TD3 : Premiers programmes assembleur

Objectif(s)

- ✯ Familiarisation avec la structure d'un programme assembleur MIPS
- ✯ Familiarisation avec les instructions arithmétiques et logiques MIPS et les appels système
- ✯ Utilisation du mémento MIPS
- ✯ Codage et décodage d'instructions
- ✯ À la fin de cette semaine, sont considérées comme acquises les notions ci-dessus et la maîtrise de votre mémento MIPS

Note pédagogique:
Rappels : les registres `$16` à `$23` sont persistants, donc leur valeur est préservée en cas d'appel système ou d'appel de fonction. La conséquence est qu'ils sont à sauvegarder sur la pile (dans le prologue des fonctions) pour garantir la préservation de leur contenu au travers des appels de fonction. A priori, il faudrait aussi le faire dans le main, qui est aussi une fonction. On ne le fera pas car on terminera les programmes principaux par un appel système de terminaison.

On prendra l'habitude dans les semaines 3 à 6 de préférer l'usage des registres non persistants `$8` à `$15` dans les codes des programmes principaux à donner. Toutefois, il faut savoir que les appels système peuvent modifier les registres non persistants (par définition de non persistance). Ainsi, en cas de valeur à conserver dans un registre au travers d'un appel système (et de fonction dans les semaines 7 et plus), il est important d'utiliser un registre persistant (`$16` à `$23`) ou de ne pas oublier que la valeur contenuee dans un registre non persistant n'est plus celle d'avant appel système et donc qu'il faut la recalculer/recharger.

Exercice(s)

Exercice 1 – Structure d'un programme assembleur

Question 1

1. Rappelez les sections qui composent un programme assembleur et expliquez ce que l'on trouve dans ces sections.
2. Quel est le point d'entrée d'un programme ?
3. Que doit contenir au minimum un programme ?

Solution:
Il y a deux types de section dans un programme assembleur.

1. La section `.data` contient les déclarations de variables globales du programme. Nous prendrons l'habitude de toujours la mettre tout en haut d'un fichier assembleur (pas de mélange data/text dans ce cours). Elle sera vide cette semaine. La section `.text` contient les instructions du programme. Il y a dans tous les cas un programme principal, il peut aussi y avoir des fonctions.
2. Le point d'entrée du programme correspond à la première instruction de la section de code.

3. Tout programme doit se terminer par un retour à l'environnement de lancement. Ce retour est réalisé au moyen d'un appel système équivalent à `exit`. Un programme minimal (c'est-à-dire qui ne fait rien) contient donc uniquement cet appel système.

Ce qui donne la structure suivante :

```
.data
   # zone de déclaration de variables globales/mots mémoire

.text
   # section contenant les instructions = programme principal + fonctions
   # debut du programme principal
   ...
   ori $2, $0, 10
   syscall             # appel système de fin de programme
```

Exercice 2 – Jeu d'instructions MIPS : instructions arithmétiques et logiques

Question 1

Parcourez sur votre mémento toutes les opérations arithmétiques (opérations sur les entiers) du jeu d'instructions MIPS. Notez, pour chacune, son code opération ainsi que le nombre de ses opérandes sources et résultats, et la nature de ces opérandes ainsi que le format de codage. En déduire une réponse aux questions suivantes :

— Combien d'opérandes ont ces instructions ?
— Quelles sont les exceptions ?
— Déduisez-en une relation entre le type des opérandes sources et le format de codage.

Solution:

Les instructions arithmétiques sont : `add`, `sub`, `addu`, `subu`, `addi`, `addiu`, `mult`, `multu`, `div`, `divu`. On peut aussi donner les instructions de comparaison : `slt`, `sltu`, `slti`, `sltiu`.

Hormis les opérations de multiplication et de division, toutes les instructions arithmétiques ont deux opérandes sources et produisent un opérande résultat, ces opérandes étant spécifiés dans l'instruction. L'opérande résultat est toujours un registre. Les opérandes sources sont soit deux registres et le format de codage est R, soit un immédiat et un registre et le format de codage est I.

La multiplication de deux entiers sur 32 bits donne potentiellement un entier sur 64 bits, il faut donc deux registres pour stocker le résultat. Celui-ci est mis dans les registres HI et LO (high et low pour la partie haute et basse du résultat). De la même manière, la division produit un quotient et un reste qui sont stockés dans les registres LO et HI. Le format de codage est R.

Question 2

Regardez le format de codage I. Quelle est la taille du champ où est encodé l'immédiat ? Qu'en déduisez vous sur les valeurs possibles ?
Comment cet immédiat est il étendu à l'exécution ?

Solution:

Le champ immédiat est sur 16 bits. Les opérations arithmétiques considèrent des valeurs signées, on ne peut donc pas mettre de valeurs en dehors de l'intervalle $[-2^{15}, 2^{15} - 1]$. Les opérations logiques considèrent des valeurs non signées, on ne peut donc pas mettre de valeurs en dehors de l'intervalle $[0, 2^{16} - 1]$

Comme les immédiats dans les opérations arithmétiques sont considérés comme signés (même pour `addu`), à l'exécution, l'extension de 16 à 32 bits de l'immédiat est donc réalisée en étendant le bit de signe sur les 16 bits de poids fort (les opérations ont *toujours* lieu sur des mots de 32 bits).

Question 3

À l'aide de votre mémento, donnez le codage binaire des instructions suivantes :
`addiu $12, $18, 15`
`addu $12, $18, $4`

Solution:

Il y a des explications dans le cours si besoin.

Le codage de `addiu $12, $18, 15` est le suivant :

OPCOD	Rs	Rt	Imm
0010 01	10 010	0 1100	0000 0000 0000 1111

Donc `addiu $12, $18, 15` se code `0x264C000F`. On peut le vérifier en utilisant MARS.

OPCOD	Rs	Rt	Rd	Sh	Func
0000 00	10 010	0 0100	0110 0	000 00	10 0001

Donc `addu $12, $18, $4` se code en `0x02446021`.

Question 4

Parcourez sur votre mémento toutes les opérations logiques (opérations sur des vecteurs de bits) du jeu d'instructions MIPS. Notez, pour chacune, son code opération, le nombre de ses opérandes sources et résultats, et la nature de ces opérandes ainsi que le format de codage. La règle trouvée précédemment est elle correcte ? Quelle est l'exception à la règle ? Pourquoi ?

Solution:

Les opérations logiques sont : `or`, `and`, `xor`, `nor`, `ori`, `andi`, `xori`. Celles de décalages sont : `sllv`, `srlv`, `srav`, `sll`, `srl`, `sra`. Ces intructions ont toutes 2 opérandes sources et un opérande résultat.

Toutes ces instructions sont codées au format R, sauf celles dont le mnémonique fait apparaitre un `i` : elles ont un opérande immédiat et sont codées au format I.

Or les opérations de décalage dont le mnémonique ne fait pas apparaitre de `v`, bien qu'elles soient codées au format R, ont aussi un opérande immédiat (on n'a donc plus "immédiat = I"). La raison est que cet immédiat code un décalage et cela n'a pas de sens de décaler de plus de 31 bits. Comme 5 bits suffisent pour encoder une valeur comprise entre 0 et 31, cet immédiat est encodé et placé dans le champ `sh` (shift amount) du format R.

Question 5

1. Si le registre $8 contient la valeur 0x0000000F, quelle est la valeur contenue dans le registre $9 après l'opération `sll $9,$8,8` ?
2. Si le registre $8 contient la valeur 0xF0000000, quelle est la valeur contenue dans le registre $9 après l'opération `srl $9,$8,28` ?
3. Si le registre $8 contient la valeur 0xF0000000, quelle est la valeur contenue dans le registre $9 après l'opération `sra $9,$8,28` ?
4. Si le registre $8 contient la valeur 0x00000036, quelle est la valeur contenue dans le registre $9 après l'opération `andi $9,$8,0x000F` ?

Solution:

1. l'opération décale de 8 bits à gauche la valeur contenue dans le registre $8 et la stocke dans le registre $9. Les 8 bits de poids faible sont mis à 0. Ainsi le registre $9 contient la valeur 0x00000F00 après l'opération.
2. l'opération décale de 28 bits à droite la valeur contenue dans le registre $8 et stocke le résultat dans le registre $9. Les 28 bits de poids fort sont mis à 0, il n'y pas de propagation du bit de signe sur les 28 bits de poids fort car c'est une opération logique. Ainsi, après l'opération, le registre $9 contient la valeur 0x0000000F.

3. Contrairement à l'opération `srl`, `sra` est une opération arithmétique, il y a propagation de la valeur du bit de signe sur les 28 bits correspondant à la valeur immédiate. Après l'opération, le registre $9 contient donc la valeur 0xFFFFFFFF.
4. Après l'opération, le registre $9 contient la valeur 0x00000006. On dit que l'on a masqué les 28 bits de poids fort (mise à 0 de tous les bits pour lesquels le bit correspondant de l'immédiat vaut 0, l'immédiat étant étendu sur 32 bits sans extension de signe car il s'agit d'une opération logique).

Question 6

À l'aide de votre mémento, donnez le codage binaire des instructions suivantes :

```
sll $12, $18, 5
mult $8, $9
```

Solution:

On obtient donc le codage suivant pour `sll $12, $18, 5` :

OPCOD	Rs	Rt	Rd	Sh	Func
0000 00	00 000	1 0010	0110 0	001 01	00 0000

Donc `sll $12, $18, 5` se code `0x00126140`.
On obtient le codage suivant pour `mult $8, $9` :

OPCOD	Rs	Rt	Rd	Sh	Func
0000 00	01 000	0 1001	0000 0	000 00	01 1000

Donc `mult $8, $9` se code `0x01090018`.

Question 7

1. Quelle est la particularité du registre $0 ?
2. Donnez plusieurs instructions qui permettent de mettre la valeur 0 dans le registre $8.
3. Donnez plusieurs instructions qui permettent de copier le contenu du registre $10 dans le registre $8.

Solution:

1. Le registre $0 contient toujours 0.
2. Pour mettre la valeur 0 dans le registre $8, on peut utiliser une des instructions suivantes :

```
addi $8, $0, 0
add  $8, $0, $0
xor  $8, $8, $8
sub  $8, $8, $8
andi $8, $0, 0
or   $8, $0, $0
...
```

3. Pour recopier le contenu de $10 dans $8, on peut utiliser une des instructions suivantes :

```
or   $8, $10, $0
ori  $8, $10, 0
add  $8, $10, $0
addi $8, $10, 0
...
```

Exercice 3 – Chargements d'une valeur dans un registre

Question 1

1. Donnez une instruction permettant de charger (mettre) la valeur `0x1234` dans le registre $8.
2. Rappelez le nombre de bits du champ immédiat des instructions avec un opérande immédiat ? Comment mettre la valeur `0x12345678` dans le registre $8 ? Donnez un exemple.
3. Les séquences suivantes sont elles équivalentes ? Que contient le registre $8 après ces deux séquences d'instructions ?

```
xor $8, $8, $8
addiu $8, $8, 0x8765
```

```
xor $8, $8, $8
ori $8, $8, 0x8765
```

4. Comment charger la valeur -1 dans un registre ?

Solution:

1. `ori $8, $0, 0x1234` permet de mettre la valeur `0x1234` dans le registre $8.
2. Le champ immédiat comporte 16 bits. Pour mettre la valeur `0x12345678` dans le registre $8, il faut décomposer le chargement en deux parties : le chargement des 16 bits de poids fort avec une instruction `lui` et le chargement des 16 bits de poids faible ensuite.

   ```
   lui $8, 0x1234
   ori $8, $8, 0x5678
   ```

   On peut aussi écrire à la main la mise en registre des 16 bits de poids fort en utilisant une opération `ori` puis une opération de décalage (mais `lui` le fait en une seule instruction et est à préférer).

   ```
   ori $8, $0, 0x1234
   sll $8, $8, 16
   ori $8, $8, 0x5678
   ```

3. Pour la première séquence (celle de gauche) : le registre $8 contient après la première instruction la valeur `0`, et après la deuxième `0xFFFF8765`. En effet, il y a extension aux 16 bits de poids fort du signe de l'immédiat sur 16 bits.

   Pour la deuxième séquence : le registre $8 contient après la première instruction la valeur `0`, et après la deuxième `0x00008765`. En effet, `ori` étant une opération logique, il n'y a pas d'extension aux 16 bits de poids fort du signe de l'immédiat sur 16 bits.

   Lors du chargement d'une valeur, il faut donc faire très attention à cette extension du bit de signe. Notamment lors d'un chargement en deux temps ou lors du chargement d'une valeur négative. Il vaut mieux prendre l'habitude d'utiliser des opérateurs logiques pour les valeurs à charger en deux temps, par contre, pour les valeurs négatives il faut *obligatoirement* utiliser un opérateur arithmétique !
4. Conséquence de la question précédente, pour charger -1 dans un registre il faut utiliser un opérateur arithmétique par exemple : `addiu $4, $0, -1`.

Question 2

Écrivez un programme complet (toutes les sections doivent apparaître) qui met respectivement les valeurs 0x34 et 34 dans les registres $8 et $9, et produit le résultat de l'addition de ces deux valeurs dans le registre $10.

Solution:

```
.data
.text
   ori $8, $0, 0x34
   ori $9, $0, 34
   addu $10, $8, $9
   ori $2, $0, 10
   syscall
```

Exercice 4 – Appels système de lecture et d'écriture d'entiers

Question 1

Qu'est-ce qu'un appel système ? Comment réaliser un appel système ?

Solution:

C'est un service demandé au système, qui requiert un accès privilégié à certaines ressources (typiquement les périphériques d'entrée/sortie : clavier, écran, etc.).

Celui-ci est réalisé avec l'instruction `syscall`. Au préalable, il faut avoir mis le numéro/code de l'appel système dans le registre `$2`. Ceux pour MARS sont donnés dans le Mémento et la documentation MIPS disponible sur le site web de l'UE. Suivant les appels, les valeurs des paramètres éventuels doivent être mises dans les registres `$4` à `$7`. La valeur de retour éventuelle se trouve dans `$2`.

Question 2

1. Complétez le programme assembleur de l'exercice précédent afin d'afficher la valeur résultant de l'addition à la fin du programme (cherchez dans le mémento l'appel système permettant d'afficher un entier).
2. Modifiez votre programme pour que la valeur du premier entier soit lue au clavier (cette valeur doit ensuite être ajoutée à la valeur 34).

Solution:

```
.data

.text
   ori  $2, $0, 5
   syscall          # lecture 1ere valeur

   ori  $9, $0, 34 # valeur 34

   addu $4, $2, $9
   ori  $2, $0, 1
   syscall          # affichage de la somme

   ori  $2, $0, 10
   syscall          # exit
```