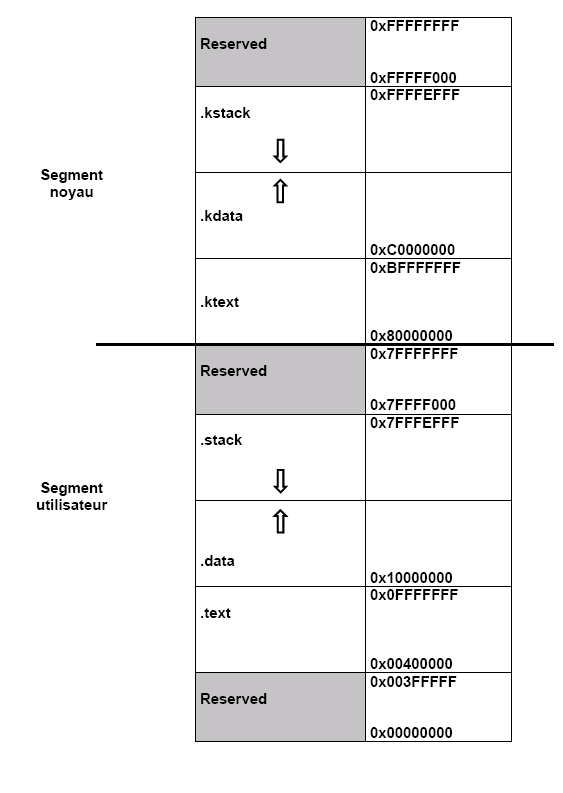
Processeur MIPS R3000

Langage d’assemblage

# ORGANISATION DE LA MEMOIRE

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 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. Ainsi, le code binaire généré par l’assemblage du programme source 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.

# REGLES SYNTAXIQUE

Pour écrire un programme assembleur, nous devons respecter un certain nombre de règles syntaxiques. On définit ci-dessous les principales règles d’écriture d’un programme source.

## Les entiers

Une valeur entière décimale est notée sans préfixe (ex : **85)**, et une valeur entière hexadécimale est notée préfixée par zéro suivi de x (ex : **0xBA**).

## Les chaînes de caractères

Elles sont représentées entre guillemets, et peuvent contenir les caractères d’échappement du langage C. **Exemple :** "c’est une chaîne de caractère avec retour à la ligne\n".

## Les labels (étiquettes)

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 « : ». Aussi on ne doit pas utiliser de mots réservés pour les labels.

*Exemple* **:**

.data

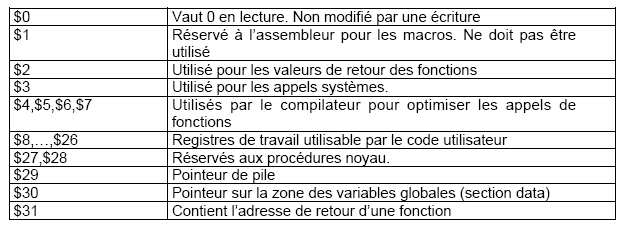
message : .asciiz "Ceci est une chaine de caractères...\n" # le mot « message » est un label

## Les immédiats

On appelle immédiat un opérande contenu dans l’instruction. Ce sont soit des entiers, soit des labels. La taille maximum d’un immédiat dépend du format de l’instruction qui l’utilise : 16 bits (format I) ou 26 bits (format J).

## 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 30 sera noté **$30** dans l’assembleur.



## Les modes d’adressage

MIPS est une architecture de type chargement / rangement, ce qui signifie que seules les instructions de chargement et de rangement accèdent à la mémoire. Les instructions de calcul opèrent uniquement sur des valeurs contenues dans des registres. MIPS ne possède qu’un unique mode d’adressage pour lire ou écrire des données en mémoire: l’adressage indirect registre avec déplacement : **I(rx)**. L’adresse est obtenue en additionnant le déplacement I (positif ou négatif) au contenu du registre **rx**.

*Exemples:*lw $12, 13($10) # $12 <= Mem[$10 + 13] chargement (load)

sw $20, -60($22) # Mem[$22-60] <= $20 rangement (store)

# DIRECTIVES SUPPORTEES PAR L'ASSEMBLEUR MIPS

Les directives (ou pseudo instruction) ne sont pas des instructions exécutables par la machine, mais permettent de donner des ordres à l’assembleur. Pour les différencier des instructions, ces directives commencent par le caractère « . ».

## 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. Ces six directives sont :

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

## 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

* **.ascii** chaîne1

*Description :* Cette directive enregistre en mémoire la chaîne de caractères *chaîne1*, mais ne la termine pas par un zéro binaire. 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, Madame!\n\0"

* **.asciiz** chaîne1, chaine2…

*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, madame"

* **.byte** b1, b2, …, bn

*Description :* Enregistre les n valeurs (b1, b2, …, bn) dans des octets consécutifs en mémoire. Les valeurs sont, donc, 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** h1, h2, …, hn

*Description :* Enregistre les n valeurs (h1, h2, …, hn) de 16 bits dans des demi mots consécutifs en mémoire.

*Exemple*

coordonnées: .half 0, 1024

* **.word**  w1, w2, …, wn

*Description :* Enregistre les n valeurs (w1, w2, …, wn) de 32 bits dans des mots consécutifs en mémoire.

*Exemple*

tab : .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.

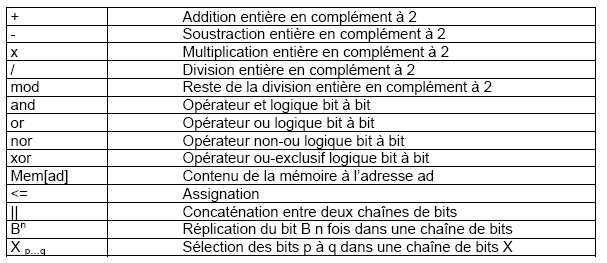
*Exemple*

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

# INSTRUCTIONS

Dans ce qui suit, le registre noté **$rr** est le registre destination, les registres notés **$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**, 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 :



**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 <= (imm1516 || imm15...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 <= ( imm1516 || imm15..0 ) + ri

**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

**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

**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 <= (016 || imm15..0 ) and ri

**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

**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

**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

**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

**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

**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

**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

**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 != rj ) pc <= adresse associée à label

**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 le registre spécial hi.

lo <= ri / rj

hi <= ri mod rj

**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

**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

**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

**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

**jr**

Branchement inconditionnel registre

**Syntaxe :**  jr $ri

**Description :** Le programme saute à l’adresse contenue dans le registre $ri.

pc <= ri

**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])724 || (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 <= 024 || (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 16bits, 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])1516|| (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 <= 016 || (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 || 016

**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

**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

**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.

**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.

**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 x rj)31 .. 0

hi <= (ri x rj)63 .. 32

**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 x rj)31 .. 0

hi <= (ri x rj)63 .. 32

**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

**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

**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 <= (016 || imm 15…0) or ri

**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|| sr5 … 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] <= rj15 .. 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 || 0imm

**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|| 0rj

**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 .tant 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

**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 <(imm1516 || imm15…0)) rr <= 1

else rr <= 0

**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 < (imm1516 || imm15..0)) rr <= 1

else rr <= 0

**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

**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 <= (ri31 )imm ||( ri)31...imm

**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 <= (ri31 )rj ||( ri)31...rj

**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 <= 0imm || (ri) 31..imm

**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 <= 0rj ||( ri) 31..rj

**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 :** subu $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

**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 aligne 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

**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 <= (016 || imm15..0) xor ri

# 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.

**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 .

# 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. Il y’a cinq appels système :

## Ecrire un entier

Pour écrire (afficher) un entier, il faut :

1. Mettre l’entier à écrire dans le registre $4.
2. Mettre la valeur 1 dans le $2.
3. syscall

*Exemple :*

li $4, 0x1234567 # stocke la valeur hexadécimale 1234567 dans $4

ori $2, $0, 1 # code de « print\_integer » dans $2

syscall # affiche 1234567

## Lire un entier sur la console

Lire un entier consiste à exécuter l’appel système numéro 5 et récupérer le résultat dans le registre $2.

*Exemple :*

ori $2, $0, 5 # code de « read\_integer » dans $2

syscall # $2 contient la valeur lue

## Ecrire une chaîne de caractères

Pour écrire une chaîne de caractères, il faut :

1. Mettre l’adresse de la chaîne à afficher dans le $4.
2. Mettre la valeur 4 dans le $2
3. syscall

*Exemple :*

.data

str: .asciiz "Chaîne à afficher\n"

.text

la $4, str # charge l’adresse de la chaîne dans $4

ori $2, $0, 4 # code de « print\_string » dans $2

syscall # affiche la chaîne pointée

## 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

## Terminer un programme

Pour terminer un programme, il faut exécuter l’appel système numéro 10. C’est à dire :

1. Mettre la valeur 10 dans le $2
2. syscall

*Exemple :*

ori $2, $0, 10 # code de « exit »

syscall # quitte pour de bon

# EXEMPLES DE PROGRAMMES

## Exemple 1 : programme permettant de lire un entier au clavier et de l’afficher.

.text

\_\_start :

ori $2, $0, 5 # code de « read\_integer » dans $2

syscall # $2 contient la valeur lue

add $4, $2, $0 # déplacer la valeur lue du $2 vers le $4.

ori $2, $0, 1 # code de « print\_integer » dans $2.

syscall # affiche le contenu du $4.

ori $2, $0, 10 # code de « exit »

syscall # quitte pour de bon

## Exemple 2 : programme permettant de charger une variable et de l’afficher.

.data

a : .word 5

.text

\_\_start :

la $20, a # charge l’adresse ‘a ‘ dans $20

lw $4, 0($20) # charge le contenu de l’adresse ‘a’ dans $4 c-à-d la valeur 5

addi $2, $0, 1 # code de « print\_integer » dans $2

syscall # affiche 5

addi $2, $0, 10 # code de « exit »

syscall # quitte pour de bon

## Exemple 3 : if … else

Dans le fragment de code C suivant, f, g, h, i et j sont des variables :

if ( i = = j)

f = g + h ;

else

f = g – h;

En supposant que les cinq variables correspondent aux cinq registres $16 à $20, le programme assembleur est le suivant :

Bne $19, $20, Else # aller en Else si i est différents de j

Add $16, $17, $18 # f = g + h

J Exit

Else : sub $16, $17, $18 # f = g - h

Exit :

# CONVENTIONS POUR LES APPELS DE FONCTIONS

L’exécution de fonctions nécessite une pile en mémoire qui correspond à la section stack.

* La pile s’étend vers les adresses décroissantes.
* Le pointeur de pile pointe toujours sur le dernier mot occupé dans la pile. Ceci signifie que tous les mots d’adresse inférieure au pointeur de pile sont libres.
* Pour accéder à la pile, on utilise les instructions lw et sw.
* 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.

A 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 possédera 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. Par convention, on place toujours le premier argument de la fonction appelée à l’adresse la plus petite dans la zone des arguments.

1. **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. On note *nr* le nombre de registres sauvegardés en plus du registre $31. Le registre $31 contenant l’adresse de retour est toujours sauvegardée à l’adresse la plus grande de la zone de sauvegarde.

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

Les valeurs stockés 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. On note *nv* le nombre de mots de 32 bits constituant la zone des variables locales.

## Organisation de la pile

Dans le cas général, une fonction f souhaite appeler une fonction g. La fonction f appelante doit effectuer la séquence suivante :

* décrémenter le pointeur de pile : $29 <= $29 – 4\*na, de façon 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 :

* 1. 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.

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 registre $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.
  1. 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.
  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.

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 déclarés 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



### Adresses basses de la pile

Adresses hautes de la pile

**prologue**

## Exemple

On traite ici l’exemple d’une fonction calculant la moyenne d’un vecteur (x,y). Les coordonnées du vecteur sont des variables globales initialisées dans le segment « data ».

En code C :

int x = 5 ;

int y = 4 ;

int main()

{

printf (" %x ", moyenne (x,y)) ;

}

int moyenne (int a, int b)

{ int somme, val; /\* Variables locales \*/

somme = a + b;

val = somme / 2 ;

return val;

}

La fonction main() n’est pas une fonction comme les autres puisqu’elle n’est appelé par personne. Elle appelle la fonction moyenne() qui a deux arguments, donc *na* = 2. La fonction moyenne() 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. La fonction moyenne renvoie son résultat dans le registre $2.

**Le programme assembleur est le suivant :**

.data

x : .word 5

y : .word 15

.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 2eme paramètre dans la pile

lw $9, 0($8)

sw $9, 4($29)

jal moyenne

addiu $29, $29, +8 # incrémentation pointeur de pile

or $4, $2, $0 # récupération résultat moyenne 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

moyenne :

# 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

add $8, $8 , $9 # calcul a + b

sra $2, $8, 1 # une division sur 2 équivaut à un

#décalage arithmétique à droite avec sh =1

#Le résultat est dans le registre $2.

# épilogue

lw $31, +16($29) # restaure adresse de retour

lw $8, +12($29) # restaure registre de travail $8

lw $9, +8($29) # restaure registre de travail $9

addiu $29, $29, +20 # incrémentation pointeur de pile

jr $31 # retour à la fonction appelante