Exécuter un programme en assembleur

Programme assembleur

- Dans ce chapitre, nous allons étudier comment un processeur exécuter un programme assembleur.
- Le programme assembleur est codé en RAM

Exemple

```
L1:
.word a
.word b
.word c
.comm a,4,4
.comm b,4,4
.comm c,4,4
mov r0,#10
ldr r1,L1
str r0,[r1]
mov r0,#20
ldr r1,L1+4
str r0,[r1]
```

```
ldr r0,L1
ldr r1,[r0]
ldr r0,L1+4
ldr r2,[r0]
add r1,r1,r2
ldr r0,L1+8
str r1,[r0]
```

Comment ce programme est-il codé en RAM

```
1) mov r0,#10
2) ldr r1,L1
3) str r0,[r1]
4) mov r0,#20
5) ldr r1,L1+4
6) str r0,[r1]
7) ldr r0,L1
8) ldr r1,[r0]
9) ldr r0,L1+4
10) ldr r2,[r0]
11) add r1,r1,r2
12) ldr r0,L1+8
13) str r1,[r0]
```

CODE_OP

- Chaque instruction est codée par un CODE_OP qui contient le codage
  - du nom de l'instruction
  - des modes d'adressage
  - des valeurs des registres
- Ensuite il pet y avoir ou non un paramètre
  - une cconstante
  - une adresse
- Chacun de ces éléments occupe 32 bits.

Les étiquettes

- Lors du transfert du programme en RAM, le système d'exploitation choisit la valeur de L1 et calcule les valeur de L1+4, L1+8, ...

Codage à partir de l'adresse 0001 0000

0001 0000-> CODE_OP1 CONSTANTE_10
0001 0008 ->CODE_OP2 ADRESSE_L1
0001 0010 ->CODE_OP3
0001 0014-> CODE_OP4 CONSTANTE_20
0001 001C ->CODE_OP5 ADRESSE_L1+4
0001 0024 ->CODE_OP6
0001 0028 ->CODE_OP7 ADRESSE_L1
0001 0030 ->CODE_OP8
0001 0034 ->CODE_OP9 ADRESSE_L1+4
0001 003C ->CODE_OP10
0001 0040 ->CODE_OP11
0001 0044 ->CODE_OP12 ADRESSE_L1+8
0001 004C ->CODE_OP13

Traduction en micro-instructions

- Chaque instruction assembleur est traduite en micro-instructions qui est un langage encore plus rudimentaire que l'assembleur.
- Liste des micro-instructions
  - registre0 <- registre1 copie registre1 dans registre0
  - registre0 <- [registre1] copie le contenu de la case mémoire registre1 dans registre0
  - registre0 -> [registre1] écrit registre0 dans la case mémoire numéro registre1
  - registre0 <- registre1+registre2 calcule la somme de registre1 et de registre2 et stocke le résultat dans registre2
  - registre0 <- registre1+4

1) mov r0,#10

1) ri <- co ; co <- co+4
2) décodage
3) r0<- [co] ; co<-co+4

2) ldr r1,L1

4) ri <- [co]; co <- co+4
5) décodage
6) rp <-[co]; co<- co+4
7) r1 <- [rp]

3) str r0,[r1]

8) ri <- [co]; co <- co+4
9) décodage
10) r0 -> [r1]

4) mov r0,#20

11) ri <- co ; co <- co+4
12) décodage
13) r0<- [co] ; co<-co+4

5) ldr r1,L1+4

14) ri <- [co]; co <- co+4
15) décodage
16) rp <-[co]; co<- co+4
17) r1 <- [rp]

6) str r0,[r1]

18) ri <- [co]; co <- co+4
19) décodage
20) r0 -> [r1]

7) ldr r0,L1

21) ri <- [co]; co <- co+4
22) décodage
23) rp <-[co]; co<- co+4
24) r0 <- [rp]

8) ldr r1,[r0]

25) ri <- [co]; co <- co+4
26) décodage
27) r1<-[r0]

9) ldr r0,L1+4

28) ri <- [co]; co <- co+4
29) décodage
30) rp <-[co]; co<- co+4
31) r0 <- [rp]

10) ldr r2,[r0]

32) ri <- [co]; co <- co+4
33) décodage
34) r2<-[r0]

11) add r1,r1,r2

35) ri <- [co]; co <- co+4
36) decodage
37) r1<-r2+r3

12) ldr r0,L1+8

35) ri <- [co]; co <- co+4
36) decodage
37) rp <-[co]; co<- co+4
38) r0 <- [rp]

13) str r1,[r0]

38) ri <- [co]; co <- co+4
39) décodage
40) r0 -> [r1]

Conclusion

- Un ensemble de 5 micro-intructions suffit à traduire un tel programme.
- L'assembleur est donc traduit à la volée dans un tel langage plus simple.