

Interprétation et compilation

Chapitre 2

Introduction à l'assembleur MIPS avec SPIM



Pablo Rauzy <pr@up8.edu>
`pablo.rauzy.name/teaching/ic`

Introduction à l'assembleur MIPS avec SPIM

- ▶ Philosophie de MIPS :
 - simplicité = régularité
 - bonne conception = bons compromis
 - simple = rapide

- ▶ MIPS appartient à la famille des processeurs *RISC*.
- ▶ RISC signifie *Reduced Instruction Set Computer*, par opposition à CISC, pour *Complex*.
- ▶ Cela signifie que les instructions MIPS sont toutes “simples”.
- ▶ C’est à dire qu’elles ne prennent que peu de cycles d’horloge pour s’exécuter.
- ▶ D’autres exemples de la famille RISC incluent :
 - ARM, Atmel AVR, Alpha, SPARC...

Registres

- ▶ Les processeurs MIPS sont basés sur l'utilisation de *registres*.
 - L'idée est que l'on travaille toujours avec les registres et jamais directement dans la mémoire.
 - Il y a donc des instructions spécifiques servant à lire et écrire en mémoire.
 - Il existe d'autres types d'architecture basés sur des accumulateurs (comme x86 par exemple) ou sur une pile (principalement dans les machines virtuelles).

Registres génériques

- ▶ MIPS dispose de 32 registres génériques chacun de la taille d'un mot mémoire (32 bits).
 - Il y a aussi 32 registres spécifiques aux flottants, mais on ne s'y intéressera pas. Le reste de ce cours fait généralement l'impasse sur ce qui concerne les flottants.
- ▶ Ces 32 registres sont nommés **\$0, \$1, ..., \$31**.
- ▶ Ils ont aussi chacun un nom mnémotechnique pour rendre les programmes plus lisibles :

Numéro	Nom	Rôle conventionnel
\$0	\$zero	Toujours à zéro
\$1	\$at	Réservé par l'assembleur
\$2, \$3	\$v0, \$v1	Valeurs de retours
\$4, ..., \$7	\$a0, ..., \$a3	Premiers arguments des fonctions
\$8, ..., \$15	\$t0, ..., \$t7	Registres temporaires
\$16, ..., \$23	\$s0, ..., \$s7	Registres sauvegardés
\$24, \$25	\$t8, \$t9	Registres temporaires
\$26, \$27	\$k0, \$k1	Réservés par le système
\$28	\$gp	Global pointer
\$29	\$sp	Stack pointer
\$30	\$fp	Frame pointer
\$31	\$ra	Adresse de retour

Registres particuliers

- ▶ Il y a également quelques registres particuliers auquel nous n'avons pas directement accès.
- ▶ Les trois qui nous importent sont :
 - le registre **pc** (*program counter*) qui contient l'adresse de l'instruction en cours d'exécution,
 - les registres **hi** et **lo** qui contiennent les résultats des multiplications et des divisions.

- ▶ Toutes les instructions sont codés sur 32 bits.
- ▶ Il n'y a que trois formats d'instructions.
- ▶ Cela limite le nombre d'instructions et permet un traitement rapide au niveau matériel.
- ▶ Cela simplifie aussi l'apprentissage de l'assembleur MIPS par les humains.

Cycle d'exécution

- De notre point de vue, l'exécution d'un programme MIPS consiste à répéter :
1. récupérer l'instruction stockée en mémoire à l'adresse contenu dans le registre `pc`,
 2. décoder cette instruction,
 3. exécuter cette instruction,
 4. accès mémoire (seulement pour les instructions *load* et *store* en MIPS),
 5. écriture du résultat dans les registres,
 6. mise à jour du registre `pc` (par défaut, `pc += 4`).

- Un programme écrit en assembleur MIPS est composé de :
- labels,
 - sections (ou segments),
 - directives,
 - commentaires,
 - instructions (et pseudo-instructions).

Structure d'un programme

► La structure globale d'un programme écrit en MIPS est la suivante :

```
1  .text      # Section de code
2  .globl main # déclaration de main comme global
3
4  # instructions ...
5
6  main:      # Point d'entrée
7
8  # instructions ...
9
10 .data      # Section de donnée
11
12 # déclaration de variables
```

- En fait le point d'entrée est le symbole `__start` mais par défaut celui-ci appelle `main`.
- `__start` est défini dans un autre module donc il est nécessaire de lui rendre `main` accessible avec la directive `.globl`.

Labels

- ▶ Un *label* est un nom unique donné à une adresse.
- ▶ Il est suivi de “:” là où il est défini.
- ▶ Dans une section de code, il s’agit de l’adresse de l’instruction immédiatement après le label.
- ▶ Dans une section de données, il s’agit de l’adresse de la prochaine zone mémoire réservée.

Directives

- ▶ Les directives commencent pas un “.”.
- ▶ On en a déjà vu quelques unes :
 - `.text` qui passe dans la section de code,
 - `.data` qui passe dans la section des données,
 - `.globl` qui déclare un label comme global, i.e., accessible aussi à l'extérieur du module.
- ▶ On verra quelques autres directives par la suite, notamment celles réservant de la mémoire, utilisées dans la section de données.

Section de données

- ▶ La section de données contient des déclarations servant à réserver des zones en mémoire.
- ▶ La syntaxe sera systématiquement `label: .type values`.
- ▶ `.type` est une directive qui réserve un certains nombre d'octets (dépendant du type) en mémoire, directement à la suite de où on est (c'est pour ça que le label pointerà dessus).

Types de bases

- ▶ Les types de base sont :
 - `.word` pour des valeurs sur 32 bits,
 - `.half` pour des valeurs sur 16 bits,
 - `.byte` pour des valeurs sur 8 bits,
 - `.float` pour des flottants,
 - `.double` pour des flottants double précision.
- ▶ Les valeurs sont données séparées par des virgules :
 - `answer: .word 42`
 - `hello: .byte 72, 101, 108, 108, 111, 0`

Raccourcis pour chaînes de caractères

- ▶ Il y a des types raccourcis pratiques pour les chaînes de caractères :
 - `.ascii` pour une chaîne de caractères (raccourci pour `byte`)
 - `.asciiz` pour une chaîne de caractères terminée par un `NUL`.
- ▶ Exemple :
 - `hello: .asciiz "Hello"`

Réserveur générique de mémoire

- ▶ Il y a aussi un autre type spécial `.space`, dans ce cas une seule valeur est fournie : le nombre d'octets à réserver.
- ▶ Exemple :
 - `my_struct: .space 120`

Section de code

- ▶ La section de code contient une suite d'instructions.
- ▶ Il est nécessaire qu'elle contienne un `main`.

Instructions

- Il y a 3 types d'instructions dans l'assembleur MIPS.
- les instructions R (pour "register"),
 - les instructions I (pour "immediate"),
 - les instructions J (pour "jump").

Instructions R

► Les instructions R sont de la forme : `instr rd, rs, rt`.

- `rd` est le registre destination,
- `rs` et `rt` les registres sources.

► Exemple :

- `add $t0, $t1, $t2` se lit " $t0 = t1 + t2$ ".

► Une fois assemblé en code machine, ces instructions sont de la forme :

opcode	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- En pratique pour les instructions R, l'opcode est toujours à zéro.
- L'opération est à la place donnée dans "func".
- "shamt" (*shift amount*) n'est utilisé que pour les opérations de décalage (pour des questions d'optimisations matérielles).

Instructions I

► Les instructions I sont de la forme : `instr rt, rs, imm`.

- `rt` est le registre destination,
- `rs` le registre source,
- `imm` est une valeur immédiate.

► Exemple :

- `addi $t0, $t1, 42` se lit " $t0 = t1 + 42$ ".

► Une fois assemblé en code machine, ces instructions sont de la forme :

- | opcode | rs | rt | imm |
|--------|--------|--------|---------|
| 6 bits | 5 bits | 5 bits | 16 bits |

Instructions J

- ▶ Les instructions J sont de la forme : **instr addr**.
 - **addr** est l'adresse à laquelle sauter.
- ▶ Exemple :
 - **j main** se lit "sauter à l'adresse main".
- ▶ Une fois assemblé en code machine, ces instructions sont de la forme :

- | | |
|--------|---------|
| opcode | addr |
| 6 bits | 26 bits |

Pseudo instructions

- ▶ L'assembleur MIPS comporte un certain nombre de *pseudo instructions*.
- ▶ Une pseudo instruction n'est pas directement supportée par le matériel.
- ▶ À la place elle est transformée en une, deux, ou trois instructions équivalentes et directement supportées par le matériel.
- ▶ C'est transparent pour nous (sauf en cas d'exécution étape par étape dans SPIM).

Exemples

► Quelques exemples de pseudo instructions :

- L'instruction `move`, qui copie un registre dans un autre :

```
move $t0, $t1      # $t0 = $t1
```

→ `addu $t0, $0, $t1` # `$t0 = 0 + $t1.`

- L'instruction `neg`, qui donne l'opposé d'un nombre :

```
neg $t0, $t1       # $t0 = -$t1
```

→ `subu $t0, $0, $t1` # `$t0 = 0 - $t1.`

- L'instruction `li` (*load immediate*), qui charge une valeur dans un registre :

```
li $t0, 42         # $t0 = 42
```

→ `ori $t0, $0, 42` # `$t0 = 0 | 42.`

- L'instruction `li`, mais avec une valeur sur 32 bits :

```
li $t0, 147483748  # $t0 = 147483748
```

→ `lui $at, 2250` # `$at = 2250 << 16`

```
ori $t0, $at, 27748 # $t0 = $at | 27748.
```


Traductions automatiques

- ▶ L'assembleur de SPIM permet de nombreuses facilités.
- ▶ Outre les pseudo instructions, il transforme aussi certaines instructions quand c'est nécessaire.
- ▶ Exemple :
 - L'instruction `add` prend normalement trois registres :
`add $t0, $t1, 100`
 - `addi $t0, $t1, 100`

Références

- ▶ De nombreuses références des instructions MIPS supportées par SPIM sont déjà disponibles sur le web :
 - http://www-soc.lip6.fr/~marchett/Archi_Memento_MIPS-nup.pdf
 - https://en.wikibooks.org/wiki/MIPS_Assembly
 - ...
- ▶ En cas de doute, la meilleure documentation pour comprendre le comportement d'une instruction est de l'exécuter en mode pas à pas sur quelques cas différents dans SPIM.

- ▶ Notre machine cible est le simulateur SPIM.
- ▶ SPIM est capable d'exécuter un binaire assemblé pour l'architecture MIPS ou directement du code assembleur MIPS.
- ▶ SPIM peut servir de débbugger (exécution pas à pas et visualisation du contenu des registres).

Interface

- ▶ La commande **spim** peut prendre plusieurs arguments.
- ▶ Celui qui nous intéresse principalement est **-file** qui permet de spécifier un fichier contenant du code assembleur à exécuter.
- ▶ Voir **man spim** pour le reste.

Commandes de base

- Une fois lancé, SPIM offre quelques commandes dont :
- **read** (ou **load**) qui permet de charger un fichier de code assembleur en mémoire.
 - **run** qui permet de lancer l'exécution, en sautant à l'étiquette **main** par défaut.
 - **breakpoint** qui permet de mettre des breakpoints.
 - **step** qui permet d'avancer étape par étape dans l'exécution.
 - **continue** qui permet de reprendre l'exécution.
 - **print** qui permet d'afficher le contenu de registre ou de la mémoire.
 - **reinitialize** qui permet de réinitialiser le simulateur.
 - **exit** (ou **quit**) pour quitter.

Appels systèmes

► SPIM supporte 16 appels systèmes :

code	fonction	argument(s)	résultat
\$v0 = 1	print_int	\$a0	
\$v0 = 2	print_float	\$f12	
\$v0 = 3	print_double	\$f12	
\$v0 = 4	print_string	\$a0	
\$v0 = 5	read_int		\$v0
\$v0 = 6	read_float		\$f0
\$v0 = 7	read_double		\$f0
\$v0 = 8	read_string	\$a0 (buffer) \$a1 (taille)	
\$v0 = 9	sbrk	\$a0 (taille)	\$v0 (adresse)
\$v0 = 10	exit		
\$v0 = 11	print_char	\$a0	
\$v0 = 12	read_char		\$v0
\$v0 = 13	open	\$a0 (fichier) \$a1 (flags) \$a2 (mode)	\$v0 (fd)
\$v0 = 14	read	\$a0 (fd) \$a1 (buffer) \$a2 (taille)	\$v0 (taille)
\$v0 = 15	write	\$a0 (fd) \$a1 (buffer) \$a2 (taille)	\$v0 (taille)
\$v0 = 16	close	\$a0 (fd)	\$v0 (success)