Les instructions INTEL 64

L'objectif de cette section est de donner un bref aperçu des différentes instructions INTEL 64 afin de pouvoir écrire rapidement des programmes simples.

2.1. mouvements de données

Chaque instruction se traduit directement en une instruction binaire pour le processeur.

2.1.1. Syntaxe

instruction source
ou
instruction destination source

Types de l'opérande source

L'opérande source peut être de l'un des deux types suivants:
- registre
- mémoire

Elle est donné en adressage immédiat (valeur immédiate codée dans l'instruction) ou implicite (valeur qui n'apparaît pas dans l'instruction)

Types de l'opérande destination

Elle peut être soit de type registre ou mémoire.

Note : Une instruction ne peut pas avoir deux opérandes de type mémoire.

2.1.2. Utilisation de l'opérateur []

[***adresse***] représente la valeur stockée à l'adresse ***adresse***.
[***registre***] représente la valeur stockée à l'adresse contenue dans le registre ***registre***.

2.1.3. La directive dx

Elle permet de réserver de l'espace de mémoire dans les segments de données.
d***x*** permet de réserver un espaces, dont la taille dépend de ***x*** dans le segment de données ***.data*** (données initialisées).

valeur de x	**taille de l'espace réservé**
b	1 octet (byte)
w	1 mot (word = 2 octets)
d	1 double mot (4 octets)
q	1 quadruple mot (8 octets)
t	ten words (10 octets)

Pour stocker des informations dans un registre, nous disposons de la famille d'instructions assembleur `mov`. Par exemple, pour stocker la valeur décimale 2 dans le registre `EAX` (resp. `BX`, `CH`), on écrit `movd $2,%eax` (resp. `movw $2,%bx`, `movb $2,%ch`).
Dans cet exemple, on utilise la syntaxe ATT dans laquelle :

- le préfixe $ indique une opérande ne nécessitant pas d'interprétation (ici un entier) ;

- les noms des registres sont préfixés par le symbole %. Ceci permet d'utiliser du code assembleur directement dans un programme C sans risque de confusion avec un nom de macro ;
- la première opérande de l'instruction `mov` est la source et la seconde est la destination (la syntaxe Intel utilise la convention inverse) ;
- la taille des opérandes se lit sur le suffixe de l'instruction :

Le compilateur n'impose pas strictement l'usage du suffixe i.e. l'instruction `mov $2,ch` est valide tant qu'il est possible de déduire la taille des données déplacées (dans le doute, on utilise l'architecture 64 bits par défaut).

Remarque 3. Il existe plusieurs variantes de l'instruction `MOV` dont voici quelques exemples :

mov $1515, %AX
mov %BL, %DH

Il existe deux autres variantes de mov:

movzx dest, src (extension avec des 0 dans dest)
movsx dest, src (extension avec le bit de signe dans dest)

Nous reviendrons sur ce sujet plus loin dans ce cours. Pour l'instant poursuivons la présentation des notions nécessaires à l'écriture de notre premier programme assembleur.

2.2. Instructions arithmétiques

Instruction	**signification**
add op1, op2	op1 := op1 + op2
sub op1, op2	op1 := op1 - op2
neg reg	reg := -reg
inc reg	reg := reg + 1
dec reg	reg := reg - 1
imul op (signé ou ***mul*** non signé)	rdx:rax := rax * op («:» indique la concaténation)
imul dest, op	dest := dest * op
imul dest, op, immédiat	dest := op * immédiat
idiv op (div non signé)	rax := rdx:rax / op rdx := rdx:rax mod op

2.3. Opérations sur les bits

Instruction	**signification**
and op1, op2	op1 := op1 ET op2
or op1, op2	op1 := op1 OU op2
xor op1, op2	op1 := op1 OU EXCLUSIF op2
not reg	reg := NON reg
shl reg, immédiat	reg := reg << immédiat

shr reg, immédiat	reg := reg >> immédiat
sal reg, immédiat	reg := reg << immédiat signé
sar reg, immédiat	reg := reg >> immédiat signé
rol reg, immédiat	reg := reg decalageCirculaireGaucheDe immédiat
ror reg, immédiat	reg := reg decalageCirculaireDroiteDe immédiat
rcl reg, immédiat	reg:CF := reg:CF decalageCircGauchede immédiat
rcr reg, immédiat	reg:CF := reg:CF decalageCircDroitede immédiat

2.4. Comparaisons et branchements

Instruction	**signification**
cmp op1, op2	calcul de op1 - op2 et de ZF, CF et OF
jmp op	branchement inconditionnel à l'dresse op
jz op	branchement à l'adresse op si ZF = 1
jnz op	branchement à l'adresse op si ZF = 0
jo op	branchement à l'adresse op si OF = 1
jno op	branchement à l'adresse op si OF = 0
js op	branchement à l'adresse op si SF = 1
jns op	branchement à l'adresse op si SF = 0
jc op	branchement à l'adresse op si CF = 1
jnc op	branchement à l'adresse op si CF = 0
jp op	branchement à l'adresse op si PF = 1
jnp op	branchement à l'adresse op si PF = 0

2.5. Les instructions de branchement après cmp op1, op2
2.5.1. entiers signés

Instruction	**signification**
je op	branchement à l'adresse op si op1 = op2
jne op	branchement à l'adresse op si op1 <> op2
jl op (jnge)	branchement à l'adresse op si op1 < op2
jle op (jng)	branchement à l'adresse op si op1 <= op2
jg op (jnle)	branchement à l'adresse op si op1 > op2
jge op (jnl)	branchement à l'adresse op si op1 >= op2

2.5.2. entiers non signés

Instruction	signification
je op	branchement à l'adresse op si op1 = op2
jne op	branchement à l'adresse op si op1 <> op2
jb op (jnae)	branchement à l'adresse op si op1 < op2
jbe op (jna)	branchement à l'adresse op si op1 <= op2
jb op (jnbe)	branchement à l'adresse op si op1 > op2
jbe op (jnb)	branchement à l'adresse op si op1 >= op2

2.5.3. Exemples: instructions de branchement

Voici quelques exemples de structures conditionnelles classiques et leurs implantations en assembleur :

```
; si X>Y alors              ; si X<Y alors
;   <instructions>          ;   <instructions>
; fsi                       ; fsi
    mov X, reg1                 mov X, reg1
    mov Y, reg2                 mov Y, reg2
    cmp reg2, reg1              cmp reg2, reg1
    jna fsi1  ; not above       jnb fsi2  ; not below
; <instructions>            ; <instructions>
fsi1:                       fsi2:

; si X=Y alors              ; si X=0 alors
;   <instructions>          ;   <instructions>
; fsi                       ; fsi
    mov X, reg1                 mov X, reg1
    mov Y, reg2                 and reg1, reg1   ; plus rapide que cmp reg1, 0
    cmp reg1, reg2              jnz fsi4
    jne fsi3                ; <instructions>
; <instructions>            fsi4:
fsi3:
```

En combinant sauts conditionnel et inconditionnel, on peut mettre en place des structures plus complexes comme par exemple :

```
; si X=0 alors
;   <instructions_alors>
; sinon
;   <instruction_sinon>
; fsi
    mov X, AX
    and AX, AX
    jne sinon
; <instructions_alors>
    jmp fsi5
sinon:
; <instructions_sinon>
```

fsi5:

2.5.4. Instruction loop

L'instruction loop nécessite l'emploi des registres %rcx comme compteur. À chaque itération de boucle, le registre %rcx est automatiquement décrémenté. Si après décrémentation, %rcx est nul, on sort de la boucle pour exécuter l'instruction suivante ; sinon loop branche en fonction de son opérande (label ou saut court 8 bits). Par exemple, on peut écrire :

```
        movq $10,%rcx
        movq $0 ,%rax
DebutBoucle:
        add %rcx,%rax
        loop %rcx
```

pour faire la somme des entiers de 0 à 10 inclut.

Attention :
si le registre — disons %cx — est nul au premier tour, il est décrémenté et sa valeur devient 65535. Dans ce cas de figure, on peut attendre un bon moment la sortie de boucle ! De même, le registre de boucle doit être modifié dans ce cadre avec grande prudence.

L'usage de ***loop*** est souvent déconseillé car elle ne permet que des sauts au plus égaux à 127.

On peut aussi utiliser les instructions décrites dans le tableau suivant

Instruction	description
loop	décrémente RCX et saut si ce registre est non nul
loope	décrémente RCX et saut si ce registre est non nul
	et si ZF=1
loopz	décrémente RCX et saut si ce registre est non nul
	et si ZF=1
loopne	décrémente RCX et saut si ce registre est non nul
	et si ZF=0
loopnz	décrémente RCX et saut si ce registre est non nul
	et si ZF=0

Exercices

Exerice 1
Que fait le programme suivant?

```
.data
UnNom :
      .long 43
      .long 54
      .long 23
      .long 32
      .long 76
      .string "hello world" /*
      .float 3.14
UnAutre:
          .space 4
.text
.globl _start
_start:
         movl $5, %eax
         movl $0, %ebx
         movl $UnNom, %ecx
top:   addl (%ecx), %ebx

         addl $4, %ecx
         decl %eax
         jnz top
done:    movl %ebx, UnAutre

        movl    $0,%ebx
        movl    $1,%eax
        int     $0x80
```

Le programme précédent est pour une architecture INTEL 32. Réécrire ce programme en utilisant les registres 64 bits et en manipulant des entiers sur 64 bits (archi Intel 64)

Exercice 2
On se propose de calculer le plus grand commun diviseur — pgcd — de deux entiers par l'algorithme d'Euclide. Cet algorithme repose sur le fait que, étant donné deux entiers *a* et *b*, le pgcd de *a* et de *b* est égal au pgcd de *b* et de *r* où *r* est le reste de la division euclidienne de *a* par *b* et ceci tant que *r* est différent de zéro. Le pgcd est alors le dernier diviseur utilisé. Écrivez un programme assembleur qui, à partir de deux entiers stockés dans le segment de données, calcule leur pgcd (on suppose que ces entiers sont tous deux inférieurs à $2^{64}-1$).
Pour information, en assembleur le reste de la division euclidienne peut être obtenu par l'instruction `div` et la comparaison par l'instruction `cmp`.
Calculer le pgcd de 13481754 et de 1234715.

Exercice 3
Soit un entier a dans le segment de données. Écrire un programme qui vérifie si a est premier ou pas.

Exercice 4
Soit un entier *a* dans le segment de données. Écrire un programme qui donne le nombre des nombres premiers inférieurs à *a*.