

Concepts de base de la programmation en langage assembleur MIPS

Par :

Julien ROSSIGNOL

20 janvier 2019

Révisé par Yves Bérubé-Lauzière 2 février 2024

2 février 2024

Tous droits réservés ©2018-2024

Table des matières

Table des matières	1
I Liste des figures	2
1 Introduction	3
2 Structure des données	3
2.1 La mémoire	3
2.2 Registres	4
3 Les instructions de base	6
3.1 Les instructions mathématiques	7
3.2 Les instructions de chargement de constante	8
3.3 Les instructions de lecture et écriture en mémoire	8
3.4 Les instructions de saut	9
3.5 Les instructions de branchement	10
4 Les appels de fonction (forme abrégée)	11
5 La convention MIPS	11
5.1 La pile	12
5.2 Le rôle de la fonction appelante	12
5.3 Le rôle de la fonction appelée	13
6 Un exemple	13
7 Conclusion	15
8 Référence	15

Table des figures

1	Représentation approximative de la mémoire dans une application, les adresses sont arbitraires.	4
2	Les registres de l'architecture MIPS.	5
3	Exemple d'addition en assembleur.	6
4	Exemple de l'instruction sub en assembleur.	7
5	Exemple de l'instruction addi en assembleur.	7
6	Exemple de l'instruction sll en assembleur.	7
7	Exemple de l'instruction addi pour charger la mémoire en assembleur.	8
8	Exemple de l'instruction lui en assembleur.	8
9	Exemple de l'instruction lw en assembleur.	9
10	Exemple de l'instruction sw en assembleur.	9
11	Exemple de l'instruction j en assembleur.	9
12	Exemple de l'instruction jal en assembleur.	10
13	Exemple de l'instruction jr en assembleur.	10
14	Deux exemples de l'instruction beq en assembleur.	11
15	Exemple de pile dans une application.	12
16	Code d'exemple.	13
17	État initial des données statiques. On se rappellera ici que chaque chiffre dans un nombre hexadecimal (hex) représente 4 bits.	14
18	État des registres et des données statiques après l'exécution de la ligne 13.	15
19	État des registres et des données statiques après l'exécution de la ligne 16.	15

1 Introduction


Le présent document vise à présenter sommairement les concepts nécessaires pour débiter la programmation en assembleur MIPS. Le document n'est pas une référence exhaustive sur la programmation en assembleur ni un remplacement aux lectures recommandées de l'APP. Ainsi, certains concepts sont simplifiés pour faciliter la compréhension et peuvent ne pas représenter à 100% la complexité de l'architecture MIPS. Des liens vers des références plus complètes sont fournis dans le guide étudiant.

2 Structure des données

L'ensemble des opérations effectuées par les instructions du langage assembleur MIPS manipule les données de l'application qui se retrouvent dans deux types de structure : la mémoire de l'application et les registres. La mémoire peut être représentée par un tableau d'octets consécutifs dans lequel on retrouve la majorité des données de l'application ce qui comprend les instructions assembleurs. Les opérations sur la mémoire sont très limitées : il est seulement possible de faire des lectures et des écritures. Pour faire des calculs ou d'autres opérations, les données doivent d'abord être copiées dans les registres. Les registres sont des boîtes avec seulement 4 octets d'espace qui sont partagés par toute l'application.

2.1 La mémoire

Les applications utilisent généralement beaucoup de données, plus que ce que le processeur peut conserver en son sein ou à proximité pour effectuer rapidement des calculs. Parmi ces données on retrouve les instructions du code à exécuter, la pile, les variables et encore plus. Toutes ces données sont stockées dans un tableau d'octets consécutifs qui constitue la mémoire de l'application. Il est important de noter qu'il y a une distinction importante à faire entre la mémoire d'une application et la mémoire physique d'un système. De nombreux systèmes possèdent une couche d'abstraction qui sépare la mémoire de l'application, de la mémoire physique.

Chacun des octets du tableau possède une adresse qui permet de l'identifier uniquement, notamment pour la lecture et l'écriture. La figure  est une représentation approximative de la mémoire d'une application.

0x00004000	Pile (stack)
0x00004004	Pile (stack)
0x00004008	Pile (stack)
0x0000400C	Tas (heap)
0x00004010	Tas (heap)
0x00004014	Données statiques
0x00004018	Données statiques
0x0000401C	Données statiques
0x00004020	Instructions
0x00004024	Instructions
0x00004028	Instructions
0x0000402C	Instructions
0x00004030	Instructions
0x00004034	Instructions
0x00004038	Reservé
0x0000403C	Reservé

FIGURE 1 – Représentation approximative de la mémoire dans une application, les adresses sont arbitraires.

Cela a été dit précédemment, mais il est important de rappeler ici que les seules opérations qui peuvent être faites sur la mémoire sont la lecture et l'écriture. Pour pouvoir faire d'autres types d'opérations sur les valeurs contenues dans la mémoire, les valeurs doivent d'abord être déplacées vers les registres.

2.2 Registres

Pour faire des calculs le plus rapidement possible, le processeur possède quelques espaces de mémoire qu'il peut accéder très rapidement : les registres. Dans l'architecture MIPS, il y a 32 registres chacun de 32 bits (4 octets). Certains de ces registres ont des fonctions particulières dans le fonctionnement du processeur. Plusieurs autres ont des fonctions particulières par convention ; il est toutefois recommandé de respecter ces conventions.

On peut représenter les registres par des boîtes ou des variables sur lesquelles le processeur peut effectuer des opérations. Il faut toutefois faire attention, vu leur nombre limité, les registres sont régulièrement réutilisés. La convention MIPS permet de s'assurer de ne pas rencontrer des problèmes lors de l'appel de fonctions.

\$zero	0	\$at	Reservé pour pseudo-instructions	\$v0	Valeur de retour	\$v1	Valeur de retour
\$a0	Argument	\$a1	Argument	\$a2	Argument	\$a3	Argument
\$t0	Temporaire	\$t1	Temporaire	\$t2	Temporaire	\$t3	Temporaire
\$t4	Temporaire	\$t5	Temporaire	\$t6	Temporaire	\$t7	Temporaire
\$s0	Enregistré	\$s1	Enregistré	\$s2	Enregistré	\$s3	Enregistré
\$s4	Enregistré	\$s5	Enregistré	\$s6	Enregistré	\$s7	Enregistré
\$t8	Temporaire	\$t9	Temporaire	\$k0	Réservé au système d'opération	\$k1	Réservé au système d'opération
\$gp	Pointeur des données globales	\$sp	Pointeur de la pile	\$fp	Pointeur du cadre	\$ra	Adresse de retour

FIGURE 2 – Les registres de l'architecture MIPS.

La figure 2 présente les 32 registres de l'architecture MIPS. Chacun des registres peut être référencé par son numéro ou par son code de deux caractères. Généralement la deuxième méthode est privilégiée, car plus parlante.

Le registre \$zero ou plus généralement \$0 est particulier : il contient toujours la valeur zéro. L'écriture dans ce registre est impossible. Ce registre est très pratique pour initialiser des valeurs.

Les registres \$v0 et \$v1 sont les registres dans lesquels sont stockées les valeurs de retour (v pour valeur de retour).

Les registres \$a0 à \$a3 sont les registres de passage d'arguments (a pour argument). Si plus de 4 arguments sont nécessaires, les arguments supplémentaires sont déplacés en mémoire, plus précisément sur la pile.

Les registres \$t0 à \$t9 sont les registres temporaires (t pour temporaire). Ce sont des registres utilisés généralement pour de courtes séquences d'instructions. La convention MIPS ne prévoit pas qu'ils soient protégés par un appel de fonction. **Attention, cela veut dire que la valeur d'un registre temporaire peut être modifiée par une fonction appelée.** C'est pourquoi on préfère les utiliser que pour des valeurs qui sont utilisées dans les instructions qui suivent. Même à cela, il faut vraiment être prudent, car si une interruption survient, les valeurs qu'ils contiennent peuvent être altérées.

Les registres \$s0 à \$s7 sont les registres sauvegardés (s pour sauvegardé). Ce sont des registres utilisés pour contenir des valeurs qui sont utilisées tout au long d'une fonction appelée. La convention MIPS prévoit que ces registres ne soient pas modifiés par un appel de fonction. Il est donc sécuritaire de les utiliser de part et d'autre d'un appel de fonction. Pour ce faire, il faut protéger leurs valeurs à l'entrée d'une fonction en copiant ces valeurs ailleurs en mémoire (généralement sur la pile) et restaurer ces valeurs dans ces registres avant la sortie de la fonction. Ce procédé de sauvegarde et restitution est appelé protection du contexte. Avec cette protection, on est donc assuré que lorsqu'on utilise ces registres après un appel de fonction, ils ont la même valeur qu'avant l'appel.

Les registres \$sp, \$fp et \$ra ont des usages très précis qui seront détaillés plus loin.

Les registres \$at, \$k0, \$k1 et \$gp ont des usages particuliers et ne devraient pas être utilisés lors de l'écriture d'un code assembleur.

Le pointeur d'instruction (program counter)

Le pointeur d'instruction, appelé «program counter» (PC) en anglais, est une donnée particulière de l'architecture MIPS (et dans la plupart des autres architectures). En effet, celui-ci n'est pas accessible par les instructions. Toutefois sa valeur est très importante et il est nécessaire de comprendre le rôle du PC pour comprendre le fonctionnement de certaines instructions.

Nous avons indiqué précédemment que toutes les instructions se retrouvent dans la mémoire de l'application. Ainsi, chacune des instructions possède une adresse pour la désigner (l'adresse de la première case mémoire dans laquelle l'instruction se trouve - en MIPS cette case est de longueur de 1 octet, mais les instructions complètes sont dans ce qui est appelé un mot qui a une longueur de 4 octets). Le PC contient l'adresse de l'instruction actuelle à laquelle est rendue l'exécution d'un programme. En fait, un programme peut être représenté par une simple boucle en deux étapes : exécuter l'instruction à l'emplacement indiqué par le pointeur d'instruction puis incrémenter celui-ci de 4 (puisque chaque instruction est représentée sur 4 octets). C'est la seule valeur du programme qui est modifiée implicitement.

Même si le programmeur n'a pas directement accès au PC, de nombreuses instructions permettent de le modifier par exemple pour faire des boucles, pour appeler des fonctions ou pour faire des branchements conditionnels (if, switch, etc.).

3 Les instructions de base

Maintenant que nous connaissons l'architecture MIPS, nous pouvons commencer à faire des opérations sur des données à l'aide des instructions.

Il existe trois types d'instructions lues par le processeur en MIPS, soient les types R, I et J. Toutefois, pour faciliter la vie du programmeur, il existe ce qu'on appelle des pseudo-instructions. Les pseudo-instructions sont traduites en une série d'instructions qui peuvent être lues par le processeur. Les types d'instructions et la traduction de celles-ci en langage machine ne sont pas couverts dans le présent document.

Les instructions en assembleur débutent toutes par l'opération (ou fonction) à faire suivie des arguments séparés par des virgules.

add \$s1, \$t1, \$t2

Cette instruction fait une addition (add). Dans ce cas-ci trois paramètres sont passés à la fonction add : \$s1, \$t1 et \$t2. **Cette instruction fait l'addition des valeurs des registres \$t1 et \$t2 et stocke le résultat dans le registre \$s1 (voir figure 3).**

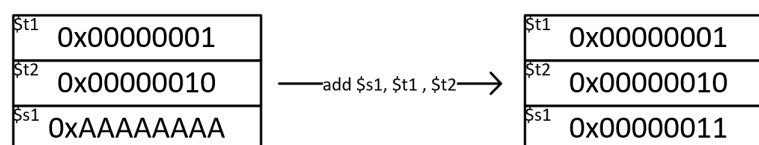


FIGURE 3 – Exemple d'addition en assembleur.

L'aide-mémoire pour l'assembleur MIPS est un excellent outil pour connaître l'effet d'une fonction.

3.1 Les instructions mathématiques

Les instructions les plus simples à comprendre sont celles pour effectuer les opérations mathématiques. Rappelons que ces instructions, comme la majorité des autres instructions, n'opèrent que sur des registres. Les données en mémoire doivent être transférées vers un registre avant d'effectuer une opération. Les instructions de lecture et d'écriture en mémoire seront présentées plus loin. La majorité des opérations mathématiques sont faciles à comprendre et fonctionnent comme l'addition.

`sub $s1, $t1, $t2`

Cette instruction soustrait \$t2 à \$t1 et stocke le résultat dans \$s1 (voir figure 4). Les instructions and, or, xor, nor, addu (addition non signée) et subu fonctionnent selon le même principe.

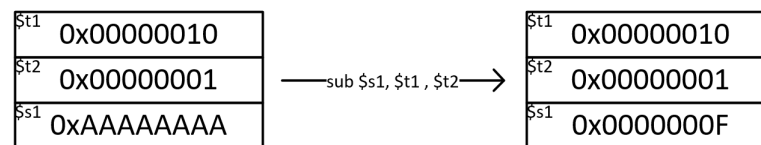


FIGURE 4 – Exemple de l'instruction sub en assembleur.

Il peut être pratique de faire ces opérations non pas entre deux registres, mais entre un registre et une constante. Par exemple pour incrémenter un registre.

Ex : `addi $s1, $t1, 1`

Cette instruction ajoute 1 à \$t1 et stocke le résultat dans \$s1 (donc \$s1 = \$t1 + 1, voir figure 5). Toutes les instructions présentées précédemment ont une version «immediate» (andi, ori, xori, nori, addui, subui), c'est-à-dire avec une constante.

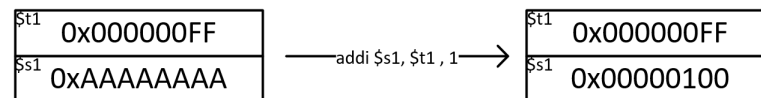


FIGURE 5 – Exemple de l'instruction addi en assembleur.

Les opérations de décalage sont aussi un peu différentes puisqu'un seul registre est décalé d'un nombre x de fois, x étant une constante et non la valeur d'un registre.

`sll $s1, $t1, 4`

Le contenu de \$t1 est décalé de 4 bits vers la gauche et le résultat est stocké dans \$s1 (voir figure 6). L'instruction slr permet de faire le décalage vers la droite. Ces instructions sont pratiques pour la multiplication et la division.

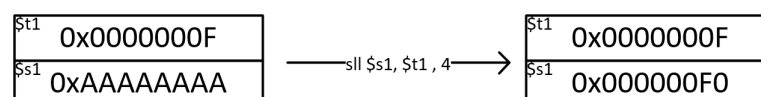


FIGURE 6 – Exemple de l'instruction sll en assembleur.

3.2 Les instructions de chargement de constante

Une opération fréquente en assembleur est le chargement d'une constante dans un registre. Généralement, cette opération sera faite avec une opération mathématique, par exemple avec l'opération suivante :

`addi $s0, $0, 16`

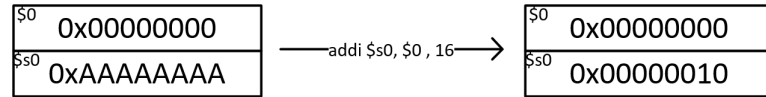


FIGURE 7 – Exemple de l'instruction addi pour charger la mémoire en assembleur.

Cette instruction fait l'addition de la valeur du registre \$0 (ou \$zero qui contient, rappelons-le, TOUJOURS la valeur 0) et de la valeur 16 et stocke le résultat dans le registre \$s0 (voir figure 7). Cette approche est fonctionnelle, mais limitée ! En effet, les instructions sont traduites en une série de 32 bits. Pour laisser de l'espace pour le code de la fonction et le code des deux registres, la valeur de l'«immediate» est limitée à 16 bits. Or les registres ont 32 bits et il est parfois commode d'utiliser l'entièreté de ces bits. Il existe donc une instruction pour nous aider.

`lui $s0, 16`

Cette instruction («load upper immediate») charge la constante fournie dans la partie haute du registre, donc dans les 16 bits les plus significatifs et met à zéro les 16 bits les moins significatifs (voir figure 8). Cette instruction est souvent utilisée pour traduire des pseudo-instructions utilisant des constantes de 32 bits. Ensuite pour charger les 16 bits les moins significatifs, on peut utiliser l'instruction ori qui fait un OU logique bit à bit entre deux mots.

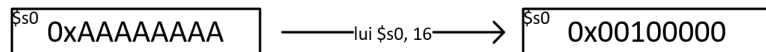


FIGURE 8 – Exemple de l'instruction lui en assembleur.

3.3 Les instructions de lecture et écriture en mémoire

Puisque l'espace disponible dans les registres est limité, les programmes ont souvent besoin d'enregistrer et de charger des valeurs dans la mémoire de l'application. Ces instructions ont une syntaxe particulière.

`lw $s0, 0($s1)`

Cette instruction («load word») charge dans le registre \$s0 les quatre octets (un mot) à l'adresse contenue dans \$s1 (voir figure 9). En fait, puisque chaque adresse ne correspond qu'à un seul octet, cette instruction va charger l'octet à l'adresse \$s1 puis celui à l'adresse \$s1+1 et ainsi de suite. Le 0 avant la parenthèse est particulier, il s'agit d'un décalage appliqué sur l'adresse. Si le 0 était changé pour un 4 par exemple, les adresses des octets chargées auraient été \$s1+4, \$s1+5, \$s1+6 et \$s1+7. Cela permet notamment de se déplacer dans un tableau par exemple (l'adresse du début du tableau étant toujours dans \$s1).

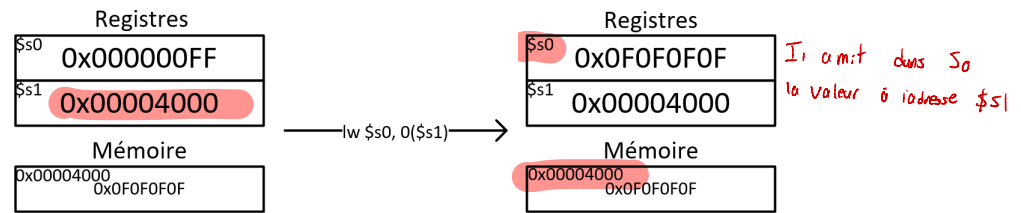


FIGURE 9 – Exemple de l’instruction lw en assembleur.

Il est aussi possible de charger un seul octet avec l’instruction «load byte» (lb). De nombreuses autres possibilités sont aussi disponibles, mais généralement moins communes que lw et lb.

Les instructions de stockage sw et sb fonctionnent de la même manière.

sw \$s0, 4(\$s1)

La valeur contenue dans le registre \$s0 est enregistrée à l’adresse contenue dans \$s1 décalée de 4 (et les trois cases mémoires subséquentes), la figure 10 présente un exemple de cette instruction.

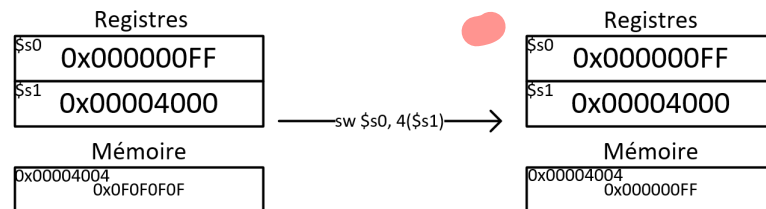


FIGURE 10 – Exemple de l’instruction sw en assembleur.

3.4 Les instructions de saut

Les instructions de saut permettent de réaliser des applications assembleurs avec plus d’une fonction !

Pour ce faire, il faut d’abord introduire le concept d’étiquette («label»). Les étiquettes sont des noms que l’on peut donner à une adresse mémoire pour la référencer plus tard. C’est un outil pratique pour le programmeur puisque cela évite de calculer la véritable adresse d’une instruction ou d’une donnée à chaque fois que l’on modifie quelque chose dans le programme. Les étiquettes sont placées entre deux instructions ou deux déclarations de données et vont automatiquement prendre la valeur de cet emplacement mémoire. Des exemples seront donnés plus loin dans ce document.

Trois instructions de saut sont communes, la première est de loin la plus simple :

j etiquette

La valeur du PC est mise à la valeur de l’étiquette. La prochaine instruction sera donc celle suivante l’étiquette (voir figure 11).

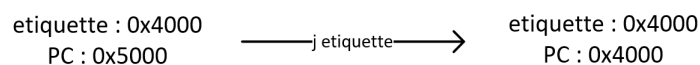


FIGURE 11 – Exemple de l’instruction j en assembleur.

Ce n’est toutefois pas pratique pour faire un appel de fonction. En effet, comment savoir à quelle

instruction reprendre lors du retour de la fonction ? Le registre `$ra` sert à cela, celui-ci étant utilisé pour les adresses de retour (ra pour «return adress»). Or, il est impossible de savoir à quelle adresse nous nous trouvons facilement dans un code puisque le PC n'est pas disponible à la lecture. Le langage MIPS possède une instruction pour cela.

jal étiquette

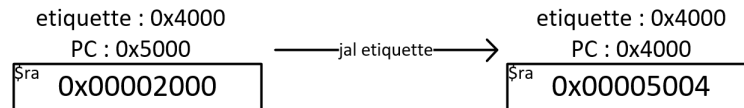


FIGURE 12 – Exemple de l’instruction jal en assembleur.

Cette instruction («jump and link») fait un saut vers l’étiquette comme l’instruction `j`, mais en plus stocke dans le registre `$ra` l’adresse de l’instruction suivant l’appel où retourner une fois la fonction exécutée (voir figure 12). On peut ainsi utiliser cette instruction pour revenir à l’instruction où on est rendu dans le programme une fois l’appel et l’exécution de la fonction complétés :

jr `$ra`

Cette instruction fait un saut vers l’adresse contenue dans le registre `$ra` (voir figure 13). On peut donc effectuer de cette façon un appel de fonction et un retour. On pourrait théoriquement spécifier n’importe quel registre, mais cela est peu recommandé. En général, on essaie d’éviter de trop manipuler le PC, l’instruction `jr` est directement une écriture sur le PC. C’est d’ailleurs comme cela que de nombreuses failles dans des systèmes sont exploitées en faisant sauter le PC vers un code injecté dans un programme qui peut avoir été mis dans l’espace des données par exemple (il est assez facile de mettre de données malveillantes dans cet espace).

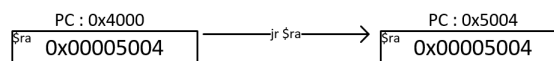


FIGURE 13 – Exemple de l’instruction jr en assembleur.

Une chose importante à noter : si vous appelez une fonction au sein d’une fonction, la valeur du registre `ra` sera réécrite. Il faut donc absolument l’enregistrer en mémoire puis la remettre en place après l’appel de fonction ; il faut donc protéger sa valeur.

3.5 Les instructions de branchement

Il existe de nombreuses autres classes d’instructions d’assembleur, mais nous allons terminer la revue de celles-ci avec une dernière classe bien utile : les branchements. Ces instructions permettent de faire des sauts dans le code, mais seulement si une condition est remplie (à l’instar des «if», «switch», etc.).

Il existe plusieurs instructions de branchements pour imiter les opérateurs d’égalité, d’inégalité, «plus grand que» et «plus petit que». Référez-vous à la carte verte pour la liste des instructions les plus courantes. Ces instructions fonctionnent toutes de la même manière.

beq `$s1, $s2, étiquette`

Cette instruction («branch on equal») positionne le PC à la valeur de l'étiquette si les valeurs contenues dans les deux registres sont égales (voir figure 14), la prochaine instruction exécutée dépendra donc de la valeur des registres. Notez que la comparaison se fait toujours de gauche à droite pour les opérations «plus petit que» et «plus grand que».

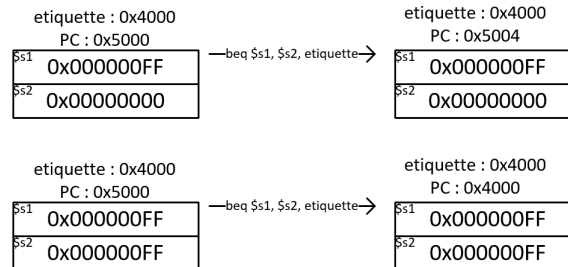


FIGURE 14 – Deux exemples de l'instruction `beq` en assembleur.

4 Les appels de fonction (forme abrégée)

Comme en programmation C/C++, Python ou Java, l'assembleur MIPS permet de subdiviser un programme en fonctions et sous-fonctions. Cependant, contrairement aux langages évolués, c'est au programmeur de gérer entièrement les étapes au frontières de l'appel. Dans le cas d'une fonction qui n'en appelle pas une autre, le mécanisme d'appel de fonction se résume à 4 actions (deux au début et deux à la fin) : 1) transférer les arguments requis par la fonction (registres `$a`) ; 2) déplacer l'exécution du code vers l'implémentation de la fonction (*jump and link*, *JAL*, section 3.4) ; après l'exécution de la fonction 3) revenir au point d'appel d'origine tout en 4) y renvoyant la ou les valeurs de retour (par les registres `$v`), si pertinent. Voyez en exemple la fonction *swap* de la section 2.13 de COD5 [1].

5 La convention MIPS

La nature de la programmation en assembleur oblige à une attention particulière lorsque l'on utilise des fonctions. En programmation C, la protection de contexte entre les fonctions est faite automatiquement et empêche que des données de la fonction appelante soient modifiées par la fonction appelée. En assembleur, c'est le programmeur qui doit s'assurer de ne pas récrire par dessus des données importantes. La convention MIPS permet de le faire et permet le partage de codes entre programmeurs puisqu'elle garantit qu'aucun effet indésirable ne peut résulter de l'appel d'une fonction. L'utilisation de la convention MIPS est requise dans le développement de la solution de la problématique.

La convention MIPS intervient à trois moments :

1. Juste avant l'appel d'une fonction
2. Au début d'une fonction
3. Juste avant le retour d'une fonction

5.1 La pile

Les instructions permettent de modifier la mémoire à n'importe quelle adresse. Il est toutefois risqué de mettre des données un peu n'importe où en mémoire. La pile permet d'avoir une façon plus structurée pour la gestion des données qui peut facilement être utilisée dans toutes les fonctions, même celles qui n'ont pas été conçues spécifiquement pour l'application.

La pile est une section de la mémoire dans laquelle les fonctions placent leurs données. Chacune des fonctions ajoute ses données sur le dessus de la pile et les retire avant de retourner à la fonction appelante. La section de la pile réservée à une fonction est appelée le «cadre» (ou «frame» en anglais). Le dernier cadre sur la pile correspond toujours à la fonction actuelle et celui en dessous, à la fonction ayant appelé la fonction actuelle.

Deux registres permettent de faire la gestion de la pile : \$sp et \$fp. Le registre \$sp est le pointeur de la pile («stack pointer») et contient l'adresse du dessus de la pile. Il faut donc le déplacer à chaque fois que des données sont ajoutées ou retirées de la pile. Le registre \$fp correspond au pointeur du cadre («frame pointer»), il contient l'adresse de fin du premier mot du cadre de la fonction. En pratique, on a rarement besoin de cette valeur, mais elle peut s'avérer utile lorsque des éléments sont ajoutés sur la pile en boucle ou dans des conditions. Le registre \$fp permet donc de toujours retrouver le début du cadre de la fonction, même si la position du pointeur de pile par rapport à celui-ci n'est plus connue. La figure 15 présente l'organisation de la pile d'une application dans laquelle la fonction 1 appelle la fonction 2 qui appelle la fonction 3.

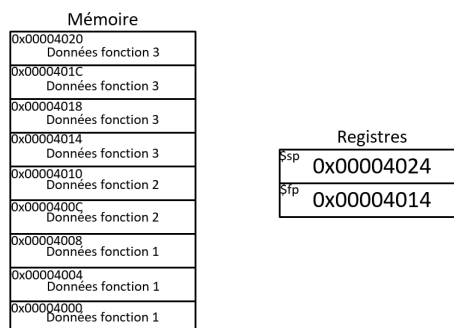


FIGURE 15 – Exemple de pile dans une application.

Important : la pile est construite à l'envers dans la mémoire, pour ajouter des octets sur la pile il faut donc diminuer le pointeur de pile et non l'augmenter !

5.2 Le rôle de la fonction appelante

Pour éviter de devoir enregistrer tous les registres utilisés à chaque appel de fonction, ce qui serait coûteux en performance, la convention MIPS prévoit que les registres \$t0 à \$t9 et \$a0 à \$a3 ne sont pas protégés lors de l'appel d'une fonction. La valeur de chacun de ces registres peut donc changer lors de l'appel d'une fonction. Si cette valeur doit être réutilisée après l'appel de la fonction, c'est le rôle de la fonction appelante de les enregistrer sur la pile.

5.3 Le rôle de la fonction appelée

Selon la convention MIPS, les registres \$s0 à \$s7, \$fp et \$ra doivent être protégés par le programmeur lorsqu'il y a un appel de fonction. Leur valeur est donc assurée d'être la même avant et après un appel de fonction si la convention MIPS est respectée. Cela ne veut pas dire qu'ils ne peuvent pas être utilisés dans une fonction ! La valeur de ces registres est placée sur la pile au début de la fonction et replacée dans les registres tout juste avant de faire le retour. Les valeurs sont donc préservées du point de vue de la fonction appelante. Toutefois, pour éviter de devoir enregistrer chacun de ces neuf registres à chaque appel de fonction, seuls les registres utilisés par la fonction doivent être enregistrés. Les autres registres ne seront de toute façon pas modifiés par les instructions de la fonction. **Lorsqu'il y a un appel de fonction dans une fonction, il est important de ne pas oublier d'enregistrer \$ra et \$fp !**

6 Un exemple

Le code suivant est un exemple de programme langage assembleur mettant en application une partie des concepts présentés dans ce document. Celui-ci fait l'addition de deux valeurs et stocke le résultat dans la mémoire de l'application.

```
1  .data    0x10010000
2                                     # donnees
3  x: .byte 1, 2
4      .align 2
5  r: .word 0                        # resultat
6
7  .text    0x400000
8  .globl  main
9
10 main:
11      la    $t0, x
12      lb    $s0, 0($t0)
13      lb    $s1, 1($t0)
14      add   $t0, $s0, $s1
15      la    $t1, r
16      sw    $t0, 0($t1)
17
18      ori   $v0, $0, 0xA
19      syscall
```

FIGURE 16 – Code d'exemple.

Examinons ce code ligne par ligne. Les lignes 1 à 5 initialisent des données statiques dans l'application. D'abord, la ligne 1, avec la directive «.data» indique qu'il s'agit de la section initialisant les données statiques. La valeur 0x10010000 est l'adresse de départ dans l'espace mémoire des données qui seront initialisées dans la section. La spécification explicite de cette adresse n'est pas toujours nécessaire (elle l'est généralement pour un simulateur, mais pour un dispositif réel, le compilateur dispose habituellement de cette information selon le plan d'adressage du dispositif). La ligne 3 commence par l'étiquette «x»,

celle-ci pourra être utilisée dans le code pour référencer l'adresse actuelle (0x10010000). Ensuite, la directive «.byte» indique que les données qui suivent seront des octets et les valeurs «1» et «2» qui suivent initialisent ces deux octets, l'un avec la valeur 1 et l'autre avec la valeur 2. Ces octets seront respectivement à l'adresse 0x10010000 et l'adresse 0x10010001 (pour rappel, les adresses sont en octets). La directive «.align n» de la ligne 4 aligne les données sur une limite qui est un multiple de 2^n octets. Ici, la valeur de l'argument n est 2, ce qui correspond à des limites alignées sur des multiples de 4 octets. Comme 4 octets correspondent à un mot (word) et que l'architecture MIPS travaille avec des mots de 32 bits (4 octets), il est naturel d'aligner les données sur des multiples de 4 octets comme c'est fait ici, autrement les données en mémoire seront mal organisées.

L'adresse où on est rendu à ce stade dans l'assignation des données, c.à.d. où la prochaine donnée (s'il y en a une) serait inscrite, est 0x10010002. Or, le prochain multiple de 4 est 0x10010004 et à cause de la directive .align, la prochaine donnée initialisée sera donc à l'adresse 0x10010004 plutôt qu'à 0x10010002. En d'autres mots, la directive .align 2 permet de faire un saut dans l'espace mémoire des données pour que le début d'une prochaine donnée qui suivra la directive soit alignée sur une limite qui est un multiple de 4 octets. Évidemment, si on avait eu la directive .word qui avait suivi «x :», alors on n'aurait pas eu besoin de se soucier d'un alignement, car les données auraient été des mots déjà alignés sur des limites qui sont des multiples de 4 octets. Finalement la ligne 5 commence elle aussi par une étiquette : «r», la valeur de celle-ci sera donc 0x10010004. Le reste de la ligne initialise 4 octets (un mot) à la valeur 0. La figure 17 présente l'allure des données en mémoire au début du programme.

Données statiques	
x:	0x10010000 0x00000201
r:	0x10010004 0x00000000

FIGURE 17 – État initial des données statiques. On se rappellera ici que chaque chiffre dans un nombre hexadecimal (hex) représente 4 bits.

La ligne 7 indique que les lignes suivantes correspondent à la section de texte, c'est-à-dire aux instructions. L'adresse du début des instructions est fournie (0x400000, noter de nouveau que donner l'adresse n'est pas toujours nécessaire), il y a donc beaucoup d'espace entre les données statiques et les instructions au sein de la mémoire. La ligne 8 indique avec la directive «.globl» que l'étiquette «main» peut être utilisée dans d'autres fichiers.

Le code commence à la ligne 11, toutefois, la ligne 10 déclare l'étiquette «main» qui pourrait servir pour appeler cette section de code ou retourner au début de la fonction par exemple. La valeur de l'étiquette est 0x400000. La ligne 11 est une instruction qui n'est pas présentée dans ce document, mais qui est très utile. Il s'agit en fait d'une pseudo-instruction qui charge la valeur de l'étiquette dans le registre spécifié. Ainsi, la valeur de «x» qui est de 0x10010000 se retrouve dans le registre \$t0.

Les lignes 12 et 13 chargent dans les registres \$s0 et \$s1 les valeurs se trouvant en mémoire aux adresses 0x10010000 et 0x10010001. En effet, \$t0 contient la valeur 0x10010000 et puisqu'à la ligne 12 aucun décalage n'est appliqué, la valeur à l'adresse 0x10010000 est donc lue. À la ligne 13 un décalage

d'un octet est appliqué, la valeur à l'adresse 0x10010001 est alors lue. Ces deux valeurs correspondent respectivement à 1 et à 2, la figure 18 présente l'état des données dans le programme suite à l'instruction de la ligne 13.

Données statiques		Registres	
x:	0x10010000 0x00000201	\$t0	0x10010000
r:	0x10010004 0x00000000	\$s0	0x00000001
		\$s1	0x00000002

FIGURE 18 – État des registres et des données statiques après l'exécution de la ligne 13.

La ligne 14 fait simplement l'addition des valeurs de \$s0 et \$s1 ; le registre \$t0 contient donc la valeur 3. La ligne 15, à l'instar de la ligne 11, charge la valeur de l'étiquette «r» (0x10010004) dans le registre \$t1. Finalement, la ligne 16 enregistre dans la mémoire, à l'adresse contenue dans le registre \$t1 (0x10010004), la valeur contenue dans le registre \$t0 (3). L'état des registres et de la mémoire est présenté à la figure 19.

Données statiques		Registres	
x:	0x10010000 0x00000201	\$t0	0x00000003
r:	0x10010004 0x00000003	\$t1	0x10010004
		\$s0	0x00000001
		\$s1	0x00000002

FIGURE 19 – État des registres et des données statiques après l'exécution de la ligne 16.

Finalement le programme se termine par l'instruction «syscall». Il s'agit d'une instruction particulière qui permet de faire appel à une série de fonctions fournies par le système. Ces fonctions servent, par exemple, pour la lecture et l'écriture en console. La fonction appelée par cette instruction est déterminée par la valeur contenue dans le registre \$v0. La valeur 0xA est placée dans le registre \$v0 à la ligne 18. Cette valeur correspond à la fonction «exit» qui termine l'exécution de l'application, vous trouverez régulièrement ces deux lignes à la fin des applications MIPS.

7 Conclusion

Cela termine cette introduction rapide à la programmation en assembleur MIPS. Il est fortement conseillé de faire la lecture des documents de références présentés dans le guide étudiant pour comprendre les détails du fonctionnement de l'architecture MIPS et les subtilités de la traduction du code en langage machine.

8 Référence

Le document suivant a servi de référence à l'écriture du présent document :

David Patterson, John Hennessy, " Computer Organization and Design, The Hardware/Software Interface, revised 5th Edition ", Elsevier / Morgan Kaufman, 2014, ISBN : 978-0-12-407726-3.