Architecture des ordinateurs : TP2

Université de Tours

`Département informatique de Blois`

Assembleur MIPS

*
* *

Dans ce TP, on va apprendre les rudiments du langage machine à l'aide du jeu d'instructions de l'assembleur MIPS. L'objectif n'étant pas de devenir un professionnel de l'assembleur, mais plutôt de découvrir qu'une instruction qui peut sembler aussi rudimentaire d'un `for` en Java est en fait loin d'être trivial dans les couches basses.

P.S : Le MIPS était aussi le langage qui servait à programmer la PlayStation 2. `;)`

1 Pré-requis d'installation

Pour commencer, il va vous falloir un environnement de développement pour coder en MIPS.
Rendez-vous sur le site suivant :
`http://courses.missouristate.edu/KenVollmar/mars/download.htm`
afin de télécharger l'IDE Mars. Il s'agit d'un fichier .jar donc qui est indépendant de votre système d'exploitation. Exécutez-le.
Ceci fait, téléchargez sur Celene le dossier methodes_assembleur.zip, il contient le squelette des différentes méthodes assembleur que vous allez devoir réaliser pour ce TP.
Pour ouvrir un fichier .asm, il suffit d'aller dans l'onglet File -> Open, puis de sélectionner le fichier désiré.
Pour tester/compiler votre programme, vous devez au préalable l'assembler en cliquant sur l'icône  , puis l'exécuter en cliquant sur l'icône  .

2 Les instructions MIPS

Au sujet du jeu d'insctructions MIPS, vous pouvez reprendre les exemples du cours *Chapitre 3 : Processeur et jeu d'instructions*. On détaille néanmoins ici les commandes les plus utilisées.

- Chargement immédiat - `li` :

  `li $r0, n` effectue ($\$r_0 \leftarrow n$), c'est-à-dire le chargement de la constante n dans le registre `$r0`, où `n` est un entier ou un caractère.

- Lecture mémoire - `lb (byte)` et `lw (word)` :

  `lb $r0, n($r1)` effectue ($\$r_0 \leftarrow RAM[\$r_1 + n]$), c'est-à-dire la lecture mémoire de l'octet contenu à l'adresse du registre `$r1 + n`, où `n` est un entier relatif (On peut écrire `lb, $r0, ($r1)` si $n = 0$).

- Écriture mémoire - `sb (byte)` et `sw (word)` :

  `sb $r0, n($r1)` effectue ($RAM[\$r_1 + n] \leftarrow \$r_0$), c'est-à-dire l'écriture mémoire de l'octet contenu à l'adresse du registre `$r1 + n`, où `n` est un entier relatif (On peut écrire `sb, $r0, ($r1)` si $n = 0$).

- Affectation - `move`

  `move $r0, $r1` effectue ($\$r_0 \leftarrow \$r_1$), c'est-à-dire la recopie du contenu du registre `$r1` vers le registre `$r0`.

- Opérations de calcul - logiques, arithmétiques et décalages :

  Ces opérations possèdent toutes les deux signatures suivantes :

  `Op` $\$r_1, \$r_2, \$r_3$ telle que $\$r_1 \leftarrow Op(\$r_2, \$r_3)$, et

  `Op` $\$r_1, \$r_2, n$ telle que $\$r_1 \leftarrow Op(\$r_2, n)$.

  - Logiques - `and, or, xor`
  - Décalage - `sll` (décalage à gauche $\ll$), `srl` (décalage à droite $\gg$)
  - Arithmétique - `add, sub, mul, div`

- Les instructions de saut ou de branchement :

  - Saut inconditionnel - `j` :

    `j label` permet de faire pointer le compteur ordinal vers l'instruction où se trouve l'adresse du `label`.

  - Branchement conditionnel

    - `beq $r0, $r1, label` permet de faire pointer le compteur ordinal vers l'instruction où se trouve l'adresse du `label` si et seulement si $\$r_0 = \$r_1$, sinon, le programme se poursuit séquentiellement. (`bne` pour $\$r_0 \neq \$r_1$).
    - `blt $r0, $r1 label` permet de faire pointer le compteur ordinal vers l'instruction où se trouve l'adresse du `label` si et seulement si $\$r_0 < \$r_1$, sinon, le programme se poursuit séquentiellement. (`ble` est utilisée pour la relation $\leq$).
    - `bgt $r0, $r1 label` permet de faire pointer le compteur ordinal vers l'instruction où se trouve l'adresse du `label` si et seulement si $\$r_0 > \$r_1$, sinon, le programme se poursuit séquentiellement. (`bge` est utilisée pour la relation $\geq$).

    Ces opérations sont aussi valables pour la signature `Op $r0, n, label`. Avec `Op` $\in \{$`beq, bne, blt, ble, bgt, bge`$\}$. Dans ce cas, $\$r_1$ est substitué par n. La sémantique de l'opération de branchement est conservée.

- Les appels système :

  Pour imprimer quelque chose dans la console, ***vous devez au préalable le stocker dans le registre*** `$a0`, puis appeler la méthode correspondante à l'aide des commandes :

  ```
  li $v0, n
  syscall
  ```

  La valeur `n` sert à spécifier le code service de l'appel à la primitive `syscall`.

  On peut résumer les principaux services rendus par le tableau suivant :

Service	Code Service	Arguments	Valeur de retour
Afficher un entier	1	$a0: Adresse l'entier à afficher	
Afficher un flottant	2	$a0: Adresse du flottant à afficher	
Afficher une ch. de car.	4	$a0: Adresse du début de la ch.	
Lire un entier	5		$v0: Entier lu
Lire une ch. de car.	8	$a0: Adresse du tampon	
		$a1: Nombre max de car. lus	
Sortie programme (exit)	10		

Pour vous facilier la tâche (et le code), les appels système les plus importants (String et int) sont déjà définis. Ainsi, pour afficher une valeur numérique il vous suffit de taper `jal print_int` ou `jal print_string` pour afficher une chaine.

J'insiste encore ici sur le fait que la valeur à afficher doit être stockée dans le registre `$a0` !

Les curieux pourront retrouver l'ensemble des mnémoniques à leur disposition sur le document commandes.pdf disponible sur Celene.

3 Un exemple pour commencer

On définit ci-dessous une fonction `affparite`. Elle prend un paramètre $n \in \mathbb{Z}$ qu'on suppose contenu dans le registre $t0, et on affiche sa parité.

```
##
# @author Clément Moreau
# File_name : parite.asm
# Description : Affiche la parité d'un nombre n contenu dans le registre $t0.
##
##### Data section #####
.data
chaine_impair : .asciiz " est impair"
chaine_pair   : .asciiz " est pair"
#### Text section #####
.text
.globl _main_
# Début du programme
_main_ :
   li $t0, 42              # Chargement du nombre n
   move $a0, $t0           # On déplace n dans le registre d'impression
   jal print_int           # On imprime n grâce à la routine print_int
   and $t0, $t0, 1         # On calcule la parité de n : n&1 <=> n%2
   beqz $t0, pair          # $t0 == 0 ? branchement à pair : sinon continue
   la $a0, chaine_impair   # $t0 est impair, on charge la chaine_impair à imprimer
   j fin                   # On saute à la fin
pair :
   la $a0, chaine_pair     # $t0 est pair, on charge la chaine_pair à imprimer
# Sorti du programme
```

```
fin :
   jal print_string        # On imprime la chaine contenu dans $a0
   li $v0, 10              # On quitte le programme
   syscall

## Routines d'impression ##
# La routine imprime le contenu du registre $a0
##
print_int :
   li $v0, 1
   syscall
   jr $ra
print_string :
   li $v0, 4
   syscall
   jr $ra
```

4 Questions

Il est important de rappeler que l'assembleur est très différent des langages haut niveau (comme Java) que vous avez l'habitude d'utiliser. Ainsi, il peut-être raisonnable de réfléchir au préalable sur papier à la forme linéaire de l'algorithme à implémenter.
Dans la suite, les fichiers mentionnés se trouvent dans le dossier methodes_assembleur.zip à télécharger sur la page CELENE du cours.

1. Compléter le fichier affSigne.asm qui permet d'afficher le signe de l'entier relatif n contenu dans le registre `$t0`.

   Ex : Si $n = 0$, on affichera NUL, si $n > 0$, on affichera POSITIF, si $n < 0$, on affichera NEGATIF.

2. Compléter le fichier affFormule.asm qui permet d'afficher le résultat de somme des n premiers entiers, soit $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. On considèrera que n est contenu dans le registre `$t0`.
   Ex : Si $n = 100$, on affichera 5050. Si $n = 42$, on affichera 861.

3. Compléter le fichier affDist2.asm qui affiche la distance euclidienne au carrée $(x_b - x_a)^2 + (y_b - y_a)^2$ entre deux points $A = (x_a, y_a)$ et $B = (x_b, y_b)$. On considèrera que les registres `$t0`, `$t1`, `$t2`, `$t3` contiennent respectivement les données x_a, y_a, x_b et y_b.

   Ex : Si $x_a = 0, y_a = 0, x_b = 1$ et $y_b = 1$, on affichera 2.

4. Compléter le fichier min_max.asm qui affiche le min et le max d'un ensemble E de trois nombres $E = \{a, b, c\}$. On considèrera que les registres `$t0`, `$t1`, `$t2` contiennent respectivement a, b et c. Le min sera placé dans le registre `$a2` et le max dans le registre `$a1`.

   De plus, on supposera que l'on dispose ici de la routine d'impression `print_min_max` qui imprime le contenu des registres `$a2` et `$a1` de la forme "min = `$a2` et max = `$a1`".

   Ex : Si $E = \{7, -3, 10\}$, on affichera min $= -3$ et max $= 10$.

5. Compléter le fichier checksum.asm qui encode le contenu n du registre $t0 selon la méthode de codage φ par contrôle de parité (i.e. checksum) et imprime l'interprétation décimale correspondant à $\varphi(n)$.

   On note que n peut être un nombre ou un caractère. On rappelle que par défaut, les registres ont une capacité de 32 bits.

   Ex : Si $n = 12$, on affichera 24 car $n = \langle 1100 \rangle_2$ et $\varphi(12) = \langle 11000 \rangle_2 = 24$.

   Si $n =$ 'a', on affichera 195, car 'a' en ASCII se code tel que 'a' $= \langle 1100001 \rangle_2$ et φ('a') $= \langle 11000011 \rangle_2 = 2^7 + 2^6 + 2^1 + 1 = 128 + 64 + 2 + 1 = 195$

6. Compléter le fichier parite_lt.asm qui, selon le contenu d'un tableau TAB donné au préalable, calcule un tableau TAB_CODE selon la méthode de codage Φ par parité longitudinale et transversale.

   On rappelle que :

$$\texttt{TAB\_CODE} = \Phi(\texttt{TAB}) = \begin{pmatrix} \varphi(\texttt{TAB}[0]) \\ \vdots \\ \varphi(\texttt{TAB}[i]) \\ \vdots \\ \varphi(\texttt{TAB}[n-1]) \\ \bigoplus_{k=0}^{n-1} \varphi(\texttt{TAB}[k]) \end{pmatrix}$$

   Où φ désigne le codage par contrôle de parité de la question 5.

   On supposera que l'on dispose ici de la routine d'impression `print_tab` qui imprime le contenu d'un tableau dont le registre $a0 contient l'adresse et le registre $a1 sa taille.

   Ex : Si TAB $= [12, \text{'a'}, \text{'Z'}, 3, 1]$, alors sous forme matricielle on a :

$$\texttt{TAB} = \begin{pmatrix} \langle 12 \rangle_2 \\ \langle \text{'a'} \rangle_2 \\ \langle \text{'Z'} \rangle_2 \\ \langle 3 \rangle_2 \\ \langle 1 \rangle_2 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

   Et donc TAB_CODE $= \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 24 \\ 195 \\ 180 \\ 6 \\ 3 \\ 106 \end{pmatrix}$.

   On affichera 24 195 180 6 3 106

7. Compléter le fichier modifier.asm qui remplace tout caractère c_1 d'un message MESSAGE par le caractère c_2. On considèrera que c_1 est contenu dans le registre $t8 et c_2 est contenu dans le registre $t9.

   Ex : Si MESSAGE $=$ "Hello world !", $c_1 =$ ' ' et $c_1 =$ '-', on affichera "Hello-world- !".

8. Compléter le fichier capitaliser.asm qui remplace chaque minuscule (ASCII sans accent, entre 'a' et 'z') d'un message `MESSAGE` par la majuscule correspondante.

   Ex : Si `MESSAGE` = "{Hello world !}", on affichera "{HELLO WORLD !}".

9. Compléter le fichier fibo.asm qui remplit un tableau d'entiers `FIBS` défini au préalable de longueur $n > 1$ selon la suite de Fibonacci puis l'affiche. Ainsi `FIBS`$[i] = \mathcal{F}_i$. Où $\mathcal{F}_i$ désigne la suite de Fibonacci au rang i. On considèrera que n est contenu dans le registre `$t9`.

   On rappelle que la suite de Fibonacci $(\mathcal{F}_n)_{n\in\mathbb{N}}$ est définie telle :

$$\begin{cases} \mathcal{F}_0 & = 0 \\ \mathcal{F}_1 & = 1 \\ \mathcal{F}_{n+2} & = \mathcal{F}_{n+1} + \mathcal{F}_n \end{cases}$$

   Ex : Si $n = 10$, on affichera 0 1 1 2 3 5 8 13 21 34 55.

10. Compléter le fichier erathostene.asm qui exécute le crible d'Erathostene jusqu'à l'entier n et affiche l'ensemble des nombres premiers inférieurs à n. On considèrera que n est contenu dans le registre `$t9` et qu'un tableau `TAB` est défini au préalable de longueur n.

    Ex : Si $n = 42$, on affichera 2 3 5 7 11 13 17 19 23 29 31 37 41.