Introduction au langage Assembleur MIPS

Modèle de Programme en Langage Assembleur MIPS

Un modèle de programme en Langage Assembleur MIPS est affiché ci-dessous à la figure 2.1.

```
# Title:
# Author:
# Date:
# Description:
# Input:
# Output:
###################  Data segment ######################
.data
 . . .
###################  Code segment ######################
.text
.globl main
main:                        # main function entry
 . . .
li $v0, 10
syscall                      # system call to exit program
```

Figure 2.1: Modèle de programme en Langage Assembleur MIPS

Il existe trois types d'instructions qui peuvent être utilisées dans le langage assembleur, où chaque instruction apparaît sur une ligne distincte :

1. *Directives de l'assembleur* : Celles-ci fournissent des informations à l'assembleur pour la traduction d'un programme en code machine. Les directives sont utilisées pour définir des segments et allouer de l'espace pour les variables dans la mémoire. Une directive de l'assembleur commence toujours par un point. Un programme typique en langage assembleur MIPS utilise les directives suivantes :

.data Définit le segment de données du programme, contenant les variables du programme.

.text Définit le segment de code du programme, contenant les instructions à exécuter.

.globl Définit un symbole comme global qui peut être référencé à partir d'autres fichiers.

2. *Instructions exécutables* : Celles-ci génèrent du code machine qui sera exécuté par le processeur. Les instructions indiquent au processeur ce qu'il faut faire.
3. *Pseudo-Instructions et macros* : Traduit par l'assembleur en instructions réelles. Ces pseudo-instructions simplifient la tâche du programmeur.

En outre, des commentaires peuvent être insérés dans le code. Les commentaires sont très importants pour les programmeurs, mais ignorés par l'assembleur. Un commentaire commence par le symbole **#** et se termine à la fin de la ligne. Les commentaires peuvent apparaître au début d'une ligne ou après une instruction. Ils expliquent l'objet du programme, quand il a été écrit, révisé, et par qui. Ils expliquent les données et les registres utilisés dans le programme, les entrées, les sorties, la séquence d'instructions et les algorithmes implémentés.

Le Cycle Edition-Assemblage-Liaison-Exécution

Avant de pouvoir exécuter un programme MIPS, vous devez convertir le texte source écrit en langage assembleur en une forme exécutable par le processeur. Il s'agit de deux étapes :

1. *Assembler* : Traduit le texte en langage assembleur MIPS vers un *fichier objet* binaire. Cela est réalisé par l'assembleur. S'il y'a plus d'un fichier source en langage assembleur, alors chacun de ces fichiers sera assemblé séparément.
2. *Edition de Liens* : Combine tous les fichiers objets ensemble (s'il y'en a plus d'un) et éventuellement avec des bibliothèques de fonctions. Cela est réalisé par l'éditeur de liens (*linker*). Le *linker* vérifie et lie les appels de fonctions dans les fichiers objets avec des bibliothèques ou d'autres fichiers objets. Le résultat de cette étape produit un *fichier exécutable*.

La figure 2.2 résume le cycle *Edition-Assemblage-Liaison-Exécution* pour produire un programme. Pour un texte source écrit dans le langage assembleur, *l'Assembleur* détecte toutes les *erreurs* de *syntaxe* et les signalera au programmeur. Par conséquent, vous devez modifier votre programme et l'assembler à nouveau s'il y'a des erreurs de syntaxe.

Il est typique que la première version exécutable de votre programme ait quelques erreurs *d'exécution* (anglais : *runtime errors*). Ces erreurs ne sont pas détectables par l'*Assembleur*, mais se produisent lorsque vous exécutez votre programme. Par exemple, votre programme peut calculer des résultats erronés. Par conséquent, vous devez *déboguer* votre programme pour identifier les erreurs au moment de l'exécution. Vous pouvez exécuter votre programme avec différentes entrées et dans différentes conditions pour vérifier qu'il fonctionne correctement.

Dans MARS, vous pouvez utiliser le mode « exécution lente », la fonctionnalité « exécution pas-à-pas », ou encore insérer des « points d'arrêt » dans votre programme pour identifier les sources d'erreur. L'« exécution pas-à-pas » est une caractéristique standard et essentielle dans tout débogueur. Elle permet d'inspecter l'effet de chaque instruction sur les registres du processeur et la mémoire principale.

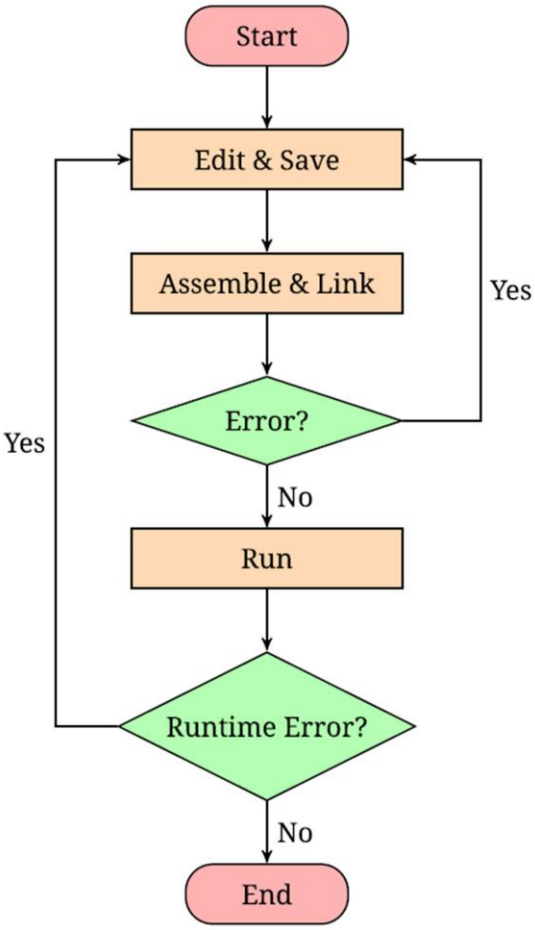


Figure 2.2: *Le* cycle *Edition-Assemblage-Liaison-Exécution*

Registres et formats d'instructions MIPS

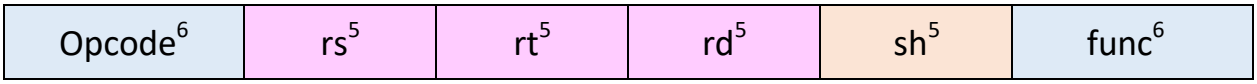
Toutes les instructions MIPS sont sur 32 bits et occupent donc quatre octets en mémoire. L'adresse d'une instruction MIPS en mémoire est toujours un multiple de quatre octets. Il existe trois formats d'instructions MIPS de base : le format Registre (type R), le format Immédiat (type I) et le format Jump (type J) comme indiqué à la figure 2.3.

Toutes les instructions MIPS ont un champ « **`Opcode`** » sur six bits qui définit le format et parfois le fonctionnement de l'instruction. Le format de type R comporte deux champs de registre source : « **`rs`** » et « **`rt`** », et un champ pour le registre de destination « **`rd`** ». Tous les champs de registre sont sur cinq bits et indexent donc 32 registres polyvalents. Le champ « **`sh`** » est utilisé pour indiquer la quantité de ***déplacement*** pour les instructions de décalage ; et le champ « **`func`** » définit la fonction UAL pour les instructions de type **R**.

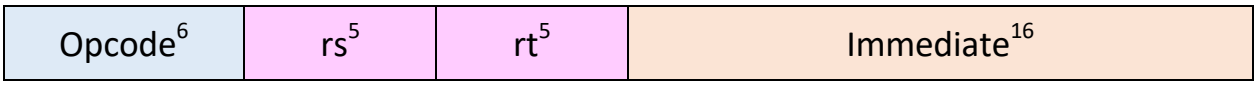
Le format de type **I** a deux champs de registre seulement : « **`rs`** » et « **`rt`** », où « **`rs`** » est toujours un registre source, tandis que « **`rt`** » peut être un registre de destination ou un deuxième registre source en fonction de l'instruction. Le champ « **`Immediate`** » de 16 bits est utilisé comme une constante dans les instructions arithmétiques, ou comme un indice/déplacement (anglais : *offset*) dans une instruction de lecture/écriture mémoire et les instructions de branchement.

Le format de type J ne possède pas de champ « **`registre`** ». Le champ « **`Immediate`** » de 26 bits est utilisé pour coder une adresse mémoire pour les instructions de saut et d'appels de fonction.

Format R-Type



Format I-Type



Format J-Type

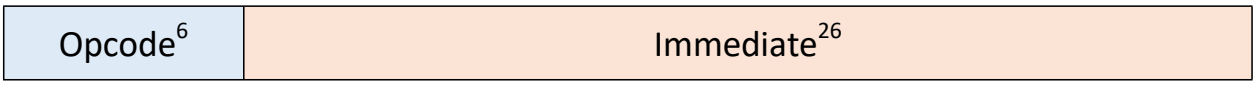


Figure 2.3: Formats d'instructions MIPS

L'architecture MIPS définit 32 registres à usage général, numérotés de **`$0`** à **`$31`**. Le symbole **`$`** est utilisé pour désigner un registre. Pour simplifier le développement de logiciels en assembleur, on peut également se référer aux registres par nom comme indiqué dans le tableau 2.1. L'outil assembleur convertira ensuite les noms des registres en leurs numéros correspondants.

Nom du registre	Numéro	Usage en programmation
$zero	**$0**	Toujours zéro, forcé par le matériel
$at	**$1**	Registre temporaire, réservé à l'utilisation de l'assembleur
$v0 – $v1	**$2 – $3**	Résultats d'une fonction
$a0 – $a3	**$4 – $7**	Arguments d'une fonction
$t0 – $t7	**$8 – $15**	Registres pour stocker des valeurs temporaires
$s0 – $s7	**$16 – $23**	Registres à sauvegarder entre les appels de fonction
$t8 – $t9	**$24 – $25**	Registres pour stocker plus de valeurs temporaires
$k0 – $k1	**$26 – $27**	Registres réservés à l'utilisation par le noyau OS
$gp	**$28**	Registre Pointeur Global qui pointe vers des données globales
$sp	**$29**	Registre de Stack qui pointe vers le haut de la pile
$fp	**$30**	Registre de structure qui pointe vers la structure de la pile
$ra	**$31**	Contient l'adresse de retour pour revenir d'un appel de fonction

Tableau 2.1 : Registres à usage général

La syntaxe générale d'une instruction du langage assembleur MIPS est la suivante :

```
[label:] mnemonic [operands] [# comment]
```

`label` (ou **`étiquette`** en français) est facultative. Elle marque l'adresse de mémoire de l'instruction et doit être suivie du caractère deux-points. En outre, une **`étiquette`** peut être utilisée pour se référer à l'adresse d'une variable en mémoire.

Le **`mnémonique`** spécifie l'opération : **`add`**, **`sub`**, etc.

Les **opérandes** spécifient les données requises par l'instruction. Différentes instructions ont un nombre différent d'opérandes. Les opérandes peuvent être des registres, des variables de mémoire ou des constantes. La plupart des instructions arithmétiques et logiques ont trois opérandes.

Un exemple d'instruction MIPS est illustré ci-dessous. L'exemple utilise l'opération **addiu** pour incrémenter le registre **$t0** :

```
L1: addiu $t0, $t0, 1    # incrémenter $t0
```

Un programme écrit en assembleur est constitué d'un ensemble d'instructions de base. Seules quelques instructions sont décrites dans les tableaux suivants. Le tableau 2.2 montre quelques instructions arithmétiques de base et le tableau 2.3 illustre des instructions de contrôle.

Instruction	Signification
`add rd, rs, rt`	`rd = rs + rt`
`sub rd, rs, rt`	`rd = rs - rt`
`addi rt, rs, Imm`	`rt = rs + Imm` (Imm est une constante signée sur 16 bits)
`li rt, Imm`	`rt = Imm` (pseudo-instruction d'affectation de constante)
`la rt, var`	`rt = adresse de var` (pseudo-instruction)
`move rd, rs`	`rd = rs` (pseudo-instruction d'affectation de registre)

Tableau 2.2 : Instructions arithmétiques de base.

Instruction	Signification
`beq rs, rt, label`	si `(rs == rt)` alors se brancher à l'étiquette `label` .
`bne rs, rt, label`	si `(rs != rt)` alors se brancher à l'étiquette `label`.
`J label`	Sautez à l'étiquette `label`.

Tableau 2.3 : Instructions de contrôle de base.

Appels système

Un programme effectue des entrées et des sorties à l'aide d'instructions spécifiques qui génèrent des appels système. Sur une machine réelle, ces appels sont gérés par le système d'exploitation (Windows, macOS, Un*x, etc) de la machine. Dans l'architecture MIPS, c'est l'instruction spéciale (**syscall**) qui permet de générer des « appels système ».

Les routines de gestion des appels système sont spécifiques au système d'exploitation utilisé par la machine. Chaque système d'exploitation fournit son propre ensemble de routines. Sur MARS, qui est un simulateur et non un système réel, il n'y a pas de système d'exploitation impliqué. Le simulateur MARS gère **l'exception initiée** par une instruction **syscall** et fournit des services système aux programmes. Le tableau 2.4 montre un petit ensemble de services fournis par MARS pour des opérations d'entrée/sortie de base.

Avant d'appeler l'instruction **`syscall`,** le numéro de service requis est chargé dans le registre **`$v0`**. De plus, les registres **`$a0`–`$a3`** peuvent être utilisés pour passer tout argument additionnel à la routine. Après avoir émis l'instruction **`syscall`,** la valeur éventuellement retournée est récupérée depuis le registre **`$v0`**.

service	$v0	argument	retour
Afficher un entier	1	**`$a0`** = entier à afficher	
Affiche une chaîne de caractères	4	**`$a0`** = adresse de la chaîne à afficher	
Lire un entier	5		**`$v0`** = entier lu
Lire une chaîne de caractères	8	**`$a0`** = adresse de mémoire tampon **`$a1`** = nombre maximum de caractère à lire	
Quitter le programme	10		Quitte le programme
Afficher un caractère	11	**`$a0`** = caractère à afficher	
Lire un caractère	12		**`$v0`** = caractère lu

Tableau 2.4 : Services d'appel système de base fournis par MARS.

L'exemple de code ci-dessous montre un programme simple qui demande à l'utilisateur d'entrer une valeur (un entier), puis affiche cette valeur sur la console de MARS (notre écran simulé).

Cinq appels système sont utilisés. Le premier appel système affiche la chaine de caractères *`str1`*. Le deuxième appel système lit un entier en entrée. Le troisième appel affiche la chaine de caractères *`str2`*. Le quatrième appel système affiche la valeur qui a été entrée par l'utilisateur. Le cinquième appel système quitte le programme.

Edit Execute

syscall.asm

```
1  .data
2  str1:       .asciiz     "Enter an integer value: "
3  str2:       .asciiz     "You entered "
4
5  .globl      main
6  .text
7  main:
8      li      $v0, 4      # service code for print string
9      la      $a0, str1   # load address of str1 into $a0
10     syscall             # print str1 string
11     li      $v0, 5      # service code for read integer
12     syscall             # read integer input into $v0
13     move    $s0, $v0    # save input value in $s0
14     li      $v0, 4      # service code for print string
15     la      $a0, str2   # load address of str2 into $a0
16     syscall             # print str2 string
17     li      $v0, 1      # service code to print integer
18     move    $a0, $s0    # copy input value
19     syscall             # print integer
20     li      $v0, 10     # service code to exit program
21     syscall             # exit program
```

Figure 2.4: Programme MIPS qui utilise les appels système.