Architecture externe du microprocesseur 32 bits MIPS R3000

1. Introduction

L'architecture externe représente ce que doit connaitre un programmeur souhaitant programmer en assembleur, ou la personne souhaitant écrire un compilateur pour ce processeur:

- MIPS (Microprocessor without Interlocked Pipeline Stages) est un processeur 32 bits industriel conçu dans les années 80. Son jeu d'instructions est de type RISC (Reduced instruction set computer) développée par la compagnie MIPS Computer Systems Inc.
- Il existe plusieurs réalisations industrielles de cette architecture (SIEMENS, NEC, LSI LOGIC, SILICON GRAPHICS, etc...)
- On les retrouve aussi dans plusieurs systèmes embarqués, comme les ordinateurs de poche, les routeurs Cisco et les consoles de jeux vidéo (Nintendo 64 et Sony PlayStation, PlayStation 2 et PSP).

2. Registres Visibles

Tous les registres visibles du logiciel, c'est à dire ceux dont la valeur peut être lue ou modifiée par les instructions, sont des registres 32 bits. Afin de mettre en œuvre les mécanismes de protection nécessaires pour un système d'exploitation multi-tâches, le processeur possède deux modes de fonctionnement : utilisateur/superviseur. Ces deux modes de fonctionnement imposent d'avoir deux catégories de registres.

1. Registres non protégés
2. Registres protégés.

2.1 Registres non protégés

Le processeur possède 35 registres manipulés par les instructions standard (c'est à dire les instructions qui peuvent s'exécuter aussi bien en mode utilisateur qu'en mode superviseur).

- **Ri** ($0 \leq i \leq 31$) 32 registres généraux. Ces registres sont directement adressés par les instructions, et permettent de stocker des résultats de calculs intermédiaires.

  Le registre **R0** est un registre particulier:

  - la lecture fournit la valeur constante "0x00000000"
  - l'écriture ne modifie pas son contenu.

  Le registre **R31** est utilisé par les instructions d'appel de procédures (instructions **BGEZAL**, **BLTZAL**, **JAL** et **JALR**) pour sauvegarder l'adresse de retour.
- **PC** Registre compteur de programme (Program Counter). Ce registre contient l'adresse de l'instruction en cours d'exécution. Sa valeur est modifiée par toutes les instructions.
- **HI et LO** Registres pour la multiplication ou la division. Ces deux registres 32 bits sont utilisés pour stocker le résultat d'une multiplication ou d'une division, qui est un mot de 64 bits.

La description des 32 registres est donnée dans le tableau suivant :

Name	Register number	Usage
$zero	0	the constant value 0
$at	1	reserved for the assembler
$v0–$v1	2–3	values for results and expression evaluation
$a0–$a3	4–7	arguments
$t0–$t7	8–15	temporaries
$s0–$s7	16–23	saved
$t8–$t9	24–25	more temporaries
$k0–$k1	26–27	reserved for the operating system
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

2.2 Registres protégés

- SR Registre d'état (Status Register). Il contient en particulier le bit qui définit le mode superviseur ou utilisateur, ainsi que les bits de masquage des interruptions. (Ce registre possède le numéro 12)
- CR Registre de cause (Cause Register). En cas d'interruption ou d'exception, son contenu définit la cause pour laquelle on fait appel au programme de traitement des interruptions et des exceptions. (Ce registre possède le numéro 13)
- EPC Registre d'exception (Exception Program Counter). Il contient l'adresse de retour (PC + 4) en cas d'interruption. Il contient l'adresse de

l'instruction fautive en cas d'exception (PC). (Ce registre possède le numéro 14)

- BAR Registre d'adresse illégale (Bad Address Register). En cas d'exception de type "adresse illégale", il contient la valeur de l'adresse mal formée. (Ce registre possède le numéro 8)

3. Adressage Mémoire

L'unité adressable est l'octet, et la mémoire est vue comme un tableau d'octets qui contient les données et les instructions :

- Les adresses sont codées sur 32 bits (CO =32).
- Les instructions sont codées sur 32 bits (RI=32).

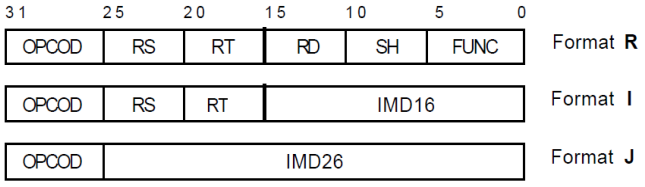
Les échanges de données avec la mémoire se font par mot (4 octets consécutifs), demi-mot (2 octets consécutifs), ou par octet. L'adresse d'un mot de donnée ou d'une instruction doit être multiple de 4. L'adresse d'un demi-mot doit être multiple de 2. (On dit que les adresses doivent être "alignées").

4. Jeu d'instructions

Le processeur possède 57 instructions qui se répartissent en 4 classes :

- 33 instructions arithmétiques/logiques entre registres
- 12 instructions de branchement
- 7 instructions de lecture/écriture mémoire
- 5 instructions systèmes

Toutes les instructions ont une longueur de 32 bits et possèdent un des trois formats suivants :



- Le format **J** n'est utilisé que pour les branchements à longue distance (inconditionnels).
- Le format **I** est utilisé par les instructions de lecture/écriture mémoire, par les instructions utilisant un opérande immédiat, ainsi que par les branchements courte distance (conditionnels).
- Le format **R** est utilisé par les instructions nécessitant 2 registres sources (désignés par RS et RT) et un registre résultat désigné par RD.

Le jeu d'instructions est "**orienté registres**". Cela signifie que les instructions arithmétiques et logiques prennent leurs opérandes dans des registres et rangent le résultat dans un registre.

- Les seules instructions permettant de lire ou d'écrire des données en mémoire effectuent un simple transfert entre un registre général et la mémoire, sans aucun traitement arithmétique ou logique.
- La plupart des instructions arithmétiques et logiques se présentent sous les 2 formes registre-registre et registre-immédiat:
  - ADD : R(rd) <--- R(rs) op R(rt) (format R)
  - ADDI : R(rt) <--- R(rs) op IMD (format I)
- L'opérande immédiat 16 bits est signé pour les opérations arithmétiques et non signé pour les opérations logiques.
- Les instructions JAL, JALR, BGEZAL, et BLTZAL sauvegardent une adresse de retour dans le registre R31. Ces instructions sont utilisées pour les appels de sous programme.
- Toutes les instructions de branchement conditionnel sont relatives au compteur ordinal pour que le code soit translatable. L'adresse de saut est le résultat d'une addition entre la valeur du compteur ordinal et un déplacement signé.
- Les instructions MTC0 et MFC0 permettent de transférer le contenu des registres SR, CR, EPC et BAR vers un registre général et inversement. Ces 2 instructions ne peuvent être exécutées qu'en mode superviseur, de même que l'instruction RFE qui permet de restaurer l'état antérieur du registre d'état avant de sortir du gestionnaire d'exceptions.

5. Programmation et format d'une instruction

Une instruction du langage d'assemblage est composée de *champs*. On identifie un champ *étiquette*, un champ *code opération*, un champ *opérandes* pouvant effectivement comporter plusieurs opérandes séparés par des virgules, et un champ commentaires.

Exemple :

boucle : add Rd,Rs,Rt ; Addition Rd=Rs+Rt (ou # Addition Rd=Rs+Rt)

Dans un programme assembleur il y'a deux sections : **data** section et **text** section.

Le **data** section commence par : **.data**, elle contient toutes les données de notre programme assembleur et le **text** section commence par le **.text** et contient toutes les instructions que le programme a besoin.

Format d'un programme MIPS : chaque programme assembleur MIPS R3000 doit avoir le format suivant :

```
.data
#---------------
#---------------
.text
#---------------
#---------------
```

5.1 Les commentaires

Ils commencent par un « # » ou un « ; » et s'achèvent à la fin de la ligne courante.

Exemple:

```
##############################################
# programme qui calcule la somme des n premiers nombres
##############################################
.... ; sauve la valeur copiée dans la mémoire
```

5.2 Code opération

Le code opération est une chaîne de caractères mnémonique du code opération binaire.

Exemple : Add, Addi, j, lb, etc…

5.3 Opérandes

Chaque opérande dans une instruction du langage d'assemblage possède un nom permettant de le référencer. Les opérandes d'une instruction sont séparés par une virgule.

Pour les opérandes variables, le nom est l'étiquette associée par le programmeur au moment de la déclaration des variables.

Les données (**constantes et variables**) doivent être déclarées dans « .Data » section. Le format général de la déclaration d'une donnée est:

<nom de variable> .<type de données> <valeur initiale>

Les données en MIPS sont de différents types:

Declaration	
`.byte`	8-bit variable(s)
`.half`	16-bit variable(s)
`.word`	32-bit variable(s)
`.ascii`	ASCII string
`.asciiz`	NULL terminated ASCII string
`.float`	32 bit IEEE floating point number
`.double`	64 bit IEEE floating point number
`.space <n>`	<n> bytes of uninitialized memory

Exemples :

Message :	.asciiz	″Hello World\n″
pi:	.float	3.14159
Tao :	.double	6.28318

5.4 Les entiers

Une valeur entière décimale est notée par exemple 3250, une valeur entière octale est notée 04372 (préfixée par un zéro), 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.

5.5 Les chaînes de caractères

Elles sont simplement entre guillemets. Exemple : "la valeur de x est :"

5.6 Les labels (étiquettes)

Une étiquette correspond à une adresse dans le programme, soit celle de l'instruction, soit celle de la variable.

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 auquel on ajoute les chiffres. Pour la déclaration, le label doit être suffixé par « : ».

Exemple :

ETIQ1 :

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

5.7 Directives

Les directives sont des ***pseudo-instructions*** : elles ne correspondent à aucune instruction exécutable par la machine; ce sont des ordres destinés au traducteur assembleur. Toutes les pseudo-instruction commencent

par le caractère « . » ce qui permet de les différencier clairement des instructions.

Les directives servent notamment à la déclaration des sections *data* et *text* réservées respectivement à la déclaration des variables et à l'écriture des instructions. Les directives servent aussi à la spécification des types de variables

6. Instructions arithmétiques et logiques

Exemples :

ADDI -- Add immediate (with overflow)

Opération : $t=$s+imm

Syntaxe : addi $t, $s, imm

Instructions Arithmétiques/Logiques entre registres

Assembleur		Opération		Format
Add	Rd, Rs, Rt	Add overflow detection	Rd <- Rs + Rt	R
Sub	Rd, Rs, Rt	Subtract overflow detection	Rd <- Rs - Rt	R
Addu	Rd, Rs, Rt	Add no overflow	Rd <- Rs + Rt	R
Subu	Rd, Rs, Rt	Subtract no overflow	Rd <- Rs - Rt	R
Addi	Rt, Rs,I	Add Immediate overflow detection	Rt <- Rs + I	I
Addiu	Rt, Rs,I	Add Immediate no overflow	Rt <- Rs + I	I
Or	Rd, Rs, Rt	Logical Or	Rd <- Rs or Rt	R
And	Rd, Rs, Rt	Logical And	Rd <- Rs and Rt	R
Xor	Rd, Rs Rt	Logical Exclusive-Or	Rd <- Rs xor Rt	R
Nor	Rd, Rs, Rt	Logical Not Or	Rd <- Rs nor Rt	R
Ori	Rt, Rs, I	Or Immediate unsigned immediate	Rt <- Rs or I	I
Andi	Rt, Rs, I	And Immediate unsigned immediate	Rt <- Rs and I	I
Xori	Rt, Rs, I	Exclusive-Or Immediate unsigned immediate	Rt <- Rs xor I	I
Sllv	Rd, Rt, Rs	Shift Left Logical Variable 5 lsb of Rs is significant	Rd <- Rt << Rs	R
Srlv	Rd, Rt, Rs	Shift Right Logical Variable 5 lsb of Rs is significant	Rd <- Rt >> Rs	R
Srav	Rd, Rt, Rs	Shift Right Arithmetical Variable 5 lsb of Rs is significant	Rd <- Rt >>* Rs	R
Sll	Rd, Rt, sh	Shift Left Logical	Rd <- Rt << sh	R
Srl	Rd, Rt, sh	Shift Right Logical	Rd <- Rt >> sh	R
Sra	Rd, Rt, sh	Shift Right Arithmetical	Rd <- Rt >>* sh * : with sign extension	R
Lui	Rt, I	Load Upper Immediate 16 lower bits of Rt are set to zero	Rt <- I || "0000"	I

SUBU -- Subtract unsigned

Operation: $d = $s - $t;

Syntaxe : subu $d, $s, $t

And -- Et logique : Et bit-à-bit registre registre

Syntaxe : and $rd, $rs, $rt

Un et bit-à-bit est effectué entre les contenus des registres $rs et $rt. Le résultat est placé dans le registre $rd.

ORI -- Ou logique immédiat Ou bit-à-bit registre, immédiat

Syntaxe : ori $rd, $rs, imm

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 $rs pour former un résultat placé dans le registre $rd.

Sll -- Décalage logique à gauche

Syntaxe : sll $rd, $rt, imm

Le registre $rt 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 $rd.

Srl -- Décalage logique à droite

Syntaxe : srl $rd, $rt, imm

Le registre $rt 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 $rd.

Instructions Arithmétiques/Logiques (suite)

Assembleur		Opération		Format
Slt	Rd, Rs, Rt	Set if Less Than	Rd <- 1 if Rs<Rt else 0	R
Sltu	Rd, Rs, Rt	Set if Less Than Unsigned	Rd <- 1 if Rs<Rt else 0	R
Slti	Rt, Rs, I	Set if Less Than Immediate sign extended Immediate	Rt <- 1 if Rs < I else 0	I
Sltiu	Rt, Rs, I	Set if Less Than Immediate unsigned immediate	Rt <- 1 if Rs < I else 0	I
Mult	Rs, Rt	Multiply	Rs * Rt LO <- 32 low significant bits HI <- 32 high significant bits	R
Multu	Rs, Rt	Multiply Unsigned	Rs * Rt LO <- 32 low significant bits HI <- 32 high significant bits	R
Div	Rs, Rt	Divide	Rs / Rt LO <- Quotient HI <- Remainder	R
Divu	Rs, Rt	Divide Unsigned	Rs / Rt LO <- Quotient HI <- Remainder	R
Mfhi	Rd	Move From HI	Rd <- HI	R
Mflo	Rd	Move From LO	Rd <- LO	R
Mthi	Rs	Move To HI	HI <- Rs	R
Mtlo	Rs	Move To LO	LO <- Rs	R

mfhi -- Copie du registre $hi dans un registre général

Syntaxe : mfhi $rd

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 $rd.

mflo -- Copie du registre $lo dans un registre général

Syntaxe : mflo $rd

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 $rd.

7. Instructions de branchement

Exemples :

Beq -- Branchement si registre égal registre

Syntaxe : beq $rs, $rt, label

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

Bne -- Branchement si registre différent de registre.

Syntaxe : bne $rs, $rt, label

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

Instructions de Branchement

Assembleur	Opération		Format
Beq Rs, Rt, Label	Branch if Equal	PC <- PC+4+(I*4) if Rs = Rt PC <- PC+4 if Rs ≠ Rt	I
Bne Rs, Rt, Label	Branch if Not Equal	PC <- PC+4+(I*4) if Rs ≠ Rt PC <- PC+4 if Rs = Rt	I
Bgez Rs, Label	Branch if Greater or Equal Zero	PC <- PC+4+(I*4) if Rs ≥ 0 PC <- PC+4 if Rs < 0	I
Bgtz Rs, Label	Branch if Greater Than Zero	PC <- PC+4+(I*4) if Rs > 0 PC <- PC+4 if Rs ≤ 0	I
Blez Rs, Label	Branch if Less or Equal Zero	PC <- PC+4+(I*4) if Rs ≤ 0 PC <- PC + 4 if Rs > 0	I
Bltz Rs, Label	Branch if Less Than Zero	PC <- PC+4+(I*4) if Rs < 0 PC <- PC+4 if Rs ≥ 0	I
Bgezal Rs, Label	Branch if Greater or Equal Zero and link	PC <- PC+4+(I*4) if Rs ≥ 0 PC <- PC+4 if Rs < 0 R31 <- PC+4 in both cases	I
Bltzal Rs, Label	Branch if Less Than Zero and link	PC <- PC+4+(I*4) if Rs < 0 PC <- PC+4 if Rs ≥ 0 R31 <- PC+4 in both cases	I
J Label	Jump	PC <- PC 31:28 || I*4	J
Jal Label	Jump and Link	R31 <- PC+4 PC <- PC 31:28 || I*4	J
Jr Rs	Jump Register	PC <- Rs	R
Jalr Rs	Jump and Link Register	R31 <- PC+4 PC <- Rs	R
Jalr Rd, Rs	Jump and Link Register	Rd <- PC+4 PC <- Rs	R

J – Saut inconditionnel

Syntaxe :j label

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

8. Instructions de lecture/écriture mémoire

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

- lw $rd, imm($rs) rd← mem[imm + rs]

Lecture d'un mot de la mémoire. L'adresse de chargement est la somme de la valeur immédiate sur 16 bits et du contenu du registre $rs. Le contenu de cette adresse est placé dans le registre $rd.

- sw $rd, imm($rs) mem[imm + rs] ← rd

Écriture d'un mot en mémoire . L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec et du contenu du registre $rs. Le contenu du registre $rd est écrit à l'adresse ainsi calculée.

Instructions de lecture/écriture mémoire

Assembleur	Opération	Format
Lw Rt, I (Rs)	Load Word Rt <- M (Rs + I) sign extended Immediate	I
Sw Rt, I (Rs)	Store Word M (Rs + I) <- Rt sign extended Immediate	I
Lh Rt, I (Rs)	Load Half Word Rt <- M (Rs + I) sign extended Immediate. Two bytes from storage is loaded into the 2 less significant bytes of Rt. The sign of these 2 bytes is extended on the 2 most signficant bytes.	I
Lhu Rt, I (Rs)	Load Half Word Unsigned Rt <- M (Rs + I) sign extended Immediate. Two bytes from storage is loaded into the the 2 less significant bytes of Rt, other bytes are set to zero	I
Sh Rt, I (Rs)	Store Half Word M (Rs + I) <- Rt sign extended Immediate. The Two less significant bytes of Rt are stored into storage	I
Lb Rt, I (Rs)	Load Byte Rt <- M (Rs + I) sign extended Immediate. One byte from storage is loaded into the less significant byte of Rt. The sign of this byte is extended on the 3 most signficant bytes.	I
Lbu Rt, I (Rs)	Load Byte Unsigned Rt <- M (Rs + I) sign extended Immediate. One byte from storage is loaded into the less significant byte of Rt, other bytes are set to zero	I
Sb Rt, I (Rs)	Store Byte M (Rs + I) <- Rt sign extended Immediate. The less significant byte of Rt is stored into storage	I

Exemple:

- lw $t2, 10($t3) copie dans le registre $t2 la valeur située dans la mémoire principale à l'adresse m obtenue en ajoutant 10 au nombre stocké dans la registre $t3.
- sw $t2, 15($t1) copie la valeur présente dans le registre $t2 dans la mémoire principale à l'adresse m obtenue en ajoutant 15 au nombre stocké dans la registre $t1.

9. Instructions systèmes

Afin d'exécuter un programme en assembleur MIPS R3000 nous faisons appel à des fonctions SYSCALL. Un nombre de services system pour les entrées et les sorties sont disponibles à l'utilisation et sont décrits dans le tableau suivant.

Service	Code dans $v0	Arguments	Résultats
Afficher un entier	1	$a0 = entier à afficher	
Afficher un réel	2	$f12 = réel à afficher	
Afficher un double	3	$f12 = double à afficher	
Afficher une chaine de caractères	4	$a0=adresse de la chaine terminée à afficher	
Lire un entier	5		$v0 contient la valeur de l'entier lu
Lire un réel	6		$f0 contient la valeur du réel lu
Lire un double	7		$f0 contient la valeur double lu
Lire une chaine	8	$a0 = adresse du buffer d'entrée $a1 = nombre maximal de caractères à lire	
Sbrk (allouer de la mémoire)	9	$a0 = nombre d'octets à allouer	$v0 contient l'adresse de la mémoire allouée
Sortir (exécution terminée)	10		
Afficher un caractère	11	$a0 = caractère à afficher	
Lire un caractère	12		$v0 contient le caractère lu
Ouvrir un fichier	13	$a0 = adresse d'une chaine terminée contenant nom de fichier $a1 = drapeaux $a2 = mode	$v0 contient description du fichier (négative s'il y a erreur)
Lire à partir d'un fichier	14	$a0 = description de fichier $a1 = adresse du buffer d'entrée $a2= nombre maximal des caractères à lire	$v0 contient un nombre de caractères lus
Ecrire dans fichier	15	$a0 = description de fichier $a1 = adresse du buffer de sortie $a2= nombre maximal des caractères à écrire	$v0 contient un nombre de caractères écrits.
Fermer fichier	16	$a0 = description de fichier	
Sortir (Terminer avec valeur)	17	$a0= résultat de terminaison	

Exemple :

Ecrire un programme en assembleur MIPS R3000 qui affiche à l'écran le message "*Hello World !!*"

```
.data
monmessage :    .asciiz     "Hello World !!\n"
.text
li $v0, 4       # Afficher une chaine de caracteres
la $a0, monmessage          #la=load address
syscall
```

.asciiz termine la chaîne de caractères par NULL (0).