Assembleur Y86, premiers pas

Registres

Exercice 1 La suite de Fibonacci est définie par la récurrence :

$$u_1 = u_2 = 1, u_{n+2} = u_n + u_{n+1} \text{ pour } n > 0$$

1. On souhaite que les termes successifs de cette suite apparaissent dans le registre `%eax`, compléter à cet effet le programme suivant :

```
        irmovl  1, %eax
        irmovl  1, %ebx
boucle: ...
        addl    %ebx, %eax
        ...
        jmp     boucle
```

2. Ajouter un compteur (utiliser le registre `%esi`) pour que le programme s'arrête après avoir calculé 16 termes de la suite.

Exercice 2 Les douze premiers termes de la suite de Fibonacci sont 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 et le contenu des registres est toujours affiché en hexadécimal par le simulateur :

1. calculer les douze premiers termes de la suite en effectuant les additions directement en hexadécimal ;
2. convertir ces termes en décimal, et vérifier qu'on retrouve la liste ci-dessus ;
3. en numération décimale 1597 et 2584 sont deux termes consécutifs de la suite de Fibonacci, calculer leurs représentations hexadécimales.

En TP, vérifier les résultats de ces calculs avec le simulateur.

Code objet

Exercice 3 Etudier le (pseudo) code objet produit par compilation du corrigé de l'exercice 1, distribué sur une feuille à part (ce code est représenté en hexadécimal) :

1. Le premier octet de chaque instruction est son *code opération* : quelles sont les instructions codées par 20 et 30 (en hexadécimal) ?
2. Quels sont les codes opérations dont le second chiffre n'est pas nul ? Pourquoi ?
3. L'octet suivant code deux registres (sauf si l'instruction n'en utilise pas) : quels sont les numéros des registres `%eax`, `%ebx`, `%ecx` et `%esi` ? *Note* : le résultat est un peu déroutant, comme souvent chez Intel, qui par ailleurs code en réalité les numéros des 8 registres sur 3 bits (logique, non ?) ; ici ils sont codés sur 4 bits et le numéro 8 désigne un registre "qui n'existe pas".
4. Quelle est la taille du code complet d'une instruction `irmovl` ? Pourquoi ?
5. Quelle est la taille du code complet d'une instruction de saut ? Pourquoi ?
6. Un entier de 32 bits est représenté par un *mot* de 4 octets, rangés dans un ordre inhabituel appelé *little endian* ; donner le code des instructions suivantes :
   `irmovl 0x1a2b,%eax`
   `irmovl 0x1234c,%ebx`

7. La colonne de gauche indique les *adresses* des instructions : quelles sont celles qui sont des multiples de 4 ?
8. Quelle est la valeur de l'étiquette *boucle* ? Où retrouve-t-on cette valeur dans le code objet ?
9. Indiquer le contenu de la mémoire à partir de l'adresse `0x18` en la représentant par un tableau dont chaque ligne est un *mot* de 4 octets, précédé de son adresse (adresses et mots sont codés en hexadécimal, les octets d'un mot sont rangés "par le petit bout") :

Adresse	Mot
18	

Exercice 4

1. Quel est le résultat de l'instruction `andl %eax,%eax` ?
2. A quoi peut bien servir une telle instruction ? Indication : elle modifie quelquechose...
3. Quel est le résultat de l'instruction `xorl %eax,%eax` ?
4. Quel est le résultat de l'instruction `iandl 1,%eax` ? A quoi peut-elle servir ?
5. Comment tester si une adresse contenue dans le registre `%edi` est un multiple de 4 ?

Mémoire

Exercice 5 On part du programme dont le code a été distribué pour l'exercice 3 et on ajoute juste après le code un entier n correctement aligné :

```
        irmovl  1, %eax
        irmovl  1, %ebx
        ...
boucle: ...
        ...
        ...
        jne     boucle
        halt
.align 4
n:      .long   16
```

Modifier et compléter ce programme :

1. pour qu'il lise la valeur de n et la place dans le registre `%esi` ;
2. pour qu'il écrive en mémoire un tableau constitué des n premiers nombres de Fibonacci, et situé juste *à la suite* de l'entier n — utiliser le registre `%edi` comme pointeur.

Suite de Syracuse

Exercice 6 Une suite de Syracuse est définie par un premier terme u_0 et par la récurrence :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ (3u_n + 1)/2 & \text{sinon} \end{cases}$$

On constate que pour tout entier positif u_0 il existe n tel que $u_n = 1$.

Les instructions `sarl` (*shift arithmetic right*) et `sall` (*shift arithmetic left*) décalent (vers la droite ou vers la gauche) les chiffres binaires du registre cible ; on utilise en général leurs versions immédiates notées `isarl` et `isall` dans le jeu d'instructions y86.

1. Ecrire l'instruction qui divise par deux le contenu du registre `%eax`.
2. Ecrire une première version d'un programme qui calcule dans le registre `%eax` les termes successifs de la suite avec $u_0 = 27$. Vérifier en TP que le test pair/impair est correctement réalisé : calculer à la main les premiers termes de la suite, effectuer les conversions en hexadécimal, et comparer avec les résultats affichés par le simulateur.
3. Ajouter un test qui stoppe l'exécution du programme lorsque $u_n = 1$. En TP vérifier avec $u_0 = 6$.
4. Placer $u_0 = 27$ au début d'un tableau s, et modifier le programme précédent pour qu'il range en mémoire les termes de la suite dans le tableau s, jusqu'à $u_n = 1$.
5. Exécuter le programme avec le simulateur et noter l'adresse du dernier terme calculé (celui qui vaut 1). Comparer avec l'adresse du premier terme, et en déduire le nombre de termes de la suite.
6. Quel est le plus grand terme de la suite ? Effectuer la conversion en représentation décimale.
7. Examiner le code de condition Z en fin de calcul.
8. Examiner le code objet des instructions arithmétiques et logiques, en version immédiate ou non.

Exercice 1, code objet

Ce code est nécessaire pour l'exercice 3 :

```
0x000: 308001000000 |          irmovl  1, %eax
0x006: 308301000000 |          irmovl  1, %ebx
0x00c: 308610000000 |          irmovl  16, %esi        # compteur
0x012: 2001         | boucle:  rrmovl  %eax, %ecx
0x014: 6030         |          addl    %ebx, %eax
0x016: 2013         |          rrmovl  %ecx, %ebx
0x018: c18601000000 |          isubl   1, %esi
0x01e: 7412000000   |          jne boucle
0x023: 10           |          halt
```

Exercice 5, corrigé

Ce programme calcule n termes de la suite de Fibonacci (ici $n = 16$), et les range en mémoire à la suite de n. Après l'exécution du programme la mémoire contient donc les valeurs suivantes, à partir de l'adresse $0x38$: 16, 2, 3, 5, 8, 13, etc.

```
0x000: 308001000000 |          irmovl  1, %eax
0x006: 308301000000 |          irmovl  1, %ebx
0x00c: 506838000000 |          mrmovl  n, %esi          # compteur
0x012: 308738000000 |          irmovl  n, %edi          # pointeur
0x018: 2001         | boucle:  rrmovl  %eax, %ecx
0x01a: 6030         |          addl    %ebx, %eax
0x01c: 2013         |          rrmovl  %ecx, %ebx
0x01e: c08704000000 |          iaddl   4, %edi
0x024: 400700000000 |          rmmovl  %eax, (%edi)
0x02a: c18601000000 |          isubl   1, %esi
0x030: 7418000000   |          jne boucle
0x035: 10           |          halt
0x038:              | .align 4
0x038: 10000000     | n:       .long   16
```

Exercice 6 (suite de Syracuse), corrigés

```
# Version 1 avec une simple boucle infinie
# Objectifs: test pair/impair, calcul de 3u+1

        irmovl  27, %eax     # u
boucle: rrmovl  %eax, %ecx
        iandl   1, %ecx
        je      pair
        # u est impair
        rrmovl  %eax, %ecx
        addl    %eax, %eax  # 2u
        addl    %ecx, %eax  # 3u
        iaddl   1, %eax     # 3u + 1
pair:   isarl   1, %eax     # division par 2
        jmp boucle
```

```
# Version 2 avec test de fin (u=1)
# Valeur initiale 6 pour eviter un calcul trop long

        irmovl  6, %eax     # u
boucle: rrmovl  %eax, %ecx
        iandl   1, %ecx
        je      pair
        # u est impair
        rrmovl  %eax, %ecx
        addl    %eax, %eax  # 2u
        addl    %ecx, %eax  # 3u
        iaddl   1, %eax     # 3u + 1
pair:   isarl   1, %eax     # division par 2
        rrmovl  %eax, %ecx
        isubl   1, %ecx     # u = 1 ?
        jne     boucle      # non
        halt
```

Version finale avec code objet ; noter que la solution utilisée pour l'adressage est une variante de celle utilisée dans le corrigé de l'exercice 5, cf. instructions d'adresses $0x006$ et $0x02d$:

```
                    | # Version finale: on range les termes
                    | # successifs de la suite dans le tableau s
                    |
  0x000: 500800010000 |         mrmovl  s, %eax     # load u
  0x006: 6377       |         xorl    %edi, %edi
  0x008: 2001       | boucle: rrmovl  %eax, %ecx
  0x00a: c28101000000 |         iandl   1, %ecx
  0x010: 7321000000 |         je      pair
                    |         # u est impair
  0x015: 2001       |         rrmovl  %eax, %ecx
  0x017: 6000       |         addl    %eax, %eax  # 2u
  0x019: 6010       |         addl    %ecx, %eax  # 3u
  0x01b: c08001000000 |         iaddl   1, %eax     # 3u + 1
  0x021: c58001000000 | pair:   isarl   1, %eax     # division par 2
  0x027: c08704000000 |         iaddl   4, %edi
  0x02d: 400700010000 |         rmmovl  %eax, s(%edi)   # store u
  0x033: 2001       |         rrmovl  %eax, %ecx
  0x035: c18101000000 |         isubl   1, %ecx     # u = 1 ?
  0x03b: 7408000000 |         jne     boucle      # non
  0x040: 10         |         halt
                    |
  0x100:            | .pos    0x100
  0x100: 1b000000   | s:      .long 27
```

Feuille mise à jour le 4 mars 2011