Architecture externe et langage d'assemblage du processeur MIPS32 en mode utilisateur

sept.-2020 — version 3.0
module LU3IN029[1]

Préambule

Le processeur MIPS est un processeur 32 bits de type RISC (Reduced Instruction Set Computer : c.-à-d. un processeur à jeu d'instructions réduit). Le MIPS dispose de deux modes d'exécution : un mode « *user* » pour les programmes utilisateur et un mode « *kernel* » pour le système d'exploitation. En mode « user », certaines instructions sont interdites et une partie de la mémoire n'est pas accessible. Ce document décrit uniquement ce qu'il faut connaître du MIPS pour écrire des programmes utilisateur en langage d'assemblage (on dit aussi : en assembleur).

Dans la deuxième partie de ce module, les programmes sont assemblés et exécutés sur MARS, un simulateur de MIPS. En fait, MARS est à la fois un programme d'assemblage (souvent nommé aussi assembleur) qui produit le code binaire, et un simulateur qui permet de visualiser le comportement du processeur instruction par instruction.

Ce document présente les registres et la mémoire accessibles aux programmes utilisateur, la syntaxe des instructions du MIPS, les appels système proposés par MARS, les conventions imposées pour les appels de fonctions et la gestion de la pile et une annexe avec les tables de codage ASCII et iso-latin1.

[1] Ce document a été rédigé initialement par Alain Greiner pour l'UE ALMO Architecture LOgicielle et Matérielle des ordinateurs et mis à jour pour l'UE Architecture LU3IN029

1. Registres utilisables en mode utilisateur

Le processeur possède 35 registres accessibles en mode utilisateur.

1.1. Les 32 registres GPR (General Purpose Register)

Il y a 32 registres numérotés de $0 à $31 que le programmeur peut utiliser pour faire ses calculs. Ils sont utilisables par toutes les instructions pour y mettre leurs opérandes et y stocker leur résultat. Ils sont tous équivalents, à 2 exceptions près :

- $0 contient la constante 0 :
  - La lecture fournit la valeur 0x00000000.
  - L'écriture ne modifie pas son contenu.
- $31 est utilisé par les instructions d'appel de fonctions pour sauver l'adresse de retour.
  - Les instructions d'appel de fonctions sont : bgezal, bltzal, jal et jalr

Même si les registres sont tous équivalents du point de vue matériel, il existe une convention d'usage qui définit lesquels sont utilisés comme argument(s) de fonctions, lesquels sont utilisés pour la gestion de la pile d'exécution, etc. Ceci est expliqué dans la section suivante.

1.2. Le registre PC

C'est le Program Counter ou compteur ordinal en français. Ce registre contient l'adresse de l'instruction en cours d'exécution. Sa valeur est modifiée par toutes les instructions.

1.3. Les registres HI et LO

Ces deux registres 32 bits sont dédiés à la multiplication entière et la division euclidienne. Ils sont utilisés pour stocker le résultat d'une multiplication ou d'une division.

- La multiplication de deux nombres de 32 bits est un mot de 64 bits dont les 32 bits de poids forts sont placés dans HI et les 32 bits de poids faibles dans LO.
- La division euclidienne de deux nombres 32 bits produit un quotient sur 32 bits placé dans LO et un reste sur 32 bits placé dans HI.

2. Conventions d'usage des registres GPR

Afin de normaliser l'écriture du logiciel, il existe des conventions d'usage des registres. Ces conventions sont nécessaires lors des appels de fonctions (cf section 9.). Dans le tableau ci-dessous, les noms entre parenthèses sont les noms symboliques des registres en rapport avec leur usage. Par exemple, $at est le nom symbolique de $1, $v0 de $2, $a0 de $4, etc.

$0	Vaut 0 en lecture. Non modifié par une écriture
$1 (at)	Réservé à l'assembleur pour les macro-instructions. L'usage est interdit pour les programmes.
$2, $3 (v0, v1)	Utilisés pour les calculs temporaires et la valeur de retour des fonctions, $2 est utilisé pour indiquer le n° de service syscall
$4 .. $7 (a0 .. a3)	Utilisés pour le passage des arguments de fonctions, leurs valeurs ne sont pas préservées lors des appels de fonctions. Les autres arguments sont placés dans la pile.
$8 .. $15,$24,$25 (t0 .. t9)	Registres de travail non-persistants, leurs valeurs ne sont pas préservées lors des appels de fonctions
$16 .. $23 (s0 .. s8)	Registres persistants, leurs valeurs sont préservées lors des appels de fonctions
$26,$27 (k0, k1)	Réservés à l'usage noyau du système d'exploitation
$28 (gp)	Pointeur sur la partie des variables globales (segment data) Utilisé par le compilateur, non utilisé dans cet UE
$29 (sp)	Pointeur de pile
$30 (fp)	Pointeur de cadre (pour les contextes de fonctions)
$31(ra)	Contient l'adresse de retour d'une fonction

Définition des acronymes

at	: Assembleur Temporary	k0, k1	: Kernel temporary 0 et 1
v0, v1	: Value 0 et Value 1	gp	: Global Pointer
a0 ... a3	: Argument 0 à Argument 3	sp	: Stack Pointer
t0 ... t9	: Temporary 0 à Temporary 9	fp	: Frame Pointer
s0 ... s8	: Saved value 0 à Saved value 8	ra	: Return Address

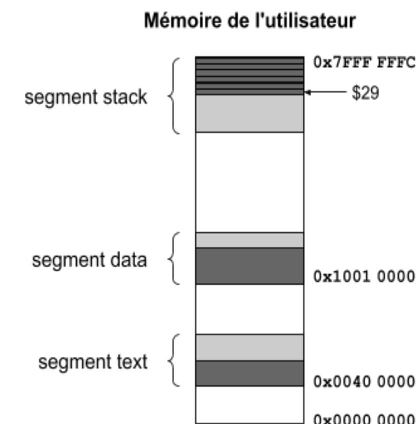
3. Organisation de la mémoire

Dans l'architecture MIPS32, l'espace adressable est divisé en deux parties : la partie accessible aux programmes utilisateur et la partie réservée au système. Dans ce document, nous ne parlons que de la partie utilisateur.

Un programme utilisateur utilise généralement trois segments d'adresses dans la partie utilisateur. Un segment d'adresses est un intervalle continu d'adresses.

- Le segment de code ou text contient le code exécutable en mode utilisateur. Il commence conventionnellement à l'adresse 0x0040 0000. Sa taille est fixe. La principale tâche de l'assembleur consiste à générer le code binaire correspondant au programme source décrit en langage d'assemblage, et ce code binaire sera chargé dans ce segment avant son exécution.
- Le segment de données ou data contient les données globales manipulées par le programme utilisateur. Il commence conventionnellement à l'adresse 0x1001 0000. Sa taille est fixe. Les valeurs contenues dans ce segment peuvent être initialisées par des directives présentes dans le programme source en langage d'assemblage. Les directives sont présentées en 4.5.
- Le segment de pile ou stack contient la pile d'exécution du programme utilisateur. Sa taille varie au cours de l'exécution du programme, et elle s'étend vers les adresses décroissantes. Elle est implantée conventionnellement en haut de l'espace d'adressage utilisateur (dernière adresse de mot : 0x7FFF FFFC).

Les segments d'adresses text et data sont remplis par des sections d'adresses définies dans le programme assembleur (voir 4.5.1.).



La figure de gauche représente l'espace d'adressage accessible quand le processeur est en mode utilisateur, c'est-à-dire quand il exécute un programme utilisateur.

En blanc, ce sont des adresses en dehors des segments autorisés, c'est-à-dire des adresses pour lesquels il n'y a pas de cases de mémoire physique[2].

En gris, ce sont les 3 segments de mémoire : text, data et stack.

Le gris foncé représente le remplissage des segments. Le programme binaire (contenant les sections **.text** et **.data** chargées en mémoire par le simulateur dans les segments **text** et **data**) n'occupe pas, en général, toute la mémoire disponible, et la pile dans le segment **stack** occupe une place variable dépendant du nombre de contextes de fonction présents.

[2] L'espace adressable peut être plus grand que la mémoire physique de la machine, donc à l'exécution certaines plages d'adresses ne sont pas associées à des cases physiques et ne sont pas autorisées.

4. Langage d'assemblage du MIPS32

On définit ci-dessous les principales règles d'écriture d'un programme en langage d'assemblage pour le simulateur de processeur MARS.

4.1. Nom d'un fichier en langage d'assemblage

Les noms des fichiers contenant un programme source en assembleur doivent être suffixés par «.s». Exemple: monprogramme.s

4.2. Commentaires dans les programmes

Les commentaires commencent par un # et s'achèvent à la fin de la ligne courante.

Exemple :
```
# fonction XYZ
lw   $10, 0($11)     # écrit $10 en mémoire à l'adr. $11
```

4.3. Forme des entiers dans les programmes

C'est comme en langage C. Une valeur entière exprimée en décimal (base 10) est une suite de chiffres de 0 à 9 ne commençant pas par 0, par exemple 250. Une valeur entière exprimée en hexadécimal (base 16) est préfixée par 0x (zéro puis x) suivi d'une suite de chiffres hexadécimaux de 0 à 9 et de A à F, par exemple 0xF4 (les lettres de A à F peuvent être écrites en majuscule ou en minuscule). Une valeur entière exprimée en octal (base 8) est une suite de chiffres de 0 à 7 commençant par 0, par exemple 025.

Par exemple, trois représentations de l'entier 42 en base 10, 16 et 8 : 42 = 0x2A = 052

4.4. Forme des chaînes de caractères dans les programmes

Elles sont simplement entre guillemets, et peuvent contenir les caractères d'échappement du langage C.

Exemple avec le retour à la ligne : "Oh la jolie chaine avec retour à la ligne\n".

4.5. Directives du langage d'assemblage

Les directives ne sont pas des instructions exécutables par le processeur, mais elles permettent de donner des ordres au programme d'assemblage. Toutes les directives commencent par le caractère «.» ce qui permet de les différencier des instructions.

Directives de déclaration des sections d'adresses

En langage d'assemblage, une section d'adresses est un intervalle d'adresses. Dans un programme en langage d'assemblage, il faut toujours indiquer la section dans laquelle doivent être placées les instructions et les données.

Le simulateur MARS supporte deux directives qui permettent de définir la section en cours de remplissage : **.text** pour le code et **.data** pour les données.

Lors de la fabrication du code binaire, toutes les sections .text vont dans le segment text (dit aussi de code) et toutes les sections .data vont dans le segment data (dit aussi de données).

.text Toutes les instructions qui suivent la directive .text sont placées dans une section .text. Lors de l'assemblage, l'ensemble des sections .text sont concaténées de manière contigüe dans le segment text.

.data Toutes les données qui suivent la directive .data sont placées dans une section .data. Lors de l'assemblage, l'ensemble des sections .data sont concaténées dans le segment data.

Pour chaque section, l'assembleur dispose d'un compteur de remplissage qu'il incrémente au fur et à mesure.

Directives de déclaration et d'initialisation des variables globales

Le remplissage de la section .data est défini par 7 directives : .**align** ; .**ascii** ; .**asciiz** ; .**byte** ; .**half** ; .**word** et .**space**. Ces directives permettent d'allouer de la place dans la section et pour 5 d'entres elles d'y initialiser des valeurs.

.align n

Aligne le compteur de remplissage sur une adresse telle que les n bits de poids faible soient à zéro.

Exemple:
```
.align 2   # demande l'alignement sur une adresse multiple de 4
.space 12  # 12 octets à une adresse multiple de 4
.align 2   # demande l'alignement sur une adresse multiple de 4
.word 24   # Ce mot est à une adresse multiple de 4
```

Notez que MARS fait lui même l'alignement pour les directives `.word` et `.half`, l'utilisation de la directive `align` n'est donc pas obligatoires avec MARS avec ces directives. Cependant si l'on veut faire un alignement différent de celui par défaut, par exemple sur des adresses multiples de 8, alors l'utilisation d'`align` est nécessaire. L'utilisation de la directive `align` est obligatoire avec la directive `.space` lorsque l'on souhaite allouer des octets qui correspondent à des mots ou demi-mots.

.ascii chaîne, [autrechaine]•••

Place la suite de caractères entre guillemets à partir de l'adresse du compteur de remplissage. S'il y a plusieurs chaînes, elles sont placées à la suite. Une chaîne peut contenir des séquences d'échappement du langage C, et doit être terminée par le caractère '\0' qui vaut 0x00 et est le marqueur de fin de chaîne. Notez bien que la présence d'un '\0' ou 0 (zéro) binaire en fin de chaîne est le seul moyen de connaître le nombre de caractères de la chaîne, c'est absolument nécessaire si on veut l'afficher sur le terminal avec un appel système (cf. syscall section 8.).

Exemple, allocation de 20 octets et initialisation avec une chaîne de caractères (dans cet exemple `message:` est une étiquette dit aussi *label* du programme, cf. section 4.6.) :

```
message:
     .ascii "Bonjour le monde !\n\0"
```

.asciiz chaîne, [autrechaine]…

Directive strictement identique à la précédente à la seule différence qu'elle ajoute le caractère de fin de chaîne (zéro binaire) à la fin de chaque chaîne.

Exemple, allocation de la même chaîne que précédemment :

```
message:
     .asciiz "Bonjour le monde !\n"
```

.byte n, [m]

Alloue des octets et les initialise avec les valeurs données en arguments.
Les valeurs sont tronquées à 8 bits (par exemple la valeur 257 définit la valeur 1).

Exemple d'allocation et initialisation de 13 octets :

```
table:
      .byte 1, 2, 4, 8, 16, 32, 64, 32, 16, 8, 4, 2, 257
```

.half n, [m]…

Alloue des demi-mots (2 octets) et les initialise avec les valeurs données en arguments. Les valeurs sont tronquées à 16 bits.

Exemple d'allocation et initialisation de 6 octets (le dernier demi-mot a la valeur 0x2345) :

```
Table:
      .half 0, 1024, 0x12345
```

.word n, [m]…

Alloue des mots (4 octets) et les initialise avec les valeurs données en arguments.
Les valeurs sont tronquées à 32 bits.

Exemple :

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

.space n

Alloue n octets qui sont initialisés à 0 qui est la valeur par défaut.

Exemple :

```
buffer:
     .space 1024    # alloue 1 kibi octets de mémoire
```

4.6. Déclaration des labels (étiquettes en français)

Les labels sont des noms donnés aux adresses en mémoire. Ces adresses peuvent être soit des adresses de variables (dans le segment data), soit des adresses d'instructions (dans le segment text). Ils doivent être suffixés par le caractère « : », mais pour y référer, on supprime le « : ».

Exemple, les labels « **message** » et « **__start** » :

```
.data
    i:   .word  12
    message:
```



```
                 .asciiz  "Ceci est une chaîne de caractères...\n"
         .text
             __start:
                 la      $4, message  # adresse de la chaîne dans $4
                 ori     $2, $0,      4    # code de l'appel système
dans $2
                 syscall              # cf. section 8.
```

Attention :

1. Les labels qui ont le même nom qu'un mot clé ou qu'une instruction de l'assembleur sont interdits.
2. « la » est une macro-instruction c'est-à-dire une instruction utilisable dans le langage d'assemblage qui n'a pas de correspondance dans le jeu d'instructions. Au moment de l'assemblage, elle est réécrite en utilisant les 2 instructions MIPS « lui » et « ori ». Dans la deuxième partie de cette UE, il est **interdit d'utiliser les macro-instructions.** C'est à vous de calculer l'adresse des labels et d'utiliser les instructions « lui » et « ori » avec les bonnes valeurs. Si le segment .data commence à 0x1001 0000, alors `message` est après le mot d'étiquette `i` qui a une taille de 4 octets
   ⇒ le message est à l'adresse 0x1001 0004. Donc :

   `la $4, message` doit être remplacé par :
   ```
   lui   $4, 0x1001
   ori   $4, $4, 0x0004
   ```
   MARS offre toutefois la possibilité d'obtenir les adresses des labels de données, leur utilisation dans la section .data vous permettra de vérifier vos calculs.

4.7. Arguments d'instructions

La plupart des instructions nécessitent un ou plusieurs arguments. Si une instruction possède plusieurs arguments, ils sont séparés par des virgules. Dans une instruction assembleur, on aura en général comme premier argument le numéro du registre dans lequel est mis le résultat de l'opération, puis ensuite le numéro du premier registre source, puis enfin le numéro du second registre source ou une valeur immédiate ou un label.

Exemple:
```
add  $8,  $4,  $5    #  $8 ←  $4 + $5
lb   $8,  42($4)     #  $8 ←  Mem[$4+42]
```

Argument de type registres à usage général (GPR)

Le processeur MIPS possède 32 registres à usage général (GPR pour General Purpose Register). Chaque registre a un numéro qui varie entre 0 et 31 préfixé par un $. Par exemple, le registre 31 est noté $31. En dehors du registre $0 qui contient toujours la valeur 0, tous les registres sont identiques du point de vue du processeur.

Argument de type valeurs immédiates

On appelle valeur immédiate entière un opérande dont la valeur est connue et contenue directement dans l'instruction. Ce sont des entiers signés ou non signés présents par exemple dans les instructions arithmétiques et logiques (comme addi ou ori).

Argument de type label

Un label, ou une étiquette en français, est une adresse (32 bits) en mémoire identifiée par un nom (cf. section 4.6.) qui est utilisée par les instructions de branchements conditionnels (p.ex. `bne`) ou par les sauts inconditionnels directs (p.ex. `jal`). Lors du codage des

instructions par le programme d'assemblage, le label est codé dans l'instruction en faisant une opération avec le PC (Program Counter), soit sur 16 bits (IMD16), soit sur 26 bits (IMD26) (cf. section 5.),

Pour les instructions de type branchement (p.ex. `bne`) IMD16 ← (label - PC+4) / 4
Pour les instructions de type saut direct (p.ex. `jal`) IMD26 ← label $_{28..2}$

Remarques :

1. Pour IMD26, on conserve 26 bits extrait de label par suppression des 4 bits de poids fort et des 2 bits de poids faible (ces 2 bit valent toujours 0 car les adresses d'instruction sont alignées).
2. Pour IMD16, c'est la différence entre « label » et « PC + 4» divisé par 4.
   La division par 4 s'explique par le fait que les instructions sont toujours à des adresses multiples de 4. Si le résultat de « (label - PC+4) / 4 » n'est pas représentable sur 16 bits, alors une erreur détectée au moment du codage. C'est un saut relatif par rapport au PC et le dépassement signifie que le label est trop loin.

4.8. Mode d'adressage de la mémoire

Le MIPS ne possède qu'un seul mode d'adressage pour lire ou écrire des données en mémoire : l'adressage indirect registre avec déplacement. Le déplacement est une valeur immédiate entière signée. L'adresse est obtenue en additionnant la valeur immédiate (positive ou négative) au contenu du registre.

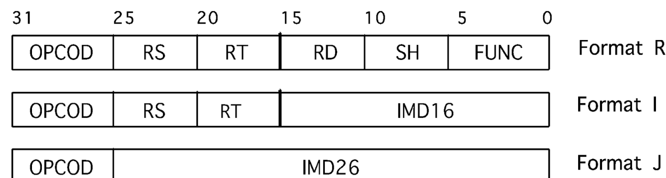
Exemples:
```
lw   $12, 16($10)    #  $12 ← Mem[$10 + 16]
sw   $20,-60($22)    #  Mem[$22 - 60] ← $20
```

5. Format des instructions

Le MIPS a 54 instructions utilisables pour les programmes en mode utilisateur :

- 32 instructions arithmétiques et logiques.
- 8 instructions d'accès à la mémoire
- 13 instructions de saut ou de branchement
- 1 instruction pour demander un service au noyau du système d'exploitation

Toutes les instructions font 32 bits et utilisent l'un des trois formats suivants :



Format R

Le format R est utilisé pour toutes les instructions ayant 2 registres sources (désignés par Rs et Rt) et un registre résultat désigné par Rd. La forme générale est :

OPCOD Rd, Rs, Rt réalisant Rd ← Rs OPCOD Rt.
Par exemple : sub $4, $8, $16 réalise $4 ← $8 - $16.

Il est aussi utilisé pour les instructions avec moins de 3 opérandes registres et aucun opérande immédiat (par exemple les multiplications et divisions), ainsi que pour toutes les instructions de décalage.

Format I

Le format I est utilisé par (a) les instructions de lecture/écriture mémoire, (b) les instructions utilisant un opérande immédiat, (c) les branchements à courte distance (conditionnels). La forme générale est

OPCOD Rt, Rs, IMD16 réalisant Rt ← Rs OPCOD IMD16

Par exemple : addi $4, $8, -42 réalise $4 ← $8 – 42
lb $4, 42($8) réalise $4 ← MEM[$8 + 42]

Format J

Le format J n'est utilisé que pour les branchements inconditionnels direct. La forme générale est

OPCOD IMD26 réalisant PC ← PC+IMD26*4.

Par exemple : j loin réalise PC ← loin

Notez que l'argument de l'instruction est un label du programme et que c'est le programme d'assemblage qui calcule la valeur IMD26. Il faut que l'adresse loin et PC aient les mêmes 4 bits de poids fort. On ne donc pas sauter à n'importe quelle adresse (taille du saut limitée). Pour ne pas avoir de contrainte, il faut utiliser l'instruction jr qui saute à une adresse contenue dans un registre et donc une adresse sur 32 bits.

6. Codage des instructions

Le codage des instructions est principalement défini par les 6 bits du champ code opération de l'instruction (INS 31:26). Cependant, trois valeurs particulières de ce champ définissent en fait une famille d'instructions : il faut alors analyser d'autres bits de l'instruction pour décoder l'instruction. Ces codes particuliers sont : SPECIAL (valeur "000000"), BCOND (valeur "000001") et COPRO (valeur "010000")

INS 31:29 \ INS 28:26	000	001	010	011	100	101	110	111
000	SPECIAL	BCOND	J	JAL	BEQ	BNE	BLEZ	BGTZ
001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
010	COPRO							
011								
100	LB	LH		LW	LBU	LHU		
101	SB	SH		SW				
110								
111								

Par exemple : l'instruction LHU possède l'OPCOD "100101".

Lorsque le code opération a la valeur SPECIAL ("000000"), il faut analyser les 6 bits de poids faible de l'instruction (**INS 5:0**):

INS 5:3 \ INS 2:0	000	001	010	011	100	101	110	111
000	SLL		SRL	SRA	SLLV		SRLV	SRAV
001	JR	JALR			SYSCALL	BREAK		
010	MFHI	MTHI	MFLO	MTLO				
011	MULT	MULTU	DIV	DIVU				
100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
101			SLT	SLTU				
110								
111								

Lorsque le code opération a la valeur BCOND, il faut analyser les bits 20 et 16 de l'instruction.

INS 20 \ INS 16	0	1
0	BLTZ	BGEZ
1	BLTZAL	BGEZAL

7. Instructions pour les programmes utilisateur

Les instructions du MIPS peuvent avoir de 0 à 3 opérandes. Dans ce qui suit, le registre de destination, c.-à-d. le registre qui reçoit le résultat de l'opération, est soit Rd soit Rt, les registres source qui contiennent les valeurs des opérandes sur lesquelles s'effectuent

l'opération sont Rs et Rt. Notez qu'un registre peut être alors fois un registre source et le registre destination de la même instruction (par exemple `add $4, $4, $2`).

Un opérande immédiat est noté imm ou sh. Lorsqu'il est noté imm sa taille dépend du codage de l'instruction, elle est de 16 ou de 26 bits pour les formats I et J respectivement, et est toujours étendue à 32 bits de manière non signée (avec des 0) ou signée (avec le bit de signe qui est le bit de poids fort) en fonction de l'instruction au moment de l'exécution. Lorsqu'il est noté sh (shift amount), sa taille est de 5 bits et correspond à un opérande d'une opération de décalage encodée dans le champ SH du format R.

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 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 :

Notation	Signification
+	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
modulo	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

Types des instructions

Instructions arithmétiques

Instruction	Opérandes	Opération	Format
add	Rd, Rs, Rt	Addition	format R
sub	Rd, Rs, Rt	Soustraction	format R
addu	Rd, Rs, Rt	Addition	format R
subu	Rd, Rs, Rt	Soustraction	format R
addi	Rt, Rs, imm	Addition	format I
addiu	Rt, Rs, imm	Addition	format I
mult	Rs, Rt	Multiplication	format R
multu	Rs, Rt	Multiplication	format R
div	Rs, Rt	Division	format R
divu	Rs, Rt	Division	format R
mfhi	Rd	Lecture de HI	format R
mflo	Rd	Lecture de LO	format R
mthi	Rd	Écriture de HI	format R
mtlo	Rd	Écriture de LO	format R

Instructions logiques

or	Rd, Rs, Rt	OU logique	format R
and	Rd, Rs, Rt	ET logique	format R
xor	Rd, Rs, Rt	OU exclusif logique	format R
nor	Rd, Rs, Rt	NON OU logique	format R
ori	Rt, Rs, imm	OU logique	format I
andi	Rt, Rs, imm	ET logique	format I
xori	Rt, Rs, imm	OU exclusif	format I

Instructions de décalages binaires

sllv	Rd, Rt, Rs	Décalage à gauche	format R
srlv	Rd, Rt, Rs	Décalage à droite	format R
srav	Rd, Rt, Rs	Décalage à droite arithmétique	format R
sll	Rd, Rt, sh	Décalage à gauche	format R
srl	Rd, Rt, sh	Décalage à droite	format R
sra	Rd, Rt, sh	Décalage à droite arithmétique	format R

Instructions de comparaison de nombre

slt	Rd, Rs, Rt	Mise à 1 si inférieur signé	format R
sltu	Rd, Rs, Rt	Mise à 1 si inférieur non signé	format R
slti	Rd, Rs, imm	Mise à 1 si inférieur signé	format I
sltiu	Rd, Rs, imm	Mise à 1 si inférieur non signé	format I
slti	Rt, Rs, imm	Mise à 1 si inférieur signé	format I
sltiu	Rt, Rs, imm	Mise à 1 si inférieur signé	format I

Instructions de lecture et d'écriture de données

lw	Rt, imm(Rs)	Lecture mot	format I
sw	Rt, imm(Rs)	Écriture mot	format I
lh	Rt, imm(Rs)	Lecture demi-mot signé	format I
lhu	Rt, imm(Rs)	Lecture demi-mot non-signé	format I
sh	Rt, imm(Rs)	Écriture demi-mot	format I
lb	Rt, imm(Rs)	Lecture octet signé	format I
lbu	Rt, imm(Rs)	Lecture octet non-signé	format I
sb	Rt, imm(Rs)	Écriture octet	format I

Instructions de branchements conditionnels

beq	Rs, Rt, Label	Saut si égalité	format I
bne	Rs, Rt, Label	Saut si différent	format I
bgez	Rs, Label	Saut si sup. ou égal à 0	format I
bgtz	Rs, Label	Saut si sup. à 0	format I
blez	Rs, Label	Saut si inf. ou égal à 0	format I
bltz	Rs, Label	Saut si inf. à 0	format I
bgezal	Rs, Label	Sauve PC et saut si sup. ou égal	format I
bltzal	Rs, Label	Sauve PC et saut si inf. ou égal	format I

Instructions de branchements inconditionnels

j	Label	Saut direct	format J
jal	Label	Sauve PC et saut direct	formal J
jr	Rs	Saut indirect	format R
jalr	Rs	Sauve PC et saut indirect	format R

Instruction de demande de service au système

Syscall		Sauve PC et saut au kernel	format R

Les instructions dans l'ordre alphabétique

add Addition registre registre signée

Syntaxe : add Rd, Rs, Rt
Description : Rd $\leftarrow$ Rs + Rt
Exception : génération d'une exception si dépassement de capacité.

Les contenus des registres Rs et Rt sont ajoutés pour former un résultat sur 32 bits qui est placé dans le registre Rd

addi Addition registre immédiat signée

Syntaxe : addi Rd, Rs, imm
Description : Rd $\leftarrow (imm_{15}^{16} \; || \; imm_{15...0}) + Rs$
Exception : génération d'une exception si dépassement de capacité.

La valeur immédiate imm sur 16 bits subit une extension de signe et est ajoutée au contenu du registre Rs pour former un résultat sur 32 bits qui est placé dans le registre Rd.

addiu Addition registre immédiat sans détection de dépassement

Syntaxe : addiu Rd, Rs, imm
Description : Rd $\leftarrow (imm_{15}^{16} \; || \; imm_{15...0}) + Rs$
Exception : pas d'exception générée en cas de dépassement de capacité

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

addu Addition registre registre sans détection de dépassement

Syntaxe : addu Rd, Rs, Rt
Description : Rd $\leftarrow$ Rs + Rt
Exception : pas d'exception générée en cas de dépassement de capacité

Les contenus des registres Rs et Rt sont additionnés
pour former un résultat sur 32 bits qui est placé dans le registre Rd

and Et bit-à-bit registre registre

Syntaxe : and Rd, Rs, Rt
Description : Rd $\leftarrow$ Rs and Rt
Exception : pas d'exception

ET bit-à-bit est effectué entre les contenus des registres Rs et Rt.
Le résultat est placé dans le registre Rd.

andi Et bit-à-bit registre immédiat

Syntaxe : andi Rd, Rs, imm
Description : Rd $\leftarrow (0^{16} \; || \; imm)$ and Rs
Exception : pas d'exception

La valeur immédiate sur 16 bits subit une extension de zéros.
ET bit-à-bit est effectué entre cette valeur étendue et le contenu du registre Rs pour former un résultat placé dans le registre Rd.

beq **Branchement si registre égal registre**

Syntaxe : beq Rs, Rt, label
Description : if (Rs == Rt) $pc \leftarrow pc+4+(imm_{15}^{14} \| imm \| 0^2)$ else $pc \leftarrow pc + 4$
Exception : Exception si adresse non alignée ou hors segment autorisé

Les contenus des registres Rs et Rt sont comparés. S'ils sont égaux, le programme saute à l'adresse label. imm est un entier signé sur 16 bits présent dans l'instruction et calculé au moment du codage comme (label - pc+4)/4 ⇒ à l'exécution pc ← label

bgez **Branchement si registre supérieur ou égal à zéro**

Syntaxe : bgez Rs, label
Description : if (Rs >= 0) $pc \leftarrow pc+4+(imm_{15}^{14} \| imm \| 0^2)$ else $pc \leftarrow pc + 4$
Exception : Exception si adresse non alignée ou hors segment autorisé

Si le contenu du registre Rs est supérieur ou égal à zéro, le programme saute à l'adresse label. imm est un entier signé sur 16 bits présent dans l'instruction et calculé au moment du codage comme (label - pc+4)/4 ⇒ à l'exécution pc ← label

bgezal **Branchement à une fonction si registre supérieur ou égal à zéro**

Syntaxe : bgezal Rs, label
Description : \$31 ← pc + 4
if (Rs >= 0) $pc \leftarrow pc+4+(imm_{15}^{14} \| imm \| 0^2)$ else $pc \leftarrow pc + 4$
Exception : Exception si adresse non alignée ou hors segment autorisé

Inconditionnellement, l'adresse de l'instruction suivant l'instruction bgezal est stockée dans le registre \$31. Si le contenu du registre Rs est supérieur ou égal à zéro, le programme saute à l'adresse label. imm est un entier signé sur 16 bits présent dans l'instruction et calculé au moment du codage comme (label - pc+4)/4 ⇒ à l'exécution pc ← label

bgtz **Branchement si registre strictement supérieur à zéro**

Syntaxe : bgtz Rs, label
Description : if (Rs > 0) $pc \leftarrow pc+4+(imm_{15}^{14} \| imm \| 0^2)$ else $pc \leftarrow pc + 4$
Exception : Exception si adresse non alignée ou hors segment autorisé

Si le contenu du registre Rs est strictement supérieur à zéro, le programme saute à l'adresse label. imm est un entier signé sur 16 bits présent dans l'instruction et calculé au moment du codage comme (label - pc+4)/4 ⇒ à l'exécution pc ← label

blez **Branchement si registre inférieur ou égal à zéro**

Syntaxe : blez Rs, label
Description : if (Rs <= 0) $pc \leftarrow pc+4+(imm_{15}^{14} \| imm \| 0^2)$ else $pc \leftarrow pc + 4$
Exception : Exception si adresse non alignée ou hors segment autorisé

Si le contenu du registre Rs est inférieur ou égal à zéro, le programme saute à l'adresse label. imm est un entier signé sur 16 bits présent dans l'instruction et calculé au moment du codage comme (label - pc+4)/4 ⇒ à l'exécution pc ← label

bltz **Branchement si registre strictement inférieur à zéro**

Syntaxe : bltz Rs, label
Description : if (Rs < 0) $pc \leftarrow pc+4+(imm_{15}^{14} \| imm \| 0^2)$ else $pc \leftarrow pc + 4$
Exception : Exception si adresse non alignée ou hors segment autorisé

Si le contenu du registre Rs est strictement inférieur à zéro, le programme saute à l'adresse label. imm est un entier signé sur 16 bits présent dans l'instruction et calculé au moment du codage comme (label - pc+4)/4 ⇒ à l'exécution pc ← label

bltzal **Branchement à une fonction si registre inférieur ou égal à zéro**

Syntaxe : bltzal Rs, label
Description : r31 ← pc + 4
if (Rs < 0) $pc \leftarrow pc+4+(imm_{15}^{14} \| imm \| 0^2)$ else $pc \leftarrow pc + 4$

Exception : Exception si adresse non alignée ou hors segment autorisé
Inconditionnellement, l'adresse de l'instruction suivant l'instruction bgezal est sauvée dans le registre \$31. Si le contenu du registre Rs est strictement inférieur à zéro le programme saute à l'adresse label. imm est un entier signé sur 16 bits présent dans l'instruction et calculé au moment du codage comme (label - pc+4)/4 ⇒ à l'exécution pc ← label

bne **Branchement si registre différent de registre**

Syntaxe : bne Rs, Rt, label
Description : if (Rs != Rt) $pc \leftarrow pc+4+(imm_{15}^{14} \| imm \| 0^2)$ else $pc \leftarrow pc + 4$
Exception : pas d'exception

Les contenus des registres Rs et Rt sont comparés. S'ils sont différents, le programme saute à l'adresse label, calculée par l'assembleur. imm est un entier signé sur 16 bits présent dans l'instruction et calculé au moment du codage comme (label - pc+4)/4 ⇒ à l'exécution pc ← label

div **Division entière signée**

Syntaxe : div Rs, Rt
Description : LO ← Rs / Rt ; HI ← Rs modulo Rt
Exception : Exception si Rt est nul

Le contenu du registre Rs est divisé par le contenu du registre Rt, le contenu des deux registres sont 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.

divu **Division entière non-signée**

Syntaxe : divu Rs, Rt
Description : LO ← (0 || Rs) / (0 || Rt) ; HI ← (0 || Rs) mod (0 || Rt)
Exception : Exception si Rt est nul

Le contenu du registre Rs est divisé par le contenu du registre Rt, le contenu des deux registres sont des nombres non signés. Le quotient de la division est placé dans le registre spécial LO, et le reste dans dans le registre spécial HI.

j **Saut inconditionnel immédiat**

Syntaxe : j label
Description : $pc \leftarrow pc_{31..29} \| label_{28..2} \| 0^2$
Exception : Exception si adresse hors segment autorisé

Le programme saute inconditionnellement à l'adresse correspondant au label.

jal **Appel de fonction inconditionnel immédiat**

Syntaxe : jal label

Description : $\$31 \leftarrow pc + 4 \; ; \; pc \leftarrow pc_{31..29} \,||\, label_{28..2} \,||\, 0^2$
Exception : Exception si adresse hors segment autorisé

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

jalr Appel de fonction inconditionnelle registre

Syntaxe : jalr Rs
Description : $\$31 \leftarrow pc + 4 \; ; \; pc \leftarrow Rs$
Exception : Exception si adresse non alignée ou hors segment autorisé

Le programme saute à l'adresse contenue dans le registre Rs . L'adresse de l'instruction suivant l'instruction jalr est sauvée dans le registre $31. Si le registre n'est pas spécifié, alors c'est par défaut le registre $31 qui est utilisé.

jr Branchement inconditionnel registre

Syntaxe : jr Rs
Description : $pc \leftarrow Rs$
Exception : Exception si adresse non alignée ou hors segment autorisé

Le programme saute à l'adresse contenue dans le registre Rs.

lb Lecture d'un octet signé en mémoire

Syntaxe : lb Rd, imm(Rs)
Description : $Rd \leftarrow (Mem[imm + Rs])_7^{24} \,||\, (Mem\}[imm + Rs])_{7...0}$
Exception : Exception si adresse non alignée ou hors segment autorisé

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre Rs. L'octet lu à cette adresse subit une extension de signe et est placé dans le registre Rd.

lbu Lecture d'un octet non-signé en mémoire

Syntaxe : lbu Rd, imm(Rs)
Description : $Rd \leftarrow (0)^{24} \,||\, (Mem\}[imm + Rs])_{7...0}$
Exception : Exception si adresse non alignée ou hors segment autorisé

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre Rs. L'octet lu à cette adresse subit une extension avec des zéros et est placé dans le registre Rd

lh Lecture d'un demi-mot signé en mémoire

Syntaxe : lh Rd, imm(Rs)
Description : $Rd \leftarrow (Mem[imm + Rs])_{15}^{16} \,||\, (Mem\}[imm + Rs])_{15...0}$
Exception : Exception si adresse non alignée ou hors segment autorisé

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre Rs. Le demi-mot de 16 bits lu à cette adresse subit une extension de signe et est placé dans le registre Rd.

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

Syntaxe : lhu Rd, imm(Rs)
Description : $Rd \leftarrow (0)^{16} \,||\, (Mem\}[imm + Rs])_{15...0}$
Exception : Exception si adresse non alignée ou hors segment autorisé

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre Rs. Le demi-mot de 16 bits lu à cette adresse subit une extension avec des zéro et est placé dans le registre Rd.

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

Syntaxe : lui Rd, imm
Description : $Rd \leftarrow imm \,||\, (0)^{16}$
Exception : pas d'exception

La constante immédiate de 16 bits est décalée de 16 bits à gauche, et est complétée de zéro. La valeur ainsi obtenue est placée dans le registre Rd.

lw Lecture d'un mot de la mémoire

Syntaxe : lw Rd, imm(Rs)
Description : $Rd \leftarrow Mem[imm + Rs]$
Exception : Exception si adresse non alignée ou hors segment autorisé

L'adresse de chargement est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre Rs. Le mot de 32 bits lu à cette adresse est placé dans le registre Rd.

mfhi Copie le registre HI dans un registre général

Syntaxe : mfhi Rd
Description : $Rd \leftarrow HI$
Exception : pas d'exception

Le contenu du registre HI --- qui est mis à jour par les opérations de multiplication ou de division --- est recopié dans le registre général Rd.

mflo Copie le registre LO dans un registre général

Syntaxe : mflo Rd
Description : $Rd \leftarrow LO$
Exception : pas d'exception

Le contenu du registre LO --- qui est mis à jour par les opérations de multiplication ou de division --- est recopié dans le registre général Rd.

mthi Copie un registre général dans le registre HI

Syntaxe : mthi Rd
Description : $HI \leftarrow Rd$
Exception : pas d'exception

Le contenu du registre général Rd est recopié dans le registre général HI.

mtlo Copie un registre général dans le registre LO

Syntaxe : mtlo Rd
Description : $LO \leftarrow Rd$
Exception : pas d'exception

Le contenu du registre général Rd est recopié dans le registre général LO.

mult Multiplication signée

Syntaxe : mult Rs, Rt
Description : $LO \leftarrow (Rs \times Rt)_{31...0} \; ; \; HI \leftarrow (Rs \times Rt)_{63...32}$
Exception : pas d'exception.

Le contenu du registre Rs est multiplié par le contenu du registre Rt, le contenu des deux registres sont 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.

multu Multiplication non-signée

Syntaxe : multu Rs, Rt
Description : LO $\leftarrow$ $((0 \mid\mid Rs) \times (0 \mid\mid Rt))_{31\ldots0}$; HI $\leftarrow$ $((0 \mid\mid Rs) \times (0 \mid\mid Rt))_{63\ldots32}$
Exception : pas d'exception.

Le contenu du registre Rs est multiplié par le contenu du registre Rt, le contenu des deux registres sont 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.

nor Non-ou bit-à-bit registre registre

Syntaxe : nor Rd, Rs, Rt
Description : Rd $\leftarrow$ Rs nor Rt
Exception : pas d'exception.

Un non-ou bit-à-bit est effectué entre les contenus des registres Rs et Rt.
Le résultat est placé dans le registre Rd.

or Ou bit-à-bit registre registre

Syntaxe : or Rd, Rs, Rt
Description : Rd $\leftarrow$ Rs or Rt
Exception : pas d'exception.

Un ou bit-à-bit est effectué entre les contenus des registres Rs et Rt.
Le résultat est placé dans le registre Rd.

ori Ou bit-à-bit registre immédiat

Syntaxe : ori Rd, Rs, imm
Description : Rd $\leftarrow$ $((0)^{16} \mid\mid imm)$ or Rs
Exception : pas d'exception.

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.

sb Écriture d'un octet en mémoire

Syntaxe : sb Rt, imm(Rs)
Description : Mem[imm + Rs] $\leftarrow$ $Rt_{7\ldots0}$
Exception : Exception si adresse non alignée ou hors segment autorisé

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

sh Écriture d'un demi-mot en mémoire

Syntaxe : sh Rt, imm(Rs)
Description : Mem[imm + Rs] $\leftarrow$ $Rt_{15\ldots0}$
Exception : Exception si adresse non alignée ou hors segment autorisé

L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre Rs. Les deux octets de poids faible du

registre Rt sont écrit à l'adresse ainsi calculée. Le bit de poids faible de cette adresse doit être à zéro.

sll Décalage à gauche immédiat

Syntaxe : sll Rd, Rs, sh
Description : Rd $\leftarrow$ $Rs_{(31 - imm)\ldots0} \mid\mid (0)^{sh}$
Exception : pas d'exception

Le registre est décalé à gauche de la valeur immédiate sh codée sur 5 bits, des zéros sont mis dans les bits de poids faibles. Le résultat est placé dans le registre Rd.

sllv Décalage à gauche registre

Syntaxe : sllv Rd, Rs, Rt
Description : Rd $\leftarrow$ $Rs_{(31 - Rt)\ldots0} \mid\mid (0)^{Rt}$
Exception : pas d'exception.

Le registre Rs est décalé à gauche du nombre de bits codés dans les 5 bits de poids faible du registre Rt, des zéros étant introduits dans les bits de poids faibles. Le résultat est placé dans le registre Rd.

slt Comparaison signée registre registre

Syntaxe : slt Rd, Rs, Rt
Description : If (Rs < Rt) Rd $\leftarrow$ 1 else Rd $\leftarrow$ 0
Exception : pas d'exception

Le contenu du registre Rs est comparé au contenu du registre Rt, les deux valeurs étant considérées comme des nombres signés. Si la valeur contenue dans Rs est strictement inférieure à celle contenue dans Rt, alors Rd prend la valeur un, sinon il prend la valeur zéro.

slti Comparaison signée registre immédiat

Syntaxe : slti Rd, Rs, imm
Description : If (Rs < imm) Rd $\leftarrow$ 1 else Rd $\leftarrow$ 0
Exception : pas d'exception

Le contenu du registre Rs 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 Rs est strictement inférieure à celle de l'immédiat, alors Rd prend la valeur 1, sinon il prend la valeur 0.

sltiu Comparaison non-signée registre immédiat

Syntaxe : sltiu Rd, Rs, imm
Description : if ((0||Rs) < (0||imm)) Rd $\leftarrow$ 1 else Rd $\leftarrow$ 0
Exception : pas d'exception

Le contenu du registre Rs 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 Rs est strictement inférieure à celle de l'immédiat étendu, alors Rd prend la valeur 1, sinon il prend la valeur 0.

sltu Comparaison non-signée registre registre

Syntaxe : sltu Rd, Rs, Rt
Description : if ((0||Rs) < (0||Rt)) Rd $\leftarrow$ 1 else Rd $\leftarrow$ 0
Exception : pas d'exception

Le contenu du registre Rs est comparé au contenu du registre Rt, les deux valeurs sont des nombres non-signés. Si la valeur dans Rs est strictement inférieur à celle dans Rt, alors Rd prend la valeur 1, sinon il prend la valeur 0.

sra Décalage à droite arithmétique immédiat

Syntaxe : sra Rd, Rs, sh
Description : Rd ← $(Rs_{31})^{sh} \| (Rs)_{31...sh}$
Exception : pas d'exception

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

srav Décalage à droite arithmétique registre

Syntaxe : srav Rd, Rs, Rt
Description : Rd ← $(Rs_{31})^{Rt} \| (Rs)_{31...Rt}$
Exception : pas d'exception.

Le registre Rs est décalé à droite du nombre de bits spécifié dans les 5 bits de poids faible du registre Rt, le bit de signe de Rs étant introduit dans les bits de poids fort. Le résultat est placé dans le registre Rd.

srl Décalage à droite logique immédiat

Syntaxe : srl Rd, Rs, sh
Description : Rd ← $(0)^{sh} \| (Rs)_{31...sh}$
Exception : pas d'exception

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

srlv Décalage à droite logique registre

Syntaxe : srlv Rd, Rs, Rt
Description : Rd ← $(0)^{Rt} \| (Rs)_{31...Rt}$
Exception : pas d'exception

Le registre Rs est décalé à droite du nombre de bits spécifié dans les 5 bits de poids faible du registre Rt des zéros étant introduits dans les bits de poids fort ainsi libérés. Le résultat est placé dans le registre Rd.

sub Soustraction registre registre signée

Syntaxe : sub Rd, Rs, Rt
Description : Rd ← Rs - Rt
Exception : génération d'une exception si dépassement de capacité

Le contenu du registre Rt est soustrait du contenu du registre Rs pour former un résultat sur 32 bits qui est placé dans le registre Rd.

subu Soustraction registre registre non-signée

Syntaxe : sub Rd, Rs, Rt
Description : Rd ← Rs – Rt
Exception : pas d'exception

Le contenu du registre Rt est soustrait du contenu du registre pour former un résultat sur 32 bits qui est placé dans le registre .

sw Écriture d'un mot en mémoire

Syntaxe : sw Rt, imm(Rs)
Description : Mem[imm + Rs] ← Rt
Exception : Exception si adresse non alignée ou hors segment autorisé

L'adresse d'écriture est la somme de la valeur immédiate sur 16 bits, avec extension de signe, et du contenu du registre Rs. Le contenu du registre Rt est écrit en mémoire à l'adresse ainsi calculée. Les deux bits de poids faible de cette adresse doivent être nuls

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

Syntaxe : syscall
Description : Pc ← 0x80000180 et passage en mode « kernel »
Exception : pas d'exception

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.
Les éventuels arguments de l'appel système sont dans les registres $4 à $7.

xor Ou-exclusif bit-à-bit registre registre

Syntaxe : xor Rd, Rs, Rt
Description : Rd ← Rs xor Rt
Exception : pas d'exception

Un ou-exclusif bit-à-bit est effectué entre les contenus des registres Rs et Rt.
Le résultat est placé dans le registre Rd

xori Ou-exclusif bit-à-bit registre immédiat

Syntaxe : xori Rd, Rs, imm
Description : Rd ← $((0)^{16} \| imm)$ xor Rs
Exception : pas d'exception

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 Rs pour former un résultat placé dans le registre Rd.

8. Appels système supportés par MARS

Certains traitements 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 caractères sur la console). En assembleur, un programme utilisateur doit effectuer un « appel système », en utilisant 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, $5, $6 et $7.

L'environnement de simulation MARS est limité, car il vise l'apprentissage de la programmation en assembleur. Il utilise comme seul périphérique un contrôleur de terminal TTY (Écran/Clavier). Le code des appels système n'est pas réellement exécuté

par le processeur MIPS, il est directement exécuté sur la station de travail qui effectue la simulation. Les appels système suivants sont supportés par MARS :

Écrire (afficher) un entier sur l'écran du terminal (syscall n° 1)

Il faut mettre l'entier à afficher dans le registre $4 et exécuter l'appel système numéro 1. Le nombre est représenté en base 10.

```
lui  $4,  0x1234    # stocke la valeur 0x12345678 dans $4
ori  $4,  $4, 0x5678
ori  $2,  $0, 1     # code de « print_integer » dans $2
syscall             # demande d'afficher la valeur numérique
```

Lire un entier depuis le clavier du terminal (syscall n° 5)

Pour lire un entier, il faut exécuter l'appel système numéro 5 et récupérer le résultat dans le registre $2.

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

Écrire un caractère seul sur l'écran du terminal (syscall n° 11)

Il faut mettre le code ascii du caractère à écrire dans le registre $4 et exécuter l'appel système numéro 11.

```
ori  $4,  $0, 'A'   # stocke le code ascii de 'A' dans $4
ori  $2,  $0,  11   # code de « print_char » dans $2
syscall             # affiche  le caractère
```

Lire un caractère seul depuis le clavier du terminal (syscall n° 12)

Pour lire un caractère, il faut exécuter l'appel système numéro 12 et récupérer le résultat dans le registre $2.

```
ori  $2,  $0,  12   # code de « read_char »
syscall             # $2 contient la valeur lue
```

Écrire une chaîne de caractères sur l'écran du terminal (syscall n° 4)

Une chaîne de caractères est identifiée par l'adresse de son premier caractère en mémoire et elle s'arrête au premier caractère nul (code ascii 0). Pour afficher une chaîne, il faut écrire l'adresse du premier octet dans $4 et exécuter l'appel système numéro 4.

```
.data               # début de la section de données
[...]               # on a déjà alloué 0x24 octets
str: .asciiz "Bonjour\n" # adresse : 0x10010024

.text
lui  $4,  0x1000    # charge le pointeur dans $4
ori  $4,  $4,  0x24 #
ori  $2,  $0,  4    # code de « print_string » dans $2
syscall             # affiche la chaîne pointée
```


Lire une chaîne de caractères sur la console (syscall n° 8)

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 d'octets). On place le pointeur que le buffer dans $4 et la taille du buffer dans $5, et on exécute l'appel système numéro 8.
Attention, la fin de la saisie se fait par un retour chariot et donc la fin de la chaîne est toujours `'\n'` suivi de `'\0'`le marqueur de fin chaîne (sauf lorsque la chaîne entrée est de taille égale à la taille du buffer -1).

```
.data               # début de la section de données
[...]               # supposons avoir déjà alloué 0x40 octets
buf:  .space 0x100  # adresse : 0x10010040

.text
lui  $4,  0x1000    # charge l'adresse du pointeur dans $4
ori  $4,  $4,  0x40 #
ori  $5,  $0,  0x100# charge la taille du buffer buf dans $5
ori  $2,  $0,  8    # numéro de « read_string » dans $2
syscall             # lit les caractères dans la chaîne buf
```

Terminer un programme (syscall n° 10)
L'appel système numéro 10 permet d'arrêter l'exécution du programme.

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

9. Conventions pour les appels de fonctions

9.1. Fonction appelante et fonction appelée

Pour expliquer les conventions d'appel de fonction, nous allons distinguer explicitement la fonction appelante et la fonction appelée. La fonction appelante (nommons-là **f()**) appelle la fonction appelée (nommons-là **g()**) en lui fournissant des arguments. Les propriétés « appelante » et « appelée » sont propres à chaque appel. En effet, la fonction « appelée », **g(),** peut à son tour devenir une fonction « appelante » si son code contient des appels de fonctions.

9.2. Type de registres

Lorsque l'on programme une fonction **f()**, on distingue deux types de registres utilisables (cf. section 2. page 2) :

- d'une part, les registres temporaires que la fonction **f()** peut modifier sans en restaurer la valeur initiale en sortant (sortir signifie un retour vers la fonction appelante de la fonction **f()** à la fin de son exécution),
- d'autre part, les registres persistants que la fonction **f()** peut utiliser, mais qu'elle doit sauvegarder pour pouvoir les restaurer dans leur état initial avant de sortir.

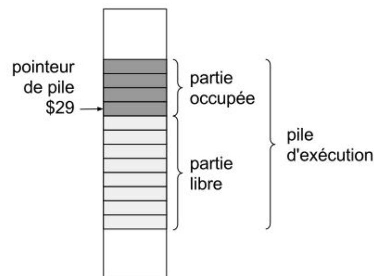
Seuls les registres $16 à $23 sont persistants, en conséquence, s'ils sont utilisés par la fonction alors ils doivent être sauvegardés au début de la fonction et restaurés à la fin. Notez que le registre $31 doit aussi être sauvegardé au début et restauré à la fin, mais c'est parce que la fonction appelante y a mis son adresse de retour et que la fonction appelée va peut-être appeler des fonctions et donc elle aussi utiliser $31.
Les registres $1 à $15 et $24, $25 sont temporaires et donc utilisables sans sauvegarde préalable. Certains registres temporaires ont un rôle spécifique :

- $1 est réservé pour l'assembleur
- $2 et $3 sont utilisés pour la valeur de retour de la fonction,
- $4 à $7 sont utilisés pour le passage des arguments.

9.3. Pile d'exécution des fonctions

Pour s'exécuter, une fonction **f()** utilise une zone en mémoire que l'on nomme contexte d'exécution que nous définissons juste après. Les contextes d'exécution des fonctions sont alloués dynamiquement dans la pile d'exécution.

- La pile d'exécution est la structure de données contenant les contextes d'exécution de toutes les fonctions.
- Le pointeur de pile se trouve dans registre **$29** par convention.
- La pile s'étend vers les adresses décroissantes.
- La pile est découpée en deux parties, une partie occupée et une partie libre.
- Le pointeur de pile contient toujours l'adresse de la dernière case occupée dans la pile. Ceci signifie que toutes les cases d'adresse inférieure au pointeur de pile sont libres.



9.4. Contexte d'exécution

Dans cette section, nous définissons le contexte de la fonction **f()** qui appelle plusieurs fonctions g() (c.-à-d. g0(), g1(), etc.).

Un contexte d'exécution contient 3 zones (R, V et A) correspondant à trois usages :

- la zone R permet à **f()** de sauvegarder les registres persistants qu'elle utilise, ainsi que le registre $31 contenant l'adresse de retour de **f()**,
- la zone V permet à **f()** de stocker ses variables locales

- la zone A permet à **f()** de passer les arguments aux fonctions qu'elle appelle.

Les arguments que la fonction **f()** a reçus de sa fonction appelante ne se trouvent pas dans le contexte de **f()**, ils se trouvent dans la zone A du contexte de sa fonction appelante.

9.4.1. Zone R : zone de sauvegarde des registres persistants

La fonction **f()** doit sauvegarder tous les registres persistants qu'elle utilise de façon à pouvoir restaurer leur valeur avant de revenir à la fonction appelante. Le registre $31 doit aussi être sauvegardé. En effet, le MIPS32 utilise le registre $31 pour stocker l'adresse de retour d'un appel de fonction. Cela signifie que lors d'un appel de fonction, lorsque le PC prend la valeur de la première instruction d'une fonction, le registre $31 contient l'adresse de retour de la fonction.

Les registres sont placés dans la pile suivant l'ordre croissant, le registre d'indice le plus petit à l'adresse la plus petite. Ainsi le registre $31, contenant l'adresse de retour, est toujours stocké à l'adresse la plus haute de la zone R.

On note **nr** le nombre de registres persistants, de taille 4 octets, à sauvegarder. Comme, le registre $31 n'est pas inclus dans ce nombre, il y aura donc nr + 1 registres à sauvegarder, et la taille de la zone R en octets doit être égale à 4*(nr +1).

9.4.2. Zone V : zone des variables locales de la fonction

Les valeurs stockées dans cette zone sont toujours lues et écrites par la fonction elle-même. Cette zone est utilisée pour stocker les variables et les structures de données locales à la fonction (c'est-à-dire l'ensemble des variables déclarées à l'intérieur de la fonction). La première variable déclarée se trouve à l'adresse la plus petite de la zone V, les autres variables sont rangées à la suite, en respectant leur ordre de déclaration et les contraintes d'alignement dictées par leur taille respective.

La contrainte d'alignement du pointeur de pile impose que cette zone V ait une taille multiple de 4 octets. On note **nv** le nombre de mots de 4 octets contenus la zone V. La taille de la zone V en octets est donc égale à 4*nv.

9.4.3 Zone A : zone des arguments des fonctions appelées

La fonction courante doit réserver de place pour le passage des arguments des fonctions qu'elle appelle. Ici, **f()** appelle une ou plusieurs fonctions **g()**.

On note **na** le nombre de mots de 4 octets nécessaires pour le passage des arguments.

Dans le cas général, une fonction **f()** appelle plusieurs fonctions (nommons ces fonctions les g0(), g1(), etc.). Ces fonctions ont généralement des nombres d'arguments différents et donc besoin d'un nombre de mots de 4 octets différents pour le passage de leurs arguments. On note na_{g0}, na_{g1}, … na_{gn}, avec na_{gi} les nombres de mots de 4 octets nécessaires pour le passage des arguments des fonctions g0(), g1(),... , gi(). Le nombre **na** est égale à la valeur maximale des na_{gi} : **na** = MAX(na_{gi}).

Le MIPS impose une optimisation[3] pour le passage des arguments. La fonction **f()** réserve toujours dans la pile l'espace nécessaire **pour tous** les arguments de la fonction appelée, mais elle n'y écrit pas les 4 premiers. Les valeurs des 4 premiers arguments sont écrites dans les registres $4, $5, $6 et $7. Si la fonction appelée contient plus de 4 arguments, les

[3] C'est une optimisation car cela évite des écritures et lectures mémoire à l'exécution pour les 4 premiers arguments.

valeurs des arguments suivants (à partir du 5e) sont écrites dans la pile, à la place qui leur a été réservée.

Cette optimisation impose de réserver 4 octets (1 mot) sur la pile **pour chacun** des 4 premiers arguments (on ne considérera pas, dans ce cours, le cas de premiers arguments de taille supérieure à 4 octets). Pour les arguments suivants, pour un appel de fonction donné, le nombre d'octets nécessaires sur la pile dépend de la taille des arguments et des contraintes d'alignement dictées par celles-ci. De plus, ce nombre doit être un multiple de 4 pour conserver l'alignement du pointeur de pile sur des mots, il correspond donc à un nombre de mots.

Quand on entre dans une fonction, le pointeur de pile $29 pointe sur la case de la pile où le premier argument devrait être (devrait puisqu'en réalité il est dans $4).

9.5. Accès à la pile

Le processeur MIPS32 ne possède pas d'instructions spécifiques à la gestion de la pile. On utilise les instructions de lecture (**lw, lh, lhu, lb, lbu**) et d'écriture (**sw, sh, sb**) pour lire et écrire dans la pile. L'instruction **addiu** est utilisée pour modifier la valeur du pointeur de pile.

L'accès à la pile est toujours relatif par rapport au pointeur de pile $29, par exemple:

- Pour lire un mot dans la pile, on utilise :
  `lw $i,d($29)` avec **d ≥ 0** et **$i € [$1...$31]**
- Pour écrire un mot dans la pile, on utilise :
  `sw $i,d($29)` avec **d ≥ 0** et **$i € [$0...$31]**

9.6. Usage de la pile dans le prologue et épilogue d'une fonction

À l'entrée de la fonction f(), le pointeur de pile pointe sur la case réservée pour le premier argument de f(). La fonction va exécuter une séquence d'instructions nommée **prologue** qui contient les étapes suivantes :

- Allocation du contexte d'exécution de f() en décrémentant le pointeur de pile :
  $29 ← $29 – 4*(nv + (nr + 1) + na)
- Écriture dans la pile des valeurs des nr registres persistants que f() va modifier ainsi que $31 en commençant par le registre $31 à l'adresse la plus grande et en rangeant ensuite les registres par numéro décroissant.
- Écriture éventuelle dans la pile des 4 premiers arguments reçus dans les registres $4 à $7. En effet, nous avons vu qu'à l'entrée de la fonction f(), les quatre premiers arguments sont placés dans les registres $4 à $7, mais si f() appelle d'autres fonctions, elle peut avoir besoin d'écraser les registres $4 à $7. Elle doit alors les écrire dans la pile dans les cases réservées par la fonction appelante de f().

À la sortie de la fonction f(), il faut exécuter une séquence nommée **épilogue** qui sert à :

- Restaurer les valeurs des nr registres ainsi que le registre $31.
- Restaurer le pointeur de pile à sa valeur initiale $29 ← $29 + 4*(nv + (nr + 1) + na)
- Sauter à l'adresse contenue dans $31

Entre le **prologue** et **l'épilogue** se trouve le **corps de la fonction** qui réalise les calculs en utilisant les registres temporaires et les registres persistants sauvés dans le prologue

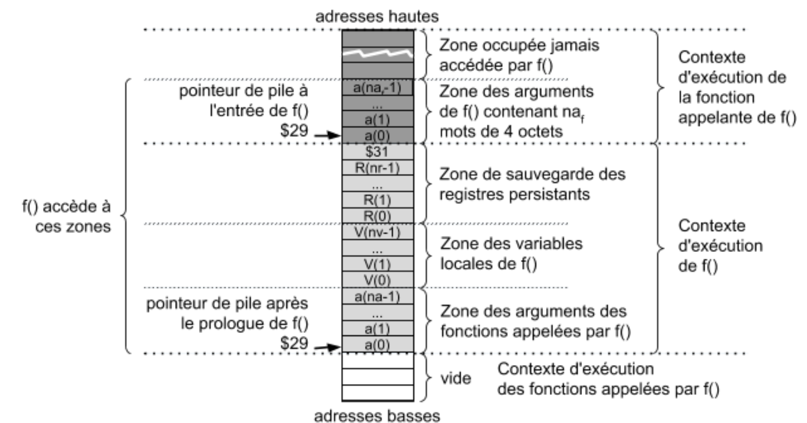
ainsi que les variables locales stockées dans la pile. Elle écrit la valeur de retour dans le registre $2.

Lors de l'appel d'une fonction g() par la fonction f(), le pointeur de pile $29 n'est pas modifié par f(), il faut juste :

- Placer les 4 premiers arguments dans les registres $4 à $7 et les autres dans la pile au-delà de la place réservée pour les 4 premiers.
- Effectuer le branchement à la première instruction de la fonction g() en utilisant une instruction de type jal (jump and link) pour enregistrer l'adresse de retour dans $31.

La figure suivante représente l'état de la pile après l'exécution du prologue de la fonction f(). On peut voir que la partie des arguments de la fonction f() a bien été allouée dans la pile par la fonction appelante de la fonction f().

C'est dans le prologue que la fonction f() alloue son propre contexte. f() accède donc à son contexte ET dans la zone A du contexte de sa fonction appelante pour récupérer ou sauver ses propres arguments.



9.7 Exemple complet

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 vue comme un entier aussi. 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) ); /* réalisé par l'appel système n° 1 */
      exit() ;                    /* réalisé par l'appel système n° 10 */
```

```
}

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

Quelques remarques :

- Le programme principal *main()* n'est pas vu comme une fonction, il se termine **toujours** par un appel système *exit()* qui met fin **définitivement** à l'exécution du programme. Pour cette raison, il n'y a pas de sauvegarde ni restauration des registres persistants qu'il utilise, ni de $31. Si le programme principal était vu comme une fonction, alors il faudrait suivre les conventions pour les registres persistants et $31.
- La fonction *main()* n'a pas de variables locales (nv = 0) et appelle la fonction *norme()* avec deux arguments. Pour les 4 premiers arguments, conformément aux conventions, il faut réserver 1 mot par argument dans la pile, donc il faut réserver 2 mots (na = 2) dans la pile pour les arguments de *norme()*.
  NB : il faut penser à restaurer le pointeur de pile avant l'appel système *exit()*.
- La fonction *norme()* déclare deux variables locales *somme* et *val* de type *int* donc nv = 2. Elle utilise deux registres de travail temporaires $8 et $9 mais aucun registre persistant donc nr=0. Elle appelle la fonction *isqrt()* qui a 1 argument et donc na = 1 (même raison que pour la fonction *norme()*). Il faut donc réserver 4 mots dans la pile (1 pour $31, 2 pour les variables locales, 1 pour l'argument de *isqrt()*)
- Les deux fonctions *isqrt()* et *norme()* renvoient leur résultat dans le registre $2. L'exemple ne contient pas le code de *isqrt()*.

Le programme assembleur est le suivant :

```
.data

x :  .word    5           # adresse 0x10010000
y :  .word    4           # adresse 0x10010004

.text

main :

addiu  $29,  $29,  -8    # décrémente pointeur de pile pour x et y
lui    $4,   0x1001      # Ecriture 1er parametre x
lw     $4,   0($4)
lui    $5,   0x1001      # Écriture 2e  paramètre y
ori    $5,   $5,   4
lw     $5,   0($5)
jal    norme
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
addiu  $29, $29,   8     # restaure l'état du pointeur de pile
ori    $2, $0, 10        # code de « exit » dans $2
syscall                  # sortie du programme
```



```
norme :

# prologue nv=2, nr=0 + $31, na = 1

addiu  $29,  $29,  -16   # décrémente pointeur de pile (2+1+1)
sw     $31,  12($29)     # sauvegarde adresse de retour

# corps de la fonction

mult   $4,   $4          # calcul a*a
mflo   $4
mult   $5,   $5          # calcul b*b
mflo   $5
addu   $4,   $4,   $5    # calcul somme dans $4 (1er argument)
jal    isqrt             # appel de isqrt (résultat dans $2)

# épilogue

lw     $31,  12($29)     # restaure adresse de retour
addiu  $29,  $29,  16    # restaure l'état du pointeur de pile
jr     $31               # retour de la fonction norme
```

Annexe

Codage ASCII et extension iso-latin1

Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8																
9																
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ