Chapitre 2

Assembleur MIPS

Sommaire

2.1 Présentation de l'architecture de la machine MIPS R2000 . . . . . **1**
2.1.1 Généralités sur les processeurs 1
2.1.2 Le MIPS R2000 2
2.2 Registres, mémoire, directives **4**
2.2.1 Les registres 4
2.2.2 Machine virtuelle assembleur 5
2.2.3 Assembleur 6
2.3 Les instructions **7**
2.3.1 Accès à la mémoire 8
2.3.2 Opérations arithmétiques 8
2.3.3 Opérations logiques 9
2.3.4 Instructions de comparaison 10
2.3.5 Instructions de contrôle 10
2.4 Programmation en assembleur **11**
2.4.1 Les appels système MARS 11
2.4.2 Assembleur avancé 11
2.4.3 Exemples 12
Factorielle (itératif) 12

2.1 Présentation de l'architecture de la machine MIPS R2000

2.1.1 Généralités sur les processeurs

Architecture des ordinateurs

Microprocesseur Exécute des instructions élémentaires (langage machine)

Mémoire Contient les programmes en cours d'exécution et leurs données

Disque, etc.

La mémoire est divisée en cases d'un octet (ou *byte*, 8 bits).
Chaque case a une adresse (nombre entier).
Le microprocesseur est cadencé par une *horloge* (signal régulier rapide, imposant un rythme au circuit et, assurant éventuellement une synchronisation avec les autres composants, tel que la mémoire).
Le microprocesseur contient un certain nombre de *registres* (cases mémoire à accès très rapide).
Taille des registres : 1 *mot* (word) (32 ou 64 bits sur les processeurs actuels)

Micro-Processeurs

Intel/AMD x86 (8086, 80286, 80386, pentium,...) utilisés dans les compatibles PC (et les Macintosh depuis 2006).

IBM/Motorola PowerPC utilisés notamment dans les Macintosh, jusqu'en 2006, des consoles de jeux, etc.

Motorola 6809, 6502, 68000,... Odinateurs Thomson (6809), vieux Apple (6502, 68000)

Sun Sparc stations de travail Sun

MIPS stations Silicon Graphics, consoles de jeux, routeurs, freebox...

DEC Alpha stations DEC

ARM systèmes embarqués

CISC/RISC

Deux catégories de processeurs :

CISC	RISC
AMD/Intel, Motorola...	PowerPC, MIPS, SPARC, Alpha, ARM
Complex Instruction Set Computer	*Reduced Instruction Set Computer*
Instructions complexes (plusieurs cycles d'horloge)	Instructions simples (un cycle)
Instructions de taille variable	Instructions de taille fixe (1 mot)
Instructions séquentielles	Pipeline
Peu de registres	Beaucoup de registres

2.1.2 Le MIPS R2000

Le processeur MIPS R2000 et l'émulateur MARS

Pour ce cours, on utilisera comme machine cible le processeur MIPS R2000, un RISC pour lequel on dispose d'un émulateur, MARS.

Un exemple

```
        .text
size:
        lw $a0,4($29)    # address of asciiz passed on the stack
        li $v0,0         # $v0 will contain the length
__loop: lbu $v1,0($a0)   # load the next byte in the string in $3
        beqz $v1,__end   # end if zero
        add $a0,$a0,1    # result in $a0
        add $v0,$v0,1    #
        j __loop
__end:  jr $ra           # return to CTigre calling function
```

.text est une directive de l'assembleur size: est une étiquette

Architecture du processeur MIPS 2000

L'architecture des processeurs MIPS est simple et régulière, ce qui en facilite la compréhension et l'apprentissage. On peut résumer en quelques points les choix architecturaux essentiels :

registres 32 registres génériques de 32 bits. *Toutes les opérations sont disponibles sur tous les registres !*

taille du mot *un `word` fait 4 octets, i.e. 32 bits (la taille d'un registre)*
L'accès à la mémoire est en général aligné *sur les mots*
- On peut adresser 2^{32} octets
- Les adresses sont des multiples de 4

(comment sont les 2 derniers bits d'une adresse alignée ?)

load/store l'accès à la mémoire se fait *seulement* par des instructions explicites de chargement et mémorisation à partir de registres *On adresse l'octet, mais on charge un mot*

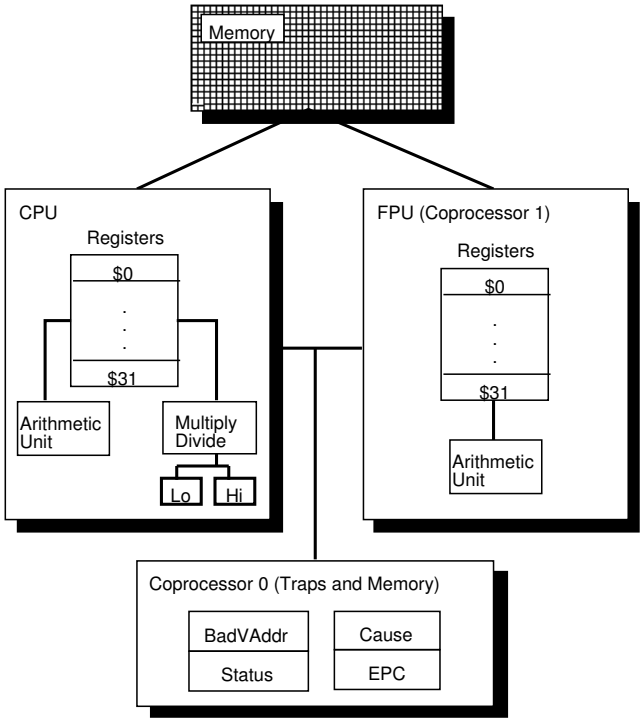
arithmétique les opérations arithmétiques utilisent deux registres en entrée et un registre en sortie

format les instructions tiennent toutes sur 32 bits (un mot) *Cela a des conséquences de taille sur les valeurs immédiates !*

	31 30 29 28 27 26	25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 09 08 07 06	05 04 03 02 01 00
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (19)		
J	opcode (6)	address (26)				

opcode = code opération
funct = complément à l'opcode
shamt = décalage (pour les instructions de décalage)
rs,rt,rd = registres source1, source2, destination
immediate = valeur immédiate
address = décalage (adresse relative)
Exemples :

Instruction	Format	Opcode	funct
add	R	0	$0x20$
addu	R	0	$0x21$
addi	I	$0x16$	



Exercice : comparez avec ce que vous savez de la famille x86.

2.2 Registres, mémoire, directives

2.2.1 Les registres

Registres, et conventions

Les 32 registres sont tous équivalents, *mais* pour garantir l'intéropérabilité entre programmes assembleurs produits par des compilateurs différents, on a fixé des *conventions d'usage* (voir table 2.1) :

- Les registres $a0...$a3 sont utilisés pour passer les premiers 4 paramètres d'une fonction lors d'un appel
- Les temporaires « sauvegardés » doivent être préservés par la fonction appelée, les temporaires « non sauvegardés » peuvent être écrasés lors d'un appel de fonction
- Le registre $sp pointe vers le bas de la pile (dernière case utilisée)
- $fp sert à stocker le haut de la portion de pile utilisée par la fonction (ancien $sp).
- $gp pointe vers le milieu du tas (variables globales). Permet d'accéder plus rapidement au données (en spécifiant un décalage par rapport à $gp plutôt qu'une adresse absolue sur 32 bits, qui ne tiendrait pas dans une seule instruction).
- $ra contient l'adresse de retour après l'appel à une fonction
- Le registre zero contient toujours la constante 0.

Registre	Numéro	Usage	Préservé par l'appelé
zero	0	Constante 0	
at	1	Réservé pour l'assembleur	N
v0	2	Évaluation d'expressions et	N
v1	3	résultats d'une fonction	N
a0-a3	4-7	Arguments 1 à 4	N
t0-t7	8-15	Temporaires	N
s0-s7	16-23	Temporaires sauvegardés	O
t8	24	Temporaire	N
t9	25	Temporaire	N
k0	26	Réservé pour le système d'exploitation	N
k1	27	Réservé pour le système d'exploitation	N
gp	28	Pointeur vers la zone des variables globale	O
sp	29	Pointeur de pile (stack pointer)	O
fp	30	Pointeur de bloc (frame pointer)	O
ra	31	Adresse de retour (pour les fonctions)	O

TAB. 2.1 – Registres MIPS et conventions d'usage

2.2.2 Machine virtuelle assembleur

Pseudo-instructions

Les instructions (RISC surtout) étant très élémentaires, on ajoute des *pseudo-instructions*, qui n'existent pas réellement pour le processeur, mais sont traduites vers une suite d'instructions réelles par l'assembleur.

Exemple :

```
la $a0, OxA123B999
```

est remplacé par l'assembleur par :

```
lui $1, 0xA123
ori $4, $1, 0xB999
```

Ici `lui` signifie *load upper immediate*, c'est-à-dire « charge le demi-mot de poids fort de la valeur immédiate dans le demi-mot de poids fort du registre ». `ori` fait un « ou logique » avec une valeur immédiate. Rappelez-vous que les instructions MIPS font toutes 32 bits, donc on ne peut manipuler des valeurs immédiates de 32 bits en une seule instruction.

Machine virtuelle assembleur

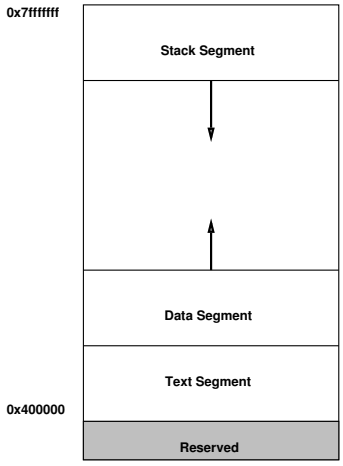
Pour des soucis d'efficacité, la machine MIPS réelle dispose d'instructions *retardées*, permettant d'accélerer l'éxécution des sauts non locaux, et impose des restrictions sur les modes d'adressage, qui compliquent la compréhension du système.

L'assembleur MIPS offre à l'utilisateur une interface qui cache la complexité de l'architecture matérielle réelle.

Dans ce cours, nous nous intéressons exclusivement à la *machine virtuelle* assembleur.

Machine virtuelle assembleur : mémoire

La mémoire sur l'émulateur MARS est divisée en zones, comme suit :



Machine virtuelle assembleur : Byte Order

Un word occupe plusieurs octets consécutifs : Soit le mot 00000000000000000000010000000001 (l'entier 1025). On dispose pour le mémoriser des 4 octets consécutifs aux adresses 10, 11, 12, et 13, on peut le mémoriser comme ça :

Little endian (PC...)

addr	contenu
10	00000001
11	00000100
12	00000000
13	00000000

ou comme ça :

Big endian (SPARC, 68000, PowerPC...)

addr	contenu
10	00000000
11	00000000
12	00000100
13	00000001

Pour s'en souvenir, Wikipedia nous dit : « Endianness does not denote what the value ends with when stored in memory, but rather which end it begins with » ("big end in" - the big end goes in first).

Les MIPS peuvent suivre les deux conventions. MARS suit la convention de la machine hôte !

2.2.3 Assembleur

Machine virtuelle assembleur : syntaxe

Commentaires tout ce qui suit # est ignoré

```
# ceci est un commentaire
```

Identificateurs séquence alphanumérique ne commençant pas par un entier ; les symboles _ et . sont admis ; les noms des instructions assembleur sont réservés

```
j end_loop.2
```

Étiquettes identificateurs qui se trouvent au debut d'une ligne et sont suivis de « : »

```
v:              .word 33
end_loop.2:     lw $a0, v
```

Chaînes de caractères sont délimitées par ". Les caractères spéciaux suivent les conventions habituelles :

fin de ligne	\n
tabulation	\t
guillemets	\"

Machine virtuelle assembleur : directives

Pour le projet, les principales directives à connaître sont les suivantes (pour les autres, voir le document de référence sur la page du cours) :

`.ascii str` Met la chaîne en mémoire, sans terminaison

`.asciiz str` Met la chaîne en mémoire, terminée par un octet à 0

`.data <addr>` Ce qui suit doit aller dans le segment DATA (éventuellement à partir de l'adresse addr)

`.text <addr>` Ce qui suit doit aller dans le segment TEXT (éventuellement à partir de l'adresse addr)

`.word w1, ..., wn` Met les n valeurs sur 32-bit dans des mots successifs

Modes d'adressage

La machine virtuelle assembleur fournit les modes d'adressage suivants pour accéder à la mémoire :

Format	Adresse
(register)	contenu du registre
imm	la valeur est immédiate
imm (register)	imm + contenu du registre
symbol	adresse du symbole
symbol ± imm	adresse du symbole ± valeur immédiate
symbol ± imm (register)	adresse du symbole ± (imm + contenu du registre)

N.B. : la plupart des accès mémoire sont *alignés*

2.3 Les instructions

Les instructions marquées « † » sont en fait des pseudo-instructions.

2.3.1 Accès à la mémoire

la Rdest, adresse *Load Address* †
Charge l'*adresse*, (et non le contenu de la case mémoire correspondante), dans le registre Rdest.

```
        .data
start:  .word 0,0,0,0,0,0,0,0,0,0,1
        .text
        la $t0, start+28
```

lw Rdest, adresse *Load Word*
Charge l'entier 32-bit (word) qui se trouve à l'*adresse* dans le registre Rdest.

```
lw $t0, 32($sp)
lw $t1, start+2($a0)
```

li Rdest, imm *Load Immediate* †
Charge la valeur imm dans le registre Rdest.

```
li $t0, 0x12345678
```

sw Rsrc, adresse *Store Word*
Mémorise le mot contenu dans le registre Rsrc à l'*adresse*.

```
sw $t0, 32($sp)
```

Copie de registres

move Rdest, Rsrc *Move* †
Copie le contenu de Rsrc dans Rdest.

```
move $t0, $a1
```

2.3.2 Opérations arithmétiques

Dans ce qui suit, Src2 peut-être un registre ou une valeur immédiate sur 16 bits.

add Rdest, Rsrc1, Src2 *Addition (with overflow)*
addi Rdest, Rsrc1, Imm *Addition Immediate (with overflow)*
addu Rdest, Rsrc1, Src2 *Addition (without overflow)*
addiu Rdest, Rsrc1, Imm *Addition Immediate (without overflow)*
Somme de registre Rsrc1 et Src2 (ou Imm) dans le registre Rdest.

div Rdest, Rsrc1, Src2 *Divide (signed, with overflow)* †
Met dans le registre Rdest le quotient de la division de Rsrc1 par Src2 dans Rdest.

mul Rdest, Rsrc1, Src2 — *Multiply (without overflow)* †
mulo Rdest, Rsrc1, Src2 — *Multiply (with overflow)* †

mulou Rdest, Rsrc1, Src2 — *Unsigned Multiply (with overflow)* †
Met le produit de Rsrc1 et Src2 dans le registre Rdest.

rem Rdest, Rsrc1, Src2 — *Remainder* †
remu Rdest, Rsrc1, Src2 — *Unsigned Remainder* †
Met le reste de la division de Rsrc1 par Src2 dans Rdest.

sub Rdest, Rsrc1, Src2 — *Subtract (with overflow)*
subu Rdest, Rsrc1, Src2 — *Subtract (without overflow)*
Met la différence entre Rsrc1 et Src2 dans Rdest.

2.3.3 Opérations logiques

and Rdest, Rsrc1, Src2 — *AND*
andi Rdest, Rsrc1, Imm — *AND Immediate*
AND logique de Rsrc1 et Src2 (ou Imm) dans le registre Rdest.

not Rdest, Rsrc — *NOT* †
Met la negation logique de Rsrc dans Rdest.

or Rdest, Rsrc1, Src2 — *OR*
ori Rdest, Rsrc1, Imm — *OR Immediate*
Met le OU logique de Rsrc1 et Src2 (ou Imm) dans Rdest.

xor Rdest, Rsrc1, Src2 — *XOR*
xori Rdest, Rsrc1, Imm — *XOR Immediate*
Met le XOR de Rsrc1 et Src2 (ou Imm) dans Rdest.

rol Rdest, Rsrc1, Src2 — *Rotate Left* †
ror Rdest, Rsrc1, Src2 — *Rotate Right* †
Rotation du contenu de Rsrc1 à gauche (droite) du nombre de places indiqué par Src2 ; le résultat va dans le registre Rdest.

sll Rdest, Rsrc1, Src2 — *Shift Left Logical*
sllv Rdest, Rsrc1, Rsrc2 — *Shift Left Logical Variable*
sra Rdest, Rsrc1, Src2 — *Shift Right Arithmetic*
Pour les valeurs signées. Le bit de signe est propagé à droite, ce qui a l'effet escompté : division par 2^k.
sra 100100 2 = 111001
srav Rdest, Rsrc1, Rsrc2 — *Shift Right Arithmetic Variable*
srl Rdest, Rsrc1, Src2 — *Shift Right Logical*
srlv Rdest, Rsrc1, Rsrc2 — *Shift Right Logical Variable*
Décale Rsrc1 à gauche (droite) du nombre de places indiqué par Src2 (Rsrc2) et met le résultat dans Rdest.

2.3.4 Instructions de comparaison

(Src2 est un registre, ou une valeur 32 bits)

seq Rdest, Rsrc1, Src2 — *Set Equal* †
Met le registre Rdest à 1 si le registre Rsrc1 est égal à Src2, et à 0 sinon.

sge Rdest, Rsrc1, Src2 — *Set Greater Than Equal* †
sgeu Rdest, Rsrc1, Src2 — *Set Greater Than Equal Unsigned* †
Met le registre Rdest à 1 si le registre Rsrc1 est plus grand que, ou égal à Src2, et à 0 sinon.

sgt Rdest, Rsrc1, Src2 — *Set Greater Than* †
sgtu Rdest, Rsrc1, Src2 — *Set Greater Than Unsigned* †
Met registre Rdest à 1 si le registre Rsrc1 est plus grand que Src2, et à 0 sinon.

sle Rdest, Rsrc1, Src2 — *Set Less Than Equal* †
sleu Rdest, Rsrc1, Src2 — *Set Less Than Equal Unsigned* †
Met le registre Rdest à 1 si le registre Rsrc1 est plus petit ou égal à Src2, et à 0 sinon.

slt Rdest, Rsrc1, Src2 — *Set Less Than*
slti Rdest, Rsrc1, Imm — *Set Less Than Immediate*
sltu Rdest, Rsrc1, Src2 — *Set Less Than Unsigned*
sltiu Rdest, Rsrc1, Imm — *Set Less Than Unsigned Immediate*
Met le registre Rdest à 1 si le registre Rsrc1 est plus petit que Src2 (or Imm), et à 0 sinon.

sne Rdest, Rsrc1, Src2 — *Set Not Equal* †
Met le registre Rdest à 1 si le registre Rsrc1 is not equal to Src2, et à 0 sinon.

2.3.5 Instructions de contrôle

On dispose de sauts « courts », appelés branch
court = le décalage est sur 16 bit, donc on peut sauter $2^{15} - 1$ instructions *en avant ou* 2^{15} *instructions en arrière (*Src2 *est un registre ou une constante)*

b label — Branch instruction †
Saut (court) inconditionnel à `label`.

beq Rsrc1, Src2, label — Branch on Equal
Saut (court) conditionnel à `label` *si le contenu de* Rsrc1 *est égale à* Src2.

bgt Rsrc1, Src2, label — Branch on Greater Than †
Saute à label *si le contenu de* `Rsrc1` *est plus grand que celui de* `Src2`

blt Rsrc1, Src2, label — Branch on Less Than †
bne Rsrc1, Src2, label — Branch on Not Equal

etc.

On dispose aussi de sauts « longs » (décalage sur 26 bits).

Service	Code	Arguments	Résultat
print_int	1	`$a0` = integer	
print_string	4	`$a0` = string	
read_int	5		integer (in `$v0`)
read_string	8	`$a0` = buffer `$a1` = length	
sbrk (alloue un bloc de mémoire)	9	`$a0` = amount	adresse (in `$v0`)
exit	10		

TAB. 2.2 – Appels système MARS

`j label` *Jump*
Saut long à `label` .

`jal label` *Jump and Link*
Saut long à `label` . Sauve l'adresse de la prochaine instruction dans le registre 31. Cela sert pour les appels de fonction.

2.4 Programmation en assembleur

2.4.1 Les appels système MARS

MARS fournit quelques appels système minimalistes (voir table 2.2). On charge dans `$v0` le code de l'appel et les arguments dans les registres `$a0`. . .`$a3` (ou `$f12` pour les valeurs en virgule flottante). Le résultat se trouve dans `$v0` (ou `$f0`).

Par exemple, pour imprimer « `deux plus trois = 5` », on peut écrire :

```
        .data
str:    .asciiz "deux plus trois = "
        .text
        li $v0, 4           # system call code for print_str
        la $a0, str         # address of string to print
        syscall             # print the string

        li $v0, 1           # system call code for print_int
        li $a0, 5           # integer to print
        syscall             # print it
```

2.4.2 Assembleur avancé

Pas traité

Nous n'avons pas (encore) parlé, notamment :

- de la pile (et registres `$fp` et `$sp`) et des appels de fonctions (la semaine prochaine)
- Allocation de mémoire dans le tas
- de la gestion des exceptions (overflow, division par zéro...) (coprocesseur 0)
- des calculs sur les flottants (coprocesseur 1)

2.4.3 Exemples

Factorielle (itératif)

```
        .data
str1:   .asciiz "Entrez un entier :"
str2:   .asciiz "Sa factorielle est "

        .text

main:   li $v0, 4          # system call code for___ print_str
        la $a0, str1       # address of__ string to__ print
        syscall            # print the string

        li $v0, 5          # system call code for___ read_int
        syscall            # read int, result in __ $v0

        li  $3, 1          # initialisation du resultat
loop:   blt $v0 1 sortie # on utilise $v0 comme variable de boucle,
                           # si <=0, on sort
        mul $3 $v0 $3
        sub $v0 $v0 1
        b loop             # sinon, $3=$3*$v0, $v0=$v0-1

sortie: li $v0, 4         # system call code for ___ print_str
        la $a0, str2       # address of__ string to__ print
        syscall            # print the string

        li $v0 1           # system call code for___ print_int
        move $a0 $3        # integer to print
        syscall            # print the integer

        li $v0 10          # on sort proprement du programme
        syscall            #
```