Chap. II : Initiation au Langage Machine

Laurent Poinsot

UMR 7030 - Université Paris 13 - Institut Galilée

Cours "Architecture et Système"

Dans cette partie du cours, nous allons étudier la programmation en langage machine d'un microprocesseur. L'étude complète d'un processeur réel, comme un Core Duo, dépasse largement le cadre de ce cours : le nombre d'instructions et de registres est très élevé. Nous allons ici nous limiter à un langage machine " jouet ", c'est-à-dire un langage machine très simplifié qui nous permettra de comprendre les principes de fonctionnement des langages machines.

Le langage machine ou code machine est la suite de bits qui est interprétée par le processeur de l'ordinateur lors de l'exécution d'un programme. C'est le langage natif du processeur, et le seul qui soit reconnu par celui-ci.
Chaque processeur possède son propre langage machine. Si un processeur A est capable d'exécuter toutes les instructions du processeur B, on dit que A est compatible avec B. L'inverse n'est pas forcément vrai, A peut avoir des instructions supplémentaires que B ne connaît pas.

Plan du chapitre

1. Un processeur idéalisé ;
2. L'adressage ;
3. Les instructions du langage machine ;
4. Équivalence entre les langages LM0 et C ;

Nous définissons un microprocesseur simplifié que nous appellerons MP0. Il saura exécuter des programmes dans un langage machine, également simplifié, appelé LM0.
MP0 ne possède que des registres huit bits (c'est bien trop peu pour un vrai microprocesseur). Il ne peut donc que manipuler des octets et accéder à des adresses de 0 à 255.

Les registres de MP0

1. Un compteur ordinal (CO) : contient l'adresse de la prochaine instruction à exécuter ;
2. Un registre d'état (SR) qui contient les bits Z et N.
   - Quand Z = 1, cela signifie que la dernière instruction a produit un résultat égal à zéro ;
   - Quand N = 1, cela signifie que la dernière instruction a produit un résultat négatif ;
3. Deux registres de données D0 et D1 qui stockent chacun un octet à traiter ;
4. Deux registres d'adresse A0 et A1 qui stockent chacun une adresse pour accéder à des octets de la RAM ;
5. Un pointeur de pile PP : registre d'adresse réservé pour la pile.

Informations complémentaires (1/3)

- Une adresse est un nombre entier représentant un endroit précis d'une partie de la mémoire ;
- Un registre est une zone mémoire où l'on peut stocker une valeur ;
- Un pointeur est une zone mémoire contenant comme valeur une adresse.

Informations complémentaires (2/3)

Une pile est une zone de mémoire dans laquelle on peut stocker temporairement des données. Il s'agit également d'un moyen d'accéder à des données en les empilant, telle une pile de livres, puis en les dépilant pour les utiliser. Ainsi il est nécessaire de dépiler les valeurs stockées au sommet (les dernières à avoir été stockées) pour pouvoir accéder aux valeurs situées à la base de la pile.
En réalité il s'agit d'une zone de mémoire et d'un pointeur qui permettent de repérer le sommet de la pile. Le pointeur contient l'adresse de la dernière valeur empilée. La pile est de type LIFO (Last In First Out), c'est-à-dire que la première valeur empilée sera la dernière sortie. (Si vous empilez des livres, il vous faudra les dépiler en commençant par enlever les livres du dessus. Le premier livre empilé sera donc le dernier sorti !)

Informations complémentaires (3/3)

Les instructions PUSH et POP sont les instructions qui servent à empiler et dépiler les données dans une pile. " PUSH registre " met le contenu du registre dans la pile (empilement) et incrémente le pointeur de la pile (il y a un élément en plus dans la pile). " POP registre " récupère le contenu de la pile, le stocke dans le registre (dépilage), puis décrémente le pointeur de la pile (il y a un élément de moins dans la pile).

Format numérique des instructions

- Chaque instruction de LM0 est codée sur 2 octets :
  - Premier octet : code le type de l'opération et le type de son ou ses opérandes (les valeurs) ;
  - Second octet : peut être utilisé pour coder une valeur associée à une opérande.
- Par exemple, " mettre la valeur 23 dans D0 "
  - Premier octet : 121 qui code l'instruction " mettre la valeur [octet suivant] dans D0 " ;
  - Second octet : 23 (la valeur à mettre dans D0).
- S'il n'y a pas de valeur à associer à une opérande, alors le second octet est égal à 0 ;
- Par exemple, " mettre la valeur de D0 dans D1 "
  - Premier octet : 248 qui code l'instruction " mettre la valeur de D0 dans D1 " ;
  - Second octet : 0.

Format symbolique des instructions

- Comme il n'est pas pratique de désigner une instruction du langage machine par son numéro, on lui attribuera un nom et une syntaxe.
- On écrira chaque instruction sous forme symbolique (c'est-à-dire en langage assembleur). Par exemple, MOVE (transfert), ADD (addition), SUB (soustraction), JMP (saut), etc.
- On ajoutera l'opérande après le nom symbolique. Par exemple, JMP #120.
- S'il y a deux opérandes, on les séparera avec une virgule. Par exemple, MOVE #100, D0.

Par adressage on entend la façon de coder l'accès aux donnés manipulées par le processeur. Les instructions de LM0 ne permettent de désigner qu'une seule opérande au maximum grâce à un numéro, l'autre opérande devant se référer à un registre. Il existe trois modes d'adressage :

1. Adressage immédiat ;
2. Adressage direct ;
3. Adressage indirect.

Adressage immédiat

1. Si on connait la valeur que l'on veut traiter, on peut la spécifier explicitement ;
2. La valeur désignée est stockée dans le second octet de l'instruction ;
3. Syntaxe : on préfixe la valeur avec le caractère dièse " # " ;
4. Par exemple : " MOVE #3, D0 " place la valeur 3 dans le registre D0. Code correspondant : 1er octet : " MOVE # ?, D0", 2e octet : " 3 ".

Adressage direct

1. L'adressage direct est l'accès à une valeur située à une adresse dont on spécifie le numéro ;
2. Syntaxe : on écrit simplement adresse (et rien d'autre) ;
3. Par exemple : " MOVE 100, D0 " place la valeur située à l'adresse 100 dans le registre D0. Code correspondant : 1er octet : " MOVE ?, D0 ", 2e octet : " 100 ".

Adressage indirect

1. L'adressage indirect est l'accès à une valeur située à une adresse mémorisée dans un registre d'adresse ;
2. Syntaxe : on spécifie le registre d'adresse entre parenthèses ;
3. Par exemple : " MOVE (A1), D1 " place la valeur située à l'adresse contenue dans le registre d'adresse A1 dans le registre D1. Code correspondant : 1er octet : " MOVE (A1), D1 ", 2e octet : " 0 ".

Types d'instructions

On peut regrouper l'ensemble des instructions en trois groupes :

1. les instructions de transfert telles que MOVE, PSH, POP ;
2. les opérations arithmétiques et logiques ADD, SUB, MUL, DIV, NOT, AND, OR, ...
3. les instructions de saut comme JMP, JEQ, JSR, ...

Instructions de transfert (1/4)

- L'instruction MOVE permet de transférer un octet d'un emplacement (registre ou case mémoire) à un autre ;
- Syntaxe : MOVE source, destination. Attention, " source " ou " destination " est nécessairement un registre parmi D0, D1, A0, A1 ;
- Par exemple :
  - MOVE #10, A1 (on place la valeur - représentant une adresse - 10 dans le registre A1)
  - MOVE D0, 123 (on place la valeur contenue dans D0 à l'adresse 123)
  - MOVE (A0), (A1) (on place la valeur se situant l'adresse contenue dans A0 dans la case mémoire dont l'adresse est contenue dans A1)
  - MOVE #10,110 n'est pas valide. On doit faire MOVE #10,D0 puis MOVE D0,110.

Instructions de transfert (2/4)

- De plus :
  - si la valeur transférée est nulle, le bit Z du registre d'état passe à 1 ;
  - si la valeur transférée est négative, le bit N du registre d'état passe à 1.

Instructions de transfert (3/4)

- Les instructions PSH et POP respectivement empile et dépile un octet grâce au registre PP ;
- L'instruction PSH décrémente PP et place la valeur de l'opérande à l'adresse pointée par PP. Ainsi l'instruction

  PSH source

  est équivalent à la séquence d'instructions

  MOVE source, (PP)

  puis

  SUB #1, PP

Instructions de transfert (4/4)

- L'instruction POP place la valeur pointée par PP dans l'opérande et incrémente PP. Ainsi l'instruction

  POP dest

  est équivalent à la séquence d'instructions

  ADD #1, PP

  puis

  MOVE (PP), dest

- Au démarrage de la machine, le registre PP contient l'adresse 255.

Opérations arithmétiques et logiques (1/6)

Les opérations arithmétiques et logiques permettent d'effectuer les calculs élémentaires (addition, soustraction, multiplication, *etc.*) ainsi que l'évaluation des conditions booléennes (conjonction, disjonction, négation, test de comparaison, *etc.*). Les opérations arithmétiques et logiques permettent la modification de la valeur de l'unique opérande ou bien le calcul d'une valeur à partir de deux opérandes. Dans le langage simplifié LM0, le résultat du calcul remplace la valeur de la deuxième opérande. Par exemple, " ADD D0, D1 " effectue la somme (" ADD " pour " addition ") des valeurs de D0 et D1, et place le résultat dans D1. Le résultat d'une opération affecte les bits Z et N du registre d'état de la façon habituelle (*i.e.*, Z prend la valeur 1 si le résultat de l'opération est nul, et *N* prend la valeur 1 si le résultat de l'opération est négatif).

Opérations arithmétiques et logiques (2/6) : Arithmétique

- L'addition : ADD source, destination. On effectue l'addition de *source* et de *destination*, et le résultat est placé dans *destination* ;
- La soustraction : SUB source, destination. On soustrait *source* à *destination*, le résultat est placé dans *destination* ;
- La multiplication : MUL source, destination. La valeur de " *destination* $\times$ *source* " est placée dans *destination* ;
- La division entière : DIV source, destination. On effectue la division entière de *destination* par *source*, *destination* prend alors comme valeur le résultat de ce calcul ;
- Exemples :
  - SUB #3, D0 : si D0 contenait 5, D0 passe à 2 ;
  - DIV #5, D0 : si D0 contenait 18, D0 passe à 3.
- *destination* doit être un registre car on va y stocker une valeur.

Opérations arithmétiques et logiques (3/6) : Logique

Les opérations logiques sont des opérations binaires : on travaille donc directement sur les représentations binaires.
La négation : NOT destination. Cette opération transforme le contenu de *destination* en substituant chaque bit à 0 de *destination* en un bit à 1, et réciproquement pour les bits à 1. Le résultat final est stocké dans *destination*. Par exemple, " NOT D0 " : si D0 contenait 55, D0 passe à 200. En effet, en format binaire nous avons

$$\begin{array}{ll} 00110111 & (55) \\ & NOT \\ \hline 11001000 & (200) \end{array} \qquad (1)$$

Opérations arithmétiques et logiques (4/6) : Logique

La conjonction : AND source, destination. Cette opération réalise la conjonction bits-à-bits selon la règle " $0\ AND\ x = x\ AND\ 0 = 0$ " quel que soit la valeur du bit x, et " $1\ AND\ 1 = 1$ ". Par exemple, " AND #179, D0 " : si D0 contenait 241, D0 passe à 177. En effet, en format binaire nous avons

$$\begin{array}{ll} 10110011 & (179) \\ 11110001 & (241) \\ & AND \\ \hline 10110001 & (177) \end{array} \qquad (2)$$

Opérations arithmétiques et logiques (5/6) : Logique

La disjonction : OR source, destination. Cette opération réalise la disjonction bits-à-bits selon la règle " $1\ OR\ x = x\ OR\ 1 = 1$ " quel que soit la valeur du bit x, et " $0\ OR\ 0 = 0$ ". Par exemple, " OR #179, D0 " : si D0 contenait 241, D0 passe à 243. En effet, en format binaire nous avons

$$\begin{array}{ll} 10110011 & (179) \\ 11110001 & (241) \\ & OR \\ \hline 11110011 & (243) \end{array} \qquad (3)$$

Opérations arithmétiques et logiques (6/6) : Comparaison

- Instruction de comparaison : CMP source, destination. Cette instruction effectue le même traitement que l'instruction SUB mais ne modifie pas la deuxième opérande ;
- Par contre, elle modifie les bits Z et N du registre d'état ;
- Par exemple, " CMP #3, D0 "
  - Z passe à 1 si D0 contient 3 (et Z passe à 0 sinon) ;
  - N passe à 1 si D0 contient une valeur < 3 (et N passe à 0 sinon) ;
- Cette instruction s'utilise en préparation d'une instruction de saut conditionnel, que l'on verra plus tard, dont l'exécution dépend de Z et de N.

Instructions de saut (1/6)

- Les instructions de saut modifient la valeur du compteur ordinal CO afin d'exécuter une autre série d'instructions que celle qui suivent l'instruction courante ;
- Les instruction de saut se regroupent en trois types :
  1. Saut inconditionnel : JMP destination ;
  2. Saut conditionnel : JEQ/JNE/JGT/... destination ;
  3. Saut vers un sous-programme : JSR destination et retour de sous-programme : RTS.

Instructions de saut (2/6) : Saut inconditionnel

- JMP destination : la prochaine instruction à exécuter est désignée par destination. (On " saute " vers l'instruction *destination*.)
- L'instruction " JMP destination " est équivalente à " MOVE destination, CO " c'est-à-dire que l'on remplace l'adresse de la prochaine instruction à exécutée (contenue dans CO) par *destination* ;
- Par exemple,
  - " JMP # 210 " : la prochaine instruction à exécuter est (codée par) 210 ;
  - " JMP 10 " : la prochaine instruction à exécuter est stockée à l'adresse 10 ;
  - " JMP A0 " : la prochaine instruction à exécuter est à l'adresse mémorisée dans le registre A0.

Instructions de saut (3/6) : Saut conditionnel

Les saut conditionnels ne sont effectués que si des conditions sur les bits Z et N du registre d'état sont réalisés.

Instruction	*Signification*	*Condition*
JEQ	Jump if EQual	$Z = 1$
JNE	Jump if Not Equal	$Z = 0$
JLT	Jump if Less Than	$N = 0$
JLE	Jump if Less or Equal	$N = 0$ *ou* $Z = 1$
JGT	Jump if Greater Than	$N = 1$
JGE	Jump if Greater or Equal	$N = 1$ *ou* $Z = 1$

(4)

Instructions de saut (4/6) : Saut conditionnel

- Un saut conditionnel se trouve généralement après une instruction de comparaison CMP ;
- Interprétation de la séquence des deux instructions :
  - " CMP #5, D0 " suivi de " JEQ #100 " : saut à l'instruction 100 si le contenu de D0 égale 5 ;
  - " CMP D0, D1 " suivi de " JLT #100 " : saut à l'instruction 100 si le contenu de D0 est $<$ au contenu de D1 ;
  - " CMP 88, D0 " suivi de " JGE #100 " : saut à l'instruction 100 si la valeur à l'adresse 88 est $\geq$ au contenu de D0.

Instructions de saut (5/6) : Saut vers un sous-programme (et retour)

- Saut (inconditionnel) vers un sous-programme : JSR destination. Cette instruction réalise les opérations suivantes :
  1. empile l'adresse de l'instruction suivante (celle contenue dans CO) ;
  2. puis saute à l'adresse désignée par l'opérande *destination*.
- Cette instruction correspond à la séquence d'instructions suivante : " ADD #2, CO " (on ajoute deux au contenu du compteur ordinal), " PSH CO " (on place la valeur de CO à l'adresse pointée par le pointeur de pile), puis on saute à l'instruction *destination* par " JMP destination " ;
- Retour de sous-programme RTS :
  1. dépile une adresse (normalement celle empilée par un JSR) et saute à cette adresse ;
  2. " RTS " est équivalent à " POP CO ".

Instructions de saut (6/6) : Saut vers un sous-programme (et retour)

110 : JSR #150
112 : ...
...
150 : ... (début du sous-programme)
...
162 : RTS (fin du sous-programme)

Ordre d'exécution des instructions : 110, 150, ..., 162, 112.

On peut faire en LM0 les mêmes choses que l'on fait en C. Les deux langages sont équivalents. Pour garantir cela, il suffit de montrer que l'on peut simuler dans le langage LM0, les instructions du langage C, à savoir,

- l'affection ;
- l'instruction conditionnelle if ;
- l'instruction de boucle while ;
- l'appel d'une fonction.

Affectation d'une variable

Le nom d'une variable en C correspond à une adresse. Affecter une valeur à une variable correspond à faire un transfert de valeur grâce à une (ou plusieurs) instruction(s) MOVE. Par exemple, quand en C, on a l'instruction " x = 1 ; ", en LM0, on fait l'équivalent grâce aux deux instructions " MOVE #1,D0 " puis " MOVE D0, 110 " (en associant x et l'adresse 110). On rappelle ici que l'on n'a pas le droit de faire " MOVE #1,110 " directement.

Exemple : L'affectation de variable à un résulat d'expression

L'instruction suivante
" x = 3 * y + 5 ; "
se simule en LM0 ainsi :
(x : adresse 100, y : adresse 101)

```
MOVE 101, D0
MUL #3, D0
ADD #5, D0
MOVE D0, 100
```

L'instruction conditionnelle if

L'instruction " if(...) [...] else [...] " se simule en LM0 ainsi :

En C	*En LM0*
if (test)	*... : CMP test* *... : Jtest siFaux*
[Instructions si vrai]	*... : [Instructions si vrai]* *... : JMP suite*
else	
[Instructions si faux]	*siFaux : [Instructions si faux]*
[suite du programme]	*suite : [suite du programme]*

(5)

L'instruction conditionnelle if : un exemple

Dans le programme ci-dessous, on utilise le registre D0 pour mémoriser la valeur de la variable x.

En C	*En LM0*
if $(x > 3)$	100 : $CMP\ \#3, D0$
$x = x - 5;$	102 : $JGE\ \#108$
else	104 : $SUB\ \#5, D0$
$x = x + 10;$	106 : $JMP\ \#110$
[...]	108 : $ADD\#10, D0$
	110 : [...]

(6)

La boucle while

L'instruction while (...) [...] se simule en LM0 ainsi :

En C	*En LM0*
while (*test*)	*boucle* : *CMP test*
	... : *Jtest siFaux*
[*Instructions si vrai*]	... : [*Instructions si vrai*]
	... : *JMP boucle*
[...]	*siFaux* : [...]

(7)

La boucle while : un exemple

En C	*En LM0*
while $(x > 10)$	104 : *CMP* $\#10, D0$
	106 : *JGE* $\#112$
$x = x - 1;$	108 : *SUB* $\#1, D0$
	110 : *JMP* $\#104$
[...]	112 : [...]

(8)