

Travaux Dirigés Structure des ordinateurs: Feuille 2

Télécom 1ère année. ENSEIRB-MATMECA 2014/2015

—Denis Barthou—

►Exercice 1. Du C à l'assembleur

Un programme C est compilé (traduit) en assembleur par le compilateur. Celui ci réalise cette traduction en plusieurs étapes :

- La génération de l'assembleur : c'est la traduction du C en assembleur `fichier.s` avec `gcc -S <fichier.c>`
- L'assemblage : c'est la traduction de l'assembleur en fichier objet. Cela crée un fichier `fichier.o` avec la commande `gcc -c <fichier.s>`
- Le linkage : c'est la création d'un binaire avec `gcc -o <executable> <fichier.o>`

1. Ecrire un code C qui affiche les puissances de 2 pour les exposants de 1 à 128. On le fera par multiplications successives, sur des flottants. On mettra le calcul dans une fonction `calcule`.
2. Compiler votre fichier C en assembleur. Chercher votre fonction `calcule` et une instruction `..mul..` : c'est la multiplication de votre code.
3. Créer un fichier objet. Lire le contenu en faisant `objdump -d <fichier.o>`. Chercher à nouveau votre fonction et la multiplication. Les adresses sont indiquées à gauche. Quelle est l'adresse du début du code ? Le compilateur a-t-il rajouté d'autres fonctions ? (commençant avec des labels)
4. Créer un fichier exécutable. Lire le contenu avec la même commande que précédemment. Qu'est-ce qui change ?

►Exercice 2. Programmation MIPS

On va utiliser le simulateur MIPS en javascript sur la page <http://morriswmz.jit.su/static/simple-mips.html>.

La structure d'un code assembleur est décrite par 2 sections, l'une des données et l'autre du code. Le schéma général est le suivant :

```
# Comment giving name of program and description of function
# Template.s
# Bare-bones outline of MIPS assembly language program

.data          # variable declarations follow this line
               # ...

.text          # instructions follow this line

main:          # indicates start of code (first instruction to execute)
               # ...
```

Dans la section `.data`, on peut utiliser des lignes avec :

- `.space n` déclare un espace de n octets, initialisés à 0.
- `.word n1, n2, ...` avec $n1, n2$ des valeurs entières. Place une liste de valeurs entières (4 octets chacune) en mémoire.
- `.byte n1, n2, ...` avec $n1, n2$ des valeurs sur un octet.

Dans la section instructions, on y place les instructions MIPS. Le code doit terminer par `break`.

1. Copier le code suivant et l'exécuter pas à pas. Que fait-il ?

```
# first program
.data
val: .word 33,47
.text
la    $r2, val      # copy address of val into r2
lw    $r3, 0($r2)   # load value 33 into r3
addi  $r2, $r2, 4    # add 4 to address
lw    $r4, 0($r2)   # load value 47 into r4
add   $r3, $r3, $r4
print $r3
break
```

2. Combien d'instructions sont exécutées ? En combien de cycles ?
3. Créer un code qui alloue un tableau de 128 entiers (avec `.space 512`) et écrit dans la case i du tableau la valeur i entière (sur 4 octets). Pour cela, on fera une boucle qui énumère toutes les valeurs de 1 à 127.

►**Exercice 3.** MIPS pipeliné On utilise maintenant le simulateur MIPS avec pipeline <http://morriswmz.jit.su/static/simple-mips-pipelined.html>. Le simulateur indique en face de chaque instruction dans quel étage du pipeline elle est. Les noms des étages sont, dans l'ordre du pipeline : F : instruction fetch, D : instruction decode, E : execute, M : memory, W : write back.

1. Reexecuter le premier code assembleur. 1 cycle du simulateur non pipeliné compte maintenant comme 5 cycles. En combien de cycles s'exécute le code ? Combien de stall ? Expliquer pourquoi
2. Faire un programme qui calcule la suite de Syracuse à partir d'une valeur initiale (donnée par un `.word`). A partir du terme s_n de la suite, le terme s_{n+1} est calculé comme suit : si s_n est pair, alors $s_{n+1} = s_n/2$ sinon $s_{n+1} = 3s_n + 1$. Si on atteint la valeur de 1, alors on arrête de calculer la suite. Pour ce code, on fera la multiplication par 3 par des additions et pour la division par 2, on utilisera `srl reg, reg, 1`.