Architecture des ordinateurs

TP4 : programmation en langage machine

C. Alias

Le but de ce TP est d'écrire quelques programmes simples pour notre processeur MIPS.

Exercice 0. *L'assembleur ASM*

Il existe un outil qui génére automatiquement les fichiers .ram à partir d'un programme écrit en langage machine. Cet outil est appelé *assembleur*. Voilà un exemple de programme accepté par notre assembleur:

```
      ldi r0,1           // compteur := 1
      ldi r1,1
      ldi r2,10          // max := 10
loop:
      ble r0,r2,body     // compteur <= 10? => itérer.
      j end_loop
body:
      add r0,r0,r1       // compteur := compteur + 1
      j loop
end_loop:
      j end_loop
```

`loop` et `end_loop` sont des *labels*. Ils nomment les points d'exécution du programme sur lesquels effectuer un saut. L'assembleur se charge de calculer le décalage pour ble et l'adresse absolue pour j.
Questions.

- Ajouter le répertoire `/home/tpetu/INF2003L/asm/bin/` à votre variable `$PATH`.
- Entrez le programme ci-dessus dans le fichier `test.asm`.
- Assemblez-le avec `asm test.asm` L'assembleur produit:
  - Le code hexadecimal correspondant sur la sortie standard.
  - Les deux fichiers **test_hi.ram** et **test_lo.ram** nécessaires au chargement dans DigMips.
- Copiez le fichier `/home/tpetu/INF2003L/diglog/tp/digmips.tgz` dans votre répertoire, puis décompactez-le avec `tar xvfz digmips.tgz`. Vous disposez maintenant du processeur complet...
- Testez dans DigMips les fichiers produits par l'assembleur.

Exercice 1. *Entrées/Sorties*

DigMips possède un fichier supplémentaire, (io.lgf, onglet 5) qui permet de réaliser des entrées sur le clavier et des sorties sur l'écran. En principe, il faudrait ajouter deux instructions dédiées aux entrées/sorties. Comme on ne peut disposer que de 8 instructions, on choisi de recycler les instructions ld et st de la façon suivante:

- **Pour afficher un caractère sur l'écran.** Placer le code ASCII le caractère dans un registre r, puis exécuter st r,[n_importe_quel_registre + 63]. Le registre de base n'a pas d'importance, il faut juste que la valeur immédiate soit 63. Par exemple:

  ```
  ldi r0,'a'    // place le code ascii de 'a' dans r0
  st r0,[r7+63] // affiche r0.
  ```

- **Pour lire un caractère au clavier.** De manière analogue, il suffit d'exécuter: ld r,[n_importe_quel_registre + 63]. Par exemple:

  ```
  ld r0,[r7+63] // Lit l'état du clavier, et place le résultat dans r0
  ```

Les caractères du clavier sont stockés dans une file de taille 4. Il est possible qu'aucun caractère ne soit disponible. Dans ce cas, on lit la valeur 0. Pour saisir un caractère au clavier, il faudra donc boucler jusqu'à ce qu'un caractère soit disponible.

- Ecrire un programme `hello.asm` qui affiche le texte `Hello world!` et passe à la ligne (caractère 13).
- Ecrire un programme `affiche10.asm` qui affiche les 10 premiers entiers (de 0 à 9).
- Ecrire un programme `saisie.asm` qui affiche `nb?`, passe à la ligne, saisit un entier entre 0 et 9, passe à la ligne, et l'affiche.

Exercice 2. *Appels de fonctions*

On considère le programme C suivant:

```
void main()
{
  printf("m");
  f();
  g();
}

void f()
{
  printf("f");
  g();
}

void g()
{
  printf("g");
}
```

Questions.

- **Tracez l'arbre des appels.** Quel affichage produit ce programme ?
- On souhaite écrire ce programme en assembleur. Ouvrez le fichier `call_mask.asm`.
- **Réperez** les définitions des fonctions `main()`, `f()`, et `g()`. **Repérez les appels de fonction**.
- En fait, le problème est de revenir à la fonction appelante... Pourquoi ne peut-on pas l'implémenter par un saut `j`? **Quelle fonction pose problème?**

Avant chaque appel, il faut donc indiquer à la fonction où revenir dans le programme une fois son exécution terminée. Par exemple, le point de retour de l'appel à `f()` dans `main()` est le label `rf` (ligne 30).

- **Avant un appel.** Placer en mémoire l'adresse du point de retour. L'adresse du label `rf` peut se récupérer avec l'instruction:

  ```
  ldi r0,lo(rf) //r0 := 8 bits de poids *faible* de rf
  ldi r1,hi(rf) //r1 := 8 bits de poids *fort* de rf
  ```

  Elle peut ensuite être placée en mémoire avec l'instruction `st`.

- **A la fin de la fonction appelée.** Récupérer l'adresse de retour en mémoire (par exemple r2 := poids fort, r3 := poids faible), et sauter avec `ja r2,r3` (voir lignes 74 – 80).

La question est maintenant **où** placer l'adresse de retour...

Est-il suffisant de réserver 2 octets (par exemple 0 et 1)? Non! Plusieurs adresses de retour peuvent être **vivantes** en même temps, comme l'indique le schéma suivant:

Contexte	Mémoire 255	254	253	252	Action
main() {					
appel à f()					empiler(rf)
f()	lo(rf)	hi(rf)			
{	" "	" "			
appel à g()	" "	" "			empiler(rg2)
g()	lo(rf)	hi(rf)	lo(rg2)	hi(rg2)	
{	" "	" "	" "	" "	
}	" "	" "	" "	" "	dépiler(rg2)
rg2:	lo(rf)	hi(rf)			
}	" "	" "			dépiler(rf)
rf:					
appel à g()					empiler(rg1)
g()	lo(rg1)	hi(rg1)			
{	" "	" "			
}	" "	" "			dépiler(rg1)
rg1:					
}					

Il faut donc stocker les adresses de retour dans une **pile**. Comme on ne peut adresser que les 255 premiers octets, on place la base de la pile à l'adresse 255, et on empile "vers le bas".

Par convention, le **pointeur de pile** est placé dans le registre **r6**, et pointe immédiatement après le dernier octet empilé. Initialement, r6 vaut 255 (voir ligne 18).

- Pour **empiler** l'octet contenu dans r0, il faut donc écrire:

```
st r0,[r6]
sub r6,r6,r1 //r1 contient toujours 1 (voir ligne 19)
```

- Pour récupérer le sommet de pile dans r2, et **dépiler**, il faut écrire:

```
add r6,r6,r1
ld r2,[r6]
```

Complétez et testez le programme **call_mask.asm**.