Formulaire d'assembleur MIPS

V. 1.4, 2022-11-21
X31I050, Université de Nantes

Registres

Nom	Numéro	Utilisation
$zero	0	Constante 0
$at	1	Réservé
$v0	2	Résultat de fonction
$v1	3	Résultat de fonction
$a0	4	Paramètre 1
$a1	5	Paramètre 2
$a2	6	Paramètre 3
$a3	7	Paramètre 4
$t0	8	Temporaire
$t1	9	Temporaire
$t2	10	Temporaire
$t3	11	Temporaire
$t4	12	Temporaire
$t5	13	Temporaire
$t6	14	Temporaire
$t7	15	Temporaire
$s0	16	Temporaire (à sauver)
$s1	17	Temporaire (à sauver)
$s2	18	Temporaire (à sauver)
$s3	19	Temporaire (à sauver)
$s4	20	Temporaire (à sauver)
$s5	21	Temporaire (à sauver)
$s6	22	Temporaire (à sauver)
$s7	23	Temporaire (à sauver)
$t8	24	Temporaire
$t9	25	Temporaire
$k0	26	Réservé
$k1	27	Réservé
$gp	28	Pointeur global
$sp	29	Pointeur de pile
$fp/$s8	30	Pointeur de cadre
$ra	31	Adresse de retour

Branchements / Sauts

Instruction	Description
b étiq	Branchement inconditionnel
beq rs, imm/rt, étiq	*Branch if EQual*
beqz rs, étiq	*Branch if EQual to Zero*
bge rs, imm/rt, étiq	*Branch if Greater or Equal*
bgeu rs, imm/rt, étiq	*Branch if Greater or Equal Unsigned*
bgez rs, étiq	*Branch if Greater or Equal to Zero*
bgt rs, imm/rt, étiq	*Branch if Greater Than*
bgtu rs, imm/rt, étiq	*Branch if Greater Than Unsigned*
bgtz rs, étiq	*Branch if Greater Than Zero*
ble rs, imm/rt, étiq	*Branch if Lower or Equal*
bleu rs, imm/rt, étiq	*Branch if Lower or Equal Unsigned*
blez rs, étiq	*Branch if Lower or Equal to Zero*
blt rs, imm/rt, étiq	*Branch if Lower Than*
bltu rs, imm/rt, étiq	*Branch if Lower Than Unsigned*
bltz rs, étiq	*Branch if Lower Than Zero*
bne rs, imm/rt, étiq	*Branch if Not Equal*
bnez rs, étiq	*Branch if Not Equal to Zero*
j étiq	*Jump*
jal étiq	*Jump And Link* (modification de $ra)
jr rs	*Jump from Register*
jalr rs	*Jump And Link from Register*

Stockage

Instruction	Description
sb rs, étiq/imm(rt)	MEM[étiq]:8 = rs / MEM[rt + imm]:8 = rs
sh rs, étiq/imm(rt)	MEM[étiq]:16 = rs / MEM[rt + imm]:16 = rs
sw rs, étiq/imm(rt)	MEM[étiq]:32 = rs / MEM[rt + imm]:32 = rs
move rd, rs	rd = rs

Chargement

Instruction	Description
la rd, étiq	rd = @étiq
lb rd, étiq/imm(rs)	rd = MEM[étiq]:8 / rd = MEM[rs + imm]:8
lh rd, étiq/imm(rs)	rd = MEM[étiq]:16 / rd = MEM[rs + imm]:16
li rd, imm	rd = SignExt(imm)
lui rd, imm	rd = (imm << 16) & 0xffff0000
lw rd, étiq/imm(rs)	rd = MEM[étiq]:32 / rd = MEM[rs + imm]:32

Directives d'assemblage

Directive	Description
.ascii *chaîne*	Stocke *chaîne* en mémoire
.asciiz *chaîne*	Stocke *chaîne* suivie de '\0'
.byte *b1, ..., bn*	Stocke *n* octets consécutivement
.data	Stockage dans le segment de données
.globl *étiq*	*étiq* est un symbole global
.half *h1, ..., hn*	stocke *n* demi-mots (16 bits)
.space *n*	Réserve la place pour *n* octets
.text	Stockage dans le segment de code
.word *w1, ..., wn*	Stocke *n* mots (32 bits)

Arithmétique entière

Instruction	Description
abs rd, rs	Valeur absolue (rd = |rs|)
add rd, rs, imm/rt	Addition signée (rd = rs + [imm/rt])
addu rd, rs, imm/rt	Addition non signée (rd = rs + rt)
div rd, rs, rt	Division signée (rd = rs / rt)
divu rd, rs, rt	Division non signée (rd = rs /rt)
mul rd, rs, rt	Multiplication (rd = rs * rt)
mulo rd, rs, rt	Multiplication avec erreur sur dépassement (rd = rs * rt)
mulou rd, rs, rt	Multiplication non signée avec erreur sur dépassement (rd = rs * rt)
neg rd, rs	Négation (rd = -rs)
rem rd, rs, rt	Reste de la division signée (rd = rs % rt)
remu rd, rs, rt	Reste de la division non signée (rd = rs % rt)
sub rd, rs, imm/rt	Soustraction signée (rd = rs - [imm/rt])
subu rd, rs, imm/rt	Soustraction non signée (rd = rs - rt)

Instructions logiques

Instruction	Description
and rd, rs, imm/rt	ET bit à bit (rd = rs & [imm/rt])
not rd, rs	NON bit à bit (rd = ~rs)
or rd, rs, imm/rt	OU bit à bit (rd = rs | [imm/rt])
sll rd, rs, imm	*Shift Left Logical* (rd = rs << imm)
sllv rd, rs, rt	*Shift Left Logical Variable* (rd = rs << rt)
sra rd, rs, imm	*Shift Right Arithmetic* (rd = rs >> imm)
srav rd, rs, rt	*Shift Right Arithmetic Variable* (rd = rs >> rt)
srl rd, rs, imm	*Shift Right Logical* (rd = rs >> imm)
srlv rd, rs, rt	*Shift Right Logical Variable* (rd = rs >> rt)
xor rd, rs, imm/rt	*OU Exclusif* (rd = rs ^ [imm/rt])

Appels système (`syscall`)

Service	*Appel ($v0)*	*Entrée*	*Sortie*
print_int	1	$a0: entier à afficher	
print_string	4	$a0: adresse chaîne	
read_int	5		$v0
read_string	8	$a0: adresse de stockage $a1: taille maximum	
malloc	9	$a0: taille à allouer	$v0
exit	10		
print_char	11	$a0: caractère (octet faible)	
read_char	12		$v0
print_hex	34	$a0: entier à afficher	
print_uint	36	$a0: entier à afficher	
rand_seed	40	$a0: valeur de l'initialiseur	
random	41	$a0: n'importe quel entier	$a0

Seuls les registres de sortie sont affectés par un appel système

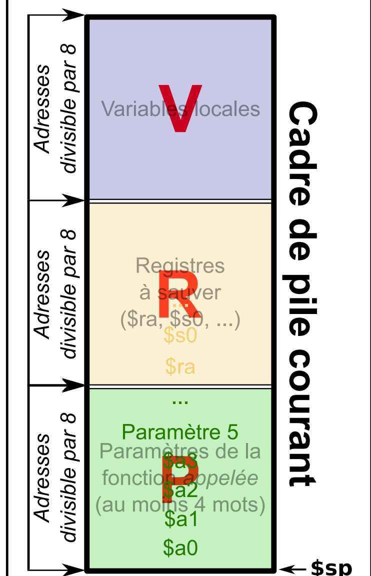
Hello World!

```
# Déclaration des données
            .data
hello_str: .asciiz "Hello World\n!"

            .text
            .globl __start

# Programme principal
__start:
        jal main
        # exit
        li $v0, 10
        syscall
main:
       la $a0, hello_str
       li $v0, 4
       syscall
       li $v0, 0
       jr $ra
```

Pile d'appels



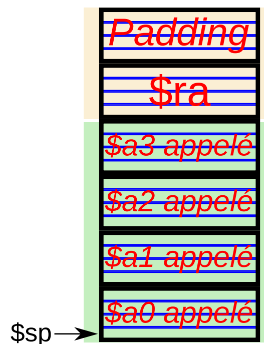
(c) 2009-2022 Frédéric Goualard, Université de Nantes
Attribution Non-Commercial Share Alike

Note: ce formulaire ne présente qu'une vue partielle des possibilités du langage d'assemblage d'un processeur MIPS tel qu'implémenté sur l'émulateur MARS.

Conventions d'appels de fonctions

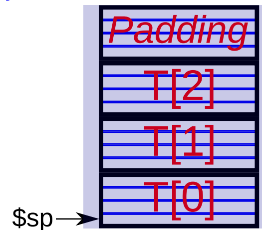
```
// Pas d'appel de fonction
// Pas de variable locale non
// scalaire
// Pas de registre à sauver
int sans_cadre(int a, int b)
{
  return a + b;
}
sans_cadre:
    add $v0, $a0, $a1
    jr $ra
```

```
// Appel de fonction
// Pas de variable locale non
// scalaire
// Registre $ra à sauver
int cadre_PR(int a, int b)
{
  return f(b,a);
}
cadre_PR:
    subu $sp, $sp, 24
    sw $ra, 16($sp)
    move $t0, $a0
    move $a0, $a1
    move $a1, $t0
    jal f
    lw $ra, 16($sp)
    addu $sp, $sp, 24
    jr $ra
```

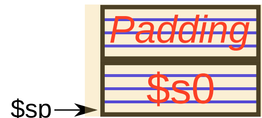



```
// Appel de fonction
// Variable locale non scalaire
// Registres  $ra et $s0 à sauver
int cadre_PRV(int a, int b)
{
  int T[3];
  T[1] = a;
  return f(a,b);
}
cadre_PRV:
    subu $sp, $sp, 40
    sw $ra, 16($sp)
    sw $s0, 20($sp)
    sw $a0, 28($sp)
    jal f
    lw $s0, 20($sp)
    lw $ra, 16($sp)
    addu $sp, $sp, 40
    jr $ra
```

```
// Pas d'appel de fonction
// Variable locale non scalaire
// Pas de registre à sauver
int cadre_V(int a, int b)
{
  int T[3];
  T[2] = a + b;
  return T[2];
}
cadre_V:
    subu $sp, $sp, 16
    add $t0, $a0, $a1
    sw $t0, 8($sp)
    lw $v0, 8($sp)
    addu $sp, $sp, 16
    jr $ra
```




```
// Pas d'appel de fonction
// Pas de variable locale non
// scalaire
// Registre $s0 à sauver
int cadre_R(int a, int b)
{
  return a + b;
}
cadre_R;
    subu $sp, $sp, 8
    sw $s0, 0($sp)
    add $s0, $a0, $a1
    move $v0, $s0
    lw $s0, 0($sp)
    addu $sp, $sp, 8
    jr $ra
```



Traduction des structures de contrôle

```
if ($t0 == 3) {
  // bloc then
} else {
  // bloc else
}
```

```
if: bne $t0, 3, else
then:
        # bloc then
        b endif
else:
        # bloc else
endif:
```

```
if ($t0 == 3 || $t1 == 4) {
  // bloc then
} else {
  // bloc else
}
```

```
if: beq $t0, 3, then
    beq $t1, 4, then
    b else
then:
    # bloc then
    b endif
else:
    # bloc else
endif:
```

```
if ($t0 == 5 && $t1 == 6) {
  // bloc then
} else {
  // bloc else
}
```

```
if: bne $t0, 5, else
    bne $t1, 6, else
then:
        # bloc then
        b endif
else:
        # bloc else
endif:
```

```
while ($t0 > 2) {
  // bloc while
}
```

```
while: ble $t0, 2, endwhile
do:     # bloc while
        b while
endwhile:
```

```
for ($t0=1; $t0 < 5; ++$t0) {
    // bloc for
}
```

```
     li $t0, 1
for: bge $t0, 5, endfor
do:  # bloc for
     add $t0, $t0, 1
     b for
endfor:
```