Généralités MIPS

Syntaxe

- Les commentaires commencent par le symbole # et se terminent à la fin de la ligne.
- Un identificateur est une séquence de caractères alphanumériques, de soulignés (_) et de points (.), qui ne commence pas par un chiffre.
- Les codes opération d'instruction sont des mots réservés qui ne peuvent pas être utilisés comme identificateurs.
- Les étiquettes sont déclarées en les plaçant au début d'une ligne et en les faisant suivre du symbole (:).
- Les nombres sont en base **10** par défaut. S'ils sont précédés de **0x** ils sont interprétés comme hexadécimaux.
- Les chaînes de caractères sont encadrées par des doubles apostrophes (").
- Certains caractères spéciaux dans les chaînes de caractères suivent la convention **C** :
- Retour-chariot : **\n**
- Tabulation : **\t**
- Guillemet : **\"**

Quelques directives

.ascii str	Enregistre en mémoire la chaîne de caractères str, mais ne la termine pas par un caractère nul.
.asciiz str	Enregistre en mémoire la chaîne de caractères **str** et la termine par un caractère nul.
.data<@>	Les éléments qui suivent sont enregistrés dans le segment de données. Si l'argument optionnel **@** est présent, les éléments qui suivent sont enregistrés à partir de l'adresse **@**.
.byte b1; : : : ;bn	Enregistre les **n** valeurs dans des octets consécutifs en mémoire.
.word w1; : : : ;wn	Enregistre les **n** quantités **32 bits** dans des mots consécutifs en mémoire.
.float f1; : : : ;fn	Enregistre les **n** nombres flottants simples précision dans des emplacements mémoire consécutifs.
.text <@>	Les éléments qui suivent sont placés dans le segment de texte de l'utilisateur. Dans **SPIM**, ces éléments ne peuvent être que des instructions ou des mots. Si l'argument optionnel **@** est présent, les éléments qui suivent sont enregistrés à partir de l'adresse **@**.
.globl sym	Déclare que le symbole **sym** est global et que l'on peut y faire référence à partir d'autres fichiers.

Les registres

Il existe **32** registres de **32 bits** numérotés **$0; : : : ; $31**. Les registres peuvent être accédé soit par leur numéro soit par leur nom.

Nom	Numéro du registre	Description
$zero	0	Constante 0
$at	1	Réservé à l'assembleur
$v0,$v1	2-3	Évaluation d'une expression et résultats d'une fonction
$a0,: : :,$a3	4-7	Arguments de sous-programmes
$t0,: : :,$t7	8-15	Valeurs temporaires (non préservées)
$s0,: : :,$s7	16-23	Valeurs temporaires (préservées)
$t8,$t9	24-25	Valeurs temporaires (non préservées)
$k0,$k1	26-27	Réservé pour les interruptions (i.e., système d'exploitation)
$gp	28	Pointeur global
$sp	29	Pointeur de pile
$fp	30	Pointeur de bloc
$ra	31	Adresse de retour

Appel de procédure - Conventions

- Par convention, lors de l'appel de procédure, les registres **$t0,: : :,$t9** sont sauvegardés par l'appelant et peuvent donc être utilisés sans problème par l'appelé. Les registres **$s0,: : :,$s7** doivent quant à eux être sauvegardés et restitués exact par l'appelé.
- La pile croit des adresses hautes vers les adresses basses : on soustrait à **$sp** pour allouer de l'espace dans la pile, on ajoute à **$sp** pour rendre de l'espace dans la pile.
- Les déplacements dans la pile se font sur des mots mémoire entiers (multiples de quatre octets).
- Lors du passage de passage de paramètres : tout paramètre plus petit que **32 bits** est automatiquement promu sur **32 bits**.
- Les quatre premiers paramètres sont passés par les registres **$a0,: : :,$a3**. Les paramètres supplémentaires sont passés dans la pile.
- Toute valeur de format inférieur ou égal à **32 bits** est retournée par le registre **$v0** (sur **64 bits $v1** est utilisé avec **$v0**).

Présentation du Simulateur MIPS (Interface QtSpim)

QtSpim est logiciel de simulateur qui exécute des programmes en assembleur **MIPS**.

L'interface **QtSpim** contient :

1. Une partie pour les registres **«Registers»**, qui affiche le contenu de tous les registres entiers
2. Un segment de texte **«Text Segment»**, qui affiche les instructions MIPS chargées en mémoire pour être exécutées. De gauche à droite, l'adresse mémoire de l'instruction, l'instruction en hexadécimal, les instructions MIPS réelles, l'assembleur MIPS que vous avez écrit ainsi que les commentaires.
3. Un segment de texte **«Data Segment»**, qui affiche les données et leurs valeurs dans les segments de données et la pile en mémoire.
4. La **« Console »** information répertorie les actions effectuées par le simulateur.

Pour une meilleure lisibilité, décochez tout ce qui correspond à *Kernel* dans les menus *Text Segment* et *Data Segment*, et décochez *FP Registers* dans le menu *Window*. Par un clic droit sur un registre ou une adresse mémoire, vous pouvez modifier son contenu dynamiquement.

Comment exécuter un programme MIPS dans QtSpim ?

Pour exécuter le programme dans **QtSpim**, nous devons suivre les étapes suivantes :

1. Utiliser un éditeur de texte pour créer un programme (par exemple Bloc-notes, Sublime, NotePad,…)
2. Le fichier contenant le programme doit avoir l'extension (le suffixe) ''**.s**''.
3. Utiliser le menu ***File--->Reinitialize and Load File*** pour charger le programme en mémoire. Le simulateur inclut un programme d'assemblage, qui effectue l'encodage en binaire et la traduction des pseudo-instructions et des labels. Il vérifie la correction syntaxique du programme. Si le programme est incorrect, une fenêtre signalant la première erreur s'affiche et le programme ne peut pas être chargé.
4. Exécuter le programme, suivant l'une des deux méthodes suivantes :
   a) **Run** (ou touche **F5**) - toutes les instructions seront exécutées
   b) **Step** (ou touche **F10**) - exécution pas à pas.
5. Pour modifier le programme, il faut :
   a) Revenir dans un éditeur de texte,
   b) Effectuer les modifications,
   c) Recharger le programme.

La structure de base d'un programme MIPS

- La section « **.data** », contient les déclarations de données, c.-à-d. les données globales manipulées par le programme utilisateur. Elle est implantée conventionnellement à l'adresse 0x10000000. Sa taille est fixe et calculée lors de l'assemblage. Les valeurs contenues dans cette section peuvent être initialisées grâce à des directives contenues dans le programme source en langage d'assemblage ;
- La section « **.text** », (code du programme) contient le code exécutable en mode utilisateur. Elle est implantée conventionnellement à l'adresse 0x00400000. Sa taille est fixe et calculée lors de

l'assemblage. La principale tâche de l'assembleur consiste à générer le code binaire correspondant au programme source décrit en langage d'assemblage, qui sera chargé dans cette section ;

- « **main** », début du programme.
- Pour sortir du programme **MIPS**, on fait un appel system « **li $v0,10 syscall** ».
- Les commentaires permettent de donner plus d'explication sur le code. Ils commencent par un « **#** » ou un « **;** ».

***Exemple* d'un simple programme MIPS**

```
1   # Exemple0
2   .data
3   # Il n'y aura pas de données à déclarer, on utilisera le mode immédiat
4   .text
5   main:
6   li $t0,3 # charger la valeur 3 dans le registre t0
7   li $t1,8 # charger la valeur 8 dans le registre t1
8   add $t2,$t1,$t0 # Faire l'addition de t0+t1 et mettre le résultat dans le registre t2
9   #Fin du programme
10  li $v0,10
11  syscall
```

Déclarations des données

Les données (constantes et variables) doivent être déclarées dans la section « **.data** ».

Les données doivent commencer par une lettre et peuvent contenir des lettres, des chiffres et/ou des caractères spéciaux.

Le format général de la déclaration d'une donnée est le suivant :

<Nom de la variable> <type de la donnée> <valeur initiale>

Exemple de déclaration de données

```
1   .data
2   C    .byte   'a'          # Octet
3   N1   .half   26           # 2 octets
4   N2   .word   353          #4 octets
5   Tap      .space 40        # réservation d'un espace de 40 octets
6   Mess .asciiz    "une chaine"    # déclaration d'une chaîne de caractères terminée par 0
```

Les données en Mips sont de différents types :

Déclaration	Description
.byte	Variables à 8 bits
.half	Variables à 16 bits
.word	Variables à 32 bits
.ascii	Chaine ASCII
.asciiz	Chaine ASCII terminée par un zéro
.float	Nombre réel à 32 bits
.double	Nombre réel à 64 bits
.space <n>	Espace mémoire à n bytes non initialisé

Les instructions du MIPS

Le MIPS propose trois types principaux d'instructions :

- Les instructions de **transfert** entre registres et mémoire ;
- Les instructions de **calcul** ;
- Les instructions de **saut**.

Seules les premières permettent d'accéder à la mémoire ; les autres opèrent uniquement sur les registres.

1. Instruction de transfert

- **Lecture** (load word):

lw dest, offset(base)

On ajoute la constante (de 16 bits) *offset* à l'adresse contenue dans le registre *base* pour obtenir une nouvelle adresse ; le mot stocké à cette adresse est alors transféré vers le registre *dest*.

On a également: **lb** (load byte)**, lbu** (load byte unisgned)**, lh** (load half)**, lhu** (load half unisgned)**.**

- **Ecriture** (store word):

sw source , offset(base)

On ajoute la constante (de 16 bits) *offset* à l'adresse contenue dans le registre *base* pour obtenir une nouvelle adresse ; le mot stocké dans le registre *source* est alors transféré vers cette adresse.

On a également: **sb** (store byte)**, sh** (store half)**.**

2. Instruction de calcul

Ces instructions lisent la valeur de 0, 1 ou 2 registres dits **arguments**, effectuent un calcul, puis écrivent le résultat dans un registre dit **destination**.

Un même registre peut figurer plusieurs fois parmi les arguments et destination.

- Les instructions de calcul nullaires

- **Ecriture d'une constante** (Load Immediate):

li dest, constant

Produit la constante *constant*.

On a également : **la** (load adrdess)**.**

- Les instructions de calcul unaires

- **Addition d'une constante** (Add Immediate):

addi dest, source, constant

Produit la somme de la constante (de 16 bits) *constant* et du contenu du registre *source*.

- **Déplacement** (Move):

move dest, source

Produit le contenu du registre *source*. Cas particulier de *addi!*

- **Négation** (Negate):

neg dest, source

Produit l'opposé du contenu du registre *source.* Cas particulier de *sub!*

- Les instructions de calcul binaires

- **Addition** (Add):

add dest, source1, source2

Produit la somme des contenus des registres *source1* et *source2*.

On a également : **sub, mul, div.**

- **Comparaison** (Set On Less Than):

slt dest, source1, source2

Produit 1 si le contenu du registre *source1* est inférieur à celui du registre *source2* ; produit 0 sinon.

On a également : **sle, sgt, sge, seq, sne**.

3. Instruction de saut

On distingue les instructions de saut selon que :

- Leurs destinations possibles sont au nombre de 1 (saut **inconditionnel)** ou bien 2 (saut **conditionnel**);
- Leur adresse de destination est **constante** ou bien **lue** dans un registre ;
- Une **adresse de retour** est sauvegarde´ e ou non.

- Saut inconditionnel

- **Saut** (Jump):

j address

Saute à l'adresse constante *address*. Celle-ci est en général donnée sous forme symbolique par une *étiquette* que l'assembleur traduira en une constante numérique.

- Saut conditionnel

- **Saut conditionnel unaire** (Branch on Greater Than Zero):

bgtz source, address

Si le contenu du registre *source* est supérieur à zéro, saute à l'adresse constante *address*.

On a également **bgez, blez, bltz.**

- **Saut conditionnel binaire** (Branch On Equal):

beq source1, source2, address

Si les contenus des registres *source1* et *source2* sont égaux, saute à l'adresse constante *address*.

On a également **bne.**

- Saut avec retour

- **Saut avec retour** (Jump And Link):

jal address

Sauvegarde l'adresse de l'instruction suivante dans le registre *ra*, puis saute à l'adresse constante *address.*

- Saut vers adresse variable

- **Saut vers adresse variable** (Jump Register)

jr target

Saute à l'adresse contenue dans le registre *target*. L'instruction **jr $ra** est typiquement employée pour rendre la main à l'appelant à la fin d'une fonction ou procédure.

4. Instruction spéciale

- **Appel système** (System Call):

  syscall

Provoque un appel au noyau. Par convention, la nature du service souhaité est spécifiée par un code entier stocké dans le registre **v0**. Des arguments supplémentaires peuvent être passés dans les registres **a0-a3**. Un résultat peut être renvoyé dans le registre **v0**.

Les appels système

- Un appel système se fait par l'instruction **syscall**.
- Les simulateurs fournissent un ensemble de services par l'intermédiaire de l'instruction d'appel **syscall**, dont le comportement dépend de la valeur du registre **$v0** :
  - si **$v0 = 1**, alors **syscall** affiche l'entier contenu dans le registre **$a0** ;
  - si **$v0 = 4**, alors **syscall** affiche la chaîne de caractère dont l'adresse est contenue dans le registre **$a0** ;
  - si **$v0 = 5**, alors **syscall** lit un entier à l'écran et met sa valeur dans le registre **$v0**.
- Pour demander un service, on charge le code du service (voir tableau ci-dessous) dans le registre **$v0** et ses arguments dans les registres **$a0,...,$a3** (ou **$f12** pour les valeurs flottantes).
- Les appels système qui retournent des valeurs placent leurs résultats dans le registre **$v0** (ou **$f0** pour les résultats flottants).

Le tableau suivant illustre les différents codes de **syscall**.

Service	Code	Arguments	Résultat
print_int	1	$a0 = entier	
print_float	2	$f12 = flottant simple précision	
print_double	3	$f12 = flottant double précision	
print_string	4	$a0 = chaine de caractères	
read_int	5		Un entier dans $v0
read_float	6		Un flottant simple dans $v0
read_double	7		Un flottant double dans $v0
read_string	8	$a0 = tampon, $a1 = longeur	
sbrk	9	$a0 = quantité	Une adresse dans $v0
exit	10		

Exemple 1 : print_int (Appel système code $v0=1)

Le code suivant imprime « **La réponse = 5** » dans la console.

```
1   .data
2   Str : .asciiz    "La reponse = "
3   .text
4   main:
5   la $a0, Str      # Mettre l'adresse de la chaine de caractère Str dans le registre $a0
6   li $v0, 4        # Charger le registre $v0 avec le code 4 pour affichage caractère (print_string)
7   syscall          # appel système pour affichage caractère
8   li $a0, 5        # Mettre la valeur à afficher dans le registre $a0
9   li $v0, 1        # Charger le registre $v0 avec le code 1 pour affichage entier (print_int)
10  syscall          # appel system pour l'opération affichage entier
11  # Fin du Programme
12  li $v0, 10
13  syscall
```

Exemple2 : print_string (Appel système code $v0 =4)

Le code suivant affiche le message « **Hello Word** » dans la console.

```
.data
Hello: .asciiz "Hello Word\n" # mettre la chaine de caractère « Hello Word » en mémoire
.text
main:           # section <code
la $a0, Hello   # charger le registre $a0 avec l'adresse de la chaine de caractère
li $v0, 4       # Charger le registre $v0 avec le code 4 pour affichage caractère
                #(print_string).
syscall         # appel system pour l'opération d'affichage
# Fin du Programme
li $v0,10
syscall
```

Exemple3 : read_int (Appel système code $v0=5)

Le code suivant affiche sur l'écran un entier saisi au le clavier.

```
.data
Str : .asciiz " Saisir un entier "
.text
main:
la $a0,Str  # Mettre l'adresse de la chaine de caractère Str dans le registre $a0
li $v0,4    # Charger le registre $v0 avec le code 4 pour affichage caractère (print_string)
syscall     # Appel système pour affichage caractère
li $v0,5    # Charger le registre $v0 avec le code 5 pour saisir un entier à partir du clavier
syscall     # Appel system pour l'opération lire entier (read_int)
move $a0,$v0 # déplacer la valeur saisie dans le register $a0
li $v0,1    # Charger le registre $v0 avec le code 1 pour affichage entier (print_int)
syscall     # Appel system pour l'opération affichage entier
# Fin du Programme
li $v0,10
syscall
```

Exemple4 : read_string (Appel système code $v0=8)

Le code suivant affiche le message écrit par l'utilisateur.

```
.data
message: .asciiz "Vous avez écrit "
UserInput: .space 20
.text
main:
# Entrer un text de 20 caractères
li $v0, 8
la $a0, UserInput
li $a1, 20
syscall
# Afficher le message « Vous avez écrit »
li $v0,4
la $a0,message
syscall
# Afficher le message
li $v0,4
la $a0,UserInput
syscall
# Fin du programme
li $v0,10
syscall
```