Chapitre 15

Les instructions du mode 64 bits

Ayant saisi le principe de compatibilité des microprocesseurs d'*Intel*, on peut penser que l'on va retrouver dans le mode 64 bits le même jeu d'instructions en code machine que d'habitude, avec possibilité d'adresse et d'opérande sur 64 bits, ainsi que des instructions nouvelles. En fait ce n'est pas tout à fait le cas : certains codes opération ont disparu dans ce mode (conduisant à une faute « code opération non valide » lorsqu'ils sont utilisés) ; la signification de certains codes opération a changée ; la signification d'un des modes d'adressage a changé. Est-ce dû au fait que le mode 64 bits a été conçu par AMD et non par *Intel* ?

Ceci a évidemment des conséquences pour les programmes. Les programmes doivent systématiquement être recompilés (lorsqu'ils sont écrits, disons, en langage C) ou réassemblés (lorsqu'ils sont écrits en langage d'assemblage) avant de pouvoir être exécutés en mode 64 bits alors que ce n'est pas le cas en mode long compatible (d'où le nom). Mais une simple recompilation ou un simple réassemblage n'est pas suffisant : certains programmes exigent une adaptation. Beaucoup d'adaptations seront transparentes au programmeur, le compilateur ou l'assembleur faisant le travail à sa place, mais ceci n'est pas toujours le cas.

15.1 Les instructions en mode 64 bits

15.1.1 Certains changements dans les codes opération

Certaines instructions disparaissent purement et simplement en mode 64 bits :

Instruction	Opcode	Signification
AAA	0x37	ASCII Adjust after Addition
AAD	0xD5	ASCII Adjust AX before Division
AAM	0xD4	ASCII Adjust AX after Multiplication
AAS	0x3F	ASCII Adjust after Substraction
BOUND	0x62	Check Array Against Bounds
CALL (far)	0x9A	Far absolute
DAA	0x27	Decimal Adjust AL after Addition
DAS	0x2F	Decimal Adjust AL after Substraction
INTO	0xCE	Interrupt to Overflow Vector
JMP (far)	0xEA	Jump Far (absolute)
POP DS	0x1F	
POP ES	0x07	
POP SS	0x17	
POPA, POPAD	0x61	POP ALL to GPR Words or Doublewords
PUSH CS	0x0E	
PUSH DS	0x1E	
PUSH ES	0x06	
PUSH SS	0x16	
PUSHA, PUSHAD	0x60	Push All to GPR Words or Doublewords
SALC	0xD6	Set AL According to CF
SYSENTER	0x0F 34	System Call
SYSEXIT	0x0F 35	System Return

Certaines disparaissent car leur opcode est réutilisé :

Instruction	Opcode	Signification	Nouvelle utilisation
ARPL	0x63	Adjust Requestor Privilege Level	MOVSXD
DEC	0x48-4F	Decrementation	REX prefix
INC	0x40-47	Incrementation	REX prefix
LDS	0xC5	Load DS Far Pointer	VEX prefix
LES	0xC4	Load ES Far Pointer	VEX prefix

Commentaires.- 1^o) Pour les instructions INC et DEC, seules disparaissent les codages de ces instructions sur un octet concernant les registres généraux, par exemple INC AX. On peut les coder autrement en utilisant l'opcode 0xFF suivi de l'octet ModR/M.

- 2^o) Le code opération 0x90 de l'instruction NOP continue à exister par compatibilité ascendante mais il s'agit en fait d'un alias de l'instruction XCHG EAX, EAX.

15.1.2 Nouvelle interprétation des déplacements

Contexte.- Pour afficher un caractère au début de la deuxième ligne de l'écran, nous avons utilisé, en mode protégé, l'instruction suivante en adressage direct :

```
movb %al,(0xB80A0)
```

c'est-à-dire que nous avons placé le contenu du registre `AL` à l'adresse absolue `0xB80A0`, l'adresse de base de `DS` étant 0.

L'adresse de base du segment (de code) étant toujours 0 en mode 64 bits, on est tenté d'utiliser la même instruction pour effectuer la même opération. Mais cela ne marche pas.

Déplacement.- Dans le cas de l'adressage direct, l'opérande (la constante placée entre crochets dans la représentation en langage symbolique) est un *déplacement* `disp`. La signification de celui-ci est différente suivant le mode d'opération utilisé :

— En mode réel, l'adresse correspondante est :

$$\texttt{CS} \times \texttt{0x10} + disp$$

— En mode protégé et en mode compatible, on a :

$$base + disp$$

où *base* est l'adresse de base du segment des données de sélecteur `DS`.

— En mode 64 bits, le déplacement est relatif au registre d'instruction `RIP` :

$$\texttt{RIP} + disp$$

`RIP` contenant l'adresse de l'instruction qui suit l'instruction dans laquelle on utilise le déplacement. Le déplacement est toujours un entier relatif codé sur 32 bits mais cela permet de parcourir tout l'espace adressable puisque `RIP` contient une valeur codée sur 64 bits.

On parle de **mode d'adressage relatif au RIP** (*RIP relative addressing*).

Exemple.- Reprenons notre premier exemple de code écrit en mode 64 bits :

```
org  0x60000
     use64
mov  rax,'L O N G '
mov  [0xB80A0],rax
jmp  $
```

dont l'assemblage conduit à :

```
48 B8 4C 20 4F 20 4E 20 47 20
48 89 05 8F 80 05 00
EB FE
```

L'instruction `JMP` suivant l'instruction `MOV` qui nous intéresse a donc un décalage de 17, soit `0x11`, relativement à l'origine `0x60000`. Le contenu de `RIP` est donc `0x60011`. Le décalage de la seconde instruction `MOV` est `0x5808F`, comme on peut le voir sur le code machine.

Le contenu de `RAX` sera donc placé en :

```
0x60000 + 0x11 + 0x5808F = 0xB80A0
```

qui est bien ce que l'on désire.

Cas du langage d'assemblage.- Comme nous venons de le voir, l'assembleur nous aide : on place comme déplacement ce que l'on aimerait obtenir et l'assembleur traduit en code machine ce qu'il faut.

Cette aide est bienvenue mais a ses limites : il faut connaître l'emplacement de l'instruction. Nous avons utilisé la directive `ORG` dans ce cas puisque nous savions où allait être placé le programme. Mais ce n'est pas toujours le cas.

Le programme ainsi obtenu n'est pas relogeable.

15.2 Exemples de programmes

15.2.1 Programme d'application en 64 bits

Jusqu'ici nous avons surtout écrit des programmes dits système. Maintenant que nous avons comment passer au mode 64 bits et revenir au mode réel, nous pouvons remplacer notre programme d'exemple en mode 64 bits, à savoir l'affichage de 'LONG', par ce que nous voulons.

Écrivons, par exemple, un programme effectuant une addition sur 64 bits et plaçant le résultat à notre adresse habituelle `6000:0`, pour pouvoir vérifier que cela a bien été effectué lorsqu'on est revenu en mode réel.

Il suffit de remplacer, dans le programme `boot5.asm`, l'exemple en mode 64 bits :

```
;
; Sample in 64-bit mode
;
        use64
suiteL:
        mov     rax,'L O N G '
        mov     ebx,0B80A0h
        mov     [ebx],rax
```

par :

```
;
; Sample in 64-bit mode
;
        use64
suiteL:
        mov     rax,12345678h
        shl     rax,32
        add     rax,12345678h
        mov     ebx,60000h
        mov     [ebx],rax
```

Pour ne pas placer de valeur immédiate de plus de 32 bits dans le registre 64 bits `RAX`, on y place une constante 32 bits, que l'on déplace dans le double mot de poids fort, à laquelle on ajoute une autre constante 32 bits (ici la même), puis on place le contenu du registre 64 bits à l'emplacement voulu.

Après avoir assemblé le programme, disons `boot6.asm`, avec `FASM` et l'avoir placé dans le répertoire BOOT de notre clé USB démarrant sur MS-DOS, on exécute le programme :

```
C:\BOOT>debug
-D 6000:0 L 10
6000:0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-q

C:\BOOT>boot6.com

C:\BOOT>debug
-D 6000:0 L 10
6000:0 78 56 34 12 78 56 34 12-00 00 00 00 00 00 00 00 xV4.xV4.........
-q
```

ce qui confirme que l'addition a bien été effectuée sur 64 bits et que le transfert à l'adresse `6000:0` a bien été effectué sur 64 bits en une seule instruction.