L'assembleur

Il est parfois nécessaire d'écrire directement des programmes en langage machine, par exemple quand un calcul précis doit être réalisé le plus vite possible par la machine. Pour cela, il n'est pas raisonnable (voir possible) décrire directement les mots binaires du langage machine. On utilisera alors **un langage *assembleur***, qui est le langage le plus bas niveau d'un ordinateur lisible par un humain. Ce langage fournit un certain nombre de facilités pour programmer, comme des étiquettes symboliques pour identifier des points de programme.

Le langage machine exprime les instructions des programmes par des combinaisons de bits, comme :

10110000 01100001

En langage d'assemblage, ces instructions sont représentées par des symboles mnémoniques bien plus lisibles, comme :

mov $0x61, %al

(qui signifie de mettre la valeur hexadécimale 61 (97 en décimal) dans le registre 'AL'.)

Vocabulaire

Jeux d'instructions des processeurs : ensemble des instructions-machine qu'un processeur d'ordinateur peut exécuter. Ces instructions-machines permettent d'effectuer des opérations élémentaires ou plus complexes. Le jeu d'instructions définit quelles sont les instructions supportées par le processeur. Le jeu d'instructions précise aussi quels sont les registres du processeur manipulables par le programmeur (les registres architecturaux).

Parmi les jeux d'instructions principaux, nous avons :

- x86 (16, puis 32 bits) ;
- AMD64 (version 64 bits de x86 par AMD, aujourd'hui la plus répandue de la famille x86) ;
- ARM (16, 32, puis 64 bits) ;
- RISC-V (32, 64 et 128 bits), un format ouvert.

Le jeu d'instructions x86 équipe tous les processeurs compatibles avec l'architecture Intel (qu'ils soient construits par Intel ou AMD).

Il a reçu plusieurs extensions dont le passage à une architecture 64 bits, AMD64 (appelé plus génériquement x86-64).

Pour ce cours, et à titre d'exemple, nous présentons ci-dessous le ***langage assembleur x86*** des microprocesseurs d'Intel.

Sections. Un programme x86 est constitué de trois parties, appeler *sections*. La première section délimitée par l'instruction **section .data**, définit une mémoire avec des déclarations de constantes, c'est à dire des zones de mémoire non modifiables. Chaque constante est déclarée de la manière suivante :

nom *type* *lval*

où **nom** est l'adresse symbolique de la constante et *type* est le nombre de cases mémoires allouées pour chaque valeur de la liste *lval*. Les types sont des puissances de deux d'octets (**db** pour un octet, **dw** pour deux octets, **dd** pour quatre octets, etc.). Par exemple, la **section .data** suivante initialise à l'adresse mémoire nommée **a** trois mots de 32 *bits* contenant les entiers 101, 278 et et 3569. Elle déclare également une constante b d'un octet contenant la valeur 42.

```
section .data
a dd 101, 278, 3569
b db 42
```

La section suivante, délimité par **section .bss** , permet de définir des variables, c'est à dire des zones modifiables initialisées ou non. Les déclarations de variables sont identiques à celles des constantes de la section **.data**, à ceci près que les zones mémoires ne sont pas nécessairement initialisées. Lorsqu'on souhaite simplement réserver de la place mémoire, on utilise les types **resb** (pour un octet), **resw** (pour deux octets), etc. La *valeur* donnée pour initialiser la zone mémoire représente alors le nombre de cases à réserver. Par exemple, la section suivante réserve deux mots de 16 *bits* à l'adresse x et un mot de 32 *bits* à l'adresse y, initialisé avec la valeur 7637.

```
section .bss
x resw 2
y dd 7637
```

Enfin, la **section .text** contient les instructions du programme. Chaque instruction peut être précédée d'une étiquette qui représente de manière symbolique l'adresse de cette instruction. En particulier, la section doit indiquer avec la directive **global** l'étiquette de l'instruction de départ du programme.

Registres. Le langage x86 à huit registres nommés **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**, **esp** et **ebp**. À ces registres s'ajoutent ceux pour manipuler les segments mémoires : **cs** et **eip** (segment de code et pointeur d'instruction), **ds** (segment de données), **ss** et **esp** (segment de pile et sommet de pile).

Le jeu d'instructions du langage x86 se compose d'instructions de transferts de données, d'opérations de calcul et d'instructions de saut. En voici ci-dessous une liste non-exhaustive.

Instructions de transfert. La principale instruction pour transférer des données entre des registres et/ou la mémoire est **mov**. Elle a la forme suivante :

mov destination, source

La source ou la destination peuvent être des registres ou des emplacements en mémoire. Par exemple :

```
mov eax, 42          ; copie la constante 42 dans eax
mov eax, ebx         ; copie le contenu de ebx dans eax
mov [x], eax         ; copie eax dans la variable x
```

Mais les copies de la mémoire vers la mémoire ne sont pas possibles directement (il faut passer par un registre).

Des instructions particulières permettent également de manipuler des données sur la pile, comme :

```
push source          ; copie source au sommet de la pile
pop destination      ; copie le sommet de la pile dans destination
```

La source ou la destination de ces instructions peuvent être soit des registres, soit des adresses de variables.

```
push 42              ; empile la constante 42
push eax             ; empile le contenu de eax
pop [x]              ; dépile à l'adresse de x
pop ebx              ; dépile dans ebx
```

Instructions de calcul. L'assembleur x86 permet d'écrire des opérations arithmétiques élémentaires (addition, soustraction, etc.), mais également des opérations logiques (et, ou, etc.). Par exemple, l'opération d'addition a la forme suivante.

add *destination*, *source*

Cette opération ajoute la valeur contenue dans *source* à celle contenue dans *destination* et stocke le résultat dans *destination*. Comme pour l'opération **mov**, la source et la destination peuvent être des registres ou des adresses, mais l'addition entre le contenu de deux adresses mémoire n'est pas possible.

Par exemple :

```
add eax, 10          ; ajoute 10 à eax
add eax, ebx         ; ajoute ebx à eax
add [x] , eax        ; ajoute eax à x
add ebx, [x]         ; ajoute x à ebx
```

L'instruction de soustraction **sub** ainsi que les opérations logiques (**and**, **or**, etc.) sont similaires. Seules les instructions de multiplication et de division (**mul** et **div**) sont plus restreintes car elles effectuent uniquement leur calcul entre le registre **eax** et un autre registre, case mémoire ou constante.

Instructions de saut. Pour contrôler le flot d'exécution[1], le langage x86 dispose d'instructions de comparaison et instructions de saut. L'instruction de comparaison a la forme suivante.

cmp *source1*, *source2*

L'opération effectue la soustraction *source1* – *source2*, mais ne stocke pas son résultat. Au lieu de cela, elle indique le résultat de la comparaison dans des registres spéciaux, appelés drapeaux (en anglais, flag) :

- Si le drapeau **zf (Zero Flag)** vaut 1, c'est que *source1* est égal à *source2*.
- Si le drapeau **sf (Sign Flag)** vaut 1, c'est que *source1* est inférieur à *source2*.

Ces drapeaux sont ensuite utilisés par les fonctions de saut. Ainsi, l'instruction de saut conditionnel **je** (pour *jump if equal*), de la forme suivante,

je *adr*	Permet de sauter à l'adresse *adr* si le drapeau **zf** vaut 1

jne *adr*	pour *jump if not equal*	Permet de sauter à l'adresse *adr* si le drapeau **zf** vaut 0
jg *adr*	pour *jump greater*	
jl *adr*	pour *jump less*	

```
cmp ax, bx              ; Comparaison : indicateurs mis à jour selon AX-BX
jl   quand_plus_petit   ; saut si AX < BX  (non signé)
jg   quand_plus_grand   ; saut si AX > BX  (non signé)
:quand_egaux
```

Enfin, il y a également une instruction de saut non conditionnel ***jmp*** qui permet de sauter à une adresse. Dans ces instructions, les adresses peuvent être contenues dans des registres, à des adresses mémoires ou être des constantes.

[1] Flot d'exécution : C'est l'ordre dans lequel les instructions d'un programme impératif sont exécutées.
La programmation impérative est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par l'ordinateur pour modifier l'état du programme.