Deuxième partie

Programmation système I : le mode réel

Nous avons vu dans la première partie comment écrire des programmes d'« application », c'est-à-dire s'exécutant sur un système d'exploitation. Nous allons voir maintenant comment démarrer un microprocesseur en mode 64 bits.

Le démarrage d'un microprocesseur *Intel* à architecture x86-64 ressemble au lancement d'une fusée à trois étages :

— Lors d'une réinitialisation, un tel microprocesseur se trouve dans un certain état : *Intel* parle de **mode**, et de **mode réel** pour ce premier mode.

Ce mode est compatible avec un `8086`, c'est-à-dire qu'un programme écrit en code machine pour un microprocesseur `8086` s'y exécutera.

Ceci n'empêche pas l'existence d'instructions nouvelles, par rapport au `8086`, exécutables dans ce mode.

— Une instruction de transfert `mov` vers un nouveau registre (c'est-à-dire qui n'existe pas sur le `8086`), de capacité 32 bits alors que les registres du `8086` ont tous une taille de 16 bits, appelé `CR0` pour *Control Register*, permet de passer au **mode protégé**, qui est le mode par excellence des microprocesseurs *Intel* d'**architecture IA-32**, avec 32 pour 32 bits.

Cette variante de l'instruction `mov` ne s'exécute pas sur un `8086` : nous voyons donc là une des instructions nouvelles. Le mode protégé exige l'initialisation d'un certain nombre de tables pour fonctionner, ce qui ne peut donc s'effectuer qu'en mode réel, d'où la nécessité d'autres nouvelles instructions.

— Une fois en mode protégé on peut, si on le désire, pour un microprocesseur d'architecture x86-64 ou **IA-32e** (avec un 'e' pour *Extended*), mais pas avec l'architecture IA-32, passer dans un nouveau mode, appelé **mode long**, comportant deux sous-modes, dont l'un est appelé **mode 64 bits**.

Bibliographie

Patrick Cégielski, **Conception des systèmes d'exploitation : Le cas linux. Deuxième édition**, Eyrolles, XIII + 680 p., septembre 2004.

Chapitre 7

Le mode réel

Nous avons vu que, lors d'une réinitialisation, un microprocesseur *Intel* se trouve toujours dans le *mode réel*. Commençons, dans ce chapitre, par décrire le mode réel.

7.1 Caractéristiques du mode réel

Il n'y a rien à dire sur le mode réel pour les microprocesseurs 8086/8088 ou 80186 : c'est le seul mode dans lequel ils peuvent fonctionner, étudié par exemple dans :

https://www.lacl.fr/cegielski/micro.html

Il existe quelques instructions nouvelles pour le 80286, permettant le passage au mode protégé. Mais c'est surtout à partir du 80386 qu'un changement intervient.

Le fonctionnement en mode réel est identique à celui des 8086/8088 en ce qui concerne la manière d'interpréter les adresses et de prendre en compte les interruptions. L'espace mémoire disponible est limité à 1 MiO. Un déplacement ne peut pas dépasser ± 64 KiO, taille maximale d'un segment dans ce mode.

Par contre on peut utiliser les nouveaux registres 32 bits pour les données et utiliser certaines des nouvelles instructions, par exemple celles portant sur le traitement des chaînes de bits.

Commençons par voir les registres 32 bits.

7.2 Les registres 32 bits en architecture IA-32

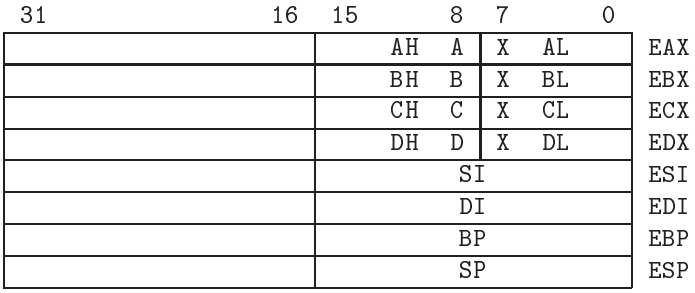
Intel a conçu un certain nombre de microprocesseurs 32 bits : le 80386 en premier puis le 80486, le *pentium* et d'autres. Du point de vue du programmeur, on parle d'**architecture IA-32** puisque les programmes s'exécutent indifféremment sur l'un ou l'autre de ces microprocesseurs, y compris sur les microprocesseurs x86-64.

Comme pour un 8086, il existe trois types de registres internes :

— les registres d'usage général, dits aussi *registres généraux* ;
— les registres de segment ;
— les registres spéciaux.

7.2.1 Les registres généraux

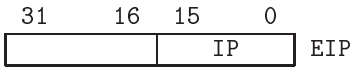
Une différence avec un 8086 est la capacité plus élevée des registres généraux. Par contre il y en a le même nombre, huit, que sur le 8086 :



On retrouve les noms des registres du 8086 avec un suffice 'E' (pour *Extended*) pour une taille de 32 bits. On peut accéder au mot de poids faible de ces huit registres EAX, EBX, ECX, EDX, ESI, EDI, EBP et ESP sous les noms connus pour le 8086, à savoir AX, BX, CX, DX, SI, DI, BP et SP. On peut accéder aux octets, de poids faible et de poids fort, des quatre premiers registres AX, BX, CX et DX sous les noms déjà connus pour le 8086 et rappelés sur la figure ci-dessus.

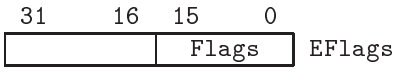
7.2.2 Les registres spéciaux

Le pointeur d'instruction.- Le pointeur d'instruction, maintenant appelé **EIP** (pour *Extended IP*), est également étendu à 32 bits :



Il contient à tout moment le décalage de la prochaine instruction à exécuter. Cependant, seuls les 16 premiers bits sont utilisés en mode réel.

Le registre des indicateurs d'états.- Là encore, le registre des indicateurs, maintenant appelé `EFlags` (pour *Extended Flags*), est étendu à 32 bits :



mais seuls les 16 premiers bits sont disponibles en mode réel.

Les indicateurs sont les suivants :

17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Seuls les indicateurs `CF` (*Carry Flag*), `PF` (*Parity Flag*), `AF` (*Auxiliary Flag*), `ZF` (*Zero Flag*), `SF` (*Sign Flag*), `TF` (*Trap Flag*), `IF` (*Interrupt Flag*), `DF` (*Direction Flag*) et `OF` (*Overflow Flag*), déjà présents sur le `8086`, ont un sens en mode réel.

Autres registres spéciaux.- Nous verrons, à propos du passage au mode protégé, qu'il existe d'autres registres spéciaux accessibles en mode réel, mais utiles uniquement pour le passage au mode protégé.

7.2.3 Six registres de segment

La largeur des registres de segment, à savoir 16 bits, n'a pas changée par rapport au `8086`. Par contre il y en a deux de plus : **FS** et **GS** en plus de `CS`, `SS`, `DS` et `ES`. Ces nouveaux registres de segment servent à charger des segments de données, tout comme `DS` et `ES`.

7.2.4 Types de données

Les trois tailles des données élémentaires sont :

— l'*octet* de huit bits,
— le *mot* de seize bits (deux octets),
— le **mot double** de trente-deux bits (quatre octets ou deux mots).

Intel affine cette classification et donne des noms à de nombreux types de données, dont le détail est donné en appendice.

7.3 Les instructions

7.3.1 Noms des instructions en langage symbolique

Le jeu d'instructions peut être subdivisé en neuf catégories fonctionnelles :

— transfert de données ;
— opérations arithmétiques ;
— décalage et rotations ;
— traitement des chaînes de caractères ;
— traitement des bits ;
— support des langages de haut niveau ;
— support des systèmes d'exploitation ;
— contrôle du processeur.

La plupart de ces instructions sont déjà présentes sur le `8086`. Nous ne détaillerons pas la sémantique de celles-ci. La liste des noms des instructions en langage symbolique *Intel* est donnée en appendice.

7.3.2 Les modes d'adressage

Le nom symbolique du code opération (*opcode*) d'une instruction est suivi des opérandes, dont la forme est très variée :

— Dans l'**adressage immédiat**, l'opérande ou l'un des opérandes est une constante (interprétée comme un entier), l'autre opérande éventuel faisant partie du code opération, par exemple :

```
IMUL EBX,5
```

en notation *Intel* et :

```
IMUL $5,%EBX
```

en notation AT&T.

La constante est ici un entier relatif (ce qu'indique le 'i' du nom symbolique), codé sur 32 bits (d'après le nom du registre).

— Dans l'**adressage à registre**, l'opérande est un registre ou les deux opérandes sont des registres, par exemple :

```
INC EAX
```

en notation *Intel* et :

```
INCL %EAX
```

en notation AT&T.

La taille de l'opérande est précisée par le nom du registre en notation *Intel*. Il y a redondance en notation AT&T puisque le suffixe 'l' le précise également.

Évidemment les deux registres doivent être de même taille dans le cas de deux opérandes.

— On parle d'**adressage à mémoire** dans les autres cas, c'est-à-dire lorsqu'un élément de la mémoire vive est impliqué. Il existe neuf cas :

— Dans l'**adressage direct**, le décalage de l'adresse est un **déplacement** (*displacement* ou *distance* en anglais), c'est-à-dire une valeur immédiate signée sur 8, 16 ou 32 bits, par exemple :

```
INC word ptr [500]
```

en notation *Intel*.

On a besoin, en notation *Intel*, de l'un des préfixes `byte ptr`, `word ptr` ou, ce qui est nouveau à l'architecture IA-32, **doubleword ptr** en langage symbolique pour spécifier la taille de l'opérande (les codes opération en langage machine, eux, sont différents).

En notation AT&T, on écrit :

```
INCL 500
```

sans le signe '$', qui désigne une constante et non une adresse, c'est le suffixe 'l' qui précise la taille.

— Dans l'**adressage indirect par registre**, le décalage est le contenu d'un registre, par exemple :

```
MOV [ECX],EDX
```

en notation *Intel*, où l'adresse est le contenu du registre `ECX` et la taille est spécifiée par le second opérande (le registre `EDX`).

En notation AT&T, on écrit :

```
MOVL %EDX,(%BX)
```

Les seuls registres permis en mode réel sont `BX` et `BP`.

Le registre de segment par défaut pour `BP` est `SS` et `DS` pour les autres registres. On peut, cependant, changer ce segment par défaut par un autre segment en utilisant un **préfixe de segment**, en écrivant, par exemple :

```
MOV AX,FS:[ECX]
```

en notation *Intel* et :

```
MOVW %FS:(%ECX),%AX
```

en notation AT&T.

— Dans l'**adressage par base**, le contenu d'un registre, dit **registre de base**, est ajouté à un déplacement pour obtenir le décalage, comme dans :

```
MOV ECX,[EAX+24]
```

en notation *Intel* et :

```
MOVL 24(%EAX),%ECX
```

en notation AT&T.

— La sémantique de l'**adressage par index**, lorsque celui-ci est utilisé seul, ressemble à l'adressage par base : le décalage est la somme d'un déplacement et du contenu d'un registre. On utilise l'un des registres `SI` ou `DI` dans le cas du mode réel.

Si la sémantique est presque la même que pour l'adressage par base, la syntaxe en langage symbolique *Intel*, elle, est différente :

```
MOV ECX,24[SI]
```

— Dans l'**adressage par base et index**, le décalage est obtenu comme somme des contenus d'un registre de base et d'un registre d'index :

```
MOV ECX,[SI][BX]
```

Pour résumer, en mode réel, une adresse est obtenue en spécifiant un registre de segment et un décalage : elle est égale à 16 fois le contenu du registre de segment plus le décalage. Le décalage est calculé en fonction du mode d'adressage par :

```
AE = [Base] + [Index] + Déplacement
```

Plus précisément, on a :

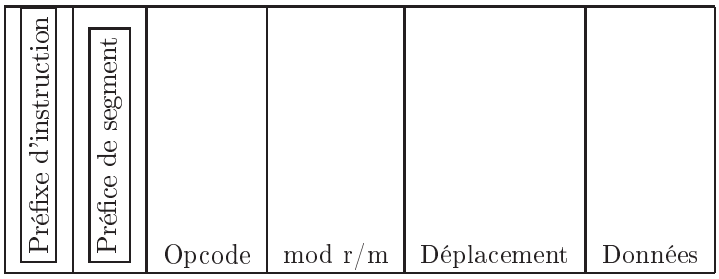
$$\begin{bmatrix} BX \\ BP \end{bmatrix} + \begin{bmatrix} SI \\ DI \end{bmatrix} + \text{Déplacement (0, 8 ou 16 bits)}$$

Cette adresse linéaire devant être comprise entre `0x0000` et `0xFFFF`.

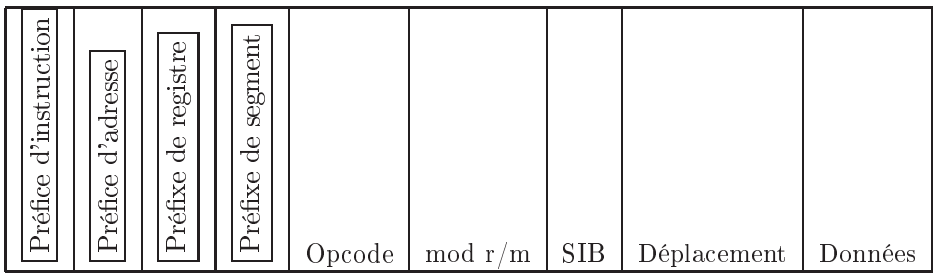
7.3.3 Codage des instructions

7.3.3.1 Format d'une instruction

Le format des instructions du `8086`, à savoir :



se complique et comporte maintenant de 1 à 16 octets suivant le format suivant :



— Le **préfixe d'instruction** éventuel, déjà connu du `8086`, sert à répéter certaines instructions ou au blocage :

Préfixe	**Signification**
`0xF3`	Répétition REP d'une opération sur les chaînes de caractères
`0xF3`	Répétition REPE/REPZ d'une opération sur les chaînes de caractères
`0xF2`	Répétition REPNE/REPNZ d'une opération sur les chaînes de caractères
`0xF0`	Préfixe pour LOCK

— La taille d'une constante est par défaut de 16 bits en mode réel. Si le **préfixe de taille de constante**, égal à `0x67`, est présent alors elle est de 32 bits.

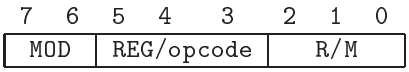
— La taille d'un registre est par défaut de 16 (ou 8) bits en mode réel. Si le **préfixe de registre**, égal à `0x66`, est présent alors elle est de 32 bits. C'est la façon d'indiquer d'accéder au registre `EAX` au lieu de `AX`, par exemple.

— On retrouve les valeurs du **préfixe de changement de segment** éventuel du `8086` plus deux nouvelles pour les segments `FS` et `GS` :

Préfixe	**Segment sous-jacent**
0x2E	CS
0x36	SS
0x3E	DS
0x26	ES
0x64	FS
0x65	GS

— Les codes opération (*opcode*), de un ou deux octets, sont les mêmes que pour le `8086` pour les instructions qui existaient déjà en langage symbolique. Nous en donnons le détail en appendice.

— L'octet éventuel mod r/m (*mode registre/mémoire*) a la même structure que pour le `8086` :



Il spécifie la forme de l'adresse utilisée et complète éventuellement le code opération.

— Le déplacement (signé) éventuel occupe 8, 16 ou 32 bits.
Sa taille est précisée par l'octet `ModR/M`, de la façon indiquée ci-dessous.

— Un opérande immédiat (non signé) éventuel occupe également 8, 16 ou 32 bits.
Sa taille est précisée par le bit `W` du code opération (8 ou 16/32) et par la présence ou non du préfixe de taille d'adresse (pour le choix entre 16 ou 32 octets).

7.3.3.2 Signification de l'octet mod r/m

Le champ MOD.- Le champ MOD spécifie le mode d'adressage de l'instruction :

MOD	Fonction
00	Pas de déplacement (sauf si R/M = 110b)
01	Déplacement sur 8 bits
10	Déplacement sur 16 bits
11	R/M est un registre

dans le cas du mode 16 bits.

Le déplacement sur 16 bits (MOD = 10b) devient un déplacement sur 32 bits dans le cas où le préfixe 0x67 de taille d'adresse est utilisé.

Le déplacement est un entier relatif.

Désignation du registre.- Le premier octet du code opération spécifie si le champ REG désigne la suite du code opération ou un registre. Lorsqu'il s'agit d'un registre, la signification du champ REG de 3 bits (et le champ R/M lorsque MOD = 11b) est donnée par le tableau suivant :

Valeur	W = 0	W = 1	
		sans préfixe 0x66	avec préfixe 0x66
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

L'octet mod r/m pour l'adressage sur 16 bits.- Laa signification de l'octet mod r/m pour l'adressage sur 16 bits est :

r/m	Adresse effective		
	mod = 00	mod = 01	mod = 10
000	[BX + SI]	[BX + SI] + disp8	[BX + SI] + disp16
001	[BX + DI]	[BX + DI] + disp8	[BX + DI] + disp16
010	[BP + SI]	[BP + SI] + disp8	[BP + SI] + disp16
011	[BP + DI]	[BP + DI] + disp8	[BP + DI] + disp16
100	[SI]	[SI] + disp8	[SI] + disp16
101	[DI]	[DI] + disp8	[DI] + disp16
110	disp16	[BP] + disp8	[BP] + disp16
111	[BX]	[BX] + disp8	[BX] + disp16

disp spécifiant un déplacement, sur 8 bits (*disp8*) ou sur 16 bits (*disp16*), ce déplacement suivant l'octet mod r/m et devant être ajouté à l'index.

Le registre de segment par défaut est SS si l'adresse effective contient BP pour index et DS sinon.

L'octet mod r/m pour l'adressage sur 32 bits.- La signification de l'octet `mod r/m` pour l'adressage sur 32 bits est :

r/m	Adresse effective		
	`mod = 00`	`mod = 01`	`mod = 10`
000	`[EAX]`	`disp8[EAX]`	`disp32[EAX]`
001	`[ECX]`	`disp8[ECX]`	`disp32[ECX]`
010	`[EDX]`	`disp8[EDX]`	`disp32[EDX]`
011	`[EBX]`	`disp8[EBX]`	`disp32[EBX]`
100	`[-][-]`	`disp8[-][-]`	`disp32[-][-]`
101	`disp32`	`disp8[EBP]`	`disp32[EBP]`
110	`[ESI]`	`disp8[ESI]`	`disp32[ESI]`
111	`[EDI]`	`disp8[EDI]`	`disp32[EDI]`

où *disp* spécifie un déplacement, sur 8 bits (*disp8*) ou sur 32 bits (*disp32*).
Le déplacement *disp8* suit l'octet SIB, doit être étendu en signe et ajouté à l'index.
Le code [–][–] indique qu'un octet SIB suit l'octet `mod r/m`.

7.4 Quelques programmes en langage machine

7.4.1 Utilisation du préfixe de taille d'opérande

Voyons comment utiliser un registre de capacité 32 bits, disons `EAX`, en mode réel. Il faut, pour cela, utiliser le préfixe de taille d'opérande. Vérifions, par exemple, que l'on peut effectuer une addition sur 32 bits en une seule instruction (alors qu'il en faut deux pour un microprocesseur 16 bits).

7.4.1.1 Le contexte

Écrivons un programme effectuant la somme de deux entiers. Plaçons-nous sous (un vrai) MS-DOS (sur clé USB, voir appendice I) et utilisons l'utilitaire `debug`.

Créons un répertoire de nom '`REAL`' sur notre clé USB.

Rappelons que nous pouvons utiliser la mémoire vive située après l'adresse `3000:400` pour nos expériences, puisqu'elle est laissée à disposition de l'utilisateur par le système d'exploitation MS-DOS. Plaçons `0x6060001` à l'emplacement `3000:400` et `0x60B0002` à l'emplacement `3000:4004`, en nous souvenant qu'*Intel* utilise la convention petit-boutienne, c'est-à-dire que, pour obtenir l'entier `0x01020304`, il faut écrire en mémoire les octets dans l'ordre `04 03 02 01` :

```
C:\REAL>debug
-E 3000:400 01 00 06 06
-E 3000:404 02 00 0B 00
-D 3000:400 L 10
3000:0400 01 00 06 06 02 00 0B 06-00 00 00 00 00 00 00 00 ................
-q
```

Écrivons maintenant un programme qui additionne le contenu de l'emplacement mémoire `3000:400` à celui de `3000:404` et place le résultat en `3000:408`.

7.4.1.2 Cas d'un programme utilisant des registres 16 bits

Ne nous fatiguons pas à traduire à la main le programme utilisant des registres 16 bits. Il suffit d'utiliser `debug` pour le traduire en langage symbolique d'*Intel* :

```
C:\REAL>debug
-a
16D3:0100 mov ax,3000
16D3:0103 mov ds,ax
16D3:0105 mov ax,[400]
16D3:0108 add ax,[404]
16D3:010C mov [408],ax
16D3:010F ret
16D3:0110
-R BX
BX 0000
:
-R CX
CX 0000
:010
-N ADD.COM
-W
Ecriture de 00010 octets
-Q
```

Lançons le programme obtenu et vérifions le résultat :

```
C:\REAL>add.com

C:\REAL>debug
-D 3000:400 L 10
3000:0400 01 00 06 06 02 00 0B 06-03 00 00 00 00 00 00 00 ................
-q
```

Conclusion.- On a bien le résultat voulu, c'est-à-dire que la somme $1 + 2 = 3$ a été effectuée. Les mots de poids fort ne sont pas intervenus dans cette addition.

7.4.1.3 Un programme utilisant des registres 32 bits

Pour effectuer la somme sur 32 bits, il faut changer partout le registre 16 bits `AX` par le registre 32 bits `EAX`.

Attention ! Tout ce qui suit doit être effectué en mode réel sur une architecture IA-32 ou x86-64. Le faire sur une architecture 16 bits conduit à un redémarrage.

Nous ne pouvons pas utiliser directement `debug` pour le codage :

```
C:\REAL>debug
-a
16D3:0100 mov eax,[400]
               ^
               Error
16D3:0100
-Q
```

Utilisons `debug` pour obtenir le code machine du programme '`add.com`' :

```
C:\REAL>debug add.com
-u
16EB:0100 B80030        MOV AX,3000
16EB:0103 8ED8          MOV DS,AX
16EB:0105 A10004        MOV AX,[400]
16EB:0108 03060404      ADD AX,[404]
16EB:010C A30804        MOV [408],AX
16EB:010F C3            RET
[...]
```

D'après ce que nous avons vu, le code pour le programme utilisant des registres 32 bits s'obtient en préfixant chaque instruction faisant référence à un registre 16 bits (disons `AX`) par `0x66` pour faire référence à un registre 32 bits (disons `EAX`) :

```
C:\REAL>debug
-E 100 B8 00 30 8E D8 66 A1 00 04 66 03 06 04 04 66 A3 08 04 C3
-R BX
BX 0000
:
-R CX
CX 0000
:013
-N ADD32.COM
-W
Ecriture de 00013 octets
-Q
```

Lançons le programme obtenu et vérifions le résultat :

```
C:\REAL>add32.com

C:\REAL>debug
-D 3000:400 L 10
3000:0400 01 00 06 06 02 00 0B 06-03 00 11 0C 00 00 00 00 ................
-q
```

qui montre, d'une part, que l'on a bien effectué en une seule étape une addition sur 32 bits, à savoir `0x6060001 + 0x60B0002 = 0xC110003`, et, d'autre part, qu'on a bien la représentation petit-boutienne indiquée ci-dessus d'après les reports effectués.

7.4.2 Utilisation du préfixe de taille de constante

Voyons comment placer un mot double en mémoire en une seule instruction (au lieu de deux pour un microprocesseur 16 bits). Plus exactement plaçons l'entier `0x100400` à l'emplacement `3000:400`.

7.4.2.1 Cas d'un programme utilisant une constante de 16 bits

Commençons par écrire un programme pour placer l'entier `0x400` de 16 bits à cet emplacement :

```
C:\REAL>debug
-a
16D3:0100 mov ax,3000
16D3:0103 mov ds,ax
16D3:0105 mov bx,400
16D3:0108 mov [400],bx
16D3:010C ret
16D3:010D
-R BX
BX 0000
:
-R CX
CX 0000
:0D
-N mov16.com
-W
Writing 0000D bytes
-Q
```

Lançons le programme obtenu et vérifions le résultat :

```
C:\REAL>debug
-D 3000:400 L 10
3000:0400 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-q

C:\REAL>mov16

C:\REAL>debug
-D 3000:400 L 10
3000:0400 00 04 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-q
```

La copie a bien été effectuée. Il n'y a rien de plus par rapport à ce que nous connaissions déjà, il s'agit juste d'un rappel.

7.4.2.2 Cas d'un programme utilisant une constante de 32 bits

Écrivons maintenant un programme utilisant une constante de 32 bits pour réaliser ce que nous voulons :

```
mov ax,3000
mov ds,ax
mov ebx,100400
mov [400],ebx
ret
```

Nous ne pouvons pas utiliser `debug` pour coder ce programme en langage machine, mais il peut nous faciliter la tâche en codant le programme précédent :

```
C:\REAL>debug mov16.com
-u
16EB:0100 B80030        MOV AX,3000
16EB:0103 8ED8          MOV DS,AX
16EB:0105 BB0004        MOV BX,400
16EB:0108 891E0004      MOV [400],BX
16EB:010C C3            RET
[...]
```

D'après ce que nous avons vu, le code pour le programme utilisant des registres de 32 bits s'obtient en préfixant chaque instruction faisant référence au registre `BX` par `0x66` pour faire référence au registre `EBX`. Ce n'est pas suffisant, il faut aussi préfixer l'instruction faisant intervenir une constante de 32 bits par le préfixe `0x67` et ne pas oublier que cette constante occupe 4 octets même si l'entier tient sur 3 octets :

```
C:\REAL>debug
-E 100 B8 00 30 8E D8 66 67 BB 00 04 10 00 66 89 1E 00 04 C3
-R BX
BX 0000
:
-R CX
CX 0000
:011
-N mov32.com
-W
Writing 00011 bytes
-Q
```

Lançons le programme obtenu et vérifions le résultat :

```
C:\REAL>mov32.com

C:\REAL>debug
-D 3000:400 L 10
3000:0400 00 04 10 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
-q
```

Le nombre voulu a bien été placé en mémoire.

Exercice 1.- *Vérifier que si on oublie le préfixe* `0x67` *ou si la constante est donnée sur 3 octets au lieu de 4, ceci conduit à un redémarrage du système.*

Exercice 2.- *Coder, lancer et vérifier l'action du programme suivant* :

```
mov ax,3000
mov ds,ax
mov ebx,400
mov byte ptr [ebx],7
ret
```

Exercice 3.- *Que se passe-t-il si, dans le programme précédent, on donne comme valeur à* `EBX` *une adresse supérieure à* `0xFFFFF`*, capacité maximum du mode réel?*

7.5 Utilisation d'un assembleur

Nous venons de donner quelques exemples de programmes en code machine, ce qui est la meilleure façon de comprendre comment cela marche. Nous allons maintenant voir comment programmer en langage d'assemblage.

Pour la suite, nous continuerons à utiliser le système d'exploitation MS-DOS sur une clé USB. Le résultat de l'assemblage étant indépendant du système d'exploitation sur lequel il est utilisé, nous utiliserons `gas` sous Linux, en s'arrangeant pour obtenir un exécutable brut, puis nous testerons l'exécutable sur MS-DOS (puisqu'il ne peut pas être exécuté, rappelons-le, même avec le format d'exécutable adéquat, en mode protégé niveau 3, que ce soit sous *Windows* ou sur *Linux*).

7.5.1 Les utilitaires GNU pour le mode réel

Les utilitaires GNU, conçu pour des microprocesseurs 32 bits, ne sont pas bien adaptés au mode réel. Montrons cependant qu'on peut, avec un peu de contorsion, les utiliser pour ce mode.

Exemple.- Rappelons ce que nous avons fait avec nos programmes en langage machine : nous avons cherché « à la main » des emplacements mémoire non utilisés pour y placer les données. On peut évidemment faire de même avec un assembleur, le programme en langage d'assemblage étant alors le même que celui en langage symbolique *Intel*.

Écrivons donc un programme en langage d'assemblage `gas` allant chercher un entier de 16 bits à l'emplacement mémoire d'adresse `6000:0`, y ajoutant celui de l'emplacement `6000:4` et déposant le résultat à l'emplacement `6000:8` :

```
.section .text
.globl   _start
.code16

_start:
movw $0x6000,%ax
movw %ax,%ds
movw 0x0,%ax
addw 0x4,%ax
movw %ax,0x8
ret
```

Remarque.- Puisque nous voulons que notre premier programme soit un programme en mode réel, on utilise la directive `.code16`, spécifiant que les instructions et constantes ont une taille de 16 bits par défaut.

Assemblage.- Nous avons déjà vu comment assembler un programme utilisateur avec `gas` et `ld`. Dans la mesure où on veut un excécutable binaire brut, il faut maintenant une étape de plus.

On peut écrire le programme avec n'importe quel éditeur de texte, en le nommant par exemple `asm1.s`. On commence ensuite par l'assembler comme dans le cas d'un programme utilisateur :

```
$ as asm1.s -o asm1.o
$ ld ./asm1.o -o ./asm1
$
```

Remarquons que si on essaie de le désassembler avec `objump`, comme dans le cas d'un programme utilisateur, on ne trouve pas ce à quoi on s'attend :

```
$ objdump -d ./asm1

./asm1:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <_start>:
  400078:       b8 00 60 8e d8          mov    $0xd88e6000,%eax
  40007d:       a1 00 00 03 06 04 00    movabs 0x8a3000406030000,%eax
  400084:       a3 08
  400086:       00 c3                   add    %al,%bl
$
```

Ceci s'explique par le fait que `objdump` analyse par défaut du code d'instructions 64 ou 32 bits et non 16 bits. On remarquera d'ailleurs que les adresses virtuelles sont données sur 64 bits, en commençant par l'adresse `0x0000000004000078`, et non sur 20 bits.

Il faut donc ajouter l'option `-m` (pour *Microprocessor*) à `objdump` pour lui indiquer qu'on a du code 16 bits, avec l'argument adéquat, ici `i8086` :

```
$ objdump -d -mi8086 ./asm1

./asm1:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <_start>:
  400078:       b8 00 60                mov    $0x6000,%ax
  40007b:       8e d8                   mov    %ax,%ds
  40007d:       a1 00 00                mov    0x0,%ax
  400080:       03 06 04 00             add    0x4,%ax
  400084:       a3 08 00                mov    %ax,0x8
  400087:       c3                      ret
$
```

Le code est désormais correctement interprété mais remarquons que les adresses sont toujours données sur 64 bits.

La taille du fichier exécutable (672 octets) et son format spécifié par `objdump` montre qu'on n'est pas en présence d'un binaire brut. On utilise l'utilitaire `objcopy` pour obtenir celui-ci :

```
$ objcopy -O binary ./asm1 ./asm1.com
$
```

ce qui produit un fichier de 16 octets.

Exécution sous MS-DOS.- Exécutons le fichier `asm1.com`, après avoir fait les préparations nécessaires :

```
C:\REAL>debug
-E 6000:0 06
-E 6000:4 09
-D 6000:0 L 10
6000:0000 06 00 00 00 09 00 00 00-00 00 00 00 00 00 00 00 ................
-q

C:\REAL>asm1.com

C:\REAL>debug
-D 6000:0 L 10
6000:0000 06 00 00 00 09 00 00 00-0F 00 00 00 00 00 00 00 ................
```

qui montre que l'addition a bien été effectuée.

7.5.2 Utilisation de macros

Nous avons vu, à propos de la programmation utilisateur, comment utiliser des macros dans un langage d'assemblage. A FAIRE DANS LA PREMIERE PARTIE Il n'y a aucune raison, on peut aussi utiliser des macros pour la programmation système.

Exemple.- Réécrivons le programme précédent pour obtenir un programme dans lequel les deux données à additionner sont placées sous forme de constantes, situées après le code, afin de bien distinguer la partie code de la partie données. Par contre, on place toujours le résultat à l'emplacement explicite `6000:0`, pour pouvoir vérifier que l'addition a bien été exécutée. Nous obtenons le programme `asm2.s` :

```
.section .text
.globl   _start
.code16
_start:
movw $0x6000,%ax
movw %ax,%ds
movw $a,%ax
addw $b,%ax
movw %ax,0x8
ret
a = 10
b = 22
```

Assemblons le programme, lions-le et désassemblons-le :

```
$ as asm2.s -o asm2.o
$ ld asm2.o -o asm2
$ objdump -d -mi8086 ./asm2

./asm2:     file format elf64-x86-64


Disassembly of section .text:

0000000000400078 <_start>:
  400078:       b8 00 60                mov    $0x6000,%ax
  40007b:       8e d8                   mov    %ax,%ds
  40007d:       b8 0a 00                mov    $0xa,%ax
  400080:       05 16 00                add    $0x16,%ax
  400083:       a3 08 00                mov    %ax,0x8
  400086:       c3                      ret
$ objcopy -O binary ./asm2 ./asm2.com
$
```

On s'aperçoit que les valeurs introduites par les macros ont été placées immédiatement dans les instructions adéquates : la structure du programme source ne se retrouve pas dans le programme exécutable.

Exécution sous MS-DOS.- Exécutons le binaire obtenu `asm2.com` sous MS-DOS :

```
C:\REAL>asm2.com

C:\REAL>debug
-D 6000:0 L 10
6000:0000 00 00 00 00 00 00 00 00-20 00 00 00 00 00 00 00 ........ ........
```

qui montre que l'addition a bien été effectuée, puisque `32 = 0x20`, code ASCII de l'espace, comme on peut le vérifier dans la troisième colonne.

Commentaire.- Nous avons vu une nouvelle directive, la déclaration d'une macro, de la forme :

```
identificateur = valeur
```

qui peut être placée n'importe où, sur une ligne à part, dans le fichier source. L'assembleur remplace toute occurrence de l'identificateur par la valeur.

7.5.3 Étiquetage des données modifiables

Introduction.- L'écriture sur des emplacements mémoire choisis par l'utilisateur peut être dangereuse lorsqu'un système d'exploitation lance un exécutable, puisque c'est le système d'exploitation qui décide l'emplacement de telle ou telle donnée : on risque de surcharger une donnée essentielle.

En effet, lorsqu'un système d'exploitation charge un programme, il cherche un emplacement mémoire libre compatible avec la taille de ce programme. Il vaut donc mieux ne pas faire appel à des emplacements explicites de mémoire pour être sûr de ne rien détruire.

Pour éviter ce problème, la plupart des systèmes d'exploitation exigent que le code d'un programme et ses données soient dans des **sections** distinctes. Il ne permet que la lecture (et interdit l'écriture) dans une section de code, ce qui empêche le programme de modifier son propre code.

Ceci est facile à mettre en place avec le mode protégé des microprocesseurs *Intel* comme nous le verrons : la section de code est placée dans un segment dans lequel on ne peut que lire alors que la section des données est placée dans un segment dans lequel on peut lire et ècrire.

Ceci n'est pas le cas pour le mode réel : on peut lire et écrire sur tous les segments. De plus, le problème demeure, même en mode protégé, lorsque c'est l'utilisateur, et non le système d'exploitation, qui décide où placer les données (dans la section des données).

Nous venons de voir comment utiliser des constantes pour bien dégager les données du code proprement dit. Ceci ne marche évidemment que pour les « données » à lire, et non pour les données de travail, qu'on veut pouvoir lire et écrire. Une autre façon de faire réserver des emplacements mémoire par le systèmes d'exploitation est d'étiqueter des emplacements mémoire.

Exemple.- Réécrivons le programme précédent de façon à ce que les deux données à additionner soient étiquetées. Par contre, on place toujours le résultat à l'emplacement explicite `6000:0`, pour pouvoir vérifier que l'addition a bien été exécutée :

```
.globl   _start
.arch i8086
.code16
_start:
movw a,%ax
addw b,%ax
movw $0x6000,%dx
movw %dx,%ds
movw %ax,0x8
ret
a: .word 10
b: .word 22
```

Remarques.- 1°) Le point essentiel est l'étiquetage, par `a` et `b`, de deux zones de mémoire, toutes les deux d'une taille de deux octets. Elles sont situées après le code. En effet, le microprocesseur ne sait pas différencier le code des données ; si le pointeur d'instruction pointe sur un emplacement mémoire étiqueté, il essaie de l'interpréter comme une instruction, et donc de l'exécuter, ce qui risque de donner lieu à un redémarrage du système puisqu'il n'y a aucune raison que cette « instruction » ait un sens. Pire, si elle a une sens, cela risque de ne pas être ce que l'on veut.

- 2^o) En `gas`, un emplacement mémoire étiqueté est déclaré de la façon suivante :

```
identificateur: type valeurInitiale
```

où :

— `identificateur` est un mot sur l'alphabet formé de lettres, minuscules et majuscules, et de chiffres et de quelques caractères spéciaux, ne commençant pas par un chiffre et n'étant pas un mot réservé. Cette définition, relativement imprécise, devrait être suffisante. On trouve évidemment une définition formelle dans le manuel de `gas`.

— Le signe deux points '`:`' indique qu'il s'agit de la déclaration d'un emplacement mémoire, et non de son utilisation.

— Le type spécifie la taille des valeurs de la variable :

  — `.byte` pour un octet ;
  — `.word` pour un mot de deux octets ;
  — `.long` pour un mot double, soit quatre octets (mais ce type et les types suivants ne s'utilisent pas en mode réel) ;

— La valeur initiale est un entier, exprimé en décimal ou en héxadécimal, compatible avec la taille annoncée.

- 3^o) En programmation utilisateur en mode protégé, une telle déclaration ne peut être effectuée que dans une section `.data` mais, comme nous l'avons déjà dit, la distinction entre code et données a peu de sens en mode réel.

- 4^o) Une étiquette placée dans une instruction, dans un programme en langage d'assemblage, est traduite, en code machine, par le décalage de l'emplacement mémoire correspondant lors de l'assemblage. L'étiquette précédée du signe '`$`' est traduite par le contenu de l'emplacement mémoire.

- 5^o) Puisque nous voulons obtenir du mode réel, nous devons utiliser ici, outre la directive `.code16` déjà rencontrée, la directive `.arch` avec la valeur `i8086`. En effet `.code16` spécifie qu'on veut un mode d'instruction 16 bits. Celui existe pour le mode réel, mais également pour le mode protégé. La différence est que les décalages sont sur 16 bits en mode réel et sur 32 bits en mode protégé. La directive `.arch i8086` spécifie d'utiliser des décalages sur 16 bits.

Assemblage sans liage.- On peut assembler le fichier source comme d'habitude mais, si nous sommes sur un GBU/Linux 64 bits, il faut utiliser l'option `-32` pour obtenir un fichier ELF 32 bits (et non 64 bits) :

```
$ as asm3.s --32 -o asm3.o
$
```

Ceci est dû au fait qu'on ne peut pas utiliser le mode d'instructions 16 bits dans le mode 64 bits alors que l'assemblage conduit par défaut à un fichier ELF 64 bits sur un système 64 bits.

Remarquons que si on essaie de lier le fichier objet obtenu, on obtient une erreur :

```
$ ld asm3.o -o asm3
ld: i386 architecture of input file ‘asm3.o’ is incompatible with i386:x86-64 output
asm3.o: In function ‘_start’:
(.text+0x1): relocation truncated to fit: R_386_16 against ‘.text’
asm3.o: In function ‘_start’:
(.text+0x5): relocation truncated to fit: R_386_16 against ‘.text’
$
```

car le lieur `ld` de GNU n'est pas adapté à l'architecture 16 bits, contrairement à d'autres assembleurs/lieurs.

Cependant tout n'est pas perdu. Si on regarde la partie code de ce fichier objet, on s'aperçoit qu'on a bien le code machine voulu :

```
$ objdump -d  -mi8086 ./asm3.o

./asm3.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start>:
   0:   a1 10 00                mov    0x10,%ax
   3:   03 06 12 00             add    0x12,%ax
   7:   ba 00 60                mov    $0x6000,%dx
   a:   8e da                   mov    %dx,%ds
   c:   a3 08 00                mov    %ax,0x8
   f:   c3                      ret

00000010 <a>:
  10:   0a 00                   or     (%bx,%si),%al

00000012 <b>:
  12:   16                      push   %ss
        ...
$
```

Remarquons que les adresses virtuelles sont maintenant données sur 32 bits et non plus sur 64 bits, grâce à l'optiopn `-32` (mais elles ne sont toujours pas données sur 20 bits).

Remarquons également que, si `objdump`, comme tout désassembleur, cette fois-ci, désassemble correctement la partie que nous considérons comme du code, il perd pied lorsqu'il rencontre du code binaire ne correspondant pas à des instructions mais à des données.

Transformons donc ce fichier objet, non lié, pour obtenir du binaire brut :

```
$ objcopy -O binary ./asm3.o ./asm3.com
$
```

On peut voir avec un éditeur hexadécimal qu'on obtient bien le binaire désiré :

```
$ ghex asm3.com &
$
```

Adaptation pour MS-DOS.- Lors de son chargement par le système d'exploitation, un exécutable `.com` est précédé d'un préfixe de 256 octets. Le décalage dont on a alors besoin (par rapport à la valeur de `CS` choisie par le système d'exploitation) pour le premier emplacement mémoire étiqueté, par exemple, est non pas `0x10` mais `0x110`.

On l'indique en utilisant la directive `.org` permettant de préciser le décalage de la ligne qui suit. On utilisera donc le programme `asm3.s` suivant :

```
.globl   _start
.arch i8086
.code16
.org 0x100
_start:
movw a,%ax
addw b,%ax
movw $0x6000,%dx
movw %dx,%ds
movw %ax,0x8
ret
a: .word 10
b: .word 22
```

Exécution sous MS-DOS.- L'exécution du binaire obtenu, `asm3.com` :

```
C:\REAL>asm3

C:\REAL>debug
-D 6000:0 L 10
6000:0000 00 00 00 00 00 00 00 00-20 00 00 00 00 00 00 00 ........ ........
```

nous montre que l'addition a bien été effectuée, puisque `32 = 0x20`.

7.5.4 Utilisation des registres 32 bits

On peut utiliser, en mode réel, un registre 32 bits sans problème ainsi que des constantes et des adresses sur 32 bits. Les préfixes adéquats sont placés lors de l'assemblage.

Exemple.- Écrivons un programme `asm4.s` utilisant le registre `EAX` :

```
.section .text
.globl   _start
.code16
.org 0x100
_start:
movl a,%eax
addl b,%eax
movw $0x6000,%dx
movw %dx,%ds
movl %eax,0x0
ret
a: .long 0x100000
b: .long 0x22
```

en remarquant que les emplacements mémoire étiquetés doivent maintenant être du type mot double.

Remarquons que la directive `.arch i8086` a disparu. Elle conduirait à une erreur puisque les registres 32 bits n'ont pas de sens pour une telle architecture.

Assemblage sans liage.- On assemble le fichier source comme dans le cas précédent :

```
$ as asm4.s --32 -o asm4.o
$
```

Comme dans l'exemple précédent, si on regarde la colonne code de ce fichier objet, on s'aperçoit qu'on a bien le code machine voulu :

```
$ objdump -d -mi8086 ./asm4.o

./asm4.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start-0x100>:
        ...

00000100 <_start>:
 100:   66 a1 13 01             mov    0x113,%eax
 104:   66 03 06 17 01          add    0x117,%eax
 109:   ba 00 60                mov    $0x6000,%dx
 10c:   8e da                   mov    %dx,%ds
 10e:   66 a3 00 00             mov    %eax,0x0
 112:   c3                      ret

00000113 <a>:
 113:   00 00                   add    %al,(%bx,%si)
```

```
 115:   10 00                   adc    %al,(%bx,%si)

00000117 <b>:
 117:   22 00                   and    (%bx,%si),%al
        ...
$
```

avec le préfixe `0x66` pour les instructions 32 bits. Mais le code commençant à l'adresse `0x0` et non `0x100`, il y a 256 octets nuls.

Commençons par transformer, comme dans l'exemple précédent, ce fichier objet, non lié, afin d'obtenir du binaire brut :

```
$ objcopy -O binary ./asm4.o ./asm4.com
$ ls -l ./asm4.com
-rwxr-xr-x 1 patrick patrick 283 sept.  4 10:33 ./asm4.com
$
```

Il comporte 283 octets dont on ne veut conserver que les 27 (= 283 - 256) derniers. On utilise pour cela de nouvelles options de `objcopy` :

```
$ objcopy -O binary -i 512 --interleave-width 27 -b 256 ./asm4.o ./asm4.com
$
```

où `-i` pour *Interleave* signifie qu'on ne doit conserver que le nombre d'octets précisé par la largeur spécifiée (`-interleave-width`) à partir de l'octet précisé par `-b` (pour *Byte*), le tout dans l'intervalle d'amplitude spcifiée par l'argument de `-i`.

On peut alors voir avec un éditeur hexadécimal qu'on obtient bien le binaire désiré :

```
$ ghex asm4.com &
$
```

Exécution sous MS-DOS.- L'exécution du binaire obtenu, `asm4.com` :

```
C:\REAL>fasm3

C:\REAL>debug
-D 6000:0 L 10
6000:0000 22 00 10 00 00 00 00 00-00 00 00 00 00 00 00 00 "...............
```

montre que l'addition est bien effectuée sur 32 bits.

7.6 Bibliographie

[Lil-89] LILEN, Henri, **80386 : modes de fonctionnement – architecture – programmation – caractéristiques**, Éditions Radio, 1989, 288 p.

7.7 Appendice I : MS-DOS sur une clé USB

Comme nous l'avons dit, nous allons étudier les instructions du mode réel en utilisant un PC muni du système d'exploitation MS-DOS. Si vous disposez déjà d'un système informatique avec MS-DOS (en mode réel, c'est-à-dire jusqu'à la version 6.22), que ce soit un ordinateur auxiliaire ou un avec multiple démarrage, il est inutile d'installer MS-DOS. Mais c'est rare de nos jours.

Si vous disposez d'un ordinateur avec un lecteur de disquettes, vous pouvez partitionner votre disque dur et installer MS-DOS sur la première partition. Mais c'est également rare de nos jours. Cela peut également se faire, en jonglant un peu, si l'ordinateur dispose d'un lecteur de CD-ROM.

Le cas le plus fréquent est un ordinateur PC ayant un microprocesseur *Intel* puissant, le plus souvent 64 bits, et, pour ce qui nous intéresse, des ports USB.

Nous allons donc installer MS-DOS sur une clé USB. On choisira soit MS-DOS 5.0 disponible gratuitement, soit MS-DOS 6.22, qui reste sous licence Microsoft mais disponible gratuitement, comme tous les autres outils de développement Microsoft, pour les étudiants dans un département d'informatique abonné au système MSDN (ou par d'autres moyens, illicites cependant).

7.7.1 Première étape : récupérer MS-DOS

Il faut récupérer le fichier `en_msdos622.exe` sur MSDN (ou de façon illicite). Le placer dans un répertoire vide, par exemple de nom '`en_MSDOS622`', et faites exécuter (sous *Windows*) ce fichier auto-extractible. On obtient deux sous-répertoires : '`DISKS`' et '`UPGRADE`'.

Ces répertoires contiennent à la fois des fichiers au nom bizarre et des fichiers exécutables. Ce sont ces derniers qui nous intéressent.

7.7.2 Seconde étape : créer une clé USB bootable

Une **clé bootable** est une clé USB qui, lorsqu'on démarre l'ordinateur avec la clé dans un des connecteurs USB (et en spécifiant au BIOS de commencer par chercher un système d'exploitation sur une telle clé, avant d'aller chercher sur un disque dur ou un CD-ROM), charge le système d'exploitation qui se trouve sur celle-ci.

Récupération d'un utilitaire.- Nous verrons plus tard comment créer une clé bootable. Utilisons pour le moment un utilitaire pour le faire. Spécifier '`HP Drive Key Boot Utility v2.1.8 download`' sur votre moteur de recherche favori. Ceci ne renvoie plus au site originel de Hewlett-Packard car l'utilitaire a évolué (mais la nouvelle version ne convient pas à ce que nous voulons faire). On peut le récupérer, par exemple (au moment où nous écrivons) sur le site Web :

`http://files.extremeoverclocking.com/file.php?f=197`

en cliquant sur '*Primary Download Site*', qui permet de télécharger le fichier '`SP27608.exe`' de 1,97 MiO. Placez-le dans un répertoire vide de votre système d'exploitation *Windows*, par exemple '`HP DriveKey`' et faites-le exécuter. On se retrouve avec quatre fichiers, dont l'exécutable `HPUSBFW.exe` (pour *HP USB Disk Storage Format Tool FreeWare*).

Fichiers MS-DOS essentiels.- Préparez un autre répertoire vide sur votre système d'exploitation *Windows*, nommé par exemple '`MSDOS622`' et placez-y les trois fichiers '`command.com`', '`io.sys`' et '`msdos.sys`' trouvés dans le sous-répertoire '`UPGRADE`' de '`en_MSDOS622`'. Attention! ces fichiers sont « cachés » car considérés comme fichiers système : il faut aller dans les onglets '`Panneau de configuration`', '`Apparence et personnalisation`' puis '`Afficher les fichiers et dossiers cachés`' pour désélectionner, au moins de façon temporaire, l'option '`Masquer les fichiers protégés du système d'exploitation (recommandé)`' afin de rendre visible les fichiers système.

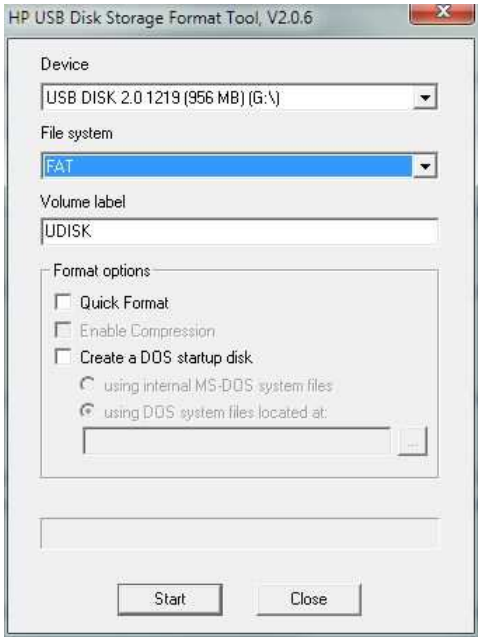


FIGURE 7.1 – Fenêtre de `HP USB Disk Storage Format Tool`

Placement de MS-DOS sur la clé USB.- Insérez une clé USB (de moins de 2 GiO) dans un connecteur USB et faites exécuter l'utilitaire `HPUSBFW.exe` en tant qu'administrateur. On obtient la fenêtre de la figure 7.1. Vérifiez que le volume indiqué dans '`Device`' est bien celui de la clé que vous voulez rendre bootable. Choisissez le système de fichiers FAT16, c'est-à-dire 'FAT'. Choisissez éventuellement un nom de volume autre que celui indiqué par défaut. Cliquez sur '*Create a DOS startup disk*' puis sur '*using DOS system files located at :*'. Appuyez sur les trois petits points pour rechercher le dossier '`MSDOS622`' précédemment préparé puis sur le bouton '*Start*'. La fenêtre de la figure 7.2 s'ouvre.

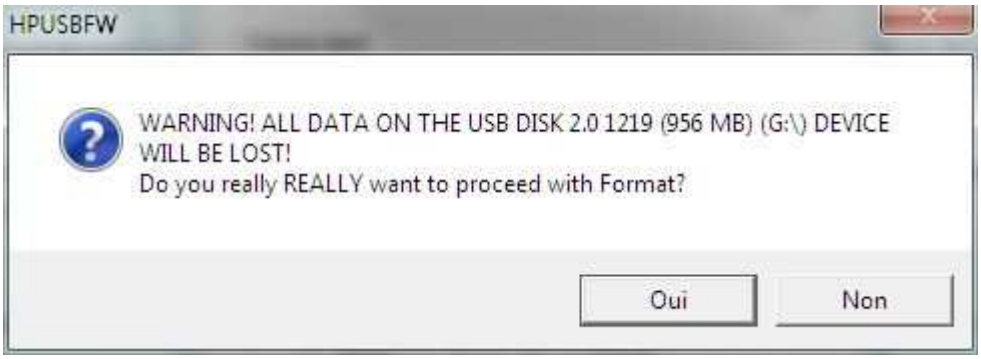


FIGURE 7.2 – Fenêtre d'avertissement de `HP USB Disk Storage Format Tool`

Après avoir vérifié une dernière fois qu'il s'agit bien de la clé USB voulue, appuyez sur le bouton '*Oui*'.

Démarrage de MS-DOS.- Une fois l'opération terminée, redémarrez l'ordinateur en laissant la clé USB dans le connecteur. Il faut indiquer au BIOS de booter sur une clé USB en changeant l'ordre du choix de périphérique de boot. Si tout s'est bien passé, on se retrouve avec MS-DOS (avec clavier Qwerty pour l'instant). Plus exactement on voit apparaître :

```
Starting MS-DOS...

Current date is Wed 01-09-2013
Enter new date (mm-dd-yy):
```

Appuyer sur le touche '`retour`'. On a alors :

```
Starting MS-DOS...

Current date is Wed 01-09-2013
Enter new date (mm-dd-yy):
Current time is 11:01:58.32a
Enter new time:
```

Appuyer sur le touche '`retour`'. On se retrouve avec :

```
Starting MS-DOS...

Current date is Wed 01-09-2013
Enter new date (mm-dd-yy):
Current time is 11:01:58.32a
Enter new time:


Microsoft(R) MS-DOS(R) Version 6.22
            (C)Copyright Microsoft Corp 1981-1994

C:\>
```

On se retrouve avec la même invite de commande que celle trouvée sur l'émulateur d'invite de commande '`command.com`' de *Windows*.

On peut commencer à explorer MS-DOS mais on se trouve dans la version anglaise avec clavier querty.

7.7.3 Troisième étape : version française complète

Pour la suite on va se placer dans la version avec clavier azerty et ajouter les utilitaires dont nous avons besoin, en particulier '`debug`'.

Utilitaires MS-DOS.- Enlevez la clé USB et redémarrez l'ordinateur pour se retrouver avec *Windows*. Placez, seulement une fois le système démarré, la clé dans un connecteur USB. Créez un répertoire '`DOS`' sur la clé et placez-y les fichiers '`COUNTRY.SYS`', '`DEBUG.EXE`', '`EDIT.COM`', '`KEYB.COM`', '`KEYBOARD.SYS`', '`MORE.COM`' et '`QBASIC.EXE`' se trouvant dans le sous-répertoire '`UPGRADE`' de '`en_MSDOS622`'.

Fichiers de configuration.- Préparer un fichier de nom '`AUTOEXEC.BAT`' (avec `notepad` par exemple ou tout autre éditeur de texte) à placer à la racine de la clé USB :

```
@ECHO OFF
PROMPT $p$g
PATH C:\DOS
KEYB FR,,C:\DOS\KEYBOARD.SYS
```

Préparer aussi un fichier de nom 'CONFIG.SYS' à placer également à la racine de la clé USB :

```
COUNTRY=033,850,C:\DOS\COUNTRY.SYS
```

Notre version finale de MS-DOS.- Redémarrez l'ordinateur en laissant la clé USB dans le connecteur. On voit maintenant apparaître :

```
Starting MS-DOS...

C:\>
```

On peut maintenant utiliser MS-DOS avec clavier azerty.

7.8 Appendice II

7.8.1 Types de données

Intel utilise les types de données suivants :

— le *bit* est, classiquement, celui de la plus petite quantité d'information ;
— l'*octet* (*byte*) est un groupe de 8 bits représentant un entier relatif compris entre - 128 et 127 ;
— l'*octet non signé* est un groupe de 8 bits représentant un entier naturel compris entre 0 et 255 ;
— le *mot* (*word*) est un groupe de 16 bits, soit deux octets ;
— le *mot entier* est un mot représentant un entier relatif ;
— le *mot double* (*doubleword*) est un groupe de 32 bits, soit quatre octets ou deux mots ;
— un **entier long** est un mot double représentant un entier relatif en complément à deux ;
— un **entier long non signé** est un mot double représentant un entier naturel ;
— un **champ de bits** est une chaîne de 32 bits au maximum ;
— un **caractère** est un octet contenant un code ASCII ;
— une **chaîne de bits** est une suite de bits d'au plus 4 GiO ;
— une **chaîne** est une suite d'octets d'une longueur comprise entre 1 et 4 GiO ;
— un *DCB compacté* est un chiffre décimal codé binaire avec 4 bits par chiffre ;
— un *DCB étendu* est un chiffre décimal codé binaire avec 8 bits par chiffre ;
— un **pointeur court** est un décalage de 16 ou 32 bits dans un segment ;
— un **pointeur long** est formé d'un sélecteur (adresse de segment) sur 16 bits et d'un pointeur court ;
— une *virgule flottante* (*point flottant*) est la représentation d'un nombre réel signé, sur 32, 64 ou 80 bits pour le coprocesseur arithmétique.

7.8.2 Noms des instructions en langage d'assemblage

Donnons la liste des instructions avec une classification plus fine que celle donnée ci-dessus.

Instructions de transfert de données

Il y a sept instructions de transfert de données en langage symbolique :

Instruction	Fonction
MOV	Déplacement d'un opérande
PUSH	Chargement d'unopérande sur la pile
POP	Extraction d'un opérande de la pile
PUSHA	Chargement de tous les registres dans la pile
POPA	Extraction de tous les registres de la pile
XCHG	Échange d'opérandes
XLAT	Traduction

dont cinq existaient déjà pour le 8086.

Instructions d'entrée-sortie

Il y a les deux mêmes instructions d'entrée-sortie en langage symbolique que pour le 8086 (et même le 8080) :

Instruction	Fonction
IN	Entrée d'un opérande
OUT	Sortie d'un opérande

Instructions arithmétiques

On retrouve les dix-sept instructions arithmétiques en langage symbolique du 8086 :

Instruction	Fonction
INC	Incrémenter de 1
DEC	Décrémenter de 1
ADD	Addition
ADC	Addition avec retenue
SUB	Soustraction
SBB	Soustraction avec emprunt
MUL	Multiplication en simple ou double précision
DIV	Division d'entiers naturels
NEG	Inversion logique en complément à 2
IMUL	Multiplication d'entiers relatifs
IDIV	Division d'entiers relatifs
CMP	Comparaison
DAA	Correction décimale après addition
AAA	Ajustement ASCII après addition
AAS	Ajustement ASCII après soustraction
AAM	Ajustement ASCII après multiplication
AAD	Ajustement ASCII après division

Instructions conditionnelles de contrôle du programme

On retrouve les dix-huit instructions conditionnelles de contrôle du programme du langage symbolique du 8086 ainsi que la nouvelle instruction SETCC :

Instruction	Fonction
JA/JNBE	Saut si supérieur/si non inférieur ni égal
JAE/JNB	Saut si supérieur ou égal/si non inférieur
JB/JNAE	Saut si inférieur/si pas supérieur ni égal
JBE/JNA	Saut si inférieur ou égal/si pas supérieur
JC	Saut si retenue
JE/JZ	Saut si égal/si nul
JG/JNLE	Saut si plus grand/si pas plus petit ou égal
JGE/JNL	Saut si plus grand ou égal/si pas plus petit
JL/JNGE	Saut si plus petit/si pas plus grand ni égal
JLE/JNG	Saut si plus petit ou égal/si pas plus grand
JNC	Saut si pas de retenue
JNE/JNZ	Saut si non égal/si non nul
JNO	Saut si pas de dépassement de cvapacité
JNP/JPO	Saut si pas de parité (impaire)/si parité impaire
JNS	Saut si pas de signe
JO	Saut si dépassement de capacité
JP/JPE	Saut si parité (paire)/si parité paire
JS	Saut si signe
SETCC	Rangement conditionnel d'octet

Autres instructions de contrôle du programme

On retrouve les douze autres instructions de contrôle du du langage symbolique du 8086 et la nouvelle instruction JECX :

Instruction	Fonction
CALL	Appel d'un sous-programme
RET	Retour d'un sous-programme
JMP	Saut inconditionnel
LOOP	Boucle
LOOPE/LOOPZ	Boucle si égal/si nul
LOOPNE/LOOPNZ	Boucle si non égal/si non nul
JCXZ	Saut si le registre CX est nul
INT	Interruption
INTO	Interruption si dépassement de capacité
IRET	Retour d'une interruption
CLI	Inhibition des interruptions masquables
STI	Autorisation des interruptions masquables
JECXZ	Saut si le registre ECX est nul

Instructions logiques

On retrouve les cinq instructions logiques du langage symbolique du 8086 :

Instruction	Fonction
NOT	Inversion logique bit à bit
AND	ET logique bit à bit
OR	OU logique inclusif bit à bit
XOR	OU logique exclusif bit à bit
TEST	Test

Instructions de décalage et de rotation

On retrouve les huit instructions de décalage et de rotation du langage symbolique du 8086 ainsi que les deux nouvelles instructions SHLD et SHRD :

Instruction	Fonction
SHL/SHR	Décalage logique à gauche/à droite
SAL/SAR	Décalage arithmétique à gauche/à droite
ROL/ROR	Rotation à gauche/à droite
RCL/RCR	Rotation *via* la retenue à gauche/à droite
SHLD/SHRD	Double décalage à gauche/à droite

Instructions de manipulation de bits

Il y a huit instructions de manipulation de bits, toutes nouvelles :

Instruction	Fonction
BT	Test de bit
BTS	Test de bit et positionnement à 1
BTR	Test de bit et réinitialisation (à 0)
BTC	Test de bit et complémentation
BSF	Balayage direct de bits
BSR	Balayage inverse de bits
IBTS	Insertion dans une chaîne de bits
XBTS	Extraction dans une chaîne de bits

Instructions sur les chaînes de caractères

On retrouve les dix instructions sur les chaînes de caractères du langage symbolique du 8086 :

Instruction	Fonction
MOVS	Copie d'une chaîne d'octets, de mots ou de mots doubles
INS	Saisie d'une chaîne de caractères
OUTS	Sortie d'une chaîne de caractères
CMPS	Comparaison de chaînes de caractères
SCAS	Analyse d'une chaîne de caractères
LODS	Chargement d'une chaîne de caractères
STOS	Rangement une chaîne de caractères
REP	Répétition d'une opération sur les chaînes de caractères
REPE/REPZ	Répétition tant que égal/nul
REPNE/REPNZ	Répétition tant que non égal/non nul

Instructions de conversion

Au vu de toutes les tailles de données élémentaires à manipuler, il y a six instructions de conversion en langage symbolique :

Instruction	Fonction
MOVZX	Copie d'un octet, d'un mot, d'un double mot, sans extension du signe
MOVSX	Copie d'un octet, d'un mot, d'un double mot, avec extension du signe
CBW	Conversion d'unoctet en un mot ou d'un mot en un mot double
CDW	Conversion d'un mot en un mot double
CDWE	Conversion étendue d'un mot en un mot double
CDQ	Conversion d'un mot double en un mot quadruple

dont seules CBW (et CDW) existaient pour le 8086.

Instructions de formation des adresses

Il y a six instructions de formation des adresses en langage symbolique, toutes nouvelles :

Instruction	Fonction
LEA	Charger l'adresse effective
LDS	Charger le pointeur dans le registre de segment DS
LES	Charger le pointeur dans le registre de segment ES
LFS	Charger le pointeur dans le registre de segment FS
LGS	Charger le pointeur dans le registre de segment GS
LSS	Charger le pointeur dans le registre de segment DS

Instructions sur les indicateurs

Il y a onze instructions sur les indicateurs en langage symbolique :

Instruction	Fonction
LAHF	Charger le registre des indicateurs dans le registre A
SAHF	Ranger le registre A dans le registre des indicateurs
PUSHF	Sauvegarder le registre des indicateurs sur la pile
POPF	Extraire le registre des indicateurs de la pile
PUSHFD	Sauvegarder EFLAGS sur la pile
POPFD	Extraire EFLAGS de la pile
CLC	Mettre l'indicateur CF de retenue à 0
CLD	Mettre l'indicateur DF de direction à 0
CMC	Complémenter l'indicateur CF de retenue
STC	Mettre l'indicateur CF de retenue à 1
STD	Mettre l'indicateur DF de direction à 1

dont les six premières sont nouvelles.

Instructions pour langages de haut niveau

On retrouve les trois instructions pour les langages de haut niveau en langage symbolique, introduites pour le 80186 :

Instruction	Fonction
BOUND	Test de limites
ENTER	Établissement du bloc des paramètres pour une procédure
LEAVE	Quitter une procédure

7.8.3 Les codes opération

Le code opération des instructions dont le nom symbolique apparaît déjà pour le 8086 est le même, à la différence près que le bit *w*, indiquant toujours la longueur de l'opérande, vaut 0 pour un octet et 1 pour la pleine longueur, c'est-à-dire 16 bits ou 32 bits suivant le cas.

Instructions de transfert de données	
Instruction	**Format**
MOV	
Registre à Registre/Mémoire	1000 100w | mod reg r/m
Registre/Mémoire à Registre	1000 101w | mod reg r/m
Immédiat à Registre/Mémoire	1100 011w | mod 000 r/m
Immédiat à Registre	1011 w reg | donnée immédiate
Mémoire à Accumulateur (forme courte)	1010 000 w | déplacement complet
Accumulateur à Mémoire (forme courte)	1010 001 w | déplacement complet
Registre/Mémoire à Registre de Segment	1000 1110 | mod 0 sreg r/m
Registre de Segment à Registre/Mémoire	1000 1100 | mod 0 sreg r/m
MOVSX	
Registre depuis Registre/Mémoire	0000 1111 | 1011 111w | mod reg r/m
MOVZX	
Registre depuis Registre/Mémoire	0000 1111 | 1011 011w | mod reg r/m
PUSH	
Mémoire	1111 1111 | mod 110 r/m
Registre	0101 0 reg
Registre de Segment (ES,CS, SS ou DS)	000 sreg 110
Registre de Segment (FS ou GS)	0000 1111 | 10 sreg 000
Immédiat	0110 10s0 | donnée immédiate
PUSHA	0110 0000
POP	
Mémoire	1000 1111 | mod 000 r/m
Registre	0101 1 reg
Registre de Segment (ES,CS, SS ou DS)	000 sreg 111
Registre de Segment (FS ou GS)	0000 1111 | 10 sreg 001
Immédiat	0110 10s0 | donnée immédiate
POPA	0110 0001
XCHG	
Registre/Mémoire avec Registre	1000 011w | mod reg r/m
Registre avec Accumulateur (forme courte)	1001 0 reg
IN	
Port Fixé	1110 010w | numéro de port
Port Variable	1110 110w
OUT	
Port Fixé	1110 011w | numéro de port
Port Variable	1110 111w
LEA	1000 1101 | mod reg r/m

Instructions arithmétiques	
Instruction	**Format**
`INC`	
Registre/Mémoire	`1111 111w` `mod 000 r/m`
Registre	`0100 0 reg`
`DEC`	
Registre/Mémoire	`1111 111w` `mod 001 r/m`
Registre	`0100 1 reg`
`ADD`	
Registre à Registre	`0000 00dw` `mod reg r/m`
Registre à Mémoire	`0000 000w` `mod reg r/m`
Mémoire à Registre	`0000 001w` `mod 000 r/m`
Immédiat à Registre/Mémoire	`1000 00 s w` `mod 000 r/m` `donnée immédiate`
Immédiat à Accumulateur (forme courte)	`0000 010 w` `donnée immédiate`
`ADC`	
Registre à Registre	`0001 00dw` `mod reg r/m`
Registre à Mémoire	`0001 000w` `mod reg r/m`
Mémoire à Registre	`0001 001w` `mod 000 r/m`
Immédiat à Registre/Mémoire	`1000 00 s w` `mod 010 r/m` `donnée immédiate`
Immédiat à Accumulateur (forme courte)	`0001 010 w` `donnée immédiate`
`SUB`	
Registre à Registre	`0010 10dw` `mod reg r/m`
Registre à Mémoire	`0010 100w` `mod reg r/m`
Mémoire à Registre	`0010 101w` `mod 000 r/m`
Immédiat à Registre/Mémoire	`1000 00 s w` `mod 101 r/m` `donnée immédiate`
Immédiat à Accumulateur (forme courte)	`0010 110 w` `donnée immédiate`
`SBB`	
Registre à Registre	`0001 10dw` `mod reg r/m`
Registre à Mémoire	`0001 100w` `mod reg r/m`
Mémoire à Registre	`0001 101w` `mod 000 r/m`
Immédiat à Registre/Mémoire	`1000 00 s w` `mod 011 r/m` `donnée immédiate`
Immédiat à Accumulateur (forme courte)	`0001 110 w` `donnée immédiate`
`MUL`	
Accumulateur avec Registre/Mémoire	`1111 011w` `mod 100 r/m`
`DIV`	
Accumulateur par Registre/Mémoire	`1111 011w` `mod 110 r/m`
`NEG`	
Immédiat	`1111 011w` `donnée immédiate`
`IMUL`	
Accumulateur par Registre/Mémoire	`1111 101w` `mod 101 r/m`
Registre par Registre/Mémoire	`0000 1111` `1010 1111` `mod reg r/m`
Registre/Mémoire par Immédiat et Registre	`0110 10s1` `mod reg r/m` `donnée immédiate`
`IDIV`	
Accumulateur par Registre/Mémoire	`1111 011w` `mod 111 r/m`

CMP	
Registre avec Registre	0011 10dw mod reg r/m
Registre avec Mémoire	0011 101w mod reg r/m
Mémoire avec Registre	0001 100w mod 000 r/m
Immédiat avec Registre/Mémoire	1000 00 s w mod 111 r/m donnée immédiate
Immédiat avec Accumulateur (forme courte)	0011 110 w donnée immédiate
DAA	0010 0111
DAS	0010 1111
AAA	0011 0111
AAS	0011 1111
AAM	1101 0100 0000 1010
AAD	1101 0101 0000 1010
CBW	1001 1000
CWD	1001 1001

Pour les instructions logiques, on utilise le champ TTT suivant :

TTT	Instruction
000	ROL
001	ROR
010	RCL
011	RCR
100	SHL/SAL
101	SHR
111	SAR

Instructions logiques	
Instruction	**Format**
NOT	1111 011w | mod 010 r/m
AND	
Registre à Registre	0010 00dw | mod reg r/m
Registre à Mémoire	0010 000w | mod reg r/m
Mémoire à Registre	0010 001w | mod reg r/m
Immédiat à Registre/Mémoire	1000 000w | mod 100 r/m | donnée immédiate
Immédiat à Accumulateur (forme courte)	0010 010w | donnée immédiate
OR	
Registre à Registre	0000 10dw | mod reg r/m
Registre à Mémoire	0000 100w | mod reg r/m
Mémoire à Registre	0000 101w | mod reg r/m
Immédiat à Registre/Mémoire	1000 000w | mod 001 r/m | donnée immédiate
Immédiat à Accumulateur (Forme courte)	0000 110w | donnée immédiate
XOR	
Registre à Registre	0011 00dw | mod reg r/m
Registre à Mémoire	0011 000w | mod reg r/m
Mémoire à Registre	0011 001w | mod reg r/m
Immédiat à Registre/Mémoire	1000 000w | mod 110 r/m | donnée immédiate
Immédiat à Accumulateur (Forme courte)	0011 010w | donnée immédiate
TEST	
Registre/Mémoire et Registre	1000 010w | mod reg r/m
Immédiat et Registre/Mémoire	1111 011w | mod 000 r/m | donnée immédiate
Immédiat et Accumulateur (Forme courte)	1010 100w | donnée immédiate
ROL, ROR, SAL, SAR, SHL, SHR RCL, RCR	
Registre/Mémoire de 1	1101 000w | mod TTT r/m
Registre/Mémoire de CL	1101 001w | mod TTT r/m
Registre/Mémoire de immédiat	1100 000w | mod 0 TTT r/m | immed 8-bit data
SHLD	
Registre/Mémoire de immédiat	0000 1111 | 1010 0100 | mod 0 reg r/m | immed 8-bit data
Registre/Mémoire de CL	0000 1111 | 1010 0101 | mod 0 reg r/m
SHRD	
Registre/Mémoire de immédiat	0000 1111 | 1010 1100 | mod 0 reg r/m | immed 8-bit data
Registre/Mémoire de CL	0000 1111 | 1010 1101 | mod 0 reg r/m

Instructions de contrôle de transfert	
Instruction	**Format**
JMP	
Court	1110 1011 déplacement sur 8 bits
Direct dans le Segment	1110 1001 déplacement complet
Indirect par Registre/Mémoire dans le Segment	1111 1111 mod 100 r/m
Direct inter-segment	1110 1010 décalage, sélecteur
Indirect inter-segment	1111 1111 mod 101 r/m
CALL	
Direct dans le Segment	1110 1000 déplacement complet
Indirect par Registre/Mémoire dans le Segment	1111 1111 mod 010 r/m
Direct inter-segment	1001 1010 décalage, sélecteur
Indirect inter-segment	1111 1111 mod 011 r/m
RET	
Dans le segment	1100 0011
Dans le segment en ajoutant un immédiat à SP	1100 0010 16-bit displ
Inter-segment	1100 1011
Inter-segment en ajoutant un immédiat à SP	1100 1010 16-bit displ

Instructions de branchement conditionnel	
Instruction	**Format**
JO	
8-Bit Displacement	0111 0000 8-bit displacement
Full Displacment	0000 1111 1000 0000 full displacement
JNO	
8-Bit Displacement	0111 0001 8-bit displacement
Full Displacment	0000 1111 1000 0001 full displacement
JB/JNAE	
8-Bit Displacement	0111 0010 8-bit displacement
Full Displacment	0000 1111 1000 0010 full displacement
JNB/JAE	
8-Bit Displacement	0111 0011 8-bit displacement
Full Displacment	0000 1111 1000 0011 full displacement
JE/JZ	
8-Bit Displacement	0111 0100 8-bit displacement
Full Displacment	0000 1111 1000 0100 full displacement
JNE/JNZ	
8-Bit Displacement	0111 0101 8-bit displacement
Full Displacment	0000 1111 1000 0101 full displacement
JBE/JNA	
8-Bit Displacement	0111 0110 8-bit displacement
Full Displacment	0000 1111 1000 0110 full displacement
JNBE/JA	
8-Bit Displacement	0111 0111 8-bit displacement
Full Displacment	0000 1111 1000 0111 full displacement
JS	
8-Bit Displacement	0111 1000 8-bit displacement
Full Displacment	0000 1111 1000 1000 full displacement

JNS	
8-Bit Displacement	0111 1001 8-bit displacement
Full Displacment	0000 1111 1000 1001 full displacement
JP/JPE	
8-Bit Displacement	0111 1010 8-bit displacement
Full Displacment	0000 1111 1000 1010 full displacement
JNP/JPO	
8-Bit Displacement	0111 1011 8-bit displacement
Full Displacment	0000 1111 1000 1011 full displacement
JL/JNGE	
8-Bit Displacement	0111 1100 8-bit displacement
Full Displacment	0000 1111 1000 1100 full displacement
JNL/JGE	
8-Bit Displacement	0111 1101 8-bit displacement
Full Displacment	0000 1111 1000 1101 full displacement
JNLE/JG	
8-Bit Displacement	0111 1111 8-bit displacement
Full Displacment	0000 1111 1000 1111 full displacement
JCXZ	1110 0011 8-bit displacement
JECXZ	1110 0011 8-bit displacement
LOOP	1110 0010 8-bit displacement
LOOPZ/LOOPE	1110 0001 8-bit displacement
LOOPNZ/LOOPNE	1110 0000 8-bit displacement

Interruptions	
Instruction	**Format**
INT	
Type spécifié	1100 1101 type
Type 3	1100 1100
INTO	1100 1110
IRET	1100 1111
BOUND	0110 0010 mod reg r/m

Instructions de contrôle des indicateurs	
Instruction	**Format**
STC	1111 1001
CLC	1111 1000
CMC	1111 0101
STD	1111 1101
CLD	1111 1100
STI	1111 1011
CLI	1111 1010
POPF	1001 1101
PUSHF	1001 1100
LAHF	1001 1111
SAHF	1001 1001

CLTS	0000 1111	0000 01100	

Instructions de contrôle du processeur		
Instruction	**Format**	
HLT	1111 0100	
NOP	1001 0000	
WAIT	1001 0000	

Instructions sur les chaînes de caractères		
Instruction	**Format**	
MOVS	1010 010w	
INS	0110 110w	
OUTS	0110 111w	
STOS	1010 101w	
LODS	1010 110w	
CMPS	1010 011w	
SCAS	1010 111w	
XLAT	1101 0111	

Instructions concernant les procédures		
Instruction	**Format**	
ENTER	1100 0100	16-bit displacement, 8-bit level
LEAVE	1100 1001	

Instructions sur les segments			
Instruction	**Format**		
LDS	1100 0101	mod reg r/m	
LES	1100 0100	mod reg r/m	
LFS	0000 1111	1011 0100	mod reg r/m
LGS	0000 1111	1011 0101	mod reg r/m
LSS	0000 1111	1011 0010	mod reg r/m

Instructions de traitement de bits				
Instruction	**Format**			
BSF	0000 1111	1011 1100	mod reg r/m	
BSR	0000 1111	1011 1101	mod reg r/m	
BT				
Register/Memory, Immediate	0000 1111	1011 1010	mod 100 r/m	immed 8-bit data
Register/Memory, Register	0000 1111	1011 0011	mod reg r/m	
BTC				
Register/Memory, Immediate	0000 1111	1011 1010	mod 111 r/m	immed 8-bit data
Register/Memory, Register	0000 1111	1011 1011	mod reg r/m	
BTR				
Register/Memory, Immediate	0000 1111	1011 1010	mod 110 r/m	immed 8-bit data
Register/Memory, Register	0000 1111	1011 0011	mod reg r/m	
BTS				
Register/Memory, Immediate	0000 1111	1011 1010	mod 010 r/m	immed 8-bit data
Register/Memory, Register	0000 1111	1010 1011	mod reg r/m	
IBTS	0000 1111	1010 0111		
XBTS	0000 1111	1010 0110		

Instructions conditionnelles sur les octets			
Instruction	**Format**		
SETO	0000 1111	1001 0000	mod 000 r/m
SETNO	0000 1111	1001 0001	mod 000 r/m
SETB/SETNAE	0000 1111	1001 0010	mod 000 r/m
SETNB	0000 1111	1001 0011	mod 000 r/m
SETE/SETZ	0000 1111	1001 0100	mod 000 r/m
SETNE/SETNZ	0000 1111	1001 0101	mod 000 r/m
SETBE/SETNA	0000 1111	1001 0110	mod 000 r/m
SETNBE/SETA	0000 1111	1001 0111	mod 000 r/m
SETS	0000 1111	1001 1000	mod 000 r/m
SETNS	0000 1111	1001 1001	mod 000 r/m
SETP/SETPE	0000 1111	1001 1010	mod 000 r/m
SETNP/SETPO	0000 1111	1001 1011	mod 000 r/m
SETL/SETNGE	0000 1111	1001 1100	mod 000 r/m
SETNL/SETGE	0000 1111	1001 1101	mod 000 r/m
SETLE/SETNG	0000 1111	1001 1110	mod 000 r/m
SETNLE/SETG	0000 1111	1001 1111	mod 000 r/m