Architecture des circuits

L'assembleur

Contenus	Capacités attendues	Commentaires
Modèle d'architecture séquentielle (von Neumann)	Distinguer les rôles et les caractéristiques des différents constituants d'une machine. Dérouler l'exécution d'une séquence d'instructions simples du type langage machine.	La présentation se limite aux concepts généraux. On distingue les architectures monoprocesseur et les architectures multiprocesseur. Des activités débranchées sont proposées. Les circuits combinatoires réalisent des fonctions booléennes.

Pré-requis :

- Représentations binaires et hexadécimales
- Notation little-endian / big-endian
- Langage Python

Introduction

Nous avons déjà dit que les microprocesseurs ne fonctionnent qu'à partir d'une suite de 0 et de 1, appelé langage binaire, ou langage machine.

Un **langage d'assemblage** ou **langage assembleur** est, en programmation informatique, le langage le plus bas niveau qui représente le langage machine sous une forme lisible par un humain. Les combinaisons de bits du langage machine sont représentées par des symboles faciles à retenir. Le programme assembleur convertit ensuite ces symboles en langage machine.

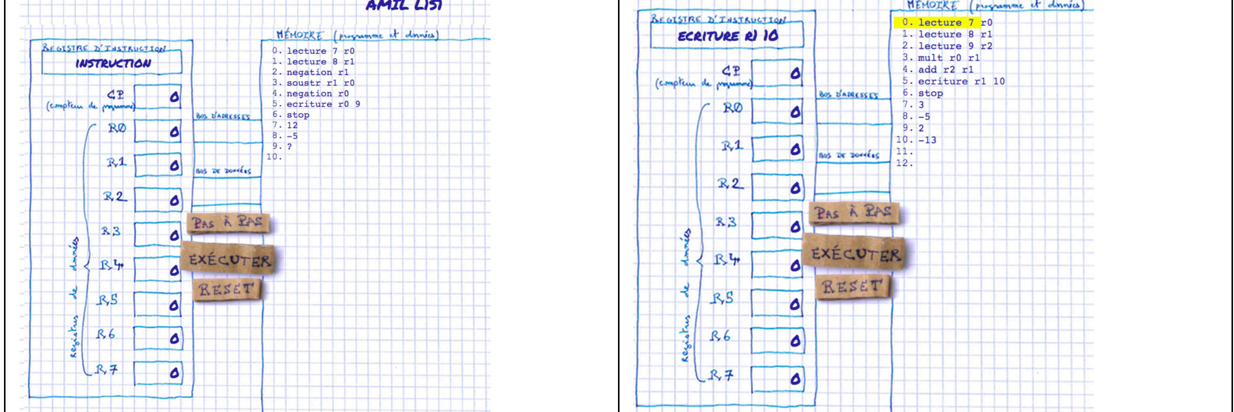
Activité n°1 : Découverte d'un langage assembleur

Exemple vidéo-projeté et travail en "débranché"

Le logiciel de simulation suivant disponible au lien suivant (http://www.fil.univ-lille1.fr/~levaire/amill1s1/) explique de façon imagée ce qu'il se passe dans un assembleur.

[Présentation de quelques instructions simples sur ce logiciel]

A vous de jouer : Faire tourner les programmes ci-dessous à la main, et remplissant les valeurs obtenues dans chaque case :



<u>Activité n°2</u> : comparaison d'un même programme dans plusieurs langages
Voici un même programme écrit en Python et dans 2 langages en assembleur (un simplifié, le AMIL, un plus proche de la réalité, le Y86)

Python	Assembleur	
	AMIL	Y86
1 a=1000 2 b=500 3 c=a+b	0 lecture 5 r0 1 lecture 6 r1 2 add r1 r0 3 ecriture r0 7 4 stop 5 1000 6 500 7 ?	1 .pos 0 2 mrmovl a, %eax 3 mrmovl b, %ebx 4 addl %eax,%ebx 5 rmmovl %ebx,c 6 halt 7 8 .align 4 9 a: .long 1000 10 b: .long 500 11 c: .long
Variables Name Value a 1000 b 500 c 1500	0 lecture 5 r0 1 lecture 6 r1 2 add r1 r0 3 ecriture r0 7 4 stop 5 1000 6 500 7 1500	0x0015: | .align 4 0x0018: e8030000 | a: .long 1000 0x001c: f4010000 | b: .long 500 0x0020: | c: .long 0018 e8030000 001c f4010000 0020 dc050000 0024 00000000 0028 00000000

La version assembleur Y86 permet de voir le langage machine (qui est le langage compréhensible par les processeurs des ordinateurs).
Les nombres d'entrée (1000 et 500) sont automatiquement convertis en binaire sur la machine et apparaissent en hexadécimal sur Y86.

- 1000 en décimal donne 3E8 en hexadécimal, et en little-endian : E8 03 00 00. Il est stocké en ligne 18 (en hexadécimal car l'adresse de la ligne commence par 0x). Ce 18 hexadécimal correspondrait en décimal à la ligne 24.
- De même, 500 en décimal est égal à 1F4, qui donne : F4 01 00 00, stocké en ligne 1c (donc en ligne n°28).

La somme, qui correspond à l'étiquette c, est stockée à l'adresse 0x0020 (en réalité, pas besoin de savoir combien ça fait en décimal) et à cette adresse, on y trouve dc 05 00 00, ce qui correspond en hexadécimal normal (big-endian) à 00 00 05 dc, soit 1500 en décimal.

<u>Explications de certaines instructions en Assembleur Y86</u> :
Dans ce simulateur de langage assembleur, la plupart des instructions se terminent par -l.

mrmovl a, %eax	peut se lire mr movl a, %eax au centre, il y a movl qui signifie qu'on va déplacer (to move) le m signifie qu'on va chercher a dans la **<u>M</u>**émoire le r signifie qu'on va mettre le résultat dans le **<u>R</u>**egistre %eax Bref, à la fin, dans le registre %eax, il y a le nombre qui était dans a.
addl %eax, %ebx	addl signifie qu'on ajoute (to add). On ne peut ajouter que 2 nombres situés dans des registres. La somme des nombres rangés dans les registres %eax et %ebx est mise dans %ebx
rmmovl %ebx, c	movl donc on déplace. r au début donc on prend la valeur du **<u>R</u>**egistre %ebx m après, donc on met le résultat dans la **<u>M</u>**émoire c

Chacune des instructions est codée en hexadécimal :

ainsi	l'instruction	mrmovl a,%eax	est codée par	500f20000000
et	l'instruction	mrmovl b,%ebx	est codée par	503f240000000
et	l'instruction	rmmovl %ebx,c	est codée par	403f28000000

A vous de jouer : Chaque instruction commence par un nombre appelé **opcode** (codé sur 1 octet) qui détermine la nature de l'instruction.
Retrouver sur le logiciel Y86 l'opcode de l'instruction addl.

Pour ceux qui vont plus vite : Pour quelle raison l'instruction addl n'est-elle codée que sur 4 demi-octets alors que l'instruction mrmovl a, %eax, elle, est codée sur 12 demi-octets ?

Bilan : Tout le programme écrit en Assembleur-Y86 peut être transformé en hexadécimal, et ensuite en binaire, il peut donc être exécuté par le processeur.

Activité n°3 : **les flags** en assembleur-Y86

```
1  a=30
2  b=10
3  c=a-b
```

1) Traduire le programme Python ci-contre en assembleur sachant que la soustraction se note subl (to subtract)
2) En testant plusieurs valeurs pour a et b, observer la valeur des "flags" SF et ZF juste après que la soustraction soit effectuée.
   a) Sur combien de bits sont-ils codés ?
   b) A quoi correspond le flag SF ?
   c) A quoi correspond le flag ZF ?

Pour comparer 2 nombres a et b, on calcule leur différence $b - a$.

- Si $a = b$, combien valent les flags ? SF = … et ZF = …
- Si $a > b$, combien valent les flags ? SF = … et ZF = …
- Si $a < b$, combien valent les flags ? SF = … et ZF = …
- Si $a \geq b$, combien valent les flags ? SF = … et ZF = …
- Si $a \leq b$, combien valent les flags ? SF = … et ZF = …

Activité n°4 : **Faire un saut** (une autre façon de voir le test Si–Alors–Sinon)
L'idée est de faire un calcul qui correspond au test et de tester le signe et/ou la nullité du résultat et de faire, en fonction de la réponse, un saut (jump) vers un endroit précis du programme.

Jump Unconditionally
`jmp Dest`

Jump When Less or Equal
`jle Dest`

Jump When Less
`jl Dest`

Jump When Equal
`je Dest`

Jump When Not Equal
`jne Dest`

Jump When Greater or Equal
`jge Dest`

Jump When Greater
`jg Dest`

```
1   .pos 0
2       mrmovl a, %eax
3       mrmovl b, %ebx
4       rrmovl %eax, %ecx
5       subl  %ebx, %ecx
6       jl L1
7       rrmovl %eax, %ecx
8       jmp L2
9   L1:
10      rrmovl %ebx, %ecx
11  L2:
12      rmmovl %ecx, c
13      halt
14
15  .align 4
16  a:  .long 20
17  b:  .long 30
18  c:  .long
```

```
1  a=20
2  b=30
3  if (a-b)< 0 :
4      c=b
5  else :
6      c=a
```

Explications : En ligne 6, l'instruction jl L1 signifie que si lors du calcul fait à la ligne précédente, le résultat était négatif (donc SF=1 mais ZF=0), il faut aller à la ligne qui a l'étiquette L1 (bref, en ligne 9), si ce n'est pas le cas, on continue et, en ligne 8, il faut obligatoirement aller à la ligne de l'étiquette L2 (donc en 11). Ainsi, la ligne 7 n'est traitée que si on n'a pas fait le saut de la ligne 6 (donc si %ecx - %ebx n'est pas négatif, donc s'il est positif ou nul).

A vous de jouer :

1) A quelle condition la ligne 10 est-elle traitée ?
2) A quelle condition la ligne 12 est-elle traitée ?

Activité n°5 (en "débranché"): Les instructions qui commencent par i- : irmovl / iaddl et isubl
Le i- signifie **I**mmediately (c'est-à-dire qu'on n'a pas besoin d'aller chercher une valeur en mémoire ou dans un registre, mais qu'on dispose de sa valeur).
Ainsi, l'instruction irmovl 5, %eax signifie qu'on met le nombre 5 dans le registre %eax (cela gagne du temps par rapport à aller chercher la valeur en mémoire puis la ranger dans le registre)

A vous de jouer : Dérouler le programme ci-contre à la main et justifier que, dans le registre %ebx, on trouve 0x00000011 à la fin (c'est-à-dire 17 en décimal).

```
1 .pos 0
2     irmovl 3, %eax
3     irmovl 2, %ebx
4 L1:
5     iaddl 5, %ebx
6     isubl 1, %eax
7     jne L1
8     halt
```

A quel type de structure cela correspond-il en algorithmique (ou en Python) ?

Activité n°6 : Décoder un programme en assembleur et le comparer à un langage de niveau haut.

1) **Lancer** le simulateur y86 (http://dept-info.labri.fr/ENSEIGNEMENT/archi/js-y86/) et **ouvrir** le fichier "prog1.ys" (load).

SOURCE CODE

```
1   .pos 0
2       mrmovl N, %eax          # mise en registre du nombre de cases du tableau
3       irmovl 0, %edi          # mise à 0 du register %edi
4
5   boucle:
6       mrmovl t1(%edi), %ecx   # récupération de la valeur
7                               # de la case courante du tableau
8       rmmovl %ecx, t2(%edi)   #
9       iaddl 4, %edi           # incrémentation de 4 octets de
10                              # l'adresse courante du tableau
11      isubl 1, %eax           # nb tours-1
12      jne boucle              # Jump When Not Equal
13      halt
14
15  .align 4                    # données positionnées 4 octets après le programme
16  N:   .long 5                # nb d'éléments du tableau
17  t1:   .long 0x00000010      # données du tableau t1
18      .long 0x00000005
19      .long 0x00000011
20      .long 0x00000004
21      .long 0x00000008
22  t2:   .long
23
```

2) A l'aide du simulateur Y86, dérouler l'exécution pas à pas du programme (>step) et déterminer la tâche qu'il remplit.

Solution : Le programme 1 recopie les valeurs du tableau t1 dans un autre tableau(t2)

3) Ecrire un algorithme en langage naturel (ou avec un schéma) remplissant la même tâche.

Solution : n ← nombre cases de tb1
i ← 0
Faire
i ← i + 1
tb2[i] ← tb1[i]
n ← n - 1
tant que n > 0

4) Ecrire le programme en Python remplissant la même tâche

Solution :

```
t1 = [8,4,10,-1,5]
t2 = [ ]
for i in range(len(t1)) :
  t2.append(t1[i])
```

Activité n°7 : Reprendre les questions 1 à 4 de l'activité précédente avec "prog2.ys"

1) **Solution : Le programme 2 permet de multiplier 2 entiers**

2) **Algo naturel :**
M ← 4 et N ← 5
res = M * N

3) **Programme**

```
M,N = 4,5
res = M * N
```

SOURCE CODE

```
1   .pos 0
2       mrmovl M, %eax
3       mrmovl N, %ecx
4   boucle:
5       addl %ecx, %edx
6       isubl 1, %eax
7       jne boucle
8       rmmovl %edx, res
9       halt
10
11  .align 4
12  N:   .long 5
13  M:   .long 4
14  res:  .long
15
```

Activité n°7 : Mêmes questions avec "prog3.ys"
Avec ou sans les commentaires suivant le niveau des élèves

SOURCE CODE

```
1  .pos 0
2      mrmovl N, %eax          # mise en registre du nombre de cases du tableau
3      isubl 1, %eax           # N-1 pour préparer le nb de tours de la boucle
4      irmovl 0, %edi          # mise à 0 du register %edi
5      mrmovl t(%edi), %ecx    # récupération de la 1ere valeur du tb
6  boucle:
7      iaddl 4, %edi           # incrémentation de 4 octets
8                              # de l'adresse courante du tb
9      mrmovl t(%edi), %edx    # récupération de la valeur
10                             # de la case courante du tb
11     si:
12         subl %ecx, %edx     # soustraction des 2 registres
13         jl sinon           # Jump When Less (%edx > %ecx)
14     alors:
15         jmp fin_si          # %edx <= %ecx
16     sinon:
17         mrmovl t(%edi), %ecx  # mémorisation dans %ecx
18     fin_si:
19     isubl 1, %eax           # nb tours-1
20     jne boucle              # Jump When Not Equal
21     rmmovl %ecx, res        # enregistrement de res dans la RAM
22     halt
23
24 .align 4                    # données positionnées 4 octets après le programme
25 res: .long                  # case mémoire accueillant le résultat du programme
26 N:   .long 9                # nb d'éléments du tableau
27 t:   .long 0x00000010       # données du tableau
28      .long 0x00000005
29      .long 0x00000011
30      .long 0x00000004
31      .long 0x00000008
32      .long 0x00000005
33      .long 0x00000020
34      .long 0x00000004
35      .long 0x00000008
```

Réponses :

1) Le programme 3 trouve la valeur minimale contenue dans un tableau

2) Algo naturel :

```
n ← nombre cases de tb
min ← tb[0]
Pour i de 1 à (n-1) faire
   val_courante ← tb1[i]
   Si (val_courante < min) alors (min ← val_courante)
```

3) Programme :

```
t1 = [10,5,11,4,8,5,20,4,8]
min = t1[0]
for i in range(1,len(t1)):
  val_courante = t1[i]
  if (val_courante < min) :
    min = val_courante
print (min)
```

Conclusion : Quels sont les avantages d'un langage de programmation de haut niveau par rapport à un de bas niveau ?

Activité 8 : Pour aller plus loin

Activités pour les élèves les plus rapides
ou en devoir maison (?)

Même exercice que l'activité 3 avec le programme suivant (prog4.ys)

Solution : Le programme 3 construit une fonction qui additionne les 2 entiers entrés en paramètres

Algo naturel :

f(x,y)=x+y

res = f(7,5)

En python :

```
def f(x,y):
    return x+y

res = f(7,5)
print (res)
```

SOURCE CODE

```
1   .pos 0
2       irmovl pile, %esp
3
4       irmovl 5, %edx
5       pushl %edx
6       irmovl 7, %edx
7       pushl %edx
8
9       call f
10      iaddl 8, %esp
11      rmmovl %eax, res
12
13      halt
14  f:
15      mrmovl 4(%esp), %ecx
16      mrmovl 8(%esp), %eax
17      addl %ecx, %eax
18      ret
19
20  .align 4
21  res: .long 0
22
23  .pos 200
24  pile:
25
```