Processeur MIPS R3000

Langage d'assemblage

Version 2.2

(septembre 2004)

JeanLou Desbarbieux
François Dromard
Alain Greiner
Frédéric Pétrot
Franck Wajsburt

1/ 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 les 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 MIPS R3000 utilisé en TP qui permet de visualiser le comportement du processeur instruction par instruction.

2/ ORGANISATION DE LA MEMOIRE

Rappelons que le but d'un programme source *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 d'être exécuté par le processeur.

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 en mémoire 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 contenues dans cette section peuvent être initialisées grâce à des directives contenues dans le programme source en langage d'assemblage.

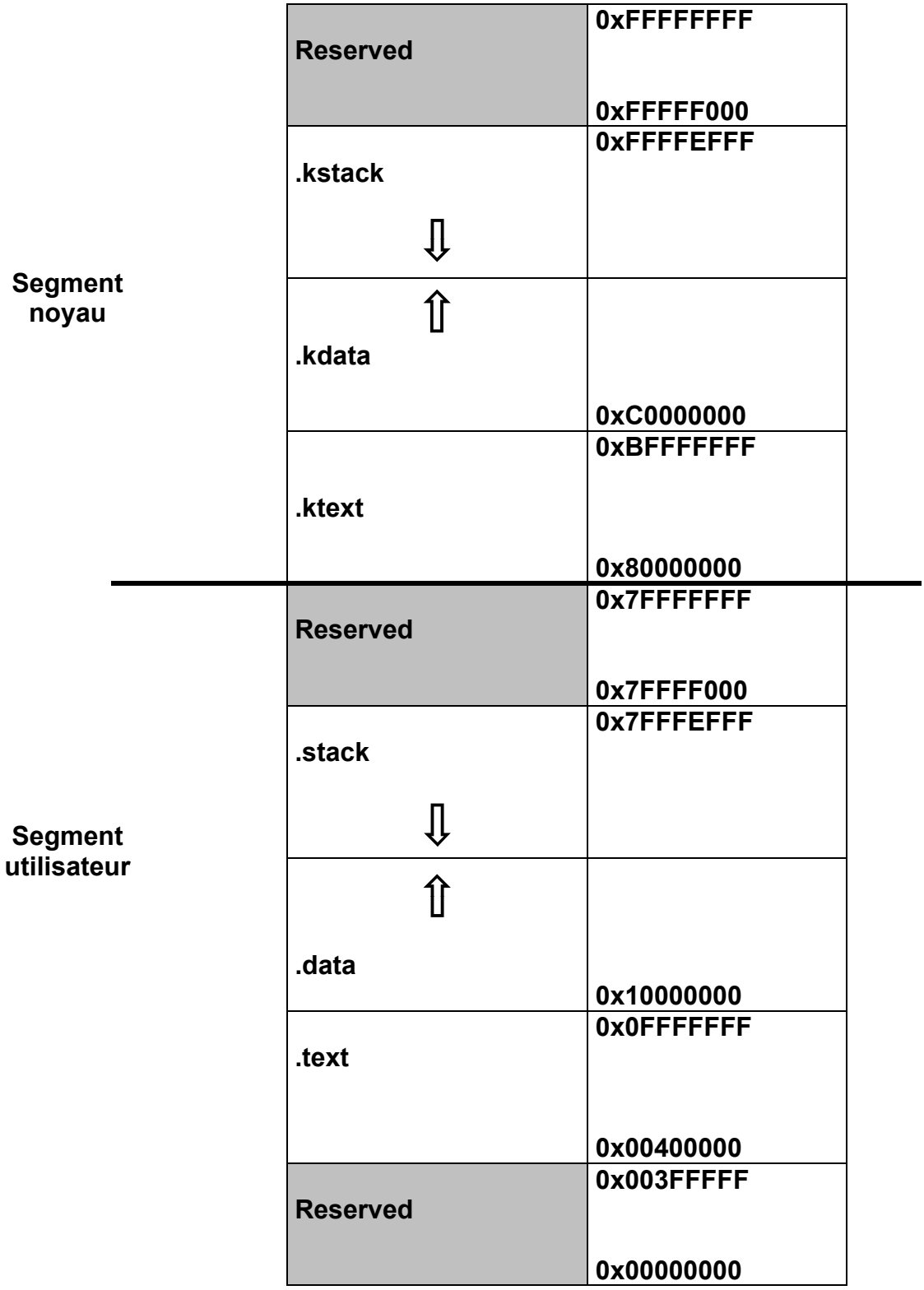
- la section **stack** contient la pile d'exécution du programme utilisateur. Sa taille varie au cours de l'exécution. Elle est implantée conventionnellement à l'adresse **0x7FFFF000**. La pile s'étend vers les adresses décroissantes.

Trois sections sont également définies dans le segment noyau (kernel):

- 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 noyau. Sa taille varie au cours de l'exécution. Elle est implantée conventionnellement à l'adresse **0xFFFFF000**. Cette pile s'étend vers les adresses décroissantes.


Segment noyau
Reserved
0xFFFFFFFF
0xFFFFF000
.kstack
0xFFFFEFFF
.kdata
0xC0000000
.ktext
0xBFFFFFFF
0x80000000
Segment utilisateur
Reserved
0x7FFFFFFF
0x7FFFF000
.stack
0x7FFFEFFF
.data
0x10000000
.text
0x0FFFFFFF
0x00400000
Reserved
0x003FFFFF
0x00000000

3/ RÈGLES SYNTAXIQUE

On définit ci-dessous les principales règles d'écriture d'un programme source.

3.1) les noms de fichiers

Les noms des fichiers contenant un programme source en langage d'assemblage doivent être suffixé par «.s». Exemple: **monprogramme.s**

3.2) les commentaires

ils commencent par un **#** et s'achèvent à la fin de la ligne courante.
Exemple :

```
        ################################################
        # Source Assembleur MIPS de la fonction memcpy
        ################################################
  ...
     lw     $10, 0($11)    # sauve la valeur copiée dans la mémoire
```

3.3) les entiers

Une valeur entière décimale est notée **250** (sans préfixe), et une valeur entière hexadécimale 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.

3.4) 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".

3.5) les labels

Ce sont des mnémoniques correspondant à des adresses en mémoire. Ces adresses peuvent être soit des adresses de variables stockées en mémoire (principalement dans la section **data**), soit des adresses de sauts (principalement dans la section **text**). Ce sont des chaînes de caractères qui doivent commencer par une lettre, un caractère « _ », ou un caractère « . ». Lors de la déclaration, ils doivent être suffixés par le caractère « : ». Pour y référer, on supprime le «:».
Exemple :

```
        .data
message :
        .asciiz  "Ceci est une chaîne de caractères...\n"
        .text
__start:
        lui     $4,     message >> 16
        ori     $4, $4, message & 0xFFFF  # adresse de la chaîne dans $4
        ori     $2,     $0,     4         # code de l'appel système dans $2
```

```
        syscall
```

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

3.6) les immédiats

On appelle immédiat un opérande contenu dans l'instruction. Ce sont des constantes. Ce sont soit des entiers, soit des labels. Les valeurs de ces constantes doivent respecter une taille maximum qui est fonction de l'instruction qui les utilise: 16 ou 26 bits.

3.7) les registres

Le processeur MIPS possède 32 registres de travail 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** qui contient toujours la valeur 0, tous les registres sont identiques du point de vue de la machine.
Afin de normaliser et de simplifier l'écriture du logiciel, des conventions d'utilisation des registres sont définies. Ces conventions sont particulièrement nécessaires lors des appels de fonctions.

$0	Vaut 0 en lecture. Non modifié par une écriture
$1	Réservé à l'assembleur pour les macros. Ne doit pas être utilisé
$2	Utilisé pour les valeurs de retour des fonctions
$3	Utilisé pour les appels systèmes.
$4,$5,$6,$7	Utilisés par le compilateur pour optimiser les appels de fonctions
$8,…,$26	Registres de travail utilisable par le code utilisateur
$27,$28	Réservés aux procédures noyau.
$29	Pointeur de pile
$30	Pointeur sur la zone des variables globales (section data)
$31	Contient l'adresse de retour d'une fonction

3.8) les arguments

La plupart des instructions nécessitent un ou plusieurs arguments. Si une instruction nécessite plusieurs arguments, ces arguments sont séparés par des virgules. Dans une instruction assembleur, on aura en général comme premier argument 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
```

3.9) l'adressage mémoire

Le MIPS ne possède qu'un unique mode d'adressage pour lire ou écrire des données en mémoire: l'adressage indirect registre avec déplacement. L'adresse est obtenue en additionnant le déplacement (positif ou négatif) au contenu du registre.

```
Exemples:   lw    $12,   13($10)     #  $12 <= Mem[$10 + 13]
            Sw    $20,   -60($22)    #  Mem[$22 – 60] <= $20
```

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

Pour ce qui concerne les sauts, il n'est pas possible d'écrire des sauts à des adresses constantes, comme par exemple un **j 0x400000** ou **bnez $3, -12**. Il faut nécessairement utiliser des labels.

4/ 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 des opérandes 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 spécifiée dans la description : de l'instruction.
Les instructions de saut prennent comme argument une étiquette (ou label), qui est utilisée pour calculer l'adresse de saut. Toutes les instructions modifient le registre **PC** (program counter), qui contient l'adresse de l'instruction suivante à exécuter. Enfin, le résultat d'une multiplication ou d'une division est rangé dans deux registres spéciaux, **HI** pour les poids forts, et **LO** pour les poids faibles.

Ceci nous amène à introduire quelques notations :

+	Addition entière en complément à 2
-	Soustraction entière en complément à 2
x	Multiplication entière en complément à 2
/	Division entière en complément à 2
mod	Reste de la division entière en complément à 2
and	Opérateur et logique bit à bit
or	Opérateur ou logique bit à bit
nor	Opérateur non-ou logique bit à bit
xor	Opérateur ou-exclusif logique bit à bit
Mem[ad]	Contenu de la mémoire à l'adresse ad
<=	Assignation
||	Concaténation entre deux chaînes de bits
B^n	Réplication du bit B n fois dans une chaîne de bits
$X_{p...q}$	Sélection des bits p à q dans une chaîne de bits X

Dans la description des instructions, pc représente l'adresse de l'instruction considérée.

add

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

rr <= ri + rj

Exception : génération d'une exception si dépassement de capacité.

addi

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.

rr <= (imm_{15}^{16} || $imm_{15...0}$) + ri

Exception : génération d'une exception si dépassement de capacité.

addiu

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.

rr <= (imm_{15}^{16} || $imm_{15...0}$) + ri

Exception : pas d'exception

addu

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

rr <= ri + rj

Exception : pas d'exception

and

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.

rr <= ri and rj

Exception : pas d'exception

andi

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.

rr <= (0^{16} || imm $_{15...0}$) and ri

Exception : pas d'exception

beq

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 associée à l'étiquette par l'assembleur.

if (ri = rj) pc <= adresse associée à label

Exception : pas d'exception

bgez

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 associée à l'étiquette par l'assembleur.

if (ri >= 0) pc <= adresse associée à label

Exception : pas d'exception

bgezal

Branchement à une fonction si registre supérieur ou égal à zéro

Syntaxe : bgezal $ri, label

Description : Inconditionnellement, l'adresse de l'instruction suivant l'instruction bgezal est stockée dans le registre $31. Si le contenu du registre $ri est supérieur ou égal à zéro, le programme saute à l'adresse associée à l'étiquette par l'assembleur.

r31 <= pc + 4
if (ri >= 0) pc <= adresse associée à label

Exception : pas d'exception

bgtz

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 associée à l'étiquette par l'assembleur.

if (ri > 0) pc <= adresse associée à label

Exception : pas d'exception

blez

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 associée à l'étiquette par l'assembleur.

if (ri <= 0) pc <= adresse associée à label

Exception : pas d'exception

bltz

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 associée à l'étiquette par l'assembleur.

if (ri < 0) pc <= adresse associée à label

Exception : pas d'exception

bltzal

Branchement à une fonction si registre strictement inférieur à zéro

Syntaxe : bltzal $ri, label

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

r31 <= pc + 4
if (ri < 0) pc <= adresse associée à label

Exception : pas d'exception

bne

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 associée à l'étiquette par l'assembleur.

If (ri not= rj) pc <= adresse associée à label

Exception : pas d'exception

break

Arrêt et saut à la routine d'exception

Syntaxe : break imm

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

Pc <= 0x80000080

Exception : déclenchement d'une exception de type point d'arrêt.

div

Division entière signée

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 quotient de la division est placé dans le registre spécial lo, et le reste dans dans le registre spécial hi.

lo <= ri / rj
hi <= ri mod rj

Exception : pas d'exception

divu

Division entière non-signée

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 quotient de la division est placé dans le registre spécial lo, et le reste dans le registre spécial hi.

lo <= ri / rj
hi <= ri mod rj

Exception : pas d'exception

j

Saut inconditionnel immédiat

Syntaxe : j label

Description : Le programme saute inconditionnellement à l'adresse associée à l'étiquette par l'assembleur.

pc <= adresse associée à label

Exception : pas d'exception

jal

Appel de fonction inconditionnel immédiat

Syntaxe : jal label

Description : L'adresse de l'instruction suivant l'instruction jal est stockée dans le registre $31. Le programme saute inconditionnellement à l'adresse associée à l'étiquette par l'assembleur.

r31 <= pc + 4
pc <= adresse associée à label

Exception : pas d'exception

jalr

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 l'instruction jalr est sauvée dans le registre $rr. Si le registre n'est pas spécifié, alors c'est par défaut le registre $31 qui est utilisé.

rr <= pc + 4
pc <= ri

Exception : pas d'exception

jr

Branchement inconditionnel registre

Syntaxe : jr $ri

Description : Le programme saute à l'adresse contenue dans le registre $ri.
. pc <= ri

Exception : pas d'exception

lb

Lecture d'un octet signé en mémoire

Syntaxe : lb $rr, imm($ri)

Description : L'adresse de lecture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri.
L'octet lu à cette adresse subit une extension de signe et est placé dans le registre $rr.

rr <= $(Mem[imm + ri])_7^{24}$ || $(Mem[imm + ri])_{7...0}$

Exceptions : - Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini

lbu

Lecture d'un octet non-signé en mémoire

Syntaxe : lbu $rr, imm($ri)

Description : L'adresse de lecture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri.
L'octet lu à cette adresse subit une extension avec des zéros et est placé dans le registre $rr.

rr <= 0^{24} || $(Mem[imm + ri])_{7...0}$

Exceptions : - Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini.

lh

Lecture d'un demi-mot signé en mémoire

Syntaxe : lh $rr, imm($ri)

Description : L'adresse de lecture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri. Le bit de poids faible de cette adresse doit être à zéro (adresse multiple de 2).

Le demi-mot de 16 bits lu à cette adresse subit une extension de signe et est placé dans le registre $rr.

rr <= $(Mem[imm + ri])_{15}^{16}$ || $(Mem[imm + ri])_{15...0}$

Exceptions : - Adresse non alignée sur une frontière de demi-mot.
- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini.

lhu

Lecture d'un demi-mot non-signé en mémoire

Syntaxe : lhu $rr, imm($ri)

Description : L'adresse de lecture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri. Le bit de poids faible de cette adresse doit être à zéro (adresse multiple de 2).
Le demi-mot de 16 bits lu à cette adresse subit une extension avec des zéro et est placé dans le registre $rr.

rr <= 0^{16} || $(Mem[imm + ri])_{15...0}$

Exceptions : - Adresse non alignée sur une frontière de demi-mot.
- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini

lui

Chargement d'une constante dans les poids forts d'un registre

Syntaxe : lui $rr, imm

Description : La constante immédiate de 16 bits est décalée de 16 bits à gauche, et est complétée de zéro à droite. La valeur sur 32 bits ainsi obtenue est placée dans le registre $rr.

rr <= $imm_{15...0}$ || 0^{16}

Exception : pas d'exception

lw

Lecture d'un mot de la mémoire

Syntaxe : lw $rr, imm($ri)

Description : L'adresse de lecture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre $ri. Les deux bits de poids faible de cette adresse doivent être à zéro (adresse multiple de 4).
Le mot de 32 bits lu à cette adresse est placé dans le registre $rr.

rr <= Mem[imm + ri]

Exceptions : - Adresse non alignée sur une frontière de mot.

- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini

mfc0

Copie d'un registre spécial dans d'un registre général

Syntaxe : mfc0 $rt, $rd

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

rt <= rd

Exception : registre spécial non défini.

mfhi

Copie le registre hi dans un registre général

Syntaxe : mfhi $rr

Description : Le contenu du registre hi --- qui est mis à jour par les opérations de multiplication ou de division --- est recopié dans le registre général $rr.

rr <= hi

Exception : pas d'exception

mflo

Copie le registre lo dans un registre général

Syntaxe : mflo $rr

Description : Le contenu du registre lo --- qui est mis à jour par les opérations de multiplication ou de division --- est recopié dans le registre général $rr.

rr <= lo

Exception : pas d'exception.

mtc0

Copie d'un registre général dans un registre spécial

Syntaxe : mtc0 $rt, $rd

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

rt <= rd

Exception : registre spécial non défini.

mthi

Copie d'un registre général dans le registre hi

Syntaxe : mthi $ri

Description : Le contenu du registre général $ri est recopié dans le registre hi.

Exception : pas d'exception.

mtlo

Copie d'un registre général dans le registre lo

Syntaxe : mtlo $ri

Description : Le contenu du registre général $ri est recopié dans le registre lo.

Exception : pas d'exception.

mult

Multiplication signée

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.

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

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

Exception : pas d'exception.

multu

Multiplication non-signée

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.

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

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

Exception : pas d'exception.

nor

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.

rr <= ri nor rj

Exception : pas d'exception.

or

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.

rr <= ri or rj

Exception : pas d'exception.

ori

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.

rr <= $(0^{16} \| imm_{15...0})$ or ri

Exception : pas d'exception.

rfe

Restauration des bits d'état en fin de traitement 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 SR à la valeur qu'il avait avant l'exécution du programme d'exception courant.

sr <= $sr_{31...4}$ || $sr_{5...2}$

sb

Ecriture 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.

Mem[imm + ri] <= $rj_{7...0}$

Exceptions : - Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini.

sh

Ecriture 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.
Le bit de poids faible de cette adresse doit être à zéro (adresse multiple de 2).
Les deux octets de poids faible du registre $rj sont écrits à l'adresse ainsi calculée.

Mem[imm + ri] <= $rj_{15...0}$

Exceptions : - Adresse non alignée sur une frontière de demi-mot.
- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini.

sll

Décalage à gauche immédiat

Syntaxe : sll $rr, $ri, imm

Description : Le registre 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.

rr <= $ri_{(31 – imm)...0}$ || 0^{imm}

Exception : pas d'exception.

sllv

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.

rr <= $ri_{(31 - rj)...0}$ || 0^{rj}

Exception : pas d'exception.

slt

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 nombres signés.
Si la valeur contenue dans $ri est strictement inférieure à celle contenue dans $rj, alors $rr prend la valeur un, sinon il prend la valeur zéro.

```
If (ri < rj)  rr <= 1
else        rr <= 0
```

Exception : pas d'exception.

slti

Comparaison signée registre immédiat

Syntaxe : slti $rr, $ri, imm

Description : Le contenu du registre est comparé à la valeur immédiate sur 16 bits qui a subi une extension de signe. Les deux valeurs sont considérées comme des nombres signés. Si la valeur contenue dans $ri est strictement inférieure à celle de l'immédiat, alors $rr prend la valeur un, sinon il prend la valeur zéro.

If (ri <(imm_{15}^{16} || $imm_{15...0}$)) rr <= 1
else rr <= 0

Exception : pas d'exception.

sltiu

Comparaison non-signée registre immédiat

Syntaxe : sltiu $rr, $ri, imm

Description : Le contenu du registre est comparé à la valeur immédiate sur 16 bits qui a subi une extension de signe.

Les deux valeurs étant considérées comme des nombres non-signés, si la valeur contenue dans $ri est strictement inférieur à celle de l'immédiat étendu, alors $rr prend la valeur un, sinon $rr prend la valeur zéro.

if (ri < (imm_{15}^{16} || $imm_{15...0}$)) rr <= 1
else rr <= 0

Exception : pas d'exception

sltu

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 considérées comme des nombres non-signés.
Si la valeur contenue dans ri est strictement inférieure à celle contenue dans $rj, alors $rr prend la valeur un, sinon il prend la valeur zéro.

If (ri < rj) rr <= 1
else rr <= 0

Exception : pas d'exception

sra

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 $ri étant introduit dans les bits de poids fort.
Le résultat est placé dans le registre .

rr <= $(ri_{31})^{imm}$ || $(ri)_{31...imm}$

Exception : pas d'exception

srav

Décalage à droite arithmétique registre

Syntaxe : srav $rr, $ri, $rj

Description : Le registre $ri est décalé à droite du nombre de bits spécifié dans les 5 bits de poids faible du registre $rj, le bit de signe de $ri étant introduit dans les bits de poids fort.
Le résultat est placé dans le registre $rr.

rr <= $(ri_{31})^{rj}$ || $(ri)_{31...rj}$

Exception : pas d'exception

srl

Décalage à droite logique immédiat

Syntaxe : srl $rr, $ri, imm

Description : Le registre est décalé à droite de la valeur immédiate codée sur 5 bits, des zéros étant introduits dans les bits de poids fort.

rr <= 0^{imm} || $ri_{31...imm}$

Exception : pas d'exception

srlv

Décalage à droite logique registre

Syntaxe : srlv $rr, $ri, $rj

Description : Le registre $ri est décalé à droite du nombre de bits spécifié 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 .

rr <= 0^{rj} || $ri_{31...rj}$

Exception : pas d'exception

sub

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.

rr <= ri - rj

Exception : génération d'une exception si dépassement de capacité.

subu

Soustraction registre registre non-signée

Syntaxe : sub $rr, $ri, $rj

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

rr <= ri – rj

Exception : pas d'exception

sw

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. Les deux bits de poids faible de cette adresse doivent être à zéro (adresse multiple de 4).
Le contenu du registre $rj est écrit en mémoire à l'adresse ainsi calculée.

Mem[imm + ri] <= rj

Exceptions : - Adresse non alignée sur une frontière de mot.
- Adresse dans le segment noyau alors que le processeur est en mode utilisateur.
- Adresse correspondant à un segment non défini

syscall

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

Syntaxe : syscall

Description : Un appel système est effectué, par un branchement inconditionnel 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.

Pc <= 0x80000080

xor

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 .

rr <= ri xor rj

Exception : pas d'exception

xori

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.

rr <= $(0^{16} \| imm_{15...0})$ xor ri

Exception : pas d'exception

5/ 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 de plusieurs instructions machine. Les macro-instructions utilisent le registre $1 si elles ont besoin de faire un calcul intermédiaire. Il ne faut donc pas utiliser ce registre dans les programmes.

la

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 .
Code équivalent
Calcul de adr par l'assembleur

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

li

Chargement d'un opérande immédiat sur 32 bits dans un registre

Syntaxe : li $rr, imm

Description : La valeur immédiate est chargée dans le registre .$rr .

Code équivalent

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

6/ DIRECTIVES SUPPORTEES 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 ces pseudo-instructions commencent par le caractère « .» ce qui permet de les différencier clairement des instructions.

6.1) 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 lors de 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 autres sections (text, data, ktext, kdata). Ces six directives sont :

- .text
- .data
- .stack
- .ktext
- .kdata
- .kstack

Toutes les instructions ou directives qui suivent une de ces six directives, concernent la section correspondante.

6.2) Déclaration et initialisation de variables

Les directives suivantes permettent d'initialiser les valeurs contenues dans certaines sections de la mémoire (uniquement text, data, ktext, et kdata).

.align n

Description : Cette directive aligne le compteur d'adresse de la section concernée sur une adresse telle que les n bits de poids faible soient à zéro.

Exemple

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

.ascii chaîne, [autrechaîne]**•••**

Description : Cette directive place à partir de l'adresse du compteur d'adresse de la section concernée 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 chaîne, [autrechaîne]…

Description : Cette directive opérateur est strictement identique à la précédente, la seule différence étant qu'elle ajoute un zéro binaire à la fin de chaque chaîne.

Exemple

```
message:
.asciiz "Bonjour, Maître"
```

.byte n, [m]…

Description : La valeur de chacune des expressions n,m,… 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 n, [m]…

Description : La valeur de chacune des expressions n,m,… est tronquée à 16 bits, et les valeurs ainsi obtenues sont placées à des adresses successives de la section active.

Exemple

```
coordonnées:
.half 0  , 1024
```

.word n, [m]…

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 n

Description : Un espace de taille n octets est réservé à partir de l'adresse courante de la section active.

Exemple

```
nuls:
.space 1024          # initialise 1024 octets de mémoire à zéro
```

7/ APPELS SYSTEME

Pour réaliser certains traitements qui ne peuvent être exécutés que sous le contrôle du système d'exploitation (typiquement les entrées/sorties consistant à lire ou écrire un nombre, ou une chaîne de caractère sur la console), le programme utilisateur doit utiliser un « appel système », grâce à l'instruction syscall.

Par convention, le numéro de l'appel système est contenu dans le registre $2, et ses éventuels arguments dans les registres $4 et $5.

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

7.1) Ecrire un entier sur la console

Il faut mettre l'entier à écrire dans le registre $4 et exécuter l'appel système numéro 1.

```
li      $4,     0x1234567   # stocke la valeur hexadécimale 1234567 dans $4
ori     $2,     $0,     1   # code de « print_integer » dans $2
syscall                     # affiche 1234567
```

On peut aussi écrire sans macro-instruction :

```
lui     $4,     0x123
ori     $4,     $0,     0x4567#  valeur hexadécimale 1234567 dans $4
ori     $2,     $0,     1   # code de « print_integer » dans $2
syscall                     # affiche 1234567
```

7.2) Lire un entier sur la console

La valeur de retour d'une fonction --- système ou autre --- doit être rangée par la fonction appelée dans le registre $2. Ainsi, lire un entier consiste à exécuter l'appel système numéro 5 et récupérer le résultat dans le registre$2.

```
ori     $2,     $0,     5   # code de « read_integer » dans $2
syscall                     # $2 contient la valeur lue
```

7.3) Ecrire une chaîne de caractères sur la console

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 » dans $2
        syscall                     # affiche la chaine pointée
```

On peut aussi écrire sans macro-instruction :

```
str:    .asciiz "Chaîne à afficher\n"
        lui     $4,     str >> 16
        ori     $4,     $0,    str & 0xFFFF          # charge le pointeur dans $4
        ori     $2,     $0,     4       # code de « print_string » dans $2
        syscall                         # affiche la chaine pointée
```

7.4) Lire une chaîne de caractères sur la console

Pour lire une chaîne de caractères, il faut un pointeur définissant l'adresse du buffer de réception en mémoire et un entier définissant la taille du buffer (en nombre de caractères). On écrit la valeur du pointeur dans $4, et la taille du buffer dans $5, et on exécute l'appel système numéro 8.

```
read:   .space 256
        la      $4,    read             # charge le pointeur dans $4
        ori     $5,     $0,     152     # charge longueur max dans $5
        ori     $2,     $0,     8       # code de « read_string »
        syscall                         # copie la chaîne dans le buffer pointé par $4
```

On peut aussi écrire sans macro-instruction :

```
read:   .space 256
        lui     $4,    read >> 16
        ori     $4,     $0, read & 0xFFFF # charge le pointeur dans $4
        ori     $5,     $0,     152     # charge longueur max dans $5
        ori     $2,     $0,     8       # code de « read_string »
        syscall                         # copie la chaîne dans le buffer pointé par $4
```

7.5) Terminer un programme

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

```
        ori     $2,     $0,     10      # code de « exit »
        syscall                         # quitte pour de bon
```

8/ 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. Les conventions définies ci-dessous sont fortement inspirées des conventions utilisées par le compilateur GCC.

- La pile s'étend vers les adresses décroissantes .
- Le pointeur de pile pointe toujours sur le dernier mot occupée dans la pile.
  Ceci signifie que tous les mots 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.
- La valeur de retour d'une fonction est conventionnellement écrite, dans le registre $2, par la fonction appelée.
- Par ailleurs, l'architecture matérielle du processeur MIPS R3000 impose l'utilisation du registre $31 pour stocker l'adresse de retour lors d'un appel de fonction.

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 $8 à $26 sont disponibles pour tout calcul dans cette fonction ; mais le contenu des registres utilisés doit être sauvegardé et restauré avant le retour au programme appelant. Dans le cas général, un contexte d'exécution d'une fonction est constitué de trois zones qui sont, dans l'ordre d'empilement :

- **La zone des arguments de la fonction appelée**

Les valeurs des arguments sont écrites dans la pile par la fonction appelante et lues dans la pile par la fonction appelée. Dans la suite de ce document, on note *na* le nombre d'arguments, et on considère que tous les arguments sont représentés par des mots de 32 bits. Par convention, on place toujours le premier argument de la fonction appelée à l'adresse la plus petite dans la zone des arguments.

- **La zone de sauvegarde des registres**

La fonction appelée est chargée de sauvegarder le registre $31, ainsi que les registres de travail qu'elle utilise de façon à pouvoir restaurer la valeur de ces registres avant de rendre la main à la fonction appelante. Dans la suite de ce document, on note *nr* le nombre de registres sauvegardés en plus du registre $31. Seuls les registres d'indice supérieur à 7 doivent être sauvegardés (s'ils sont utilisés).
Le registre $31 contenant l'adresse de retour est toujours sauvegardé à l'adresse la plus grande de la zone de sauvegarde.

- **La zone des variables locales de la fonction appelée.**

Les valeurs stockées dans cette zone ne sont en principe lues et écrites que par la fonction appelée. Elle est utilisée pour stocker les variables et structures de données locales à la fonction. Dans la suite de ce document, on note *nv* le nombre de mots de 32 bits constituant la zone des variables locales.

8.1) Organisation de la pile

Dans le cas général, une fonction f souhaite appeler une fonction g. La fonction g possède des arguments, utilise des registres et possède des variables locales.

La fonction f appelante doit effectuer la séquence suivante :

- décrémenter le pointeur de pile : $29 <= $29 – 4*_na_, de façon à réserver la place dans la pile pour les arguments.
- écrire dans la pile les valeurs des na arguments de la fonction g, en mettant le premier argument à l'adresse pointée par $29, le deuxième à $29 +4, etc...
- effectuer le branchement à la première instruction de la fonction appelée en utilisant une instruction de type jal ou bgezal.
- incrémenter le pointeur de pile pour restaurer la valeur qu'il avait avant l'appel de la fonction g : $29 <= $29 + 4*_na_

La fonction g appelée est découpée en trois parties.
La première partie est le prologue, qui commence par décrémenter le pointeur de pile de façon à réserver la place pour la sauvegarde du registre $31 et des registres qui seront utilisés par g, ainsi que la place nécessaire aux variables locales de g. Le prologue sauvegarde ensuite dans la pile : la valeur du registre $31 et les valeurs des registres qui seront utilisés par g.
La deuxième partie est le corps de la fonction qui effectue les calculs, en utilisant les registres nécessaires ainsi que les variables locales stockées dans la pile, puis écrit la valeur de retour dans le registre $2.
La troisième partie est l'épilogue chargé de : restaurer les valeurs des registres sauvegardés dans la pile, d'incrémenter le pointeur de pile pour lui redonner la valeur qu'il avait en entrant dans la fonction g.

Le prologue de la fonction g doit effectuer la séquence suivante :

- décrémenter le pointeur de pile : $29 <= $29 – 4*(*nv* + *nr* + 1)
- écrire successivement dans la pile suivant les adresses décroissantes la valeur du registres $31 et des *nr* registres qui seront utilisés par le corps de la fonction.
- lire dans la pile les arguments de la fonction g et les stocker dans les registres utilisés par le corps de la fonction g.

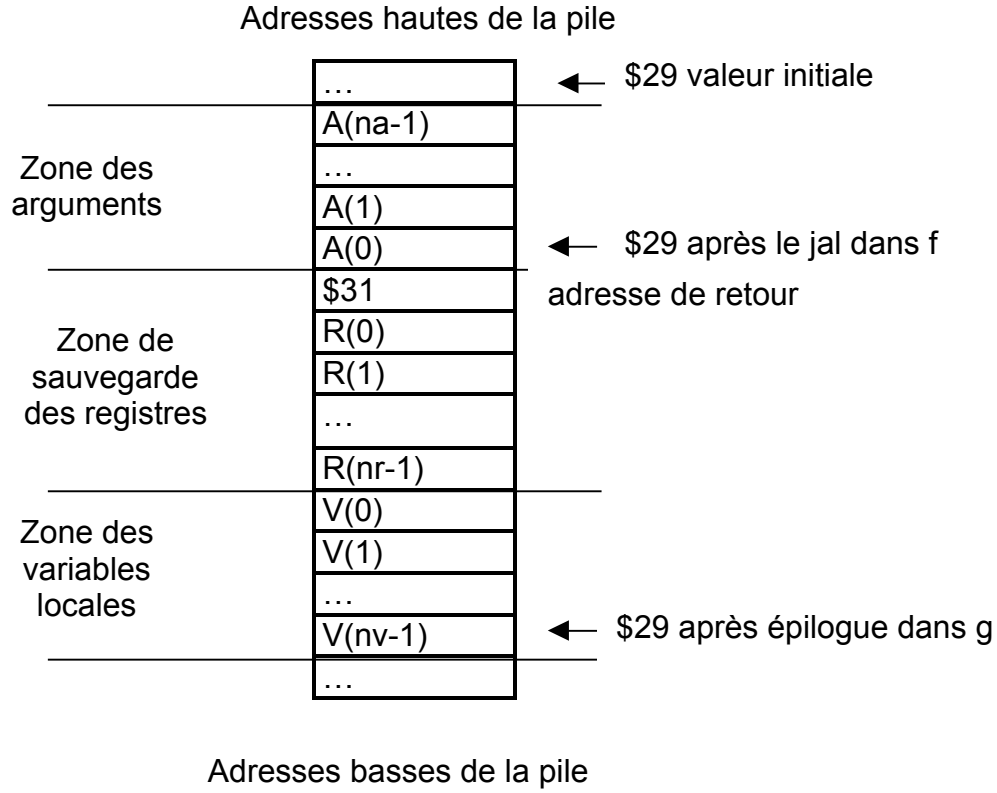
L'épilogue de la fonction g doit effectuer la séquence suivante :

- lire dans la pile les valeurs des (*nr* +1) registres sauvegardés (dont le registre $31) et restaurer les valeurs qu'avaient les registres avant l'appel de la fonction g.
- incrémenter le pointeur de pile : $29 <= $29 + 4*(*nv* + *nr* + 1)
- effectuer un branchement à l'adresse de retour contenue dans le registre $31

On note R(i) les registres utilisés par la fonction appelée, qui doivent être sauvegardés.
On note A(i) les registres utilisés dans l'appelant pour stocker les arguments.
On note P(i) les registres utilisés dans l'appelé pour la récupération des arguments.
On note v(i) les variables locales déclarées dans la fonction appelée
Le code à écrire est donc le suivant.

Dans f:

```
        ...
        addiu  $29,          $29,   -4*na          # décrémentation pointeur de pile
        sw     $A(0),        0 ($29)               # écriture premier argument
        ...
        sw     $A(na -1),    4*(na -1) ($29)       # écriture dernier argument
        jal    g                                   # branchement à la fonction g
        addiu  $29,          $29,   4*na           # incrémentation pointeur de pile
        ...
Dans g:
# prologue
g :     addiu  $29,          $29, - 4*(nv + nr + 1)   # décrémentation pointeur de pile
        sw     $31,          4*(nv +nr )($29)         # sauvegarde registre $31
        sw     $R(0),        4*(nv + nr  - 1)($29)    # sauvegarde premier registre
        ...
        sw     $R(nr  -1),   4*nv($29)                # sauvegarde dernier registre
        lw     $P(0),        4*( nr + nv + 1)($29)    # récupération premier argument
        ...
        lw     $P(na  -1),   4*(na + nr + nv)($29)    # récupération dernier argument
# corps de la fonction
        ...
# epilogue
        lw     $31,          4*(nv + nr)($29)         # restauration registre $31
        lw     $R(0),        4*(nv + nr  – 1)($29)    # restauration premier registre
        ...
        lw     $R(nr-1),     4*nv($29)                # restauration dernier registre
        addiu  $29,          $29,   4*(nv + nr + 1)   #  incrémentation pointeur de pile
        jr     $31                                    #  branchement adresse de retour
```



8.2) Exemple

On traite ici l'exemple d'une fonction calculant la norme d'un vecteur (x,y), en supposant qu'il existe une fonction `int isqrt(int x)` qui retourne la racine carrée d'un nombre entier. Les coordonnées du vecteur sont des variables globales initialisées dans le segment « data ». Ceci correspond au code C ci-dessous :

```
int     x =   5 ;
int     y =   4 ;

int main()
{
printf (« %x », norme(x,y) );
}

int norme (int a, int b)
{  int somme, val;          /* Variables locales */
   somme = a * a + b * b;
   val = isqrt(somme);
   return val;
}
```

La fonction `main()` n'est pas une fonction comme les autres puisqu'elle n'est appelée par personne. Elle appèle la fonction `norme()` qui a deux arguments, donc *na* = 2. La fonction `norme()` a deux variables locales déclarées (somme et val). Elle utilise deux registres de travail ($7 et $8). On a donc *nv* = 2 et *nr* = 2.
Les deux fonction isqrt() et norme() renvoient leur résultat dans le registre $2.
Le programme assembleur est le suivant :

```
.data
x :     .word     5
y :     .word     4

.text

__start:  addiu     $29,  $29,  -8    # décrémentation pointeur de pile
          la        $8,   x           # Ecriture 1er paramètre dans la pile
          lw        $9,   0($8)
          sw        $9,   0($29)
          la        $8,   y           # Ecriture 2e paramètre dans la pile
          lw        $9,   0($8)
          sw        $9,   4($29)
          jal       norme
          addiu     $29,  $29,  +8    # incrémentation pointeur de pile
          or        $4,   $2,   $0    # récupération résultat norme dans $4
          ori       $2,   $0,   1     # code de « print_integer » dans $2
          syscall                     # affichage résultat sur la console
          ori       $2, $0, 10        # code de « exit » dans $2
          syscall                     # sortie du programme
```

norme :

```
# prologue
        addiu   $29,  $29,  -20   # décrémentation pointeur de pile
        sw      $31,  +16($29)    # sauvegarde adresse de retour
        sw      $8,   +12($29)    # sauvegarde registre de travail $8
        sw      $9,   + 8($29)    # sauvegarde registre de travail $9
        lw      $8,   +20($29)    # récupération 1er paramètre "a"
        lw      $9,   +24($29)    # récupération 2e paramètre  "b"

# corps de la fonction
        mult    $8,   $8          # calcul a*a
        mflo    $8
        mult    $9,   $9          # calcul b*b
        mflo    $9
        addu    $8,   $8,   $9    # calcul somme
        addiu   $29,  $29,  -4    # décrémentation pointeur de pile
        sw      $8,   0($29)      # écriture paramètre isqrt dans la pile
        jal     isqrt             # branchement à la fonction isqrt
                                  # le résultat est dans $2
        addiu   $29,  $29,  +4    # incrémentation pointeur de pile
# épilogue
        lw      $31,  +16($29)    # restaure adresse de retour
        sw      $8,   +12($29)    # restaure registre de travail $8
        sw      $9,   +8($29)     # restaure registre de travail $9
        addiu   $29,  $29,  +20   # incrémentation pointeur de pile
        jr      $31               # retour à la fonction appelante
```