférieur à zéro le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$\$31 \leftarrow pc + 4$

$ri < 0 \Rightarrow pc \leftarrow pc + 4 + addr$

bne

action

Branchement si registre différent de registre

syntaxe

```
bne $ri, $rj, label
```

description

Les contenus des registres **`$ri`** et **`$rj`** sont comparés. S'ils sont différents, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$ri \neq rj \Rightarrow pc \leftarrow pc + 4 + addr$

break

action

Arrêt et saut à la routine d'exception

syntaxe

```
break imm
```

description

Un point d'arrêt est detecté, et le programme saute à l'adresse de la routine de gestion des exceptions.

opération

$pc \leftarrow 0x80000080$

exception

Déclenchement d'une exception de type point d'arrêt.

div

action

Division entière et reste signé registre registre

syntaxe

```
div $ri, $rj
```

description

Le contenu du registre **$ri** est divisé par le contenu du registre **$rj**, le contenu des deux registres étant considéré comme des nombres en complément à deux. Le résultat de la division est placé dans le registre spécial **$lo**, et le reste dans **$hi**.

opération

$lo \leftarrow \frac{ri}{rj}$
$hi \leftarrow ri \bmod rj$

divu

action

Division entière et reste non-signé registre registre

syntaxe

```
divu $ri, $rj
```

description

Le contenu du registre **$ri** est divisé par le contenu du registre **$rj**, le contenu des deux registres étant considéré comme des nombres non signés. Le résultat de la division est placé dans le registre spécial **$lo**, et le reste dans **$hi**.

opération

$lo \leftarrow \frac{0 \| ri}{0 \| rj}$
$hi \leftarrow 0 \parallel ri \bmod 0 \parallel rj$

j

action

Branchement inconditionnel immédiat

syntaxe

```
j label
```

description

Le programme saute inconditionnellement à l'adresse correspondant au label, calculé par l'assembleur.

opération

$pc \leftarrow label$

jal

action

Appel de fonction inconditionnel immédiat

syntaxe

`jal label`

description

L'adresse de l'instruction suivant le **`jal`** est sauvée dans le registre **`$31`**. Le programme saute inconditionnellement à l'adresse correspondant au label, calculée par l'assembleur.

opération

$\$31 \leftarrow pc + 4$

$pc \leftarrow label$

jalr

action

Appel de fonction inconditionnel registre

syntaxe

`jalr $ri` ou

`jalr $rr, $ri`

description

Le programme saute à l'adresse contenue dans le registre **`$ri`**. L'adresse de l'instruction suivant le **`jalr`** est sauvée dans le registre **`$rr`**. Si le registre **`$rr`** n'est pas spéficié, alors c'est par défaut le registre **`$31`** qui est utilisé. Attention, l'adresse contenue dans le registre **`$ri`** doit être aligné sur une frontière de mots.

opération

$rr \leftarrow pc + 4$

$pc \leftarrow ri$

jr

action

Branchement inconditionnel registre

syntaxe

`jr $ri`

description

Le programme saute à l'adresse contenue dans le registre **$ri**. Attention, cette adresse doit être aligné sur une frontière de mots.

opération

$pc \leftarrow ri$

lb

action

Lecture d'un octet signé de la mémoire

syntaxe

```
lb $rr, imm($ri)
```

description

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre **$ri**. Le contenu de cette adresse subit une extension de signe et est ensuite placé dans le registre **$rr**.

opération

$rr \leftarrow$ mem$[imm + ri]_{7}^{24}$ || mem$[imm + ri]_{7...0}$

exception

- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
- Mémoire inexistante à l'adresse de chargement.

lbu

action

Lecture d'un octet non-signé de la mémoire

syntaxe

```
lbu $rr, imm($ri)
```

description

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre **$ri**. Le contenu de cette adresse est étendu avec des zéro et est ensuite placé dans le registre **$rr**.

opération

$rr \leftarrow 0^{24}$ || mem$[imm + ri]_{7...0}$

- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
- Mémoire inexistante à l'adresse de chargement.

lh

action

Lecture d'un demi-mot signé de la mémoire

syntaxe

```
lh $rr, imm($ri)
```

description

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre **$ri**. Le contenu de cette adresse subit une extension de signe et est ensuite placé dans le registre **$rr**. Attention, le bit de poids faible de l'adresse résultante doit être à zéro.

opération

$rr \leftarrow$ mem$[imm + ri]_{15}^{16}$ || mem$[imm + ri]_{15...0}$

exception

- Adresse non alignée sur une frontière de demi-mot. ;
- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
- Mémoire inexistante à l'adresse de chargement.

lhu

action

Lecture d'un demi-mot non-signé de la mémoire

syntaxe

```
lhu $rr, imm($ri)
```

description

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre **$ri**. Le contenu de cette adresse est étendu avec des zéro et est ensuite placé dans le registre **$rr**. Attention, le bit de poids faible de l'adresse résultante doit être à zéro.

opération

$rr \leftarrow 0^{16}$ || mem$[imm + ri]_{15...0}$

exception

- Adresse non alignée sur une frontière de demi-mot. ;
- Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;

– Mémoire inexistante à l'adresse de chargement.

lui

action

Lecture d'une constante dans les poids forts

syntaxe

```
lui $rr, imm
```

description

La constante immédiate de 16 bits est décalée de 16 bits à gauche, et est complétée de zéro. La valeur ainsi obtenue est placée dans **$rr**.

opération

$rr \leftarrow imm \parallel 0^{16}$

lw

action

Lecture d'un mot de la mémoire

syntaxe

```
lw $rr, imm($ri)
```

description

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre **$ri**. Le contenu de cette adresse est placé dans le registre **$rr**. Attention, les deux bits de poids faible de l'adresse résultante doivent être à zéro.

opération

$rr \leftarrow$ mem$[imm + ri]$

exception

– Adresse non alignée sur une frontière de mot ;
– Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
– Mémoire inexistante à l'adresse de chargement.

mfc0

action

Copie d'un registre spécialisé dans d'un registre général

syntaxe

```
mfc0 $rt, $rd
```

description

Le contenu du registre spécialisé **$rd** — non directement accessible au programmeur — est recopié dans le registre général **$rt**. Les registres possibles pour **$rd** servent à la gestion des exceptions et interuptions, et sont les suivants : **$8** pour **BAR** (*bad address register*), **$12** pour **SR** (*status register*), **$13** pour **CR** (*cause register*) et **$14** pour **EPC** (*exception program counter*).

opération

$rt \leftarrow$ copro$[rd]$

– Utilisation de l'instruction en mode utilisateur.

mfhi

action

Copie le registre **$hi** dans un registre général

syntaxe

```
mfhi $rr
```

description

Le contenu du registre spécialisé **$hi** — qui est mis à jour par l'opération de multiplication ou de division — est recopié dans le registre général **$rr**.

opération

$rr \leftarrow hi$

mflo

action

Copie le registre **$lo** dans un registre général

syntaxe

```
mflo $rr
```

description

Le contenu du registre spécialisé **$lo** — qui est mis à jour par l'opération de multiplication ou de division — est recopié dans le registre général **$rr**.

opération

$rr \leftarrow lo$

mtc0

action

Copie d'un registre général dans un registre spécialisé

syntaxe

```
mtc0 $rt, $rd
```

description

Le contenu du registre général **$rt** est recopié dans le registre spécialisé **$rd** — non directement accessible au programmeur —. Ces registres servent à la gestion des exceptions et interuptions, et sont les suivants : **$8** pour **BAR** (*bad address register*), **$12** pour **SR** (*status register*), **$13** pour **CR** (*cause register*) et **$14** pour **EPC** (*exception program counter*).

opération

copro$[rd] \leftarrow rt$

mthi

action

Copie d'un registre général dans le registre **$hi**

syntaxe

```
mthi $ri
```

description

Le contenu du registre général **$ri** est recopié dans le registre spécialisé **$hi**.

opération

$hi \leftarrow ri$

mtlo

action

Copie d'un registre général dans le registre **$lo**

syntaxe

```
mtlo $ri
```

description

Le contenu du registre général **$ri** est recopié dans le registre spécialisé **$lo**.

opération

$lo \leftarrow ri$

mult

action

Multiplication signé registre registre

syntaxe

```
mult $ri, $rj
```

description

Le contenu du registre **$ri** est multiplié par le contenu du registre **$rj**, le contenu des deux registres étant considéré comme des nombres en complément à deux. Les 32 bits de poids fort du résultat sont placés dans le registre **$hi**, et les 32 bits de poids faible dans **$lo**.

opération

$lo \leftarrow (ri \times rj)_{31...0}$

$hi \leftarrow (ri \times rj)_{63...32}$

multu

action

Multiplication signé registre registre

syntaxe

```
multu $ri, $rj
```

description

Le contenu du registre **$ri** est multiplié par le contenu du registre **$rj**, le contenu des deux registres étant considéré comme des nombres non-signés. Les 32 bits de poids fort du résultat sont placés dans le registre **$hi**, et les 32 bits de poids faible dans **$lo**.

opération

$lo \leftarrow (0 \parallel ri \times 0 \parallel rj)_{31...0}$

$hi \leftarrow (0 \parallel ri \times 0 \parallel rj)_{63...32}$

nor

action

Non-ou bit-à-bit registre registre

syntaxe

```
nor $rr, $ri, $rj
```

description

Un **non-ou** bit-à-bit est effectué entre les contenus des registres **$ri** et **$rj**. Le résultat est placé dans le registre **$rr**.

opération

$rr \leftarrow ri$ nor rj

or

action

Ou bit-à-bit registre registre

syntaxe

```
or $rr, $ri, $rj
```

description

Un **ou** bit-à-bit est effectué entre les contenus des registres **$ri** et **$rj**. Le résultat est placé dans le registre **$rr**.

opération

$rr \leftarrow ri$ or rj

ori

action

Ou bit-à-bit registre immédiat

syntaxe

```
ori $rr, $ri, imm
```

description

La valeur immédiate sur 16 bits subit une extension de zéros. Un **ou** bit-à-bit est effectué entre cette valeur étendue et le contenu du registre **$ri** pour former un résultat placé dans le registre **$rr**.

opération

$rr \leftarrow (0^{16} \parallel imm)$ or ri

rfe

action

Restauration des bits d'état en fin d'exception

syntaxe

```
rfe
```

description

Recopie les anciennes valeurs des bits de masques d'interruption et de mode (noyau ou utilisateur) du registre d'état — un des registres spécialisé — à la valeur qu'il avait avant l'exécution du programme d'exception courant.

opération

$sr \leftarrow sr_{31...4} \parallel sr_{5...2}$

sb

action

Écriture d'un octet en mémoire

syntaxe

```
sb $rj, imm($ri)
```

description

L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre **$ri**. L'octet de poids faible du registre **$rj** est écrit à l'adresse ainsi calculée.

opération

mem$[imm + ri] \leftarrow rj_{7...0}$

– Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
– Mémoire inexistante à l'adresse de chargement.

sh

action

Écriture d'un demi-mot en mémoire

syntaxe

```
sh $rj, imm($ri)
```

description

L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre **$ri**. Les deux octets de poids faible du registre **$rj** sont écrit à l'adresse ainsi calculée. Le bit de poids faible de cette adresse doit être à zéro.

opération

mem$[imm + ri] \leftarrow rj_{15...0}$

exception

– Adresse non alignée sur une frontière de demi-mot ;
– Adresse de chargement en segment noyau alors que le code tourne avec le bit utilisateur ;
– Mémoire inexistante à l'adresse de chargement.

sll

action

Décalage à gauche immédiat

syntaxe

```
sll $rr, $ri, imm
```

description

Le registre **$ri** est décalé à gauche de la valeur immédiate codée sur 5 bits, des zéros étant introduits dans les bits de poids faibles. Le résultat est placé dans le registre **$rr**.

opération

$rr \leftarrow ri_{31-imm...0} \parallel 0^{imm}$

sllv

action

Décalage à gauche registre

syntaxe

```
sllv $rr, $ri, $rj
```

description

Le registre **$ri** est décalé à gauche du nombre de bits spécifiés dans les 5 bits de poids faible du registre **$rj**, des zéros étant introduits dans les bits de poids faibles. Le résultat est placé dans le registre **$rr**.

opération

$$rr \leftarrow ri_{31-rj_{4\ldots0}\ldots0} \parallel 0^{rj_{4\ldots0}}$$

slt

action

Comparaison signée registre registre

syntaxe

```
slt $rr, $ri, $rj
```

description

Le contenu du registre **$ri** est comparé au contenu du registre **$rj**, les deux valeurs étant considérées comme des quantités signées. Si la valeur contenue dans **$ri** est inférieure à celle contenue dans **$rj**, alors **$rr** prend la valeur un, sinon il prend la valeur zéro.

opération

$$ri < rj \Rightarrow rr \leftarrow 0^{31} \parallel 1$$
$$ri \geq rj \Rightarrow rr \leftarrow 0^{32}$$

slti

action

Comparaison signée registre immédiat

syntaxe

```
slti $rr, $ri, imm
```

description

Le contenu du registre **$ri** est comparé à la valeur immédiate sur 16 bits qui à subit une extension de signe. Les deux valeurs étant considérées comme des quantités signées, si la valeur contenue dans **$ri** est inférieure à celle de l'immédiat étendu, alors **$rr** prend la valeur un, sinon il prend la valeur zéro.

opération

$$ri < imm^{16}_{15} \parallel imm \Rightarrow rr \leftarrow 0^{31} \parallel 1$$
$$ri \geq imm^{16}_{15} \parallel imm \Rightarrow rr \leftarrow 0^{32}$$

sltiu

action

Comparaison non-signée registre immédiat

syntaxe

```
sltiu $rr, $ri, imm
```

description

Le contenu du registre **$ri** est comparé à la valeur immédiate sur 16 bits qui à subit une extension de signe. Les deux valeurs étant considérées comme des quantités non-signées, si la valeur contenue dans **$ri** est inférieur à celle de l'immédiat étendu, alors **$rr** prend la valeur un, sinon il prend la valeur zéro.

opération

$0 \parallel ri < 0 \parallel imm_{15}^{16} \parallel imm \Rightarrow rr \leftarrow 0^{31} \parallel 1$
$0 \parallel ri \geq 0 \parallel imm_{15}^{16} \parallel imm \Rightarrow rr \leftarrow 0^{32}$

sltu

action

Comparaison non-signée registre registre

syntaxe

```
sltu $rr, $ri, $rj
```

description

Le contenu du registre **$ri** est comparé au contenu du registre **$rj**, les deux valeurs étant considérés comme des quantités non-signées. Si la valeur contenue dans **$ri** est inférieur à celle contenue dans **$rj**, alors **$rr** prend la valeur un, sinon il prend la valeur zéro.

opération

$0 \parallel ri < 0 \parallel rj \Rightarrow rr \leftarrow 0^{31} \parallel 1$
$0 \parallel ri \geq 0 \parallel rj \Rightarrow rr \leftarrow 0^{32}$

sra

action

Décalage à droite arithmétique immédiat

syntaxe

```
sra $rr, $ri, imm
```

description

Le registre **$ri** est décalé à droite de la valeur immédiate codée sur 5 bits, le bit de signe du registre étant introduit dans les bits de poids fort. Le résultat est placé dans le registre **$rr**.

opération

$rr \leftarrow ri_{31}^{imm} \parallel ri_{31...imm}$

srav

action

Décalage à droite arithmétique registre

syntaxe

```
srav $rr, $ri, $rj
```

description

Le registre **$ri** est décalé à droite du nombre de bits spécifiés dans les 5 bits de poids faible du registre **$rj**, le signe de **$ri** étant introduit dans les bits de poids fort ainsi libérés. Le résultat est placé dans le registre **$rr**.

opération

$rr \leftarrow ri_{31}^{rj_{4...0}} \parallel ri_{31...rj_{4...0}...0}$

srl

action

Décalage à droite logique immédiat

syntaxe

```
srl $rr, $ri, imm
```

description

Le registre **$ri** est décalé à droite de la valeur immédiate codée sur 5 bits, des zéros étant introduits dans les bits de poids fort. Le résultat est placé dans le registre **$rr**.

opération

$rr \leftarrow 0^{imm} \parallel ri_{31...imm}$

srlv

action

Décalage à droite logique registre

syntaxe

```
srlv $rr, $ri, $rj
```

description

Le registre **$ri** est décalé à droite du nombre de bits spécifiés dans les 5 bits de poids faible du registre **$rj**, des zéros étant introduits dans les bits de poids fort ainsi libérés. Le résultat est placé dans le registre **$rr**.

opération

$rr \leftarrow 0^{rj_{4...0}} \parallel ri_{31...rj_{4...0}...0}$

sub

action

Soustraction registre registre signée

syntaxe

```
sub $rr, $ri, $rj
```

description

Le contenu du registre **$rj** est soustrait du contenu du registre **$ri** pour former un résultat sur 32 bits qui est placé dans le registre **$rr**.

opération

$rr \leftarrow ri - rj$

exception

génération d'une exception si dépassement de capacité.

subu

action

Soustraction registre registre non-signée

syntaxe

```
sub $rr, $ri, $rj
```

description

Le contenu du registre **$rj** est soustrait du contenu du registre **$ri** pour former un résultat sur 32 bits qui est placé dans le registre **$rr**.

opération

$rr \leftarrow ri - rj$

sw

action

Écriture d'un mot en mémoire

syntaxe

```
sw $rj, imm($ri)
```

description

L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre **$ri**. Le contenu du registre **$rj** est écrit à l'adresse ainsi calculée. Les deux bits de poids faible de cette adresse doivent être à zéro.

opération

mem$[imm + ri] \leftarrow rj$

exception

Adresse non alignée sur une frontière de mot.

syscall

action

Appel à une fonction du système (en mode noyau).

syntaxe

```
syscall
```

description

Un appel système est effectué, transférant immédiatement, et inconditionnellement le contrôle au gestionnaire d'exception. Note : par convention, le numéro de l'appel système, c.-à-d. le code de la fonction système à effectuer, est placé dans le registre **$2**.

opération

$pc \leftarrow 0x80000080$

exception

Déclenchement d'une exception de type appel système.

xor

action

Ou-exclusif bit-à-bit registre registre

syntaxe

```
xor $rr, $ri, $rj
```

description

Un **ou-exclusif** bit-à-bit est effectué entre les contenus des registres **$ri** et **$rj**. Le résultat est placé dans le registre **$rr**.

opération

$rr \leftarrow ri$ xor rj

xori

action

Ou-exclusif bit-à-bit registre immédiat

syntaxe

```
xori $rr, $ri, imm
```

description

La valeur immédiate sur 16 bits subit une extension de zéros. Un **ou-exclusif** bit-à-bit est effectué entre cette valeur étendue et le contenu du registre **$ri** pour former un résultat placé dans le registre **$rr**.

opération

$rr \leftarrow (\mathbf{0^{16}} \parallel imm)$ xor ri

E) MACRO-INSTRUCTIONS

Une macro-instruction est une pseudo-instruction qui ne fait pas partie du jeu d'instructions machine, mais qui est acceptée par l'assembleur qui la traduit en une séquence d'instructions machine. Les macro-instructions utilisent le registre **$1** si elles ont besoin de faire un calcul intermédiaire. Il faut donc éviter d'utiliser ce registre dans les programmes.

bge

action

Branchement si registre plus grand ou égal que registre

syntaxe

```
bge $ri, $rj, label
```

description

Les contenus des registres **$ri** et **$rj** sont comparés. Si **$ri** $\geq$ **$rj**, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$ri \geq rj \Rightarrow pc \leftarrow pc + 4 + addr$

code équivalent

```
slt     $1, $ri, $rj
beq     $1, $0,  label
```

bgt

action

Branchement si registre strictement plus grand que registre

syntaxe

```
bgt $ri, $rj, label
```

description

Les contenus des registres **$ri** et **$rj** sont comparés. Si **$ri** $>$ **$rj**, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$ri > rj \Rightarrow pc \leftarrow pc + 4 + addr$

code équivalent

```
slt     $1, $rj, $ri
bne     $1, $0,  label
```

ble

action

Branchement si registre plus petit ou égal à registre

syntaxe

```
ble $ri, $rj, label
```

description

Les contenus des registres **$ri** et **$rj** sont comparés. Si **$ri** $\leq$ **$rj**, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$ri \leq rj \Rightarrow pc \leftarrow pc + 4 + addr$

code équivalent

```
slt     $1, $rj, $ri
beq     $1, $0,  label
```

blt

action

Branchement si registre strictement plus petit que registre

syntaxe

```
blt $ri, $rj, label
```

description

Les contenus des registres **$ri** et **$rj** sont comparés. Si **$ri** $<$ **$rj**, le programme saute à l'adresse correspondant à l'étiquette, calculée par l'assembleur.

opération

$addr \leftarrow label$

$ri < rj \Rightarrow pc \leftarrow pc + 4 + addr$

code équivalent

```
slt     $1, $ri, $rj
bne     $1, $0,  label
```

div

action

Division entière et reste signé registre registre

syntaxe

```
div $rr, $ri, $rj
```

description

Le contenu du registre **$ri** est divisé par le contenu du registre **$rj**, le contenu des deux registres étant considéré comme des nombres en complément à deux. Le résultat de la division est placé dans le registre **$rr**.

opération

$rr \leftarrow \lfloor \frac{ri}{rj} \rfloor$

code équivalent

```
div     $ri, $rj
mflo    $rr
```

divu

action

Division entière non-signé registre registre

syntaxe

```
divu $rr, $ri, $rj
```

description

Le contenu du registre **$ri** est divisé par le contenu du registre **$rj**, le contenu des deux registres étant considéré comme des nombres non signés. Le résultat de la division est placé dans le registre spécial **$lo**, et le reste dans **$hi**. Le contenu de **$lo** est recopié dans **$rr**.

opération

$rr \leftarrow \lfloor 0 \parallel ri0 \parallel rj \rfloor$

code équivalent

```
divu    $ri, $rj
mflo    $rr
```

la

action

Chargement d'une adresse dans un registre

syntaxe

```
la $rr, adr
```

description

L'adresse considérée comme une quantité non-signée est chargée dans le registre **$rr**.

opération

$rr \leftarrow adr$

code équivalent calcul de **adr** par l'assembleur, puis :

```
lui     $rr,      adr >> 16
ori     $rr, $rr, adr & 0xFFFF
```

li

action

Chargement d'un immédiat sur 32 bits dans un registre

syntaxe

```
li $rr, imm
```

description

La valeur immédiate non-signée est chargée dans le registre **$rr**.

opération

$rr \leftarrow imm$

code équivalent

```
lui     $rr,      imm >> 16
ori     $rr, $rr, imm & 0xFFFF
```

lb

action

Chargement de l'octet, étendu de signe, contenu à une addresse indiquée par une étiquette dans un registre.

syntaxe

```
lb $rr, etiq
```

description

L'octet présent en mémoire à l'adresse **etiq** est étendu de signe et chargé dans le registre **$rr**.

opération

$rr \leftarrow$ mem$[etiq]$

code équivalent

```
la     $1,       etiq   # Utilisation de la macro la
lb     $rr,     0($1)
```

lbu

action

Chargement de l'octet, considéré comme une entité non signée, contenu à une addresse indiquée par une étiquette dans un registre.

syntaxe

```
lbu $rr, etiq
```

description

L'octet présent en mémoire à l'adresse **etiq** est chargé dans le registre **$rr**. Les 3 octets de poids forts sont mis à 0.

opération

$rr \leftarrow$ mem[$etiq$]

code équivalent

```
la     $1,       etiq   # Utilisation de la macro la
lbu    $rr,     0($1)
```

lh

action

Chargement du demi-mot, étendu de signe, contenu à une addresse indiquée par une étiquette dans un registre.

syntaxe

```
lh $rr, etiq
```

description

Le demi-mot présent en mémoire à l'adresse **etiq** est étendu de signe et chargé dans le registre **$rr**.

opération

$rr \leftarrow$ mem[$etiq$]

code équivalent

```
la     $1,       etiq   # Utilisation de la macro la
lh     $rr,     0($1)
```

lhu

action

Chargement du demi-mot, considéré comme une entité non signée, contenu à une addresse indiquée par une étiquette dans un registre.

syntaxe

```
lhu $rr, etiq
```

description

Le demi-mot présent en mémoire à l'adresse **`etiq`** est chargé dans le registre **`$rr`**. Les 2 octets de poids forts sont mis à 0.

opération

$rr \leftarrow$ mem$[etiq]$

code équivalent

```
la     $1,       etiq   # Utilisation de la macro la
lhu    $rr,     0($1)
```

lw

action

Chargement du mot contenu à une addresse indiquée par une étiquette dans un registre

syntaxe

```
lw $rr, etiq
```

description

Le mot de 32 bits présent en mémoire à l'adresse **`etiq`** est chargée dans le registre **`$rr`**.

opération

$rr \leftarrow$ mem$[etiq]$

code équivalent

```
la     $1,       etiq   # Utilisation de la macro la
lw     $rr,     0($1)
```

move

action

transfert registre/registre

syntaxe

```
move $rd, $rs
```

description

Le mot contenu dans le registre **$rs** est copié dans le registre **$rd**.

opération

$rd \leftarrow rs$

code équivalent

```
addu     $rd, $rs, $0
```

mult

action

Multiplication signé registre registre

syntaxe

```
mult $rr, $ri, $rj
```

description

Le contenu du registre **$ri** est multiplié par le contenu du registre **$rj**, le contenu des deux registres étant considéré comme des nombres en complément à deux. Les 32 bits de poids faible sont placés dans **$rr**.

opération

$rr \leftarrow (ri \times rj)_{31...0}$

code équivalent

```
mult $ri, $rj
mflo $rr
```

multu

action

Multiplication signé registre registre

syntaxe

```
multu $rr, $ri, $rj
```

description

Le contenu du registre **$ri** est multiplié par le contenu du registre **$rj**, le contenu des deux registres étant considéré comme des nombres non-signés. Les 32 bits de poids faible sont placés dans **$rr**.

opération

$rr \leftarrow (0 \parallel ri \times 0 \parallel rj)_{31...0}$

code équivalent

```
multu $ri, $rj
mflo  $rr
```

sb

action

Écriture de l'octet de droite d'un registre à une addresse indiquée par une étiquette

syntaxe

```
sb $rr, etiq
```

description

L'octet le plus à droite du registre **`$rr`** est écrit en mémoire à l'adresse **`etiq`**.

opération

mem$[etiq_{31...2}]_{etiq_{1...0}} \leftarrow rr_{7...0}$

code équivalent

```
la     $1,       etiq   # Utilisation de la macro la
sb     $rr,     0($1)
```

sh

action

Écriture du demi-mot (16 bits) de droite d'un registre à une addresse indiquée par une étiquette

syntaxe

```
sh $rr, etiq
```

description

Le demi-mot le plus à droite du registre **`$rr`** est écrit en mémoire à l'adresse **`etiq`**.

opération

mem$[etiq_{31...1}]_{etiq_0} \leftarrow rr_{16...0}$

code équivalent

```
la     $1,       etiq   # Utilisation de la macro la
sh     $rr,     0($1)
```

sw

action

Écriture d'un registre à une addresse indiquée par une étiquette

syntaxe

```
sw $rr, etiq
```

description

Le mot de 32 bits présent dans le registre **`$rr`** est écrit en mémoire à l'adresse **`etiq`**.

opération

mem$[etiq] \leftarrow rr$

code équivalent

```
la     $1,       etiq   # Utilisation de la macro la
sw     $rr,     0($1)
```

Les macro-instructions ci-dessus décritent servent à simplifier l'écriture des programmes, et à donner au programmeur (ou compilateur) une vision homogène de l'assembleur. En revanche, l'instruction **`la`** doit forcément être utilisée lors du chargement d'une adresse car l'assembleur ne connaît l'adresse finale d'une instruction ou d'une donnée qu'après avoir assemblé l'ensemble du code.

F) DIRECTIVES SUPPORTÉES PAR L'ASSEMBLEUR MIPS

Les directives ne sont pas des instructions exécutables par la machine, mais permettent de donner des ordres à l'assembleur. Toutes les pseudo-instruction commencent par le caractère << **.** >> ce qui permet de les différencier clairement des instructions.

1) Déclaration des sections :`text`, `data` et `stack`

Six directives permettent de spécifier quelle section de la mémoire est concernée par les instructions, macro-instructions ou directives qui les suivent. Sur ces six directives, deux sont dynamiquement gérées à l'exécution : ce sont celles qui concernent la pile utilisateur, **`stack`**, et la pile système, **`kstack`**. Ceci signifie que l'assembleur gère quatre compteurs d'adresse indépendants correspondants aux quatre sections **`text`**, **`data`**, **`ktext`** et **`kdata`**.

`.text`

action
: Passage dans la section **`text`**

syntaxe
: `.text`

description
: Toutes les instructions et directives qui suivent concernent la section **`text`** dans le segment utilisateur.

`.data`

action
: Passage dans la section **`data`**

syntaxe
: `.data`

description
: Toutes les instructions et directives qui suivent concernent la section **`data`** dans le segment utilisateur.

`.stack`

action
: Passage dans la section **`stack`**

syntaxe

`.stack`

description

Toutes les instructions et directives qui suivent concernent la section **`stack`** dans le segment utilisateur.

`.ktext`

action

Passage dans la section **`ktext`**

syntaxe

`.ktext`

description

Toutes les instructions et directives qui suivent concernent la section **`ktext`** dans le segment noyau.

`.kdata`

action

Passage dans la section **`kdata`**

syntaxe

`.kdata`

description

Toutes les instructions et directives qui suivent concernent la section **`kdata`** dans le segment noyau.

`.kstack`

action

Passage dans la section **`kstack`**

syntaxe

`.kstack`

description

Toutes les instructions et directives qui suivent concernent la section **`stack`** dans le segment noyau.

2) Déclaration et initialisation de variables

Les directives suivantes permettent de d'initialiser certaines zones dans les sections **`text`** ou **`data`** de la mémoire.

`.align` *expression*

action

Aligne le compteur d'adresse courant afin que *expression* bits de poids faible soient à zéro.

syntaxe

`align num`

description

Cet opérateur aligne le compteur d'adresse sur une adresse telle que les n bits de poids faible soient à zéro. Cette opération est effectuée implicitement pour aligner correctement les instructions, demi-mots et mots.

exemple

```
.align 2
.byte 12
.align 2
.byte 24
```

`.ascii`

action

Déclare et initialise une chaîne de caractères

syntaxe

`.ascii` *chaîne, [chaîne,]* `...`

description

Cet opérateur place à partir de l'adresse du compteur d'adresse correspondant à la section active la suite de caractères entre guillemets. S'il y a plusieurs chaînes, elles sont placées à la suite. Cette chaîne peut contenir des séquences d'échappement du langage C, et doit être terminée par un zéro binaire si elle est utilisé avec un appel système.

exemple

```
message:
        .ascii "Bonjour, Maître!\n\0"
```

`.asciiz`

action

Déclare et initialise une chaîne de caractères, en ajoutant un zéro binaire à la fin.

syntaxe

`.asciiz` *chaîne, [chaîne,]* `...`

description

Cet opérateur est strictement identique au précédent, la seule différence étant qu'il ajoute un zéro binaire à la fin de chaque chaîne.

exemple `message:`

```
            .asciiz "Bonjour, Maître!\n"
```

.byte

action

Positionne des octets successifs aux valeurs des expressions.

syntaxe

`.byte` *expression, [expression,]* `...`

description

La valeur de chacunes des expressions est tronquée à 8 bits, et les valeurs ainsi obtenues sont placées à des adresses successives de la section active.

exemple

```
table:
     .byte 1, 2, 4, 8, 16, 32, 64, 32, 16, 8, 4, 2, 1
```

.half

action

Positionne des demi-mots successifs aux valeurs des expressions.

syntaxe

`.half` *expression, [expression,]* `...`

description

La valeur de chacunes des expressions est tronquée à 16 bits, et les valeurs ainsi obtenues sont placées dans des adresses successives de la section active.

exemple

```
coordonnées:
    .half 0  , 1024
```

```
.half 968, 1024
```

.word

action

Positionne des mots successifs aux valeurs des expressions.

syntaxe

`.word` *expression, [expression,]* `...`

description

La valeur de chaque expression est placée dans des adresses successives de la section active.

exemple

```
entiers:
    .word -1, -1000, -100000, 1, 1000, 100000
```

.space

action

Reserve *expression* octets, et les mets à zéro

syntaxe

`.space` *expression*

description

Un espace de taille *expression* octets est réservé à partir de l'adresse courante de la section active.

exemple

```
nuls:
    .space 1024 ; initialise 1 kilo de mémoire à zéro
```

G) APPELS SYSTÈME

Pour exécuter certaines fonctions système, typiquement les entrées/sorties (lire ou écrire un nombre, ou un caractère), il faut utiliser des appels système.

Par convention, le numéro de l'appel système est contenu dans le registre **$2**, et son unique argument dans le registre **$4**.

Cinq appels système sont actuellement supportés dans l'environnement de simulation :

écrire un entier :

Il faut mettre l'entier à écrire dans le registre **$4** et exécuter l'appel système numéro 1. Typiquement, on aura :

```
li      $4,     1234567  ; met 1234567 dans l'argument
ori     $2, $0, 1        ; code de 'print_integer'
syscall                  ; affiche 1234567
```

lire un entier :

La valeur de retour d'une fonction — système ou autre — est positionnée dans le registre **$2**. Ainsi, lire un entier consiste à exécuter l'appel système numéro 5 et récupérer le resultat dans le registre**$2**.

```
ori     $2, $0, 5        ; code de 'read_integer'
syscall                  ; $2 contient ce qui a été lu
```

écrire une chaîne de caractères :

Une chaîne de caractères étant identifiée par un pointeur, il faut passer ce pointeur à l'appel système numéro 4 pour l'afficher.

```
str: .asciiz "Chaîne à afficher\n"

     la      $4,     str      ; charge le pointeur dans $4
     ori     $2, $0, 4        ; code de 'print_string'
     syscall                  ; affiche la chaine pointée
```

lire une chaîne de caractères :

Pour lire une chaîne de caractères, il faut un pointeur et une longueur maximum. Il faut passer ce pointeur, dans **$4**, et cette longueur, dans **$5**, et exécuter l'appel système numéro 8. Le résultat sera mis dans l'espace pointé.

```
read_str:
     .space 256

     la      $4,     read_str ; charge le pointeur dans $4
     ori     $5, $0, 255      ; charge longueur max dans $5
     ori     $2, $0, 8        ; code de 'read_string'
     syscall                  ; lit la chaîne dans l'espace
                              ; pointé par $4
```

quitter :

L'appel système numéro 10 effectue l'**`exit`** du programme au sens du langage C.

```
     ori     $2, $0, 10       ; indique l'appel à exit
     syscall                  ; quitte pour de bon!
```

H) CONVENTIONS POUR LES APPELS DE FONCTIONS

L'exécution de fonctions nécessite une pile en mémoire. Cette pile correspond à la section **`stack`**. L'utilisation de cette pile fait l'objet de conventions qui doivent être respectées par la fonction appelée et par la fonction appelante.

– la pile s'étend vers les adresses décroissantes ;
– le pointeur de pile pointe *toujours* sur la dernière case occupée dans la pile. Ceci signifie que toutes les cases d'adresse inférieure au pointeur de pile sont libres ;
– le R3000 ne possède pas d'instructions spécifiques à la gestion de la pile. On utilise les instructions **`lw`** et **`sw`** pour y accéder.

Les appels de fonction utilisent un pointeur particulier, appelé pointeur de pile. Ce pointeur est stocké conventionnellement dans le registre **`$29`**. On le désigne aussi par la notation **`$sp`**. La valeur de retour d'une fonction est conventionellement présente dans le registre **`$2`**.

Par ailleurs, l'architecture du processeur MIPS R3000 impose l'utilisation du registre **`$31`** pour stocker l'adresse de retour lors d'un appel de fonction (instructions de type **`jal`** ou **`bgezal`**).

À chaque appel de fonction est associée une zone dans la pile constituant le << contexte d'exécution >> de la fonction. Dans le cas des fonctions récursives, une même fonction peut être appelée plusieurs fois et possèdera donc plusieurs contextes d'exécution dans la pile. Lors de l'entrée dans une fonction, les registres **`$5`** à **`$26`** sont disponibles pour tout calcul dans cette fonction. Dans le cas général, un contexte d'exécution d'une fonction est constitué de quatre zones qui sont, dans l'ordre d'empilement :

1. la zone de sauvegarde des registres de travail de la fonction appelante. La fonction, qui a appelée la fonction courante, a dû sauvegarder les registres qu'elle utilise et qu'elle ne veut pas voir modifiés. Dans la suite, nous nommerons ces registres << persistants >> par opposition aux registres << temporaires >> dont la modification n'a pas d'influence dans la suite de la fonction appelante. Ces derniers n'ont pas à être sauvegardé. Par convention, on positionne toujours le registre persistant d'index le plus petit à l'adresse la plus petite ;
2. la zone de sauvegarde de l'adresse de retour à la fonction appelante (adresse à laquelle la fonction appelée doit revenir lors de sa terminaison) ;
3. la zone des arguments de la fonction appelée. Les valeurs des arguments sont écrites dans la pile par la fonction appelante et lus dans la pile par la fonction appelée. Par convention, on positionne toujours le premier argument de la fonction appelée à l'adresse la plus petite ;

4. la zone des variables locales à la fonction appelé.

1) Organisation de la pile

Dans le cas général, une fonction f, utilisant des registres persistants, souhaite appeler une fonction g avec des arguments et qui possède des variables locales.

On appelle r le nombre de registres persistants dans f, a le nombre d'argument de g, et l le nombre de variables locales de g.

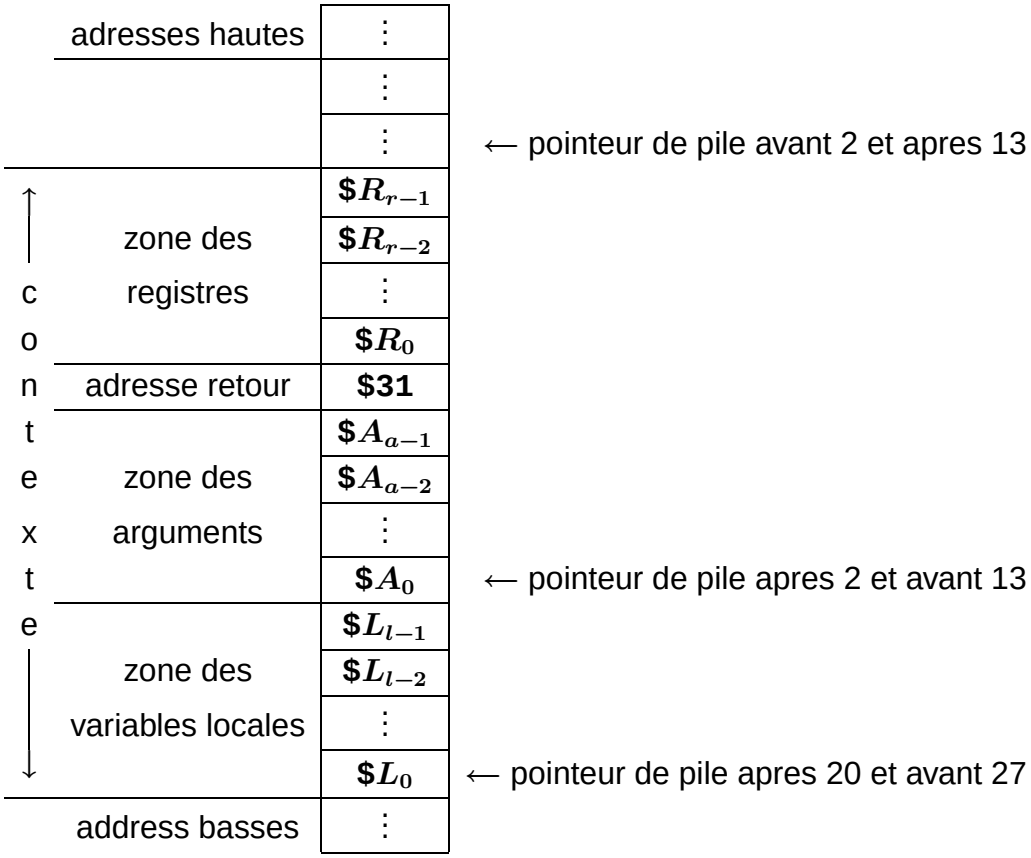
On note R_0, R_{r-1} les registres persistants. On note A_0, A_{a-1} les registres utilisés dans l'appelant pour le calcul des arguments qui peuvent être persistants ou non. On note P_0, P_{a-1} les registres utilisés dans l'appelé pour la récupération des arguments qui peuvent être persistants ou non.

Le code à écrire, dont on impose la structure, est le suivant. Dans f :

```
 1   ...
 2   addiu $sp, $sp, -(r + a + 1) × 4
 3   sw    $R0,  (a + 1) × 4($sp)
 4   ...
 5   sw    $Rr-1,  (a + r) × 4($sp)
 6   sw    $A0,  0($sp)
 7   ...
 8   sw    $Aa-1,  (a - 1) × 4($sp)
 9   jal   g
10   lw    $R0,  (a + 1) × 4($sp)
11   ...
12   lw    $Rr-1,  (a + r) × 4($sp)
13   addiu $sp, $sp, +(r + a + 1) × 4
14   ...
```

Dans g :

```
20   addiu $sp, $sp, -l × 4
21   sw    $31,  (l + a) × 4($sp)
22   lw    $P0, l × 4($sp)
23   ...
24   lw    $Pa-1, (l + a - 1) × 4($sp)
25   ...
26   lw    $31,  (l + a) × 4($sp)
27   addiu $sp, $sp, l × 4
28   jr $31
```



2) Énumération des différents cas d'utilisation du pointeur de pile

Pour être tout à fait clair, nous allons passer en revue tous les cas possibles d'appel d'une fonction g par une fonction f, suivant que f utilise ou non des registres persistants, que g a des arguments ou non, et que g a des variables locales ou non. Il y a huit cas possibles. Notez que les numéros de registres dans les exemples ont été choisi arbitrairement.

– f n'utilise pas de registres persistants et appelle g qui n'a pas d'arguments ni de variables locales.

Dans f :

```
...
# pour sauver $31
addiu $sp, $sp, -4
jal g
addiu $sp, $sp, +4
...
```

Dans g :

```
sw $31, 0($sp)
...
lw $31, 0($sp)
jr $31
```

– f utilise des registres persistants (2 dans l'exemple) et appelle g qui n'a pas d'arguments ni de variables locales.

Dans f :

```
...
# pour sauver $31, $5, $6
addiu $sp, $sp, -12
sw    $5,  4($sp)
sw    $6,  8($sp)
jal g
lw    $5,  4($sp)
lw    $6,  8($sp)
addiu $sp, $sp, +12
...
```

Dans g :

```
sw $31, 0($sp)
...
lw $31, 0($sp)
jr $31
```

– f n'utilise pas de registres persistants et appelle g qui a des arguments (3 dans l'exemple) et pas de variables locales. Les registres 5, 6, et 7 sont des registres temporaires utilisés uniquement pour la construction des arguments.

Dans f :

```
...
# pour sauver $31
# et passer trois arguments
addiu $sp, $sp, -16
sw    $5, 0($sp)
sw    $6, 4($sp)
sw    $7, 8($sp)
jal g
addiu $sp, $sp, +16
...
```

Dans g :

```
sw $31, 12($sp)
lw $10, 0($sp)
lw $17, 4($sp)
lw $23, 8($sp)
...
lw $31, 12($sp)
jr $31
```

– f utilise des registres persistants (2 dans l'exemple) et appelle g qui a des arguments (3 dans l'exemple) et pas de variables locales. Les registres 11 et 12 sont les registres persistants. Les registres 5, 6, et 7 sont des registres temporaires utilisés uniquement pour la construction des arguments.

Dans f :

```
...
addiu $sp, $sp, -24
sw     $5, 0($sp)
sw     $6, 4($sp)
sw     $7, 8($sp)
sw    $11, 16($sp)
sw    $12, 20($sp)
jal g
lw    $11, 16($sp)
lw    $12, 20($sp)
addiu $sp, $sp, +24
...
```

Dans g :

```
sw $31, 12($sp)
lw $10, 0($sp)
lw $11, 4($sp)
lw $12, 8($sp)
...
lw $31, 12($sp)
jr $31
```

– f n'utilise pas de registres persistants et appelle g qui n'a pas d'arguments mais qui a des variables locales (4 ici).

Dans f :

```
...
addiu $sp, $sp, -4
jal g
addiu $sp, $sp, +4
...
```

Dans g :

```
# pour 4 variables entières
addiu $sp, $sp, -16
sw $31, 16($sp)
...
lw $31, 16($sp)
addiu $sp, $sp, +16
jr $31
```

– f utilise des registres persistants (2 dans l'exemple) et appelle g qui n'a pas d'arguments mais qui a des variables locales (4 ici).

Dans f :

```
...
addiu $sp, $sp, -12
sw    $5,  4($sp)
sw    $6,  8($sp)
jal g
lw    $5,  4($sp)
lw    $6,  8($sp)
addiu $sp, $sp, +12
...
```

Dans g :

```
addiu $sp, $sp, -16
sw $31, 16($sp)
...
lw $31, 16($sp)
addiu $sp, $sp, +16
jr $31
```

– f n'utilise pas de registres persistents et appelle g qui a des arguments (3 dans l'exemple) et des variables locales (4 ici). Les registres 5, 6, et 7 sont des registres temporaires utilisés uniquement pour la construction des arguments.

Dans f :

```
...
# 3 registres + $31
addiu $sp, $sp, -16
sw    $5, 0($sp)
sw    $6, 4($sp)
sw    $7, 8($sp)
jal g
addiu $sp, $sp, +16
...
```

Dans g :

```
addiu $sp, $sp, -16
# 28($sp) car 3 arguments
# et 4 variables locales
sw    $31, 28($sp)
lw    $20, 16($sp)
lw    $21, 20($sp)
lw    $23, 24($sp)
...
lw    $31, 28($sp)
addiu $sp, $sp, +16
jr    $31
```

– f utilise des registres persistants (2 dans l'exemple) et appelle g qui a des arguments (3 dans l'exemple) et des variables locales (4 ici). Les registres 11 et 12 sont les registres persistants. Les registres 5, 6, et 7 sont des registres temporaires utilisés uniquement pour la construction des arguments.

Dans f :

```
...
# de la place pour $31
# 3 arguments et 2 registres
addiu $sp, $sp, -24
# pour passer 3 arguments
sw     $5, 0($sp)
sw     $6, 4($sp)
sw     $7, 8($sp)
# et 2 registres
sw    $11, 16($sp)
sw    $12, 20($sp)
jal g
lw    $11, 16($sp)
lw    $12, 20($sp)
addiu $sp, $sp, +24
...
```

Dans g :

```
addiu $sp, $sp, -16
# sauver $31
sw    $31, 28($sp)
lw    $20, 16($sp)
lw    $21, 20($sp)
lw    $23, 24($sp)
...
lw    $31, 28($sp)
addiu $sp, $sp, +16
jr    $31
```

3) Exemple

Exemple d'une fonction calculant une distance euclidienne, en supposant qu'il existe une fonction **`int isqrt(int x);`** qui retourne la racine carrée d'un nombre entier.

En C :

```
int main()
{
   eudistance(5, 4);
}

int eudistance(int a, int b)
{
int somme, distance; /* Variables locales */

   somme = a * a + b * b;
   distance = isqrt(somme);
   return distance;
}
```

Le programme assembleur est le suivant (**sans optimisations**) : **`main`** n'est pas une fonction comme les autres puisqu'elle n'est appelée par personne : On ne se préoccupe pas de sauver d'adresse de retour.

```
     .text
     .globl      main
main :
      addiu       $sp, $sp, -12
      # Met les paramètres dans la pile
      li          $5, 4
      sw          $5, 0($29)
      li          $5, 5
      sw          $5, 4($29)
      jal         eudistance
      addiu       $sp, $sp, +12

      # Sortie << propre >> avec exit
      ori         $2, $0, 10
      syscall

eudistance :

      addiu       $sp, $sp, -8
      # Sauve l'adresse de retour
      sw          $31,      +16($sp)

      # Récupération des paramètres
      lw          $5,        8($sp) # 1 parametre, valeur 4 ici
      lw          $6,       12($sp) # 2 parametre, 5 ici, si !
```

```
# Posons par convention
#  0($sp) est somme
#  4($sp) est distance

# Utilisation du premier paramètre
mult        $5, $5
mflo        $3

# Utilisation du second paramètre
mult        $6, $6
mflo        $4

# a * a + b * b rangé dans somme
addu        $4, $3,  $4
sw          $4,       0($sp)

# $5 et $6 ne sont pas des registres permanents puisqu'ils ne
# seront pas utilisés lors du retour de la fonction isqrt
# Il n'est donc pas utile de les sauver

# Met à jour le pointeur de pile en fonction de la taille du contexte
addiu       $sp, $sp, -8 # 1 argument + $31
sw          $4,  0($sp)
jal         isqrt
addiu       $sp, $sp, 8

# Range la valeur calculée par isqrt dans distance
sw          $2,        4($sp)

# Prépare la valeur à retourner
lw          $2,        4($sp)

# Restaure la valeur de l'adresse de retour
lw          $31,      -16($sp)

# Restaure le pointeur de pile
addiu       $sp,  $sp,  +8
# Retour à la fonction appelante
jr          $31
```