Programmation en langage d'assemblage *MIPS32*

Frédéric Goualard

2021-10-17, v. 0.3

1 Introduction

Les premiers processeurs à architecture MIPS ont été développés au mitan des années 80 en réaction à la complexité — toute relative pour un observateur actuel — des processeurs de l'époque, en particulier ceux utilisant l'architecture *x86* introduite par Intel.

L'objectif premier ayant guidé tout le *design* des processeurs MIPS est la simplicité. C'est pourquoi cette famille de processeurs présente des intérêts pédagogiques certains : ils sont suffisamment simples pour être étudiés dans un cours introductif de Licence Informatique, mais suffisamment avancés pour présenter les idées fondamentales sur les processeurs en général.

Dans cette note, nous nous focaliserons uniquement sur l'architecture *MIPS32* et nous ne présenterons que les éléments de ce type de processeur qui sont pertinents pour la programmation en langage d'assemblage. Les conventions utilisées sont celles décrites dans le document System V Application Binary Interface[1] avec quelques simplification pour leur utilisation dans le cadre du cours « *Architecture des ordinateurs (X31I050)* » de la Faculté des Sciences et des Techniques de l'Université de Nantes. Nous utiliserons l'émulateur mars pour assembler et exécuter le code MIPS sur des machines à architecture *x86*.

2 Le processeur *MIPS32*

Le processeur *MIPS32* est composé de trois circuits, visibles dans la figure 1 :

- Le CPU[2] proprement dit. Il contient :
  - Le *séquenceur*, chargé de récupérer en mémoire l'instruction courante, de la décoder et de faire exécuter les calculs,
  - L'*Unité Arithmétique et Logique* (UAL, ou *ALU* en anglais), exécutant les opérations `add`, `sub`, `or`, ...
  - Les trente-deux registres de 32 bits pour manipuler des valeurs entières (voir table 1) ;
- Le coprocessor 0 (*Core 0*), qui est l'unité responsable de la gestion de la mémoire, des caractéristiques modifiables du processeur *MIPS32* (par exemple, son *endianness*), du cache et des exceptions et interruptions. On n'en parlera pas plus dans cette note ;
- Le coprocesseur 1 (*Core 1*) correspondant à l'unité de calcul flottant (ou *Floating-Point Unit — FPU*) au standard IEEE 754. On n'en parlera pas non plus dans cette note.

[1] *System V Application Binary Interface : MIPS® RISC Processor Supplement.* Rapp. tech. Troisième édition

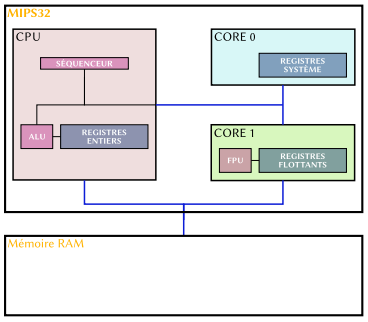


Figure 1 : Architecture du processeur *MIPS32* connecté à la mémoire centrale.

[2] **CPU** · *Central Processing Unit.*

[3] **Machine à registres** · « *Load-store architecture*, » dans la littérature anglaise.

Le processeur *MIPS32* a une architecture de machine à registres[3] ; le jeu d'instructions reconnues par le processeur se divise essentiellement en deux :

— Les instructions permettant de charger une valeur de la mémoire vers un registre, ou de stocker en mémoire une valeur présente dans un registre ;
— Les instructions exécutant des opérations sur les registres.

En particulier, contrairement à d'autres types de processeurs, comme les processeurs à architecture *x86* par exemple, aucune opération ne peut se faire directement sur une valeur en mémoire ; il faut toujours passer par un registre.

Le processeur *MIPS32* a une architecture 32 bits très régulière. Il possède trente-deux registres capables de stocker des valeurs entières sur 32 bits. Chacun des registres peut être identifié par son numéro entre 0 et 31 ou son nom symbolique : l'instruction

```
add $t0, $t1, $t2
```

pour sommer les contenus des registres `$t1` et `$2` et stocker le résultat dans `$t0` est, par exemple, équivalente à :

```
add $8, $9, $10
```

[4] **Espace d'adressage** · L'espace d'adressage d'un processeur correspond à l'ensemble des adresses en mémoire accessibles, que cette mémoire existe physiquement ou pas. Un processeur peut, par exemple avoir un espace d'adressage de 32 bits — ce qui lui permet d'accéder théoriquement à 2^{32} emplacements en mémoire, soit environ 4 GiB —en étant intégré dans un système ne possédant que 2 GiB de mémoire physique. Le système d'exploitation, aidé de certains éléments matériels, est chargé d'assurer la correspondance entre l'espace physique et la mémoire virtuelle.

L'espace d'adressage[4] du *MIPS32* est aussi de 32 bits. La figure 2 présente le découpage de l'*espace d'adressage virtuel* pour un programme chargé en mémoire avec MARS.

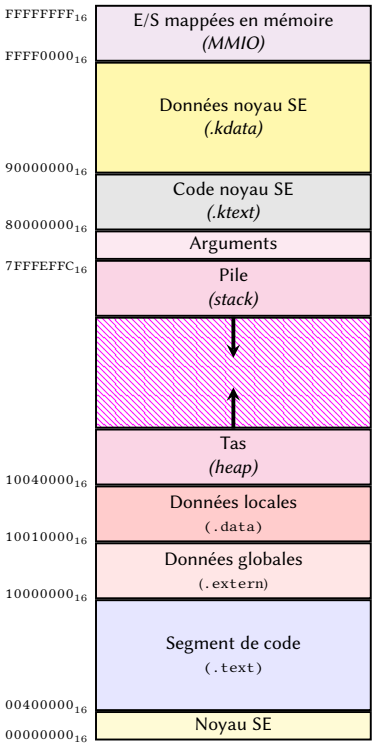


FIGURE 2 : Espace d'adressage du *MIPS32* sur MARS

TABLE 1 : Les trente-deux registres de 32 bits du *MIPS32*

Nom	Numéro	Utilisation
`$zero`	0	Registre constant valant 0
`$at`	1	Réservé (synthèse de pseudo-instructions)
`$v0`–`$v1`	2–3	Résultat d'une fonction
`$a0`–`$a3`	4–7	Paramètre d'une fonction
`$t0`–`$t7`, `$t8`, `$t9`	8–15, 24, 25	Registre temporaire
`$s0`–`$s7`, `$s8`	16–23, 30	Registre sauvegardé
`$k0`, `$k1`	26, 27	Réservé
`$gp`	28	Pointeur pour accès en mémoire globale
`$sp`	29	Pointeur sur le haut de la pile
`$ra`	31	Adresse de retour d'une fonction

3 « Hello World ! »

L'exemple canonique ci-dessous, stocké dans un fichier texte d'extension « .asm » affiche le message « *Hello World !* » sur la sortie standard. Il permet d'illustrer certaines des caractéristiques de la programmation en langage d'assemblage dont nous reparlerons en détails dans les sections suivantes.

```
# helloworld.asm
        .data
msgstr: .asciiz "Hello World!"

        .text
        .globl __start
__start:
        jal main
        li $v0, 10 # exit()
        syscall
```

```
main:
        # Appel du service d'affichage d'une chaîne
        la $a0, msgstr
        li $v0, 4
        syscall
        # Retour à l'appelant
        jr $ra
```

Le code source du programme est décomposé en deux parties délimitées par les directives d'assemblages `.data` et `.text` permettant d'identifier ce qui doit être stocké à l'exécution dans le segment de données et ce qui doit l'être dans le segment de code (figures 2 et 3). La directive `.globl` identifie une étiquette exportée à l'extérieur de la portée du fichier courant qui peut être référencée par un autre fichier source. Elle indique ici le nom du programme principal `__start`. Dans le cadre du module « Architecture des ordinateurs, » *le code du programme principal sera toujours le même* : il appelle la fonction `main()` puis demande au système d'exploitation de reprendre la main et de libérer les ressources acquises lors de l'exécution via l'appel système 10 (voir section 5.5). Lors de la programmation dans un langage « de haut niveau » comme le C ou le C++, le code de `__start` est ajouté par le compilateur et contient, outre un appel à la fonction `main()` fournie par l'utilisateur, des instructions d'initialisation des différentes bibliothèques utilisées ainsi que du code pour gérer le retour de la fonction `main()`.

Tous les noms suivis de « : » sont des *étiquettes* correspondant à des noms symboliques pour des adresses en mémoire. Une étiquette dans un programme en langage d'assemblage ressemble à une macro dans un programme C ou C++ : elle n'a pas d'existence propre, n'occupe aucune place en mémoire et est remplacée par un mécanisme de « chercher/remplacer » lors de l'assemblage. L'instruction :

```
la $a0, msgstr
```

charge dans le registre `$a0` l'adresse correspondant à la chaîne de caractères « Hello World ! ». Le fait d'écrire :

```
msgstr:   .asciiz  "Hello World!"
```

indique à l'assembleur[5] que l'on souhaite que l'étiquette `msgstr` corresponde à l'adresse du premier élément stocké en mémoire qui suit, à savoir ici la lettre `'H'` de la chaîne de caractères "Hello World !". On notera que cette définition a lieu après la directive `.data`, ce qui garantit que le stockage aura bien lieu dans le segment de données lors de la mise en mémoire du programme à l'exécution.

Une étiquette peut apparaître n'importe où dans le segment de données ou le segment de code et fait toujours référence à l'adresse de la première case en mémoire qui la suit, même si elle n'est pas utilisée. Exemple :

```
      .data
msg:  .asciiz "Un long message"
fin:
      .text
      la $a0, fin
```

Le code ci-dessus charge dans le registre `$a0` l'adresse de la première case mémoire qui suit la chaîne "Un long message". Cela peut être un moyen simple de calculer la taille de la chaîne de caractères en calculant la différence entre les adresses identifiées par fin et msg.

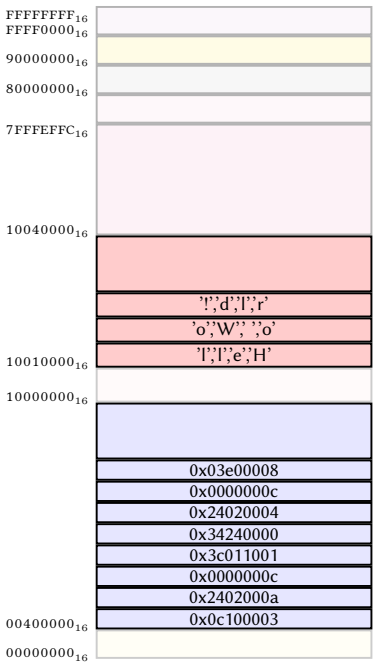


FIGURE 3 : Représentation en mémoire du programme helloworld pour une machine *MIPS32 Little Endian*. Dans ce dessin, chaque case occupée correspond à un mot (4 octets).

[5] **Assembleur** · L'*assembleur* est le programme qui traduit un code écrit en langage d'assemblage vers du code machine (c'est l'assemblage). Un abus de langage fréquent, qui ne sera pas fait dans cette note, confond « langage d'assemblage » et « assembleur ».

4 Les instructions du processeur *MIPS32*

Le processeur *MIPS32* ne supporte réellement qu'un petit jeu d'instructions, les *instructions natives*, qui ont une implémentation matérielle directe. Afin de faciliter le codage en langage d'assemblage, l'assembleur offre des instructions supplémentaires, les

pseudo-instructions, qui n'ont pas de traduction directe dans le processeur ; elles sont remplacées lors de l'assemblage par plusieurs instructions natives. De même, le processeur *MIPS32* ne reconnaît qu'une méthode pour accéder à la mémoire, mais un assembleur en propose souvent d'autres, qui sont implémentées *via* des pseudo-instructions. Parfois même, certaines pseudo-instructions portent le même nom qu'une instruction ; c'est par exemple le cas de « `add` ». Dans la suite de cette note, nous utiliserons indifféremment les pseudo-instructions et les instructions natives.

La figure 4 décrit la représentation en mémoire des instructions natives du *MIPS32*. La connaissance de ces formats permet d'inférer certaines limites des instructions. L'instruction `addi`, par exemple, ajoute le contenu d'un registre et d'une valeur immédiate[6]. Comme elle est de format *I* (voir table 3, page 6), la valeur immédiate en troisième argument ne peut faire plus de 16 bits. Avec `addi`, il s'agit d'un entier codé en complément à 2 sur 16 bits, donc devant appartenir à l'intervalle $[-2^{15}, 2^{25} - 1] = [-32768, 32767]$. En cas de dépassement (ex. : `addi $t0, $t1, 32768`), l'assembleur de MARS remplace automatiquement l'instruction `addi` par une instruction `add` sur trois registres après avoir chargé la valeur immédiate trop grande dans le registre auxiliaire `$at` :

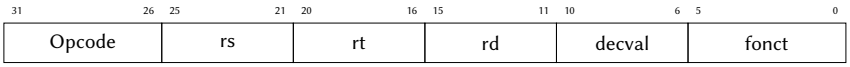
[6] **Valeur immédiate** · Une *valeur immédiate* est une constante entière ou un caractère apparaissant dans le code en langage d'assemblage. Exemple : dans l'instruction « `addi $t0, $t1, 5,` » la valeur 5 est une valeur immédiate.

Pseudo-instructions → instructions natives (MARS)

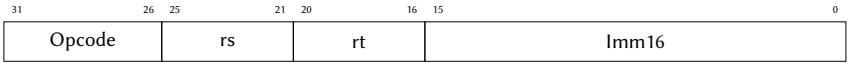
```
addi $t0, $t1, 32768
```

```
lui $at, 0x00000000
ori $at, $at, 0x00008000
add $t0, $t1, $at
```

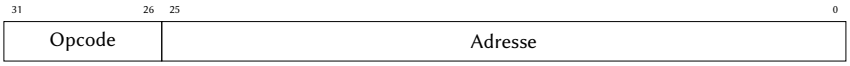
L'instruction « `lui` » charge dans le registre `$at` les 16 bits de poids forts ; l'instruction « `ori` » y ajoute les 16 bits de poids faibles en faisant un « OU » bit-à-bit.



(a) Format R.



(b) Format I.



(c) Format J.

FIGURE 4 : Formats des instructions natives *MIPS32*.

4.1 Les directives d'assemblage

Les directives d'assemblage ne sont pas à proprement parler des instructions du *MIPS32* car elles s'adressent à l'assembleur et non au processeur. La plupart d'entre elles font référence au stockage de données (voir section 5.3) et en particulier au nombre d'octets ou au format utilisé pour stocker une valeur en mémoire. La table 2 présente les directives les plus communes reconnues par MARS.

On notera que les directives `.byte`, `.half` et `.word` permettent de stocker consécutivement en mémoire une suite d'entiers sur, respectivement, 1, 2, ou 4 octets. Ces directives spécifient la taille de la représentation, pas le type. Ainsi, on peut écrire indifféremment :

```
.byte -191
.byte 'A'
.byte 65
```

TABLE 2 : Quelques directives d'assemblage reconnues par MARS.

Directive	Description
`.ascii str`	Stocke la chaîne `str` sans terminateur `'\0'` final
`.asciiz str`	Stocke la chaîne `str` avec terminateur `'\0'` final
`.byte b1, ..., bn`	Stocke chaque valeur `b1`, ..., `bn` sur un octet consécutivement en mémoire
`.data`	Indique que les éléments qui suivent doivent être stockés dans le segment de données
`.double d1, ..., dn`	Stocke les valeurs `d1`, ..., `dn` au format IEEE 754 double précision consécutivement en mémoire
`.float f1, ..., fn`	Stocke les valeurs `f1`, ..., `fn` au format IEEE 754 simple précision consécutivement en mémoire
`.globl sym`	Indique que `sym` est une étiquette de portée extérieure au fichier courant
`.half h1, ..., hn`	Stocke chaque valeur `h1`, ..., `hn` sur 2 octets consécutivement en mémoire
`.space n`	Réserve n octets dans le segment de données
`.text`	Indique que les éléments qui suivent doivent être stockés dans le segment de code
`.word w1, ..., wn`	Stocke chaque valeur `w1`, ..., `wn` sur 4 octets consécutivement en mémoire

Attention : *le tableau présente seulement une sélection des directives reconnues par* MARS.

pour un même résultat : le stockage en mémoire sur un octet de la chaîne binaire 1000001_2. De la même manière, on pourrait écrire :

```
.word -191
.word 'A'
.word 65
```

pour stocker la chaîne $00000000000000000000000001000001_2$ en mémoire sur quatre octets.

Le processeur *MIPS32* contraint les données à être positionnées en mémoire (« *alignées* ») sur une adresse multiple de leur taille en octets : un entier sur 32 bits doit être stocké à une adresse multiple de 4, un flottant double précision à une adresse multiple de 8. Si nécessaire, des octets de padding[7] sont ajoutés.

La figure 5 présente l'organisation en mémoire consécutive au chargement du programme ci-dessous à partir de l'adresse 10010000_{16}.

```
        .data
txt:    .asciiz "Hello"
w:      .word 10
b:      .byte 13
f:      .float 3.14
d:      .double 1.5
```

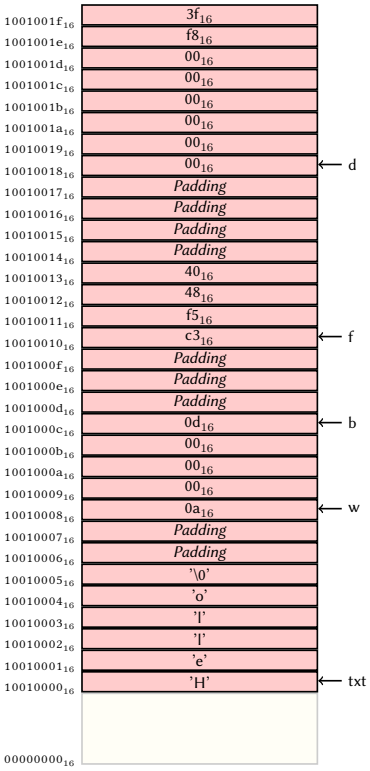


FIGURE 5 : Stockage de données en mémoire (*little endian*). Chaque case correspond à 1 octet.

[7] **Padding** · Le *padding* correspond à l'ajout de données fictives en mémoire de façon à garantir que la prochaine donnée stockée consécutivement commence à une adresse avec une propriété de divisibilité précise — exemple : que l'adresse soit divisible par 8.

4.2 Les instructions arithmétiques et logiques

Les opérations arithmétiques peuvent se faire entre deux registres ou, pour certaines instructions, entre un registre et une valeur immédiate. Le résultat est toujours stocké dans un registre, qui peut être le même que l'un des deux opérandes :

```
add $t0, $t0, $t1 # $t0 <- $t0 + $t1
add $a0, $t0, 5   # $a0 <- $t0 + 5
```

La première occurrence de l'instruction `add` fait référence à une instruction native (l'addition entre deux registres) ; la deuxième occurrence est une pseudo-instruction remplacée par l'assembleur en un appel à l'instruction native `addi` pour faire la somme d'un registre et d'une valeur immédiate.

La table 3 présente les différentes instructions arithmétiques sur les entiers ; la table 4 présente les instructions logiques (bit-à-bit).

TABLE 3 : Instructions arithmétiques. Le format « P » correspond aux pseudo-instructions. Les notations Rd, Rs, Rt font référence aux registres de la figure 4. Une valeur immédiate est noté Imm.

Instruction	Format	Description
abs Rdest, Rsrc	P	Stocke la valeur absolue du registre Rsrc dans Rdest
add Rdest, Rsrc, Imm	P	Rd ← Rsrc + SignExt_{32}(Imm)
add Rd, Rs, Rt	R	Rd ← Rs + Rt (signé)
addi Rt, Rs, Imm	I	Rt ← Rs + SignExt_{32}(Imm)
addiu Rt, Rs, Imm	I	Rt ← Rs + Ext_{32}(Imm)
addu Rd, Rs, Rt	R	Rd ← Rs + Rt (non-signé)
div Rd, Rs, Rt	P	Rd ← Rs ÷ Rt
div Rd, Rs, Imm	P	Rd ← Rs ÷ SignExt_{32}(Imm)
divu Rd, Rs, Rt	P	Rd ← Rs ÷ Rt
divu Rd, Rs, Imm	P	Rd ← Rs ÷ Ext_{32}(Imm)
mul Rd, Rs, Rt	P	Rd ← Rs × Rt
mul Rd, Rs, Imm	P	Rd ← Rs × SignExt_{32}(Imm)
neg Rd, Rs	P	Rd ← −Rs
rem Rd, Rs, Rt	P	Rd ← Rs % Rt
remu Rd, Rs, Rt	P	Rd ← Rs % Rt
sub Rd, Rs, Rt	R	Rd ← Rs − Rt
sub Rd, Rs, Imm	P	Rd ← Rs − SignExt_{32}(Imm)
subu Rd, Rs, Rt	R	Rd ← Rs − Rt
subu Rd, Rs, Imm	P	Rd ← Rs − Ext_{32}(Imm)

Note : la fonction $\text{SignExt}_{32}()$ correspond à l'extension à 32 bits d'une valeur représentée en complément à 2 ; la fonction $\text{Ext}_{32}()$ étend à 32 bits une valeur non signée.

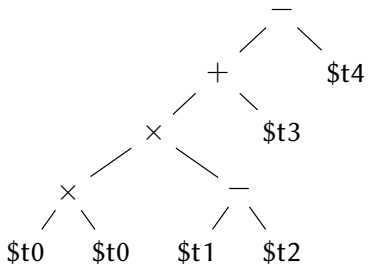


FIGURE 6 : Arbre d'expression pour $\$t0^2 * (\$t1 - \$t2) + \$t3 - \$t4$.

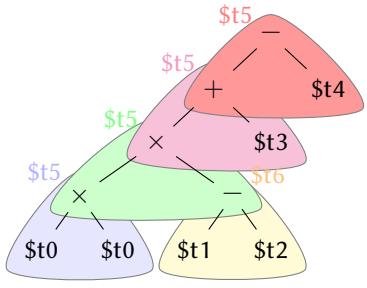


FIGURE 7 : Arbre d'expression couvert par des instructions du *MIPS32* pour $\$t0^2 * (\$t1 - \$t2) + \$t3 - \$t4$.

Le calcul d'expressions complexes se fait en les décomposant en leurs opérations atomiques et en utilisant les registres pour sauver les calculs intermédiaires.

Considérons, par exemple, l'expression $\$t0^2 * (\$t1 - \$t2) + \$t3 - \$t4$. Elle correspond à l'arbre de la figure 6. Connaissant les instructions à trois opérandes reconnues par le processeur *MIPS32*, on peut faire une *couverture* de cet arbre d'expression de façon à faire correspondre chaque triplet de nœuds à une instruction, en ajoutant sur le nœud d'opération le nom du registre devant stocker le résultat intermédiaire. Si l'on choisit les registres `$t5` et `$t6` pour les résultats intermédiaires, on obtient l'arbre couvert de la figure 7.

On en déduit le code *MIPS32* (en remontant des feuilles à la racine de l'arbre) :

```
mul $t5, $t0, $t0
```

```
sub $t6, $t1, $t2
mul $t5, $t5, $t6
add $t5, $t5, $t3
sub $t5, $t5, $t4
```

TABLE 4 : Instructions logiques. Le format « P » correspond aux pseudo-instructions. Les notations Rd, Rs, Rt font référence aux registres de la figure 4. Une valeur immédiate est noté Imm

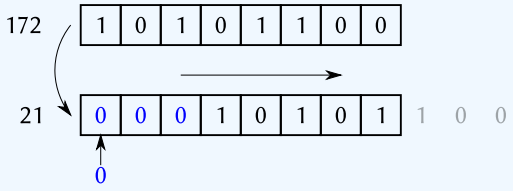
Instruction	Format	Action
and Rd, Rs, Rt	R	ET bit-à-bit : Rd ← Rs ∧ Rt
and Rd, Rs, Imm	P	ET bit-à-bit : Rd ← Rs ∧ Ext_{32}(Imm)
andi Rd, Rs, Imm	I	ET bit-à-bit : Rd ← Rs ∧ Ext_{32}(Imm)
nor Rd, Rs, Rt	R	NON-OU bit-à-bit : Rd ← $\overline{\text{Rs} \vee \text{Rt}}$
not Rd, Rs	R	NON bit-à-bit : Rd ← $\overline{\text{Rs}}$
or Rd, Rs, Rt	R	OU bit-à-bit : Rd ← Rs ∨ Rt
or Rd, Rs, Imm	P	OU bit-à-bit : Rd ← Rs ∨ Ext_{32}(Imm)
ori Rd, Rs, Imm	I	OU bit-à-bit : Rd ← Rs ∨ Ext_{32}(Imm)
sll Rd, Rs, Imm	R	décalage à gauche : Rd ← Rs << Imm
sllv Rd, Rs, Rt	R	décalage à gauche : Rd ← Rs << Rt
sra Rd, Rs, Imm	R	décalage à droite arithmétique : Rd ← Rs >> Imm
srav Rd, Rs, Rt	R	décalage à droite arithmétique : Rd ← Rs >> Rt
srl Rd, Rs, Imm	R	décalage à droite : Rd ← Rs >> Imm
srlv Rd, Rs, Rt	R	décalage à droite : Rd ← Rs >> Rt
rol Rd, Rs, Rt	P	rotation à gauche de Rt bits de Rs dans Rd
ror Rd, Rs, Rt	P	rotation à droite de Rt bits de Rs dans Rd
xor Rd, Rs, Rt	R	OU EXCLUSIF bit-à-bit : Rd ← Rs ⊕ Rt
xor Rd, Rs, Imm	P	OU EXCLUSIF bit-à-bit : Rd ← Rs ⊕ Ext_{32}(Imm)
xori Rd, Rs, Imm	I	OU EXCLUSIF bit-à-bit : Rd ← Rs ⊕ Ext_{32}(Imm)

Note : la fonction $\text{Ext}_{32}()$ étend à 32 bits une valeur non signée.

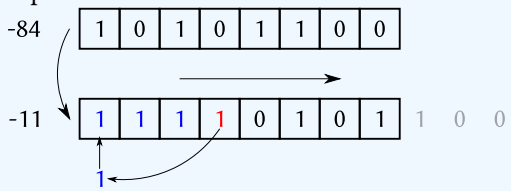
Décalage à droite arithmétique *vs.* logique et rotations

Le processeur *MIPS32* propose deux types d'instructions pour effectuer un décalage à droite des bits d'un registre :

- Le décalage logique avec `srl`/`srlv` correspond à un déplacement des bits du registre source vers la droite en injectant des « 0 » dans les emplacements libérés :



— Le décalage arithmétique avec `sra`/`srav` correspond à un déplacement des bits du registre source vers la droite en injectant le bit de poids fort initial dans les emplacements libérés, de façon à tenir compte des nombres négatifs représentés en complément à 2 :



Le logiciel MARS accepte l'utilisation de certaines instructions logiques avec seulement deux paramètres (exemple : `and $t0, 5` est reconnu comme `and $t0, $t0, 5`). Cet usage est déconseillé et on ne l'emploiera jamais dans les programmes montrés en exemples dans la suite de cette note.

4.3 Accès à la mémoire et chargement des registres

Comme on l'a vu au début de cette note, les registres sont incontournables pour tous les calculs. On doit donc disposer d'instructions pour charger une valeur dedans avant un calcul (« *load* ») ou stocker un résultat en mémoire (« *store* »).

Le processeur *MIPS32* ne connaît qu'une seule façon de faire référence à l'adresse d'une case en mémoire : la construction $\text{Imm}_{32}(\text{Reg})$ référence la case dont l'adresse est connue en ajoutant le contenu du registre Reg et la valeur immédiate sur 32 bits Imm_{32}. Par exemple, l'instruction :

```
lw $t0, 16($t1)
```

stockera dans `$t0` le mot de 4 octets commençant à l'adresse 10010010_{16} si `$t1` contient la valeur 10010000_{16}.

Avec MARS, l'accès à une case en mémoire peut se faire de sept façons différentes, [1] chacune de ces méthodes étant traduite par l'assembleur sous la forme effectivement supportée par le *MIPS32* (table 5).

TABLE 5 : Différentes formes d'adressage reconnues par MARS.

Adressage	**Exemple**	**Action**†
label	`la $t0, msg`	$t0 ← 0x10010000
Imm_{32}	`sw $t0, 0x10010000`	MEM[0x10010000−3] ← $t0
label + Imm_{32}	`lb $t0, msg+1`	$t0 ← MEM[0x10010001]
(Reg)	`lw $t0, ($t1)`	$t0 ← MEM[0x10010004−7]
label(Reg)	`lb $t0, msg($t2)`	$t0 ← MEM[0x10010002]
Imm_{32}(Reg)	`lb $t0, 2($t1)`	$t0 ← MEM[0x10010002]
label + Imm_{32}(Reg)	`lb $t0, msg+2($t2)`	$t0 ← MEM[0x10010004]

†On suppose que l'étiquette msg fait référence à l'adresse 0x10010000, que $t1 contient la valeur 0x10010004 et que $t2 contient la valeur 2. La notation « MEM[0x10010000−x] ← y » indique que les octets 0x10010000 jusqu'à 0x1001000 + x sont impactés par le stockage de la valeur y.

Un processeur *MIPS32* peut être configuré pour accéder à la mémoire en *little endian* ou *big endian* indifféremment. Cependant, la manipulation de la mémoire dans MARS est toujours faite en *little endian*. [2] La table 6 présente les différentes instructions de chargement et de stockage.

1. Source : https://courses.missouristate.edu/KenVollmar/MARS/Help/MarsHelpIntro.html.
2. Source : http://courses.missouristate.edu/KenVollmar/mars/Help/MarsHelpDebugging.html

TABLE 6 : Instructions de chargement/stockage. Le format « P » correspond aux pseudo-instructions. Les notations Rd, Rs, Rt font référence aux registres de la figure 4. Une valeur immédiate est noté Imm.

Instruction	Format	Description[†]
la Rd, *Adr*	P	Charge l'adresse *Adr* dans le registre Rd
lb Rt, *Adr*	I	Charge l'octet se trouvant en mémoire à l'adresse *Adr* dans le registre Rt. Il y a extension de signe de 8 à 32 bits
lbu Rt, *Adr*	I	Charge l'octet se trouvant en mémoire à l'adresse *Adr* dans le registre Rt. Il y a extension non signée de 8 à 32 bits
lh Rt, *Adr*	I	Charge les deux octets se trouvant en mémoire aux adresses *Adr* et *Adr* + 1 dans le registre Rt. Il y a extension de signe de 16 à 32 bits
lhu Rt, *Adr*	I	Charge les deux octets se trouvant en mémoire aux adresses *Adr* et *Adr* + 1 dans le registre Rt. Il y a extension non signée de 16 à 32 bits
lw Rt, *Adr*	I	Charge les quatre octets se trouvant en mémoire aux adresses *Adr* à *Adr* + 3 dans le registre Rt
li Rd, Imm	P	Charge les 16 bits de Imm dans le demi-mot de poids faible de Rt ; le demi-mot de poids fort de Rt est forcé à 0
lui Rt, Imm	I	Charge les 16 bits de Imm dans le demi-mot de poids fort de Rt ; le demi-mot de poids faible de Rt est forcé à 0
sb Rt, *Adr*	I	Stocke l'octet de poids faible du registre Rt en mémoire dans la case mémoire d'adresse *Adr*
sh Rt, *Adr*	I	Stocke les deux octets de poids faible du registre Rt en mémoire dans les cases d'adresse *Adr* et *Adr* + 1
sw Rt, *Adr*	I	Stocke le registre Rt en mémoire dans les cases d'adresse *Adr* à *Adr* + 3
move Rd, Rs	P	Copie le contenu du registre Rs dans le registre Rd

[†] Une adresse *Adr* peut prendre l'une des sept formes présentées dans la table 5.

4.4 Les instructions de comparaison

Les instructions de comparaison permettent de comparer deux registres ou un registre et une valeur immédiate et de stocker le résultat sous forme numérique (« 1 » pour « true » et « 0 » pour « false ») dans un registre destination. Elles sont utiles pour implémenter des alternatives sans recourir à des instructions de branchement. Les principales instructions de comparaison sont décrites dans la table 7

Considérons, par exemple, la fonction C++ `isminor()` suivante avec sa traduction en langage d'assemblage MIPS (on renvoie le lecteur aux sections suivantes pour la compréhension fine de l'implémentation d'une fonction en MIPS) :

```
int isminor(unsigned int age)
{
   if (age < 18) {
      return 1;
   } else {
      return 0;
   }
}
```

```
isminor:
if:   bgeu $a0, 18, else
then:
   li $v0, 1
   b endif
else:
   move $v0, $zero
endif:
   jr $ra
```

Elle peut — et doit — être simplifiée en :

```
int isminor(unsigned int age)
{
   return age < 18;
}
```

```
isminor:
   sltiu $v0, $a0, 18
   jr $ra
```

On notera l'utilisation de l'instruction sltiu pour obtenir directement sous forme numérique le résultat de la comparaison.

4.5 Les instructions de saut

Le registre PC (*Program Counter*) pointe à chaque instant sur l'instruction courante d'un programme. C'est un registre de seulement 30 bits car, toutes les instructions étant sur 4 octets, elles doivent être alignées en mémoire sur des adresses multiples de 4. Par conséquent, les deux bits de poids faibles des adresses de toutes les instructions sont toujours nuls. On choisit donc de ne pas les stocker. Lors de l'adressage de la mémoire, on rajoute deux bits à 0 à droite du contenu du registre PC pour obtenir une adresse sur 32 bits.

En marche normale, PC est incrémenté à chaque « tic » d'horloge pour passer à l'instruction suivante. Les instructions de saut permettent de lui ajouter ou retirer un entier *ofs* représenté en complément à 2 sur 16 bits en sus de l'incrémentation. On parle alors de *saut relatif* car on définit la nouvelle valeur de PC relativement à son ancienne valeur. Les instructions de type « *branch* » (table 8) utilisent ce type de saut.

Comme le déplacement *ofs* sur 16 bits est ajouté au registre de 30 bits, tout se passe comme si l'on avait ajouté quatre fois *ofs* à l'adresse sur 32 bits obtenue à partir de PC. Par conséquent, *on saute ofs instructions en avant ou en arrière et non ofs octets.*

Comparaisons signées et non signées

La comparaison de deux registres se fait par leur soustraction et l'étude des indicateurs OF, SF et ZF levés. Pour pouvoir interpréter correctement ces indicateurs, on a besoin de savoir si les valeurs se trouvant dans les registres sont considérées par

TABLE 7 : Instructions de comparaison. Le format « P » correspond aux pseudo-instructions. Les notations Rd, Rs, Rt font référence aux registres de la figure 4. Une valeur immédiate sur x bits est noté Imm_x.

Instruction	Format	Description
seq Rd, Rs, Rt	P	Met 1 dans le registre Rd si Rs est égal à Rt et 0 sinon
sge Rd, Rs, Rt	P	Met 1 dans le registre Rd si Rs est supérieur ou égal à Rt et 0 sinon (comparaison signée)
sgeu Rd, Rs, Rt	P	Met 1 dans le registre Rd si Rs est supérieur ou égal à Rt et 0 sinon (comparaison non-signée)
sgt Rd, Rs, Rt	P	Met 1 dans le registre Rd si Rs est strictement supérieur à Rt et 0 sinon (comparaison signée)
sgtu Rd, Rs, Rt	P	Met 1 dans le registre Rd si Rs est strictement supérieur à Rt et 0 sinon (comparaison non-signée)
sle Rd, Rs, Rt	P	Met 1 dans le registre Rd si Rs est inférieur ou égal à Rt et 0 sinon (comparaison signée)
sleu Rd, Rs, Rt	P	Met 1 dans le registre Rd si Rs est inférieur ou égal à Rt et 0 sinon (comparaison non-signée)
slt Rd, Rs, Rt	R	Met 1 dans le registre Rd si Rs est strictement inférieur à Rt et 0 sinon (comparaison signée)
sltu Rd, Rs, Rt	R	Met 1 dans le registre Rd si Rs est strictement inférieur à Rt et 0 sinon (comparaison non-signée)
slti Rd, Rs, Imm_{16}	I	Met 1 dans le registre Rd si Rs est strictement inférieur à $\text{SignExt}_{32}(\text{Imm}_{16})$ et 0 sinon (comparaison signée)
sltiu Rd, Rs, Imm_{16}	I	Met 1 dans le registre Rd si Rs est strictement inférieur à $\text{SignExt}_{32}(\text{Imm}_{16})$ et 0 sinon (comparaison non signée)
sne Rd, Rs, Rt	P	Met 1 dans le registre Rd si Rs est différent de Rt et 0 sinon

Note : la fonction $\text{SignExt}_{32}()$ correspond à l'extension à 32 bits d'une valeur représentée en complément à 2.

TABLE 8 : Instructions de saut relatif. Le format « P » correspond aux pseudo-instructions. Les notations Rd, Rs, Rt font référence aux registres de la figure 4. Une valeur immédiate est noté Imm ; une étiquette est notée label.

Instruction	Format	Description
b label	P	Saut à l'instruction d'adresse label
bgez Rs, label	I	Saute à l'instruction d'adresse label si Rs est supérieur ou égal à 0
bgtz Rs, label	I	Saute à l'instruction d'adresse label si Rs est strictement supérieur à 0
blez Rs, label	I	Saute à l'instruction d'adresse label si Rs est inférieur ou égal à 0
bltz Rs, label	I	Saute à l'instruction d'adresse label si Rs est strictement inférieur à 0
beqz Rs, label	P	Saute à l'instruction d'adresse label si Rs est égal à 0
bnez Rs, label	P	Saute à l'instruction d'adresse label si Rs est différent de 0
beq Rs, Rt, label	I	Saute à l'instruction d'adresse label si Rs est égal à Rt
bne Rs, Rt, label	I	Saute à l'instruction d'adresse label si Rs est différent de Rt
bge Rs, Rt, label	P	Saute à l'instruction d'adresse label si Rs est supérieur ou égal à Rt (comparaison signée)
bgeu Rs, Rt, label	P	Saute à l'instruction d'adresse label si Rs est supérieur ou égal à Rt (comparaison non signée)
bgt Rs, Rt, label	P	Saute à l'instruction d'adresse label si Rs est strictement supérieur à Rt (comparaison signée)
bgtu Rs, Rt, label	P	Saute à l'instruction d'adresse label si Rs est strictement supérieur à Rt (comparaison non signée)
ble Rs, Rt, label	P	Saute à l'instruction d'adresse label si Rs est inférieur ou égal à Rt (comparaison signée)
bleu Rs, Rt, label	P	Saute à l'instruction d'adresse label si Rs est inférieur ou égal à Rt (comparaison non signée)
blt Rs, Rt, label	P	Saute à l'instruction d'adresse label si Rs est strictement inférieur à Rt (comparaison signée)
bltu Rs, Rt, label	P	Saute à l'instruction d'adresse label si Rs est strictement inférieur à Rt (comparaison non signée)

l'utilisateur comme signées ou non signées.

Exemple : supposons que le registre $t0 contient la valeur $\texttt{ffffffff}_{16}$ et le registre $t1 la valeur $\texttt{00000001}_{16}$. Le registre $t0 est-il plus grand ou plus petit que $t1 ? Si l'on suppose que les deux valeurs sont non signées, $t0 vaut $2^{32} - 1$ alors que $t1 vaut seulement 1. Donc $t0 > $t1. Par contre, si l'on suppose que les deux registres contiennent des valeurs signées, $t0 vaut maintenant -1 et donc $t0 < $t1. C'est pourquoi l'on doit choisir les instructions de saut en fonction de la représentation utilisée (ex : « bgt » en signé et « bgtu » en non signé).

Certaines instructions au format J (figure 4c, page 4) peuvent aussi remplacer complètement les 26 bits de poids faible du registre PC par les 26 bits de leur champ « Adresse ». On parle alors de *saut absolu*. Il s'agit des instructions de type « *jump* » (table 9).

Que le saut soit relatif ou absolu, le programmeur l'exprime toujours sous la forme d'une étiquette représentant une adresse absolue, à charge pour l'assembleur de calculer le déplacement en cas de saut relatif.

TABLE 9 : Instructions de saut absolu. Une valeur immédiate est noté Imm ; une étiquette est notée label.

Instruction	Format	Description
j label	J	Saut à l'instruction d'adresse label
jal label	J	Saut à l'instruction d'adresse label après avoir sauvé dans le registre $ra l'adresse de l'instruction suivant jal
jalr Reg	R	Saut à l'instruction dont l'adresse se trouve dans Reg après avoir sauvé dans le registre $ra l'adresse de l'instruction suivant jalr
jr Reg	R	Saute à l'instruction dont l'adresse se trouve dans Reg

5 Programmer en langage d'assemblage *MIPS32*

L'écriture d'un programme en langage d'assemblage MIPS32 doit impérativement *se faire à partir d'un code de départ écrit en C ou C++*. Grâce à cela, il devient possible, lorsqu'un code en langage d'assemblage ne donne pas les résultats attendus, de déterminer si l'erreur est dans l'algorithme utilisé (auquel cas, la version C/C++ donne aussi un mauvais résultat) ou dans la traduction de C/C++ vers le langage d'assemblage.

5.1 Le programme principal

Lors de l'écriture d'un programme en C ou C++, le compilateur ajoute le programme principal au code de l'utilisateur. Comme on l'a déjà évoqué dans la section 3, ce programme, appelé `__libc_start_main` en C, a pour responsabilité d'appeler la fonction `main()` après avoir initialisé certaines bibliothèques. Au retour de `main()`, il se charge aussi de retourner le code d'erreur final au système d'exploitation.

Lorsque l'on écrit directement en langage d'assemblage, on doit aussi écrire le programme principal comme l'on ne peut plus se reposer sur un compilateur pour nous le fournir. Sur un système *MIPS32*, on appelle « `__start` » (avec deux tirets bas « _ ») ce programme. Comme l'étiquette correspondante doit pouvoir être accessible hors de la portée du fichier, on la déclarera avec la directive `.globl`. De même, on pensera à utiliser

Table 10 : Programme principal `__start`.

```
        .text
        .globl __start
__start:
        jal main
        li $v0, 10
        syscall
```

la directive `.text` afin que l'intégralité du programme soit bien stockée dans le segment de code.

Dans le cadre du module X31I050, le programme `__start` aura toujours la forme présentée dans la table 10 : le programme `__start` appelle la fonction `main()` puis rend la main au système par un appel au service 10 (exit).

Dans MARS, l'assemblage d'un programme se fait en deux passes sur le code source, ce qui permet d'écrire les fonctions dans n'importe quel ordre, y compris après leur premier appel. Cependant, le point d'entrée est défini par la première instruction rencontrée. Il est donc important que le code de `__start` apparaisse en premier dans le programme.

5.2 Les arguments de la ligne de commande

Sur un système *MIPS32* réel, il est possible de passer des arguments à un programme en les donnant sur la ligne de commande à la suite de son nom. Par exemple :

```
find . -name toto.txt
```

On peut alors récupérer ces arguments en déclarant des paramètres pour la fonction `main()` :

```
int main(int argc, char *argv[])
{
   cout << "Nombre d'arguments : " << argc << "\n";
   for (int i = 0; i < argc; ++i) {
      cout << *argv << "\n";
      ++argv;
   }
}
```

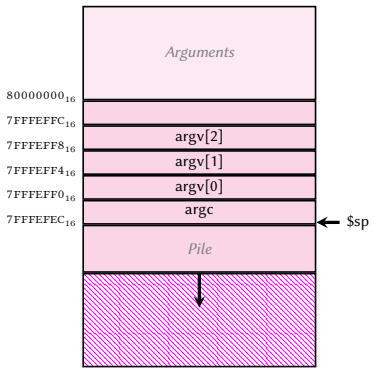


Figure 9 : Positionnement de `argc` et `argv` dans la pile avant exécution.

Le paramètre `argc` contient le nombre d'arguments et `argv` est un tableau contenant chacun des arguments sous forme d'une chaîne de caractères. On notera qu'en C, le nom du programme appelé est considéré comme un argument de la ligne de commande et se trouve à la position 0 du tableau `argv`.

Avec MARS, il est possible de passer des arguments à un programme en cochant la case Settings⟶Program arguments provided to MIPS program (figure 8).

Lorsque la case est cochée, un champ de saisie permet de fournir des arguments au programme. Avant l'exécution de la première instruction, le registre $a0 reçoit le nombre d'arguments fourni et le registre $a1 contient l'adresse de la première case du tableau `argv`.

Parallèlement, les paramètres de `main()` sont mis sur la pile. Initialement, le registre $sp contient l'adresse $7fffeffc_{16}$. Il est décrémenté de 4 octets pour stocker la valeur de `argv[argc-1]`, qui est l'adresse dans le segment commençant à l'adresse 80000000_{16} où commence la chaîne de caractères représentant le premier argument. Les valeurs de `argv[argc-2]`, ..., `argv[0]`, sont empilées à la suite, suivies de la valeur de `argc`. Si l'on passe trois arguments, on obtient alors la pile de la figure 9. On notera que, contrairement au programme C, le nom du programme lui-même ne fait pas partie des arguments.

Lorsque l'on rentre dans la fonction `main()`, on peut donc accéder aux arguments de deux façons :

- En utilisant $a0 et $a1 pour connaitre le nombre de paramètres et l'adresse de `argv[0]`;
- En accédant à `argc` et `argv` directement à partir de $sp.

Le programme ci-dessous traduit le code C++ montré précédemment en utilisant la première approche pour accéder aux paramètres de `main()` :

```
   .data
nargstr:  .asciiz "Nombre de paramètres: "
```

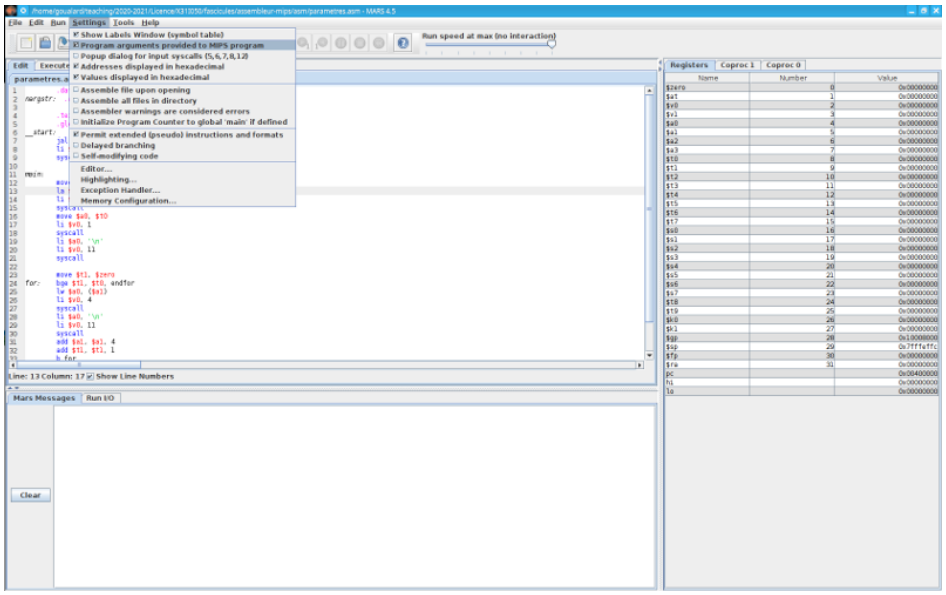


FIGURE 8 : Fourniture d'arguments à un programme sous MARS

```
   .text
   .globl __start
__start:
   jal main
   li $v0, 10
   syscall

main:
   move $t0, $a0
   la $a0, nargstr   # cout << "Nombre de paramètres";
   li $v0, 4
   syscall
   move $a0, $t0  # cout << argc;
   li $v0, 1
   syscall
   li $a0, '\n'    # cout << "\n";
   li $v0, 11
   syscall

   move $t1, $zero
for:  bge $t1, $t0, endfor
do:   lw $a0, ($a1)  # $a0 <- *argv
   li $v0, 4
   syscall
   li $a0, '\n'
   li $v0, 11
   syscall
   add $a1, $a1, 4
   add $t1, $t1, 1
   b for
endfor:
   jr $ra
```

5.3 Les données globales

Les variables globales d'un programme C/C++, et seulement celles-là, doivent être définies de façon à être insérées dans le segment de données lors de l'exécution. Cela se fait avec la directive .data. Tout ce qui suit cette directive jusqu'à la rencontre d'une directive .text sera stocké consécutivement en mémoire dans le segment de données, en tenant compte des contraintes d'alignement par une insertion d'octets de *padding* si nécessaire (figure 5).

On notera un piège terminologique : les *variables globales* d'un programme correspondent aux *données locales* de la figure 2. Dans cette figure, les *données globales* correspondent aux variables globales dont la portée dépasse le fichier de code source.

5.4 Les structures de contrôle

Le code C/C++ doit absolument être respecté dans sa traduction, ce qui implique de reprendre les structures de contrôle classiques de la programmation structurée : `for`, `while`, `if`, ...

5.4.1 La conditionnelle `if`

Considérons une conditionnelle en C/C++ :

```
if (x == 4) {
  // [Code du 'then']
} else {
  // [Code du 'else']
}
```

Si l'on traduit le test « x==4 » directement, on a deux possibilités de code en langage d'assemblage (en utilisant `$t0` pour x, par exemple) :

```
if:   beq $t0, 4, then
      b else
then:
      # [Code du 'then']
      b endif
else:
      # [Code du 'else']
endif:
```

```
if:   beq $t0, 4, then
else:
      # [Code du 'then']
      b endif
then:
      # [Code du 'else']
endif:
```

Aucune des deux solutions n'est satisfaisante : le code de gauche requiert deux instructions de saut conditionnel pour chaque test et le code de droite impose de remplacer la structure *if...then...else* par une nouvelle structure *if...else...then* sans équivalent en C/C++.

La solution est d'inverser le test comme ceci :

```
if:   bne $t0, 4, else # Sauter à 'else' si $t0 != 4
then:                 # Si $t0 == 4, on ne fait rien
                      # donc on rentre dans 'then'
      # [Code du 'then']
      b endif
else:
      # [Code du 'else']
endif:
```

Dans le cas de tests plus complexes, on peut avoir à revenir à une traduction directe ou non. Pour un test avec un « ET » logique, de la forme $X \wedge Y$, la négation donne $\overline{X \wedge Y} = \overline{X} \vee \overline{Y}$, qui se traduit naturellement en langage d'assemblage par une succession de tests niés :

```
if (a == 3 && b < 1) {
  // [Code du 'then']
} else {
  // [Code du 'else']
}
```

```
# On prend $t0 pour 'a' et $t1 pour 'b'
if:   # Aller à 'else' si $t0 != 3
      bne $t0, 3, else
      # Aller à 'else' si $t1 >= 1
      bge $t1, 1, else
then: # On n'a pas $t0 != 3 ni $t1 >= 1
      # donc on a $t0 == 3 ou $t1 < 1
      # [Code du 'then']
      b endif
else:
      # [Code du 'else']
endif:
```

Pour sauter à l'étiquette `else:`, il suffit que l'une des conditions niée soit vérifiée : la succession d'instructions de saut conditionnels correspond à un « ou » logique de toutes les conditions.

Si l'on a un test avec un « ou » logique, on ne peut plus utiliser cette approche car l'on a : $\overline{X \vee Y} = \overline{X} \wedge \overline{Y}$. Dans ce cas, on décide de ne pas nier les tests originaux :

```
if (a == 3 || b < 1) {
  // [Code du 'then']
} else {
  // [Code du 'else']
}
```

```
if:   # Aller à 'then' si $t0 == 3
      beq $t0, 3, then
      # Aller à 'then' si $t1 < 1
      blt $t1, 1, then
      b else
then:
      # [Code du 'then']
      b endif
else:
      # [Code du 'else']
endif:
```

Les schémas de traduction que l'on vient de voir doivent être scrupuleusement suivis même lorsqu'il est possible d'écrire un code plus compact.

Exemple de mauvaise traduction

```
if (a == 1) {
  // [Code de then1]
} else {
  if (a >= 2) {
    // [Code de 'then2']
  } else {
    // [Code de 'else2']
  }
}
```

```
# ATTENTION! Mauvaise traduction
if1:   bne $t0, 1, else1
then1:
       # // [Code de 'then1']
       b endif1
else1:
if2:   blt $t0, 2, else2
then2:
       # // [Code de 'then2']
       b endif1  # NON!
else2:
       # // [Code de 'else2']
endif1:
```

La bonne solution est ici d'introduire l'étiquette `endif2:` faisant partie du schéma de traduction :

```
if1:   bne $t0, 1, else1
then1:
       # // [Code de 'then1']
       b endif1
else1:
```

```
if2:   blt $t0, 2, else2
then2:
       # // [Code de 'then2']
       b endif2
else2:
       # // [Code de 'else2']
endif2:
endif1:
```

Notez qu'il n'est pas possible d'avoir plusieurs étiquettes avec le même nom. Dans ce cas, on choisit de numéroter les noms de structures « canoniques » dans leur ordre d'apparition.

Comme on l'a vu dans la traduction d'un `if` avec un test contenant des connecteurs logiques, toutes les instructions constituant le test doivent apparaitre entre les étiquettes « `if` » et « `then:` ». Cela est vrai même en l'absence de connecteurs. Considérons, par exemple, le programme C++ suivant, où `T` est un tableau d'entiers sur 4 octets :

```
if (T[i] == 3) {
  // code du 'then'
} else {
  // code du 'else'
}
```

En supposant que l'adresse de début du tableau `T` se trouve dans le registre $a0 et que la variable `i` est représentée par le registre $t0, on a :

```
if: sll $t1, $t0, 2   # $t1 <- i*4
    add $t1, $a0, $t1 # $t1 <- T+i*4
    lw $t1, ($t1)     # $t1 <- *(T+i*4)
    bne $t1, 3, else
then:
    # Code du 'then'
    b endif
else:
    # Code du 'else'
endif:
```

On notera que tout le code pour accéder à la variable `T[i]` apparaissant dans le test se trouve après l'étiquette `if` de façon à refléter qu'il fait partie du test dans le code C/C++.

5.4.2 La boucle `while`

Comme pour la conditionnelle, la traduction de la boucle `while` se fait en niant le test d'entrée :

```
while (a < 10) {
  // [Code du 'while']
}
```

```
# $t0 code la variable C 'a'
while: bge $t0, 10, endwhile
do:
       # [Code du 'while']
       b while
endwhile:
```

Notez la présence de l'étiquette « `do:` » ; elle n'est pas nécessaire au code car aucune instruction n'y fait référence. Elle est cependant utile pour déterminer visuellement où termine le test d'entrée et où commence le code de la boucle `while`. C'est d'autant plus important lorsque le test d'entrée requiert plusieurs instructions.

On pensera à ajouter le saut explicite au début de la boucle (`b while`) qui n'est qu'implicite dans le code C/C++.

5.4.3 La boucle `do...while`

La boucle `do...while` présente la particularité d'être traduite avec le test direct apparaissant dans le code C/C++ et non sa négation.

```
do {
  // [Code de la boucle]
} while (a != 1);
```

```
do:
       # //[Code de la boucle]
while:
       bne $t0, 1, do
```

5.4.4 La boucle `for`

La boucle `for` correspond à du « sucre syntaxique » pour une boucle `while` et c'est d'ailleurs par une telle boucle qu'elle est traduite (en utilisant ici `$t0` pour `i`) :

```
for (int i = 0; i < 10; ++i) {
  // [Code du 'for']
}
```

```
     # Initialisation de 'i'
     move $t0, $zero
for: bge $t0, 10, endfor
do:
     # // [Code du 'for']
     add $t0, $t0, 1 # ++i
     b for
endfor:
```

On commence par initialiser la variable d'itération à l'extérieur de la boucle. Il ne faut pas oublier d'incrémenter (ou décrémenter, en fonction de la boucle `for`) la variable d'itération à la fin du corps de la boucle, puis de sauter au test d'entrée.

5.5 Les appels au système avec `syscall`

Toutes les entrées/sorties, ainsi qu'un certain nombre d'autres *services* tels que l'allocation dynamique de mémoire, sont fournis par le système d'exploitation :

1. Le programme charge dans le registre `$v0` le numéro du service souhaité (table 11) puis exécute l'opération `syscall` ;
2. L'instruction `syscall` passe la main au système d'exploitation en arrêtant le programme courant ; le système d'exploitation regarde la valeur courante du registre `$v0`, en déduit le service demandé, récupère si nécessaire les valeurs utiles dans d'autres registres identifiés et exécute le service ;
3. Le système d'exploitation stocke, si nécessaire, dans un registre identifié une valeur de retour et rend la main au programme courant.

Lors d'un appel à `syscall`, seuls les registres de sortie sont modifiés par le système d'exploitation.

Exemple — Affichage d'un entier signé

Le code ci-dessous affiche la valeur « 42 » à l'écran en appelant le service `print_int` du système d'exploitation.

```
li $v0, 1   # Appel du service 'print_int'
li $a0, 42  # Chargement de la valeur à afficher
syscall
```

Exemple — Tirage de nombres aléatoires

Le programme suivant montre comment manipuler deux flux de nombres aléatoires différents en leur associant un identifiant (1 et 2). On tire deux nombres sur le premier flux et un seul sur le deuxième. À l'affichage, on voit trois nombres, le premier et le troisième étant identiques car les deux flux ont été initialisés avec la même valeur 13.

```
   .text
   .globl __start
__start:
   # Création du flux aléatoire 1
   li $v0, 40  # Demande du service rand_seed
   li $a0, 1
   li $a1, 13 # Initialisation du flux avec la valeur 13
   syscall
   # $a1 et $v0 contiennent toujours, respectivement, 13 et 40
   # Création du flux aléatoire 2
   li $a0, 2
   syscall
   # Récupération du nombre aléatoire suivant du flux 1
   li $v0, 41
   li $a0, 1
   syscall
   # Affichage du nombre (déjà présent dans $a0)
   li $v0, 1
   syscall
   # Récupération du nombre aléatoire suivant du flux 1
   li $v0, 41
   li $a0, 1
   syscall
   # Affichage du nombre (déjà présent dans $a0)
   li $v0, 1
   syscall
   # Récupération du nombre aléatoire suivant du flux 2
   li $v0, 41
   li $a0, 2
   syscall
   # Affichage du nombre (déjà présent dans $a0)
   li $v0, 1
   syscall
   # Exit
   li $v0, 10
   syscall
```

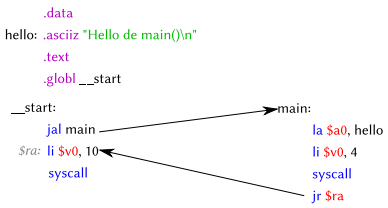


FIGURE 10 : Un appel de fonction simple

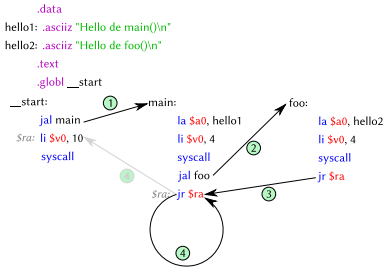


FIGURE 11 : Deux appels de fonctions en cascade

5.6 Les appels de fonctions

En langage d'assemblage, une fonction est simplement un morceau de code au début duquel on peut sauter (présence d'une étiquette pour référencer l'adresse de sa première instruction) et dont on peut revenir (connaissance de l'adresse de l'instruction suivant l'appel dans le code de la fonction appelante).

Le passage de paramètres ainsi que la manière de retourner le résultat d'une fonction font l'objet de conventions auxquelles il est crucial d'adhérer pour garantir l'interopérabilité du code et des outils associés (*debugger*, ...). Ces conventions font partie de l'*ABI* (*Application Binary Interface*). Plusieurs conventions existent pour les systèmes à base de processeur *MIPS32*. Les conventions que nous présentons ici sont une version légèrement simplifiée pour les besoins de ce module du document System V Application

TABLE 11 : Principaux services de `syscall`. La liste complète des services offerts dans MARS est disponible dans sa documentation.

Service	$v0	Description
`print_int`	1	Affiche l'entier contenu dans `$a0`
`print_string`	4	Affiche la chaîne de caractères dont l'adresse est dans `$a0`
`read_int`	5	Lit un entier au clavier et le stocke dans `$v0`
`read_string`	8	Lit une chaîne au clavier d'au plus `$a1` caractères (en comptant le `'\0'` terminal) et la stocke à l'adresse donnée par `$a0`
`malloc`	9	Alloue `$a0` octets dans le tas. Retourne dans `$v0` l'adresse de la mémoire allouée
`exit`	10	Arrête le programme et rend la main au système d'exploitation
`print_char`	11	Affiche le caractère dont le code ASCII se trouve dans l'octet de poids faible de `$a0`
`read_char`	12	Stocke dans `$v0` un caractère lu au clavier
`open`	13	Ouvre le fichier dont le nom se trouve dans `$a0` en lecture ou écriture en fonction de `$a1` — 0 : lecture seule — 1 : écriture seule (écrasement) — 9 : écriture seule (ajout) Sauve dans `$v0` le descripteur du fichier ouvert
`read`	14	Lit au plus `$a2` octets dans le fichier de descripteur `$a0` et les stocke à partir de l'adresse `$a1`. Sauve dans `$v0` le nombre d'octets effectivement lus (< 0 si erreur)
`write`	15	Écrit `$a2` octets stockés à partir de l'adresse `$a1` dans le fichier de descripteur `$a0`. Sauve dans `$v0` le nombre de caractères effectivement écrits (< 0 si erreur)
`close`	16	Ferme le fichier de descripteur `$a0`
`print_hex`*	34	Affiche en hexadécimal l'entier contenu dans `$a0`
`print_bin`*	35	Affiche en binaire l'entier contenu dans `$a0`
`print_uint`*	36	Affiche l'entier non signé contenu dans `$a0`
`rand_seed`*	40	Initialisation avec la valeur dans `$a1` du flux aléatoire dont l'identifiant est dans `$a0`
`random`*	41	Retourne dans `$a0` le nombre pseudo-aléatoire suivant du flux dont l'identifiant est dans `$a0`

Les services suivis d'une étoile « * » sont propres à MARS.

[8] *System V Application Binary Interface : MIPS® RISC Processor Supplement.* Rapp. tech. Troisième édition

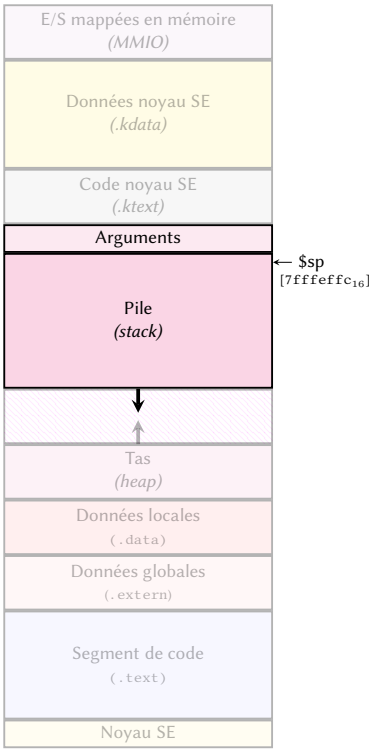


FIGURE 12 : Pile d'appels

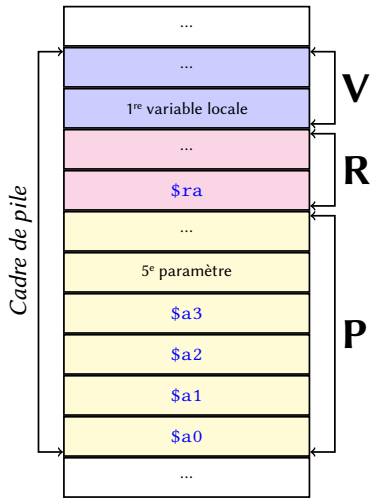


FIGURE 13 : Cadre de pile complet

[9] **Fonction terminale** · Une *fonction terminale* (*leaf function* en anglais) est une fonction qui n'appelle pas d'autre fonction.

[10] **Variable scalaire** · Une variable *scalaire* est une variable contenant une seule valeur (caractère, entier, nombre flottant). Par opposition, une variable *non scalaire* est de type tableau, structure, ...

Binary Interface[8].

— Note importante —

L'adhérence stricte aux conventions décrites ici et présentées en cours magistral est indispensable lors de la rédaction de code en langage d'assemblage durant les séances de travaux dirigés, travaux pratiques, contrôles continus et examens. Un code, même correct, utilisant une autre convention sera considéré comme invalide.

Le saut à une fonction se fait grâce à l'instruction « `jal` », qui prend en paramètre une adresse, sous forme d'une étiquette en général. Exemple :

```
jal main # Saut à la fonction main()
```

Cette instruction sauve dans le registre `$ra` l'adresse de l'instruction suivant le `jal` avant de positionner le pointeur d'instruction PC (*Program Counter*) à l'adresse donnée en paramètre, ce qui effectuera le saut au tic d'horloge suivant.

Le retour d'une fonction se fait en effectuant un saut explicite à l'adresse contenue dans le registre `$ra` par l'instruction `jr $ra` à la fin du code de la fonction appelée (figure 10, page 20).

Comme le processeur *MIPS32* ne dispose que d'un seul registre `$ra`, un problème apparaît lors de multiples sauts en cascade, comme dans l'exemple de la figure 11, page 20 : lors de l'appel initié par « `jal main` », l'adresse de retour dans le code `__start` est sauvée dans `$ra`. Cependant, lors de l'appel de la fonction `foo` dans la fonction `main` par l'instruction `jal foo`, l'adresse de retour dans `__start` est écrasée par celle dans `main`. En conséquence, on bouclera indéfiniment sur la dernière instruction dans le code de `main` au lieu de revenir au code de `__start`. Il est donc indispensable de sauver `$ra` avant de le modifier pour éviter ce genre de problème.

La seule manière pour une fonction de sauver de l'information en étant sûre qu'elle ne sera pas écrasée par une autre fonction est de réserver de la place dans la *pile* (figure 12) pour l'y stocker. La portion de pile réservée par une fonction s'appelle son *cadre de pile*.

5.6.1 Le cadre de pile

— Note importante —

Par soucis de clarté, sauf exception rendue visible par une hauteur clairement plus faible, chaque case apparaissant dans le dessin d'un cadre de pile présenté ci-après correspond à un *mot* en mémoire (quatre octets).

Le cadre de pile d'une fonction contient plusieurs types d'informations, organisées en trois blocs (figure 13) :

- Le bloc **V** pour les variables locales à la fonction ;
- Le bloc **R** pour stocker les registres à sauvegarder (`$ra`, par exemple) ;
- Le bloc **P** pour les paramètres des fonctions *appelées*. Par convention, s'il existe un bloc **P**, il doit avoir la taille suffisante pour sauver les registres `$a0`, `$a1`, `$a2` et `$a3`, *même si certains de ces registres ne sont pas utilisés*. Sa taille est donc d'au moins 16 octets.

Chacun des blocs est facultatif et sa présence dépend du type de fonction. Une fonction sans variable locale aura un cadre de pile sans bloc **V**, par exemple. Une fonction terminale[9] simple peut même se passer de cadre de pile. Les critères à considérer sont :

- Si une fonction a des variables locales non scalaires[10], elle devra avoir un bloc **V** ;

— Une fonction sans variable locale non scalaire et un nombre limité de variables locales scalaires n'a pas besoin de bloc **V** : les variables locales scalaires peuvent être directement stockées dans des registres. Cependant, si le nombre de variables locales scalaires est important, il faut alors les stocker dans un bloc **V** ;
— Une fonction qui fait appel à au moins une autre fonction doit avoir un cadre **R** pour sauver le registre `$ra` et un cadre **P** pour sauver les paramètres des fonctions appelées ;
— Une fonction qui utilise un registre à sauvegarder (`$s0`–`$s8`) doit avoir un cadre **R** pour l'y sauver avant de le modifier.

On a vu que les données en mémoire doivent être alignées sur des adresses multiples de leur taille. Par convention, il existe des contraintes d'*alignement* plus fortes pour la pile :

— Chaque valeur scalaire stockée sur la pile doit occuper au moins un *mot* (4 octets). Une valeur de taille moindre est étendue pour occuper quatre octets exactement ;
— Chaque bloc **P**, **R** et **V** doit avoir sa première adresse alignée sur un multiple de 8. En conséquence, la taille de chaque bloc doit être un multiple de 8 octets.

La création effective du cadre de pile se fait en manipulant le registre `$sp` (*stack pointer*). La pile étant orientée à l'envers et croissant vers les adresses les plus basses, la création d'un cadre de pile se fait en soustrayant sa taille de `$sp` (figure 14). À l'inverse, la destruction du cadre de pile se fait en rajoutant à `$sp` la taille du cadre de pile :

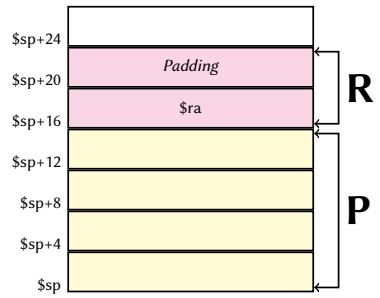


FIGURE 14 : Cadre de pile de `main()` avec un bloc **P** et un bloc **R**.

```
main:
      subu $sp, $sp, 24 # Création d'un cadre de 24 octets
      sw $ra, 16($sp)   # Sauvegarde de $ra dans le bloc R

      # Code de main() et modification éventuelle de $ra par un 'jal'
      # lors d'un appel de fonction.

      lw $ra, 16($sp)   # Récupération de la valeur sauvegardée de $ra
      addu $sp, $sp, 24 # 'Destruction' du cadre de pile
      jr $ra            # Retour à la fonction appelante
```

5.6.2 Le passage des paramètres

Si une fonction possède moins de quatre paramètres, ils sont stockés par la fonction appelante dans les registres `$a0`, `$a1`, `$a2` et `$a3` avant son appel :

```
int main(void)
{
  f(34,56);
  // Suite du code de main()
}
int f(int a, int b)
{
  // Code de f
}
```

```
main:
      subu $sp, $sp, 24
      sw $ra, 16($sp)
      li $a0, 34
      li $a1, 56
      jal f
      lw $ra, 16($sp)
      addu $sp, $sp, 24
      jr $ra
f:
   # Code de f avec $a0 et $a1
```

Si une fonction demande plus de quatre paramètres, les paramètres au-delà du quatrième sont stockés dans le cadre **P** de la fonction *appelante* (figure 15) :

```
int main(void)
{
  f(12,34,56,78,90);
  // Suite du code de main()
}
```

```
int f(int a, int b, int c,
      int d, int e)
{
  // Code de f
}
```

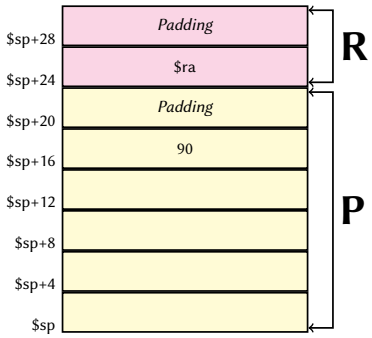


FIGURE 15 : Cadre de pile de `main()` pour appeler une fonction `f()` à cinq paramètres.

```
main:
      subu $sp, $sp, 32
      sw $ra, 24($sp)
      li $a0, 12
      li $a1, 34
      li $a2, 56
      li $a3, 78
      # On stocke le 5e paramètre
      # dans le cadre P
      li $t0, 90
      sw $t0, 16($sp)
      jal f
      # Suite du code de main()
      lw $ra, 24($sp)
      addu $sp, $sp, 32
      jr $ra
f:
   # Code de f
```

On notera qu'il est inutile de stocker dans le cadre de pile les paramètres qui sont déjà présents dans les registres `$a0` à `$a3`. Ces cases ne sont cependant pas inutiles car c'est là que la fonction *appelée* pourra sauver les paramètres qu'elle a reçu, si le besoin s'en fait sentir.

5.6.3 Retour du résultat d'une fonction

Le résultat d'une fonction doit être stocké dans le registre `$v0` avant le retour à la fonction appelante. On peut aussi utiliser le registre `$v1` si le résultat demande plus de 32 bits :

```
int foo(int a, int b)
{
  if (a > 2) {
    return 2*a + b;
  } else {
    return a - b;
  }
}
```

```
foo:
if:   ble $a0, 2, else
then:
      # $v0 <- 2*a
      sll $v0, $a0, 1
      # $v0 <- 2*a + b
      add $v0, $v0, $a1
      b endif
else:
      sub $v0, $a0, $a1
endif:
   jr $ra
```

On notera que l'instruction `return` du programme C/C++ ne se traduit pas formellement par un retour à la fonction appelante mais par une simple affectation à `$v0` de la valeur de retour.

Quelque soit le nombre d'instructions `return` dans une fonction C/C++, le code en langage d'assemblage ne doit effectuer un saut de retour à la fonction appelante qu'en un seul point, à la fin du code de la fonction appelée.

5.6.4 Exemples de traduction de fonctions

Les sections suivantes présentent des exemples de traduction de fonctions C/C++ avec le cadre de pile associé pour illustrer les différents cas possibles.

Une fonction simple prenant des pointeurs en paramètres La fonction `streq()` est une fonction terminale qui n'a pas besoin d'un cadre de pile. Elle est cependant intéressante car elle montre le cas d'une fonction prenant des pointeurs en paramètres :

```c
int streq(const char* s1,
          const char *s2)
{
   while (*s1 && *s2 && (*s1 == *s2)) {
      ++s1;
      ++s2;
   }
   return !*s1 && !*s2;
}
```

```
streq:
while:
   lb $t0, ($a0)           # $t0 <- *s1
   lb $t1, ($a1)           # $t1 <- *s2
   beqz $t0, endwhile
   beqz $t1, endwhile
   bne $t0, $t1, endwhile
do:
   addu $a0, $a0, 1
   addu $a1, $a1, 1
   b while
endwhile:
   seq $t0, $t0, $zero
   seq $t1, $t1, $zero
   and $v0, $t0, $t1
   jr $ra
```

On notera la nécessité de différentier les pointeurs, dans `$a0` et `$a1`, des caractères pointés, dans `$t0` et `$t1`.

Une fonction non terminale simple. Une fonction non terminale doit prévoir un cadre de pile avec un bloc **P** et un bloc **R**. La taille du bloc **P** est déterminée par le nombre de paramètres maximal demandé par les fonction appelées, en tenant compte de l'ajout éventuel d'un *padding* si la taille finale du bloc n'est pas un multiple de 8. La taille du bloc **R** dépend du nombre de registres à sauver.

```c
int fnts(int x,
         int y)
{
  bar1(x, y);
  bar2(x-y, -13,
       x+y, y,
       2*x, 6);
}
```

```
fnts:
     subu $sp, $sp, 32
     sw $ra, 24($sp)
     sw $a0, 32($sp)   # Sauvegarde de $a0
     sw $a1, 36($sp)   # Sauvegarde de $a1

     jal bar1          # Appel de bar1(a,b)
     # Récupération de $a0 et $a1
     # possiblement écrasés par bar1()
     lw $a0, 32($sp)
     lw $a1, 36($sp)
     add $a2, $a0, $a1 # $a2 <- x+y
     move $a3, $a1     # $a3 <- y
     sll $t0, $a0, 1   # $t0 <- 2*x
     sw $t0, 16($sp)   # '$a4' <- 2*x
     li $t0, 6
     sw $t0, 20($sp)   # '$a5' <- 6
     sub $a0, $a0, $a1 # $a0 <- x-y
     li $a1, -13       # $a1 <- -13
     # Appel de bar2(x-y, -13, x+y,
                     y, 2*x, 6)
     jal bar2

     lw $ra, 24($sp)
     addu $sp, $sp, 32
     jr $ra
```

La fonction `fnts()` appelle deux fonctions : la fonction `bar1()` demande deux paramètres et la fonction `bar2()` demande six paramètres. Il faut donc prévoir la place pour 6 paramètres dans le cadre **P** de `fnts()`. On doit aussi prévoir un cadre **R** de 8 octets pour sauver le registre `$ra`. On doit donc créer un cadre de pile de 32 octets (figure 16).

Après l'appel de `bar1()`, la fonction `fnts()` n'a aucune garantie de retrouver dans `$a0` et `$a1` les valeurs des paramètres qu'elle avait reçu. Comme elle en a besoin pour

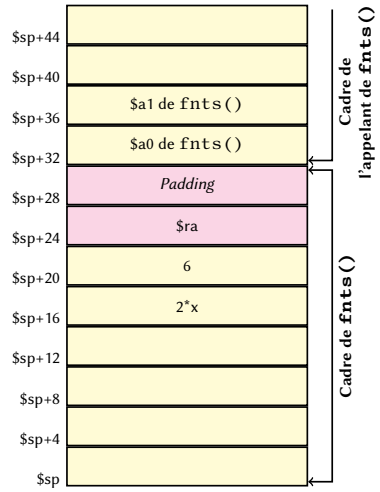


FIGURE 16 : Cadre de pile de `fnts()` lors de l'appel de `bar2()`.

appeler ensuite `bar2()`, elle doit les sauvegarder dans le bloc **P** de la fonction *qui l'a appelée*.

Une fonction avec variables locales On ne peut pas stocker dans des registres les variables locales non scalaires d'une fonction. Il faut donc prévoir un bloc **V** (figure 17).

```
int favl(int idx)
{
  int T[5] { 12, -23, 34, 45, 56 };
  int accu = 0;

  for (int i = 0; i < idx; ++i) {
    accu += T[i];
  }
  return accu;
}
```

```
favl:
   subu $sp, $sp, 24
   # Chargement du tableau local
   # dans la pile
   li $t0, 12
   sw $t0, 0($sp)
   li $t0, -23
   sw $t0, 4($sp)
   li $t0, 34
   sw $t0, 8($sp)
   li $t0, 45
   sw $t0, 12($sp)
   li $t0, 56
   sw $t0, 16($sp)
   move $v0, $zero # accu = 0
   move $t1, $zero # i = 0
for:  bge $t1, $a0, endfor
do:
   sll $t2, $t1, 2    # $t2 = i*4
   addu $t2, $sp, $t2 # $t2 = T+i*4
   lw $t2, 0($t2)     # $t2 = *(T+i*4)
   add $v0, $v0, $t2  # accu += T[i]
   add $t1, $t1, 1    # ++i
   b for
endfor:
   addu $sp, $sp, 24
   jr $ra
```

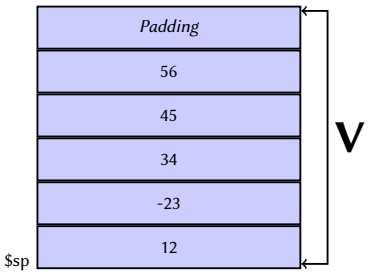


FIGURE 17 : Cadre de pile de `favl()`.

Une fonction utilisant des registres `$s*`. Une fonction est libre de modifier les registres `$t0`–`$t9` ainsi que les registres `$a0`–`$a3` et `$v0`–`$v1`. Par contre, elle a l'obligation de préserver les registres `$s0`–`$s8`. Pourquoi utiliser ces registres dans ce cas ? Considérons un exemple de programme C/C++où cela peut s'avérer utile :

```
int Tab[5] { 12, 23, 34, 45, 56};

int swap(int v)
{
   return 10*(v%10)+v/10;
}

void apply(int (*fun)(int x), int T[], unsigned int szT)
{
  for (unsigned int i = 0; i < szT; ++i) {
      T[i] = fun(T[i]);
   }
}

int main(void)
{
   apply(swap,Tab,5);
   for (unsigned int i = 0; i < 5; ++i) {
```

```
        cout << Tab[i] << " ";
    }
    cout << endl;
}
```

La fonction `apply()` reçoit en paramètres un pointeur sur une fonction prenant en entrée un entier et retournant un entier (c'est la notation « `int (*fun)(int x)`), un tableau d'entiers et la taille du tableau. Elle modifie le tableau en appliquant la fonction passée en paramètre sur chaque case.

On commence par écrire le code pour les variables globales (ici, `Tab`), pour le programme principal `__start`, la fonction `main()` et la fonction `swap()`.

La fonction `main()` n'est pas terminale, donc il lui faut un cadre de pile pour sauver `$ra` et les paramètres de la fonction appelée. À l'inverse, la fonction `swap` est terminale et n'a pas besoin d'un cadre de pile :

```
    .data
Tab:  .word 12, 23, 34, 45, 56

    .text
    .globl __start

__start:
    jal main
    # exit()
    li $v0, 10
    syscall

main:
    subu $sp, $sp, 24
    sw $ra, 16($sp)
    la $a0, swap
    la $a1, Tab
    li $a2, 5
    jal apply
    move $t0, $zero
for1: bgeu $t0, 5, endfor1
do1:
    sll $t1, $t0, 2    # $t1 = 4*i
    lw  $a0, Tab($t1)  # $t2 = *(Tab + 4*i)
    li $v0, 1
    syscall
    li $a0, ' '
    li $v0, 11
    syscall
    addu $t0, $t0, 1
    b for1
endfor1:
    li $a0, '\n'
    li $v0, 11
    syscall
    lw $ra, 16($sp)
    addu $sp, $sp, 24
    jr $ra

swap:
    li $t0, 10
    rem $t1, $a0, $t0 # $t1 = v%10
    mul $t2, $t0, $t1 # $t2 = 10*(v%10)
```

```
   div $t0, $a0, $t0 # $t0 = v/10
   add $v0, $t2, $t0 # return 10*(v%10)+v/10
   jr $ra
```

Dans la fonction `apply()`, dès lors que l'on appelle une fonction à l'intérieur de la boucle `for`, que ce soit par un appel direct avec `jal` ou un appel *via* une adresse passée en paramètre avec `jalr`, on se heurte au problème de préservation du contenu des registres à travers les appels de fonction : le compteur de boucle `i` va être stocké dans un registre et manipulé à chaque tour de boucle. Si on le stocke dans un registre comme `$t0`, on n'a aucune garantie que celui-ci contienne la même valeur avant et après l'appel de la fonction `fun()`. Une solution serait de le stocker dans le bloc **R** du cadre de pile de `apply()` avant chaque appel de `fun()` et de le récupérer après l'appel. Cela a comme inconvénient de rajouter de nombreux accès à la mémoire dans la boucle (un accès en écriture suivi d'un accès en lecture), ce qui dégradera forcément les performances. Une solution bien meilleure est de le stocker dans un registre comme `$s0` car, par convention, ce registre doit survivre à un appel de fonction : si une fonction le modifie, elle doit le réinitialiser avant de revenir à la fonction appelante. Comme cette contrainte est valable aussi pour `apply()`, cela suppose de prévoir de la place dans le bloc **R** de son cadre de pile pour sauver le registre `$s0` utilisé. Cependant, contrairement à ce qui serait nécessaire avec `$t0`, la sauvegarde de `$s0` et sa réinitialisation peut se faire une seule fois à l'extérieure de la boucle `for` (au début et à la fin de fonction, respectivement). On notera que le problème de la variable `i` se pose aussi pour `fun`, `T` et `szT` qui se trouvent, respectivement dans les registres `$a0`, `$a1` et `$a2`. De manière moins visible, on a aussi le problème du calcul de l'adresse de la i^e case de `T`, dont on a besoin pour récupérer la valeur `T[i]` avant d'appeler `fun()` et pour stocker le résultat après l'appel de `fun()`. On pourrait recalculer cette adresse après l'appel mais c'est sous-optimal. On va donc utiliser un registre sauvegardé `$s1`.

Afin d'illustrer les deux méthodes possibles pour préserver les valeurs de registres à travers l'appel d'une fonction, la version du code en langage d'assemblage ci-dessous utilise les registres sauvegardés `$s0` et `$s1` pour la variable `i` et le calcul de l'adresse de `T[i]`, mais elle sauve les registres `$a0`–`$a2` dans le bloc **P** du cadre de pile de la fonction `main()` (la fonction ayant appelé `apply()`) où se trouve une place pour les accueillir.

```
apply:
   subu $sp, $sp, 32
   sw $ra, 16($sp)
   sw $s0, 20($sp)
   sw $s1, 24($sp)
   sw $a0, 32($sp)
   sw $a1, 36($sp)
   sw $a2, 40($sp)
   move $s0, $zero
for2: bgeu $s0, $a2, endfor2
do2:
   # Calcul de l'adresse de T[i]
   sll $s1, $s0, 2
   addu $s1, $a1, $s1 # $s1 = T + i*4
   move $t0, $a0
   lw $a0, 0($s1)     # $a0 = T[i]
   jalr $t0           # fun(T[i])
   sw $v0, 0($s1)     # T[i] = fin(T[i])
   lw $a0, 32($sp)
   lw $a1, 36($sp)
   lw $a2, 40($sp)
   addu $s0, $s0, 1
   b for2
```

```
endfor2:
   lw $s1, 24($sp)
   lw $s0, 20($sp)
   lw $ra, 16($sp)
   addu $sp, $sp, 32
   jr $ra
```

La figure 18 montre l'état de la pile d'appel lorsque l'on est en train d'exécuter le code de la fonction `apply()`.

Une fonction utilisant tous les blocs L'exemple ci-dessous illustre l'utilisation de tous les blocs d'un cadre de pile avec une fonction chargée d'encrypter une chaîne de caractères.

```
char message[21+1] = "What hath God wrought";

int main(void)
{
   cout << message << endl;
   cipher(message, 21);
   for (unsigned int i = 0; i < 21; ++i) {
      cout << message[i];
   }
   cout << endl;
   cipher(message, 21);
   cout << message << endl;
}
```

La fonction `main()` requiert un cadre de pile classique de 24 octets car elle appelle une fonction de moins de cinq paramètres et n'a pas de variable globale non scalaire.

```
void cipher(char *msg, unsigned int sz)
{
   char cryptopad[6] {'T','u','r','i','n','g'};

   for (unsigned int i = 0; i < sz; ++i) {
      *msg = xor_cipher(cryptopad[i%6], *msg);
      ++msg;
   }
}
```

La fonction C++ `cipher()` possède une variable locale non scalaire (`cryptopad`). Il lui faut donc un cadre de pile avec un bloc **V** de 8 octets (6 octets pour les caractères du tableau `cryptopad` et 2 octets de *padding*). Comme elle appelle aussi une fonction avec deux paramètres (`xor_cipher()`), il lui faut le bloc **P** minimal de 16 octets. Enfin, il lui faut un bloc **R** pour sauver le registre `$ra`. Les variables `msg` et `sz` sont stockées dans les registres `$a0` et `$a1`, qui seront écrasés lors de l'appel de `xor_cipher()` à l'intérieur de la boucle `for`. Pour garantir que leurs valeurs seront toujours disponibles à chaque tour de boucle, on va copier les registres `$a0` et `$a1` dans `$s0` et `$s1`. De même, pour garantir que l'adresse de début de `cryptopad` et celle du compteur de boucle `i` ne seront pas écrasées par l'appel à `xor_cipher()`, on va les stocker dans `$s2` et `$s3`, respectivement. Cela fait cinq registres à sauver dans le cadre **R**, soit 24 octets en comptant le *padding*.

Au final, on a donc besoin d'un cadre de pile de 48 octets pour `cipher()`.

```
char xor_cipher(char c, char key)
{
```

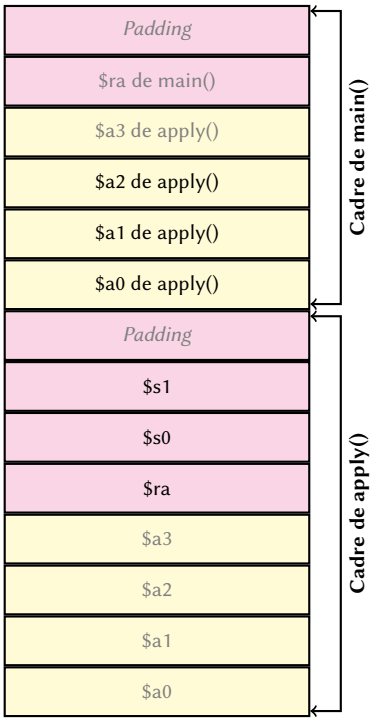


FIGURE 18 : Cadre de pile lors de l'appel de `apply()`.

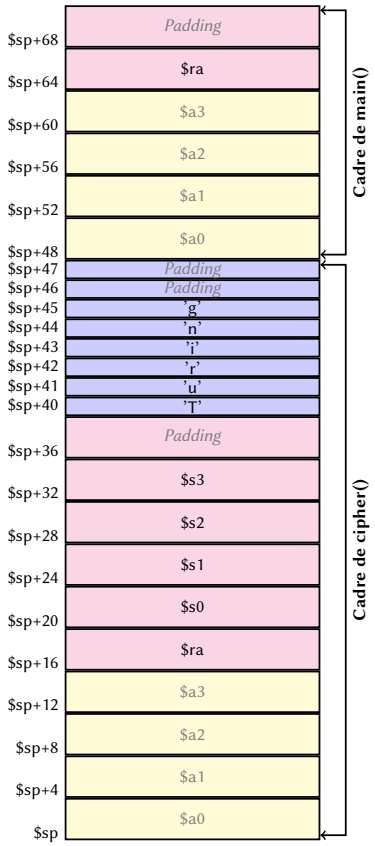


FIGURE 19 : Cadres de pile lors de l'appel de `cipher()`.

```
    return c ^ key;
}
```

La fonction `xor_cipher()` ne requiert aucun cadre de pile. Lors de l'appel de `cipher()`, on a donc dans la pile l'organisation présentée dans la figure 19.

Toutes les chaînes de caractères constantes doivent être collectées dans le segment de données. On a aussi le code habituel pour le programme principal `__start` :

```
    .data
message: .asciiz "What hath God wrought"

    .text
    .globl __start

__start:
    jal main
    # exit()
    li $v0, 10
    syscall
```

On notera l'utilisation de la directive `.asciiz`, qui assure que la chaîne est terminée par le caractère `'\0'`.

La fonction `main()` ne présente pas de difficultés particulières. On notera l'utilisation de l'adressage de la forme « *label(registre)* » pour récupérer chaque caractère de la chaîne.

```
main:
    subu $sp, $sp, 24
    sw $ra, 16($sp)
    # cout << message << endl;
    la $a0, message
    li $v0, 4
    syscall
    li $a0, '\n'
    li $v0, 11
    syscall
    # cipher(message, 21)
    la $a0, message
    li $a1, 21
    jal cipher
    move $t0, $zero # i = 0;
for1:
    bge $t0, 21, endfor1
do1:
    lb $a0, message($t0)
    li $v0, 11
    syscall
    addu $t0, $t0, 1 # ++i
    b for1
endfor1:
    # cout << endl;
    li $a0, '\n'
    li $v0, 11
    syscall
    # cipher(message, 21)
    la $a0, message
    li $a1, 21
    jal cipher
```

```
   # cout << message << endl;
   la $a0, message
   li $v0, 4
   syscall
   li $a0, '\n'
   li $v0, 11
   syscall
   lw $ra, 16($sp)
   addu $sp, $sp, 24
   jr $ra
```

Le code de la fonction `cipher()` doit créer un cadre de pile de 48 octets et remplir le bloc **V** caractère par caractère. On notera l'utilisation de l'instruction « `lb` » car chaque caractère n'occupe qu'un octet en mémoire. La fonction doit aussi sauver les registres `$s0`–`$s3` avant de les modifier et les réinitialiser avant de quitter.

```
cipher:
   subu $sp, $sp, 48
   sw $ra, 16($sp)
   sw $s0, 20($sp)
   sw $s1, 24($sp)
   sw $s2, 28($sp)
   sw $s3, 32($sp)
   # char cryptopad[6] {'T','u','r','i','n','g'};
   li $t0, 'T'
   sb $t0, 40($sp)
   li $t0, 'u'
   sb $t0, 41($sp)
   li $t0, 'r'
   sb $t0, 42($sp)
   li $t0, 'i'
   sb $t0, 43($sp)
   li $t0, 'n'
   sb $t0, 44($sp)
   li $t0, 'g'
   sb $t0, 45($sp)
   move $s0, $a0     # $s0: msg
   move $s1, $a1     # $s1: sz
   addu $s2, $sp, 40 # $s2: cryptopad
   move $s3, $zero   # $s3: i
for2:
   bge $s3, $s1, endfor2
do2:
   # $v0 = xor_cipher(cryptopad[i%6], *msg);
   remu $t0, $s3, 6
   addu $t0, $s2, $t0 # $t0 = cryptopad + i%6
   lb $a0, ($t0)      # $a0 = *(cryptopad + i%6)
   lb $a1, ($s0)
   jal xor_cipher
   sb $v0, ($s0) # *msg = xor_cipher(cryptopad[i%6], *msg);
   add $s0, $s0, 1 # ++msg
   add $s3, $s3, 1 # ++i
   b for2
endfor2:
   lw $s3, 32($sp)
   lw $s2, 28($sp)
   lw $s1, 24($sp)
   lw $s0, 20($sp)
   lw $ra, 16($sp)
```

```
    addu $sp, $sp, 48
    jr $ra
```

Enfin, la fonction `xor_cipher` est très simple ; elle n'a même pas besoin de cadre de pile :

```
xor_cipher:
    xor $v0, $a0, $a1
    jr $ra
```

Connaissant le code de la fonction `xor_cipher()`, on réalise qu'elle ne modifie que le registre `$v0` (il faut aussi modifier les registres `$a0` et `$a1` pour lui passer ses paramètres). Dans l'absolu, on aurait donc pu utiliser des registres non sauvegardés dans le code de `cipher` sans que l'exécution du code n'en pâtisse. On se souviendra cependant que l'on n'a généralement pas la connaissance de toutes les fonctions appelées et que l'on ne peut donc faire des hypothèses sur les registres modifiés ou non par une fonction. De plus, rien ne dit que le code de la fonction `xor_cipher()` ne sera pas modifié plus tard. On ne doit donc jamais faire d'hypothèse sur le comportement interne d'une fonction ni sur les registres qu'elle modifie, même si l'on en est l'implémenteur.

5.6.5 Processus de traduction d'une fonction en langage d'assemblage

Comme tout code en langage d'assemblage, une fonction doit toujours s'écrire à partir d'un code écrit en C/C++ :

```
int multrusse(int a, int b)
{
  int p = 0;
  while (b > 0) {
    if (b%2 == 1) {
      p += a;
    }
    a = 2*a;
    b = b/2;
  }
    return p;
}
```

On doit alors déterminer si la fonction C/C++ à traduire a besoin d'un cadre de pile. Ici, la fonction `multrusse()` est une fonction terminale et elle n'a pas de variable locale non scalaire donc on peut se passer d'un cadre de pile.

Ensuite, on reprend *servilement* la structure du code C/C++ en positionnant les étiquettes correspondantes (sans indentation) :

```
multrusse:
          # Initialisation de la boucle
while:
do:
if:
then:
endif:
          b while
endwhile:
```

On notera que le saut inconditionnel du `while` a déjà été positionné afin de ne pas l'oublier, comme il n'a pas d'équivalent explicite dans le code C++.

Il reste à ajouter le code à l'intérieur des structures :

```
multrusse:
          move $v0, $zero    # On utilise $v0 pour 'p'
while:    blez $a1, endwhile # si b <= 0 aller à la fin de boucle
do:
          and $t0, $a1, 1    # $t0 <- b%2
if:       bne $t0, 1, endif
then:
          add $v0, $v0, $a0
endif:
          sll $a0, $a0, 1    # a <- 2*a
          sra $a1, $a1, 1    # b <- b/2
          b while
endwhile:
          # $v0 contient le résultat à retourner
          jr $ra
```

5.6.6 Règles d'écriture d'un programme

Au-delà du respect des conventions montrées précédemment, il est important d'écrire le code en langage d'assemblage en tenant compte des points suivants :

— Le code doit contenir le minimum de commentaires simplifiant sa compréhension. Le respect de la structure générale du programme C/C++ rend superflu l'ajout systématique de commentaires. Par contre, un commentaire indiquant la correspondante registre/variable rend le code plus lisible :

```
int accu = 0;
```

```
move $t0, $zero # $t0 : accu
```

— Le cadre de pile d'une fonction doit toujours être construit dès l'entrée dans la fonction et détruit juste avant de la quitter, même si certains appels pourraient ne pas nécessiter de cadre. Exemple :

```
int recur(int x)
{
  if (x == 0) {
    return 4;
  } else {
    return recur(x-1);
  }
}
```

```
# MAUVAIS EXEMPLE !
recur:
if:   bnez $a0, else
then: li $v0, 4
      b endif
else:
      subu $sp, $sp, 24
      sw $ra, 16($sp)
      sub $a0, $a0, 1
      jal recur
      lw $ra, 16($sp)
      addu $sp, $sp, 24
endif:
      jr $ra
```

```
# BON EXEMPLE !
recur:
      subu $sp, $sp, 24
      sw $ra, 16($sp)
if:   bnez $a0, else
then: li $v0, 4
      b endif
else:
      sub $a0, $a0, 1
      jal recur
endif:
      lw $ra, 16($sp)
      addu $sp, $sp, 24
      jr $ra
```

— Contrairement au code C/C++, les structures de contrôle du code en langage d'assemblage ne doivent pas être indentées. Trois niveaux d'indentation seulement doivent être utilisés :

1. Aligné à gauche : toutes les étiquettes,
2. Une indentation : le code,
3. Deux indentations : les commentaires ;

Exemple :

```
if:     beq $t0, 4, endif  # $t0 != age ?
then:
        li $t1, -7
endif:
```

— En aucun cas le code en langage d'assemblage ne doit utiliser des raccourcis de structure ; il doit toujours représenter fidèlement la structure du code C/C++ modèle ;
— Les variables globales scalaires ainsi que les variables scalaires locales aux fonctions n'ont pas nécessairement besoin d'etre représentées en mémoire. Lorsque leur nombre n'est pas trop grand par rapport au nombre de registres disponibles, il suffit de les stocker directement dans un registre pour les manipuler ;
— On ne doit jamais faire d'hypothèse sur le comportement du code implémentant une fonction appelée, même si on l'a écrit soi-même et même si cette fonction est la fonction elle-même (appel récursif). C'est particulièrement vrai en ce qui concerne la liste des registres potentiellement modifiés par l'appel.

6 Licence de ce document

Les remarques, suggestions et indications d'erreurs peuvent être transmises à l'auteur : frederic.goualard@univ-nantes.fr.