

## 4. Initiation à l'assembleur



# 1. Introduction

Pour mieux comprendre la structure interne d'un ordinateur, nous aborderons la langage d'assemblage d'un processeur contemporain, le Pentium III d'Intel. Le but du cours n'est donc pas de faire de vous des programmeurs en assembleur.

Qu'est-ce que l'assembleur ou le langage d'assemblage ? Pour obtenir une réponse à cette question, nous vous recommandons de lire à ce stade-ci le chapitre 13 du volume de Zanella et Ligier.

Nous utilisons pour ce cours l'assembleur Masm32 de Microsoft disponible gratuitement sur le Web à l'adresse suivante :

<http://www.pbq.com.au/home/hutch/masm.htm>.

Comme débogueur, nous recommandons le Enhanced Debugger d'Intel, également disponible gratuitement à l'adresse suivante :

<http://developer.intel.com/vtune/compilers/cpp/demo.htm>.

Il est fourni avec le compilateur C/C++ 4.5 que vous pouvez utiliser avec Visual Studio de Microsoft si le cœur vous en dit.

Vous pouvez également faire de l'assembleur dans le cadre d'un compilateur C/C++, comme dans Visual C/C++ de Microsoft. Dans ce cas, vous créez une fonction en C ou en C++, puis vous y insérez la commande `_asm { }`. Vous pouvez ensuite insérer entre les accolades autant d'instructions assembleur que vous désirez. Il faut donc déclarer la fonction comme si c'était une fonction C, puis faire la déclaration en question comme suit :

```
long maFonction (long x)
{
    _asm
    {
        mov    eax, x           ; place la valeur de x dans le registre eax
somme:      add    eax, 4       ; additionne 4 à eax.
    }                       ; le résultat est le contenu de eax.
}
```

Cette fonction peut ensuite être appelée par un programme principal :

```
void main()
{
    long resultat;
        resultat = maFonction(75);
}
```

Vous remarquerez en passant la structure d'une instruction assembleur. Elle est constituée d'un **opcode**, par exemple mov, add, etc. et **d'opérandes**, par exemple, eax, x, ou 4. L'opérande de gauche est appelé **opérande destination** et c'est là que le résultat de l'instruction sera enregistré. L'opérande de droite est appelé **opérande source**.

On peut mettre un commentaire à la suite de l'instruction. Dans le contexte C, on utilise la syntaxe C (//). Avec un assembleur standard, on précède plutôt le commentaire d'un point-virgule (;). On ne peut avoir qu'une instruction par ligne. On peut avoir une étiquette au début d'une ligne pour identifier un énoncé, par exemple. somme:. Cette étiquette doit être suivie d'un deux-points (:).

Pour les travaux pratiques du présent cours, nous n'utiliserons pas l'assembleur intégré dans le C, mais l'assembleur standard.

Pour pouvoir écrire de l'assembleur, il est indispensable d'avoir une connaissance très précise de l'architecture du processeur avec lequel on travaille. Nous passons donc à une description de l'architecture du Pentium III d'Intel.

## 2. Architecture du Pentium III

### 2.1 Historique

L'histoire de la famille 80x86 d'Intel commence dans les années 70 avec le 8080, un processeur de 8 bits avec un bus d'adresses de 16 bits, qui pouvait adresser un total de 64 Ko.

Vers 1980, le 8086 et le 8088 font leur apparition, ce dernier avec le premier PC d'IBM. Ce sont des processeurs de 16 bits avec un bus d'adresses de 20 bits, qui avaient une capacité d'adressage de 1 Mo. Le 8088 diffère du 8086 par la largeur du bus de données externe qui est de 16 bits dans le 8086 et de 8 bits dans le 8088.

Toutefois, même si le bus d'adresses était de 20 bits, les registres internes d'adresses étaient toujours de 16 bits pour assurer la compatibilité avec le 8080. Comment donc accéder au reste de la mémoire? Toute la complexité des processeurs Intel vient de la solution adoptée à cette époque pour régler ce problème.

On décida que l'adresse serait constituée des 16 bits des registres internes ajoutée à 16 fois le contenu d'un de quatre registres appelés registres de segment. Ces quatre registres étaient CS (Code Segment), DS (Data Segment), SS (Stack Segment) et ES (Extra Segment).

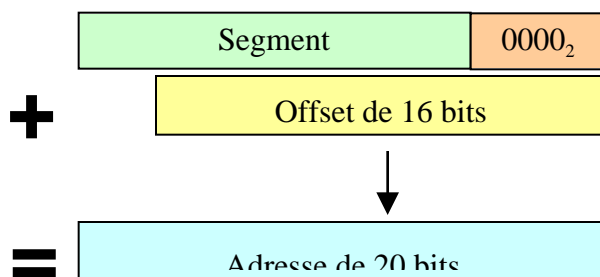


Figure 1.

On remarque que chaque segment a une taille de 64 Ko (offset 16 bits et  $2^{16}$ ), et que la distance entre chaque segment peut aller de 16 octets à 64 Ko.

La capacité totale d'adressage est  $0xFFFF0 + 0xFFFF = 0x10FFEF$ ,

qui dépasse légèrement 1 Mo ( $0xFFFFF$ ).

Le 80286 fait son apparition quelques années plus tard avec un bus d'adresses de 24 bits (capacité de 16 Mo). C'est là que les choses se compliquent.

Jusqu'alors, les processeurs fonctionnaient en ce qu'Intel appelle le « **mode réel** ». Les systèmes d'exploitation utilisés avec ces processeurs étaient mono-tâches et mono-usagers (par exemple, MS-DOS). Les registres de segment contenaient de vraies adresses, et l'utilisateur pouvait accéder sans limite à toutes les ressources du système : les périphériques, les interruptions, etc.

Toutefois, les registres de segment demeuraient de 16 bits. Comment donc accéder aux 16 Mo que permettait le bus d'adresses de 24 bits du 80286?

Pour permettre cet adressage sur une plus grande plage de mémoire ainsi que l'avènement de systèmes d'exploitation plus performants, Intel introduisit avec le 80286 le « **mode protégé** ».

Mais comme la plupart des applications roulant sous MS-DOS, qui dominait le marché, étaient incompatibles avec le mode protégé, on continua pendant des années à fonctionner en mode réel avec une capacité de mémoire de 1 Mo.

Le 80286 fut donc longtemps considéré comme un 8086 rapide parce que personne ne savait comment utiliser le mode protégé. Pourtant, ce processeur offrait la mémoire virtuelle, des droits d'accès pour la sécurité, des niveaux de privilège d'exécution, etc.

Pendant ce temps, Motorola mettait en marché la famille 68000, qui offrait des registres de 32 bits et, à partir de 1985 avec le 68020, une capacité d'adressage de 4 Go.

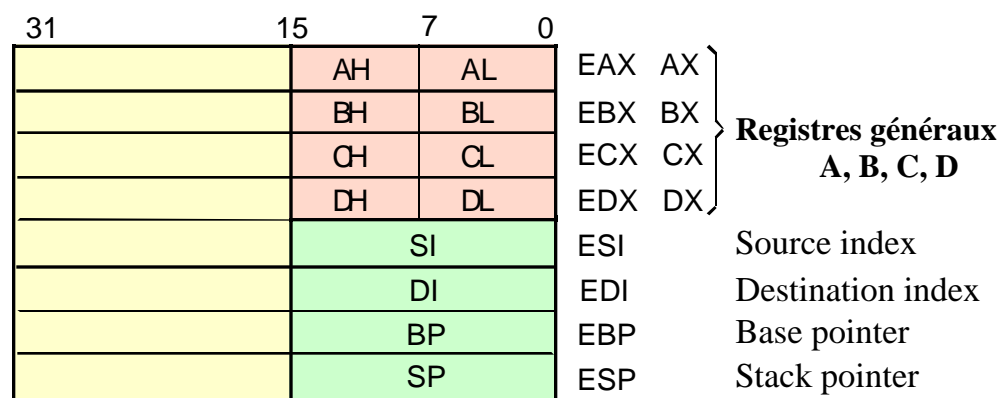
En 1987, Intel met au point le 80386, puis le 80486, ensuite, le Pentium et, finalement, en 1997, le Pentium II que nous retrouvons dans les ordinateurs que nous utilisons pour ce cours. Ils fonctionnent tous sous Windows NT 4.0.

## 2.2 Registres d'entiers

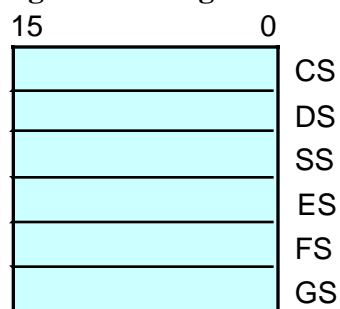
Pour l'instant, nous abordons que la partie visible au programmeur qui travaille en mode utilisateur. Nous aborderons l'architecture du Pentium II plus en détail au chapitre X.

Un **registre** est un contenant dans lequel on peut placer des motifs de bits de 8, 16 ou 32 bits. On peut appeler un registre par son nom et effectuer des opérations en utilisant les instructions machine. Par exemple, **add** eax, 4 additionne 4 au contenu du registre eax et place le résultat dans eax. Dans ce processeur, un motif de 8 bits s'appelle un octet (byte), un motif de 16 bits s'appelle un mot et un motif de 32 bits, un double mot.

La figure suivante illustre les registres disponibles au programmeur pour le traitement des entiers.



### Registres de Segment



### Registre fantômes

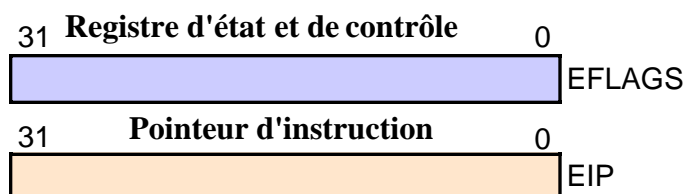
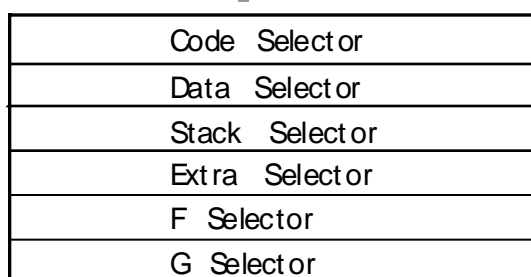


Figure 2.

Le Pentium possède huit registres généraux. quatre registres de données A, B, C et D, et quatre registres de pointeurs (adresses), ESI, EDI, EBP et ESP. Pour désigner le registre A au complet (32 bits), on utilise le nom EAX. Pour désigner les 16 bits de poids faible, on utilise AX. Pour désigner les 8 bits de poids faible, on utilise AL. On utilise AH pour désigner les bits 8 à 15 du registre A. Il en va de même pour les trois autres registres de données, B, C et D.

De la même façon, le registre ESI désigne le registre de 32 bits, tandis que SI désigne les 16 bits de poids faible de ESI.

Certains registres généraux jouent un rôle spécial. Le registre C sert souvent de compteur, par exemple dans l'instruction **loop**. Le registre D sert d'extension au registre A pour enregistrer des nombres de double précision pour la multiplication et la division. On peut enregistrer un double mot de 32 bits dans la paire DX:AX ou un quadruple mot de 64 bits dans la paire EDX:EAX.

Certains registres d'adresse jouent également un rôle spécial à l'occasion. Ainsi, le registre ESI sert d'adresse source pour les opérations de chaîne, tandis que le registre EDI sert d'adresse destination. On peut utiliser ESI et EDI comme registres généraux quand on ne les utilise pas pour des opérations de chaîne.

Les registres d'adresse EBP et ESP ont des fonctions spécialisées. EBP sert de pointeur pour les blocs de pile (paramètres et variables locales), tandis que ESP est le pointeur de pile système. Vous ne devez pas modifier EBP ou ESP à moins de savoir exactement ce que vous faites, sous peine de faire se planter l'ordinateur.

Nous étudierons le registre EFLAGS en détail un peu plus loin.

Le registre EIP est le compteur ordinal et contient l'adresse de la prochaine instruction à exécuter.

Le Pentium possède également six registres de segment: CS, DS, SS, ES, FS et GS. En mode réel (voir historique, plus haut), ces registres contiennent un nombre de 16 bits qu'on combine avec un offset de 16 bits contenu dans un des registres de pointeurs (si, di, etc.) pour former une adresse de 20 bits. En mode protégé, ils contiennent plutôt des **sélecteurs** de segment de 16 bits qui ont la forme suivante :



Figure 3. Format d'un sélecteur



Le champ **Index** est un offset de 13 bits qui sert d'index dans une table **de descripteurs de segment**. Le champ **TI** indique dans quelle table ce descripteur se trouve : 0 indique qu'il s'agit de la table globale de descripteurs (Global Descriptor Table ou **GDT**), 1 indique qu'il s'agit de la table locale de descripteurs (Local Descriptor Table ou **LDT**). Le champ **RPL** indique le niveau de privilège demandé (0 à 3, 0 étant le niveau le plus privilégié).

Les registres fantômes associés aux registres de segment ne sont pas accessible au programmeur. Ils contiennent une copie du descripteur de la table.

Un **descripteur de segment** est une structure de données de 64 bits qui contient toutes les informations relatives à un segment : sa position en mémoire, sa taille, ses privilèges, etc.

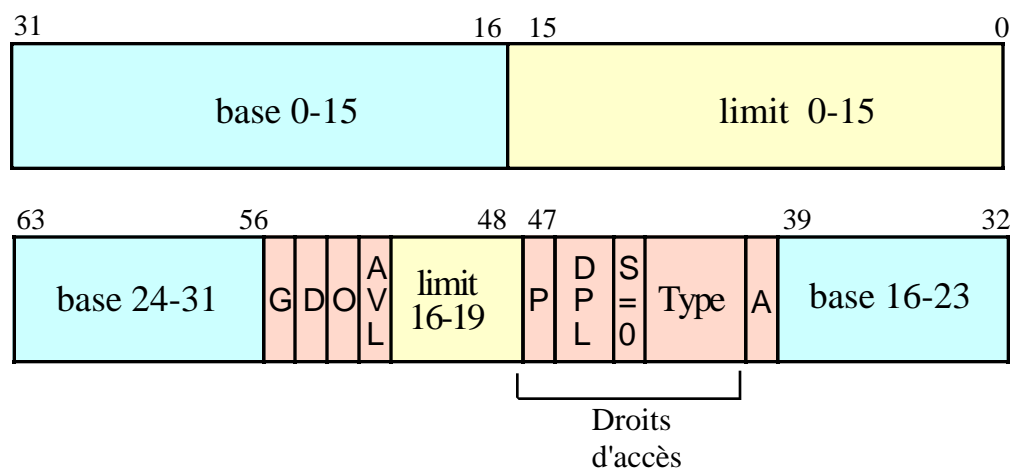


Figure 4

Le champ **base** de 32 bits donne l'adresse de départ du segment. Le champ **limit** de 20 bits donne la taille du segment, donc 1 méga unités. Ces unités peuvent être soit des octets si le bit **G** (granularité) est 0, soit des pages de 4 Ko si le bit **G** est 1. La taille maximum d'un segment est donc de 4 Go.

## 2.3 Registres de virgule flottante

Pour l'exécution d'instructions de virgule flottante, le Pentium possède huit registres de données de 80 bits, un registre de contrôle, un registre d'état et divers autres registres.

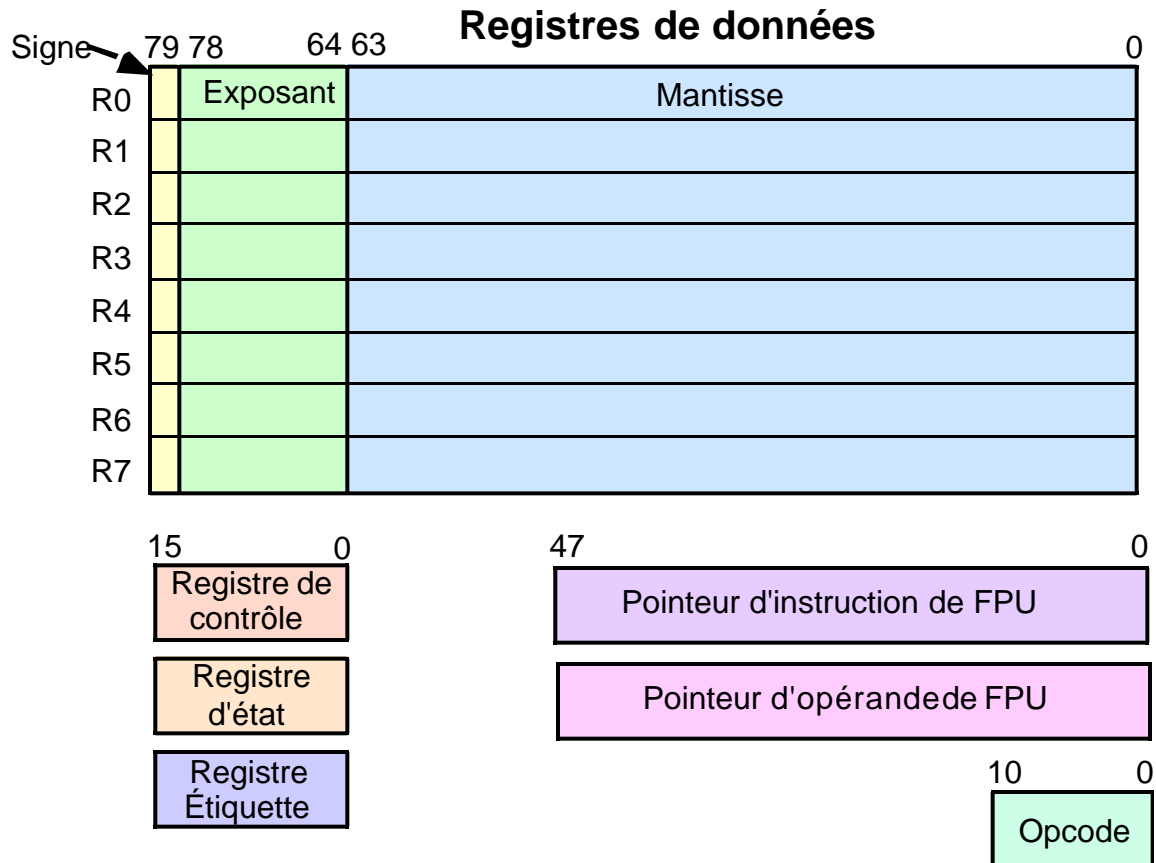


Figure 5.

Les registres de données de virgule flottante, contrairement aux registres d'entiers, ne sont pas à accès direct et ne peuvent pas être appelés par leur nom. Ils constituent plutôt une pile, comme suit :

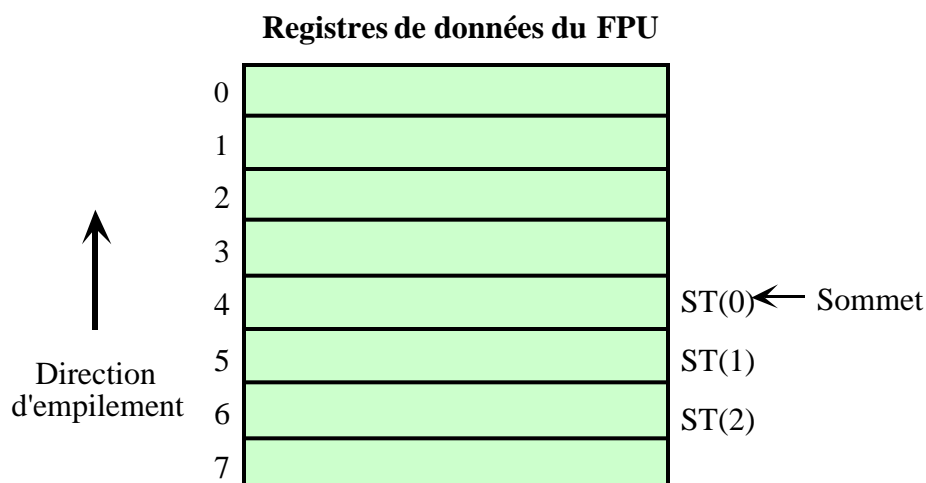


Figure 6.

Quand on effectue le chargement d'une donnée, elle est empilée, i.e. ST(0) est décrémenté de 1 et la donnée est inscrite à cette adresse. Ce qu'il y avait sur la pile auparavant devient ST(1), ST(2), etc.

## 2.4 Registres MMX

Ces mêmes 8 registres peuvent servir comme registres de 64 bits pour la programmation multimédia. On les appelle alors mm0 à mm7.

## 2.5 Registres SIMD

Dans le Pentium III, les 8 registres de virgule flottante ont été étendus à 128 bits. Ils s'appellent alors xmm0 à xmm7. Ils peuvent contenir quatre nombres de virgule flottante de simple précision ou deux nombres de virgule flottante de double précision qu'on traite simultanément.

Mentionnons en passant que notre assembleur Masm32 supporte la programmation MMX mais pas encore la programmation SIMD du Pentium III.

## 2.6 Le Registre EFLAGS

Comme tout processeur, le Pentium possède un certain nombre d'indicateurs. Ce sont des bits d'états qui sont mis à jour lors de l'exécution de la plupart des instructions pour indiquer au programme le résultat de l'instruction.

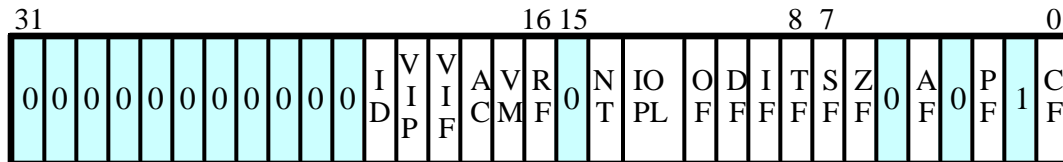


Figure 7

### Indicateurs usuels

CF = Carry Flag ou retenue

PF = Parity Flag ou parité

AF = Auxiliary Carry Flag ou retenue auxiliaire

ZF = Zero Flag ou indicateur de zéro

SF = Sign Flag, ou indicateur de signe

DF = Direction Flag ou indicateur de direction

OF = Overflow Flag ou indicateur de débordement de capacité

### Indicateurs de programmation système

TF = Trap Flag ou indicateur de trappe

IF = Interrupt Enable Flag ou indicateur d'autorisation d'interruption

IOPL = I/O Privilege Level ou indicateur de niveau de privilège d'entrée-sortie

NT = Nested Task Flag ou indicateur de tâche imbriquée

RF = Resume Flag

VM = Virtual 386 Mode

AC = Alignment Check

VIF = Virtual Interrupt Flag

VIP = Virtual Interrupt Pending

ID = Identification Flag

La plupart des instructions affectent un ou plusieurs indicateurs usuels afin de permettre au programme d'avoir des informations sur le résultat de l'opération. Ces indicateurs serviront, entre autres, à prendre une décision lors d'un branchement conditionnel. Par

exemple, après une addition, ou pourrait se demander s'il y a eu une retenue, si oui, de faire telle chose, et si non, de faire autre chose. C'est grâce à eux qu'on peut implanter les structures de contrôle comme **if**, **while**, **switch**, etc.

## 2.7 Autres registres

D'autres registres sont disponibles et servent principalement à la programmation système :

Control registers	CR0, CR1, CR2, CR3 et CR4
Global Descriptor Table Register	GDTR
Interrupt Descriptor Table Register	IDTR
Local Descriptor Table Register	LDTR
Task Register	TR
Debug Registers	DR0 à DR7
Test Registers	TR6 et TR7
Time Stamp Counter	TSC

## 3. Outils de programmation

Il existe plusieurs logiciels pour programmer un PC en C/C++ et en assembleur. Les plus connus sont :

- Microsoft Visual Studio
- Borland C/C++ et Turbo C/C++ avec TASM
- Metrowerks Code Warrior

Pour la programmation en assembleur standard, nous allons utiliser un assembleur autonome, Masm32, basé sur le MASM de Microsoft.

### 3.1 Masm32

Le programme principal de Masm32 s'appelle QEditor. C'est un éditeur de texte pour la programmation. Il comporte un menu **Project** qui permet la compilation d'un fichier assembleur (.asm).

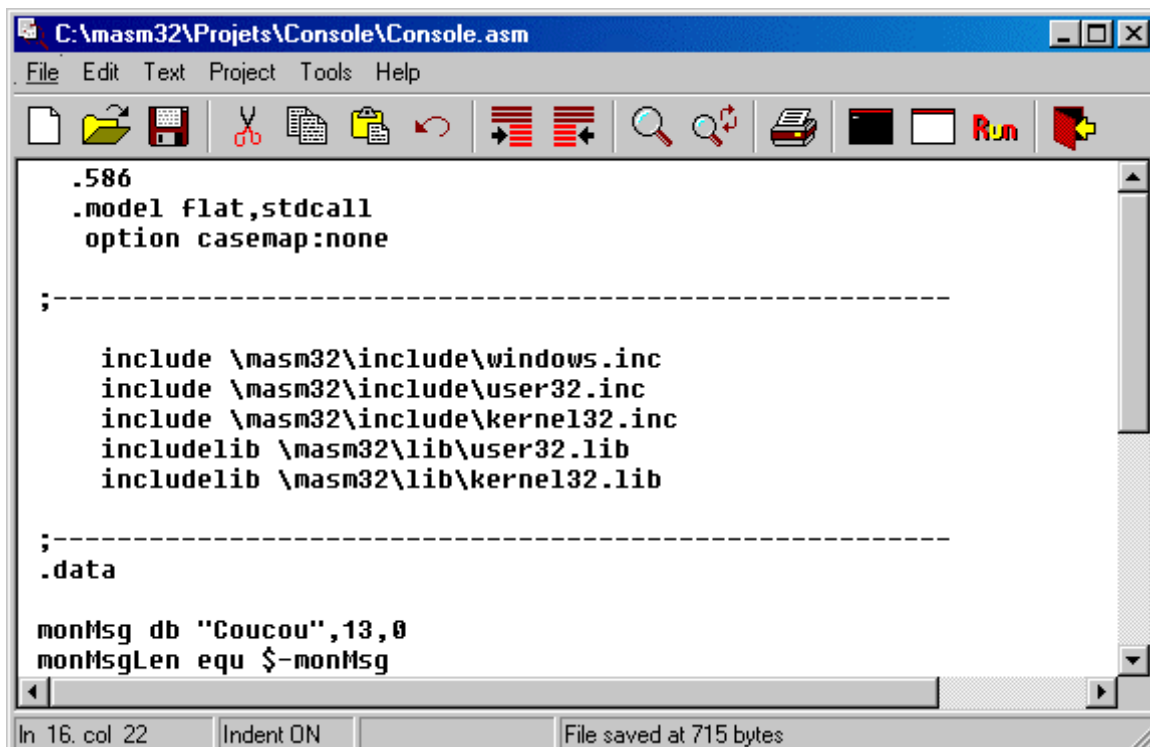


Figure 8

Pour créer un nouveau fichier, on clique sur **New** dans le menu **File**. On obtient alors une fenêtre vide. Une fois qu'on a écrit du code dedans, on peut la sauvegarder en cliquant sur **Save** ou **Save As** dans le menu **File**.

On a deux modes de fonctionnement pour les entrées/sorties des programmes en assembleur. Le mode **Console**, qui utilise une fenêtre de style DOS, et le mode normal, qui nécessite de la programmation Windows pour utiliser des champs d'édition pour les entrées-sorties. Nous utiliserons les deux dans le cadre de ce cours.

On utilise donc **QEditor** pour créer un code source en assembleur. Ensuite, on doit absolument enregistrer ce code sur disque au moyen de la commande **Save** ou **Save As** dans le menu **File**. Ensuite, pour un programme en mode normal on peut utiliser **Compile ASM file**, puis **Link OBJ File**, ou encore **Assemble & Link** dans le menu **Project**.

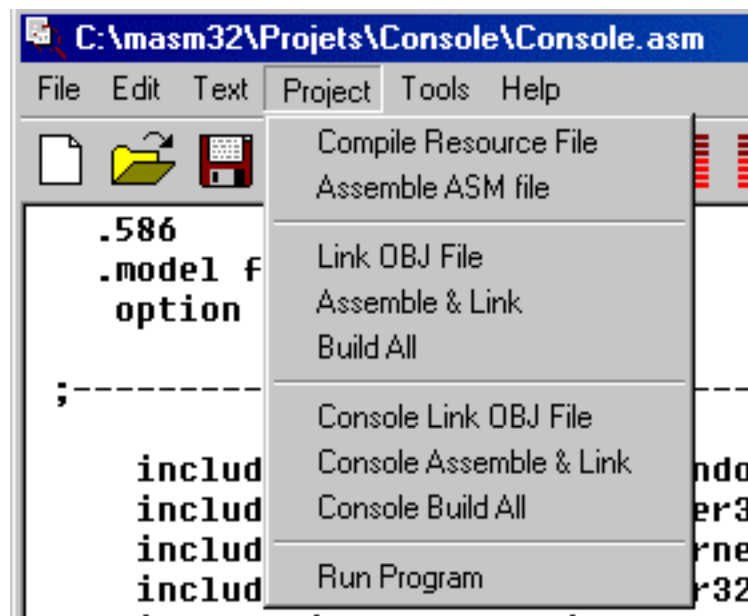


Figure 9

Pour un programme en mode console, on utilise plutôt **Assemble ASM file**, puis **Console Link OBJ File** ou encore **Console Assemble & Link**.

## 3.2 Débogage

Il est pratiquement indispensable d'utiliser un débogueur lorsqu'on programme en assembleur. Un bon débogueur permet :

- a. D'arrêter le programme à un point d'arrêt (breakpoint) spécifique.
- b. D'exécuter un programme pas à pas pour trouver une erreur.
- c. De visualiser le contenu des registres du processeur.
- d. De visualiser le contenu d'adresses mémoire.
- e. De désassembler le code généré.

Il existe deux débogueurs convenables gratuits : **WinDbg** (Microsoft Windows Debugger), version 5.1,



et Enhanced Debugger d'Intel ou **Edb32**, version 4.5.



Malheureusement, ce dernier, pourtant le meilleur des deux, ne fonctionne qu'avec Windows NT 4.0 ou Windows 2000. Vous pourrez toutefois l'utiliser dans les laboratoires du département. Si vous êtes trop loin pour pouvoir vous prévaloir de nos laboratoires et que vous utilisez Windows 95 ou Windows 98, alors téléchargez WinDbg, disponible gratuitement à l'adresse :

<http://www.microsoft.com/ddk/?RLD=460>

avec le DDK de Windows 98.

Dans les deux cas, on lance d'abord le débogueur. Ensuite, on ouvre un fichier exécutable à déboguer au moyen de la commande **Open Executable...** dans le menu **File**. Si le fichier source n'apparaît pas automatiquement, ouvrez-le avec la commande **Open Source File...**




On choisit ensuite dans le code source l'endroit où on veut commencer le débogage. On y place le curseur et on invoque la commande **Run to Cursor**. Dans WinDbg, cette commande est dans le menu **Debug**, tandis que dans le débogueur Intel, elle se trouve dans le menu **Execute**. On peut également placer un point d'arrêt en choisissant une ligne du code source et en appuyant sur la touche **F9**.



WinDbg présente les barres d'outils suivantes :




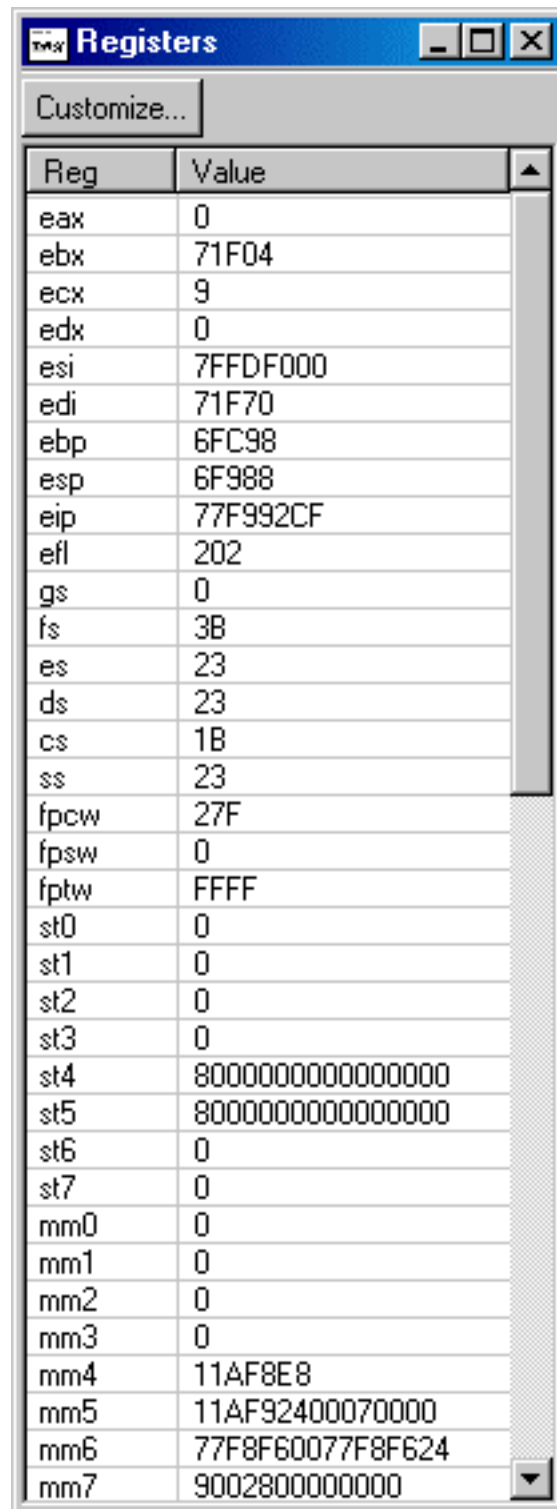


Figure 10

Pour démarrer le débogueur, cliquez sur la première icône en haut à gauche . L'exécution de votre programme devrait commencer et s'arrêter à la première instruction. Vous pouvez ensuite demander l'exécution pas à pas en cliquant sur l'icône  ou sur l'icône . La première entre dans les sous-programmes, la seconde considère un sous-programme comme une seule instruction.

L'icône  vous permet de sortir d'une fonction ou d'un sous-programme, tandis que l'icône  lance l'exécution jusqu'à l'instruction où se trouve le curseur d'édition.


Pour visualiser les registres, cliquez sur l'icône . La fenêtre suivante apparaît, dans laquelle vous pouvez observer les registres d'entiers et les registres de virgule flottante ainsi que les registres de segment, d'indicateurs, etc. Cette fenêtre est mise à jour à chaque instruction.



The image shows a 'Registers' window from a debugger. It has a title bar with a 'Registers' label and standard window controls. Below the title bar is a 'Customize...' button. The main area is a table with two columns: 'Reg' and 'Value'. The table lists various registers and their current values. The registers are grouped into general-purpose registers (eax, ebx, etc.), segment registers (gs, fs, etc.), floating-point registers (fpcw, fsw, etc.), and MMX registers (st0, mm0, etc.).

Reg	Value
eax	0
ebx	71F04
ecx	9
edx	0
esi	7FFDF000
edi	71F70
ebp	6FC98
esp	6F988
eip	77F992CF
efl	202
gs	0
fs	3B
es	23
ds	23
cs	1B
ss	23
fpcw	27F
fsw	0
fptw	FFFF
st0	0
st1	0
st2	0
st3	0
st4	8000000000000000
st5	8000000000000000
st6	0
st7	0
mm0	0
mm1	0
mm2	0
mm3	0
mm4	11AF8E8
mm5	11AF92400070000
mm6	77F8F60077F8F624
mm7	9002800000000

Figure 11

Pour visualiser le contenu de la mémoire, cliquez sur l'icône . Vous tapez ensuite une adresse dans la fenêtre d'édition en haut à gauche pour obtenir le contenu de cette adresse et des suivantes. La fenêtre de droite permet de choisir sous quel format on veut observer

## Assembleur

la mémoire. On a le choix entre ASCII, hexadécimal sur 8, 16 ou 32 bits, décimal signé ou non signé sur 16 ou 32 bits, ou virgule flottante de simple ou double précision.

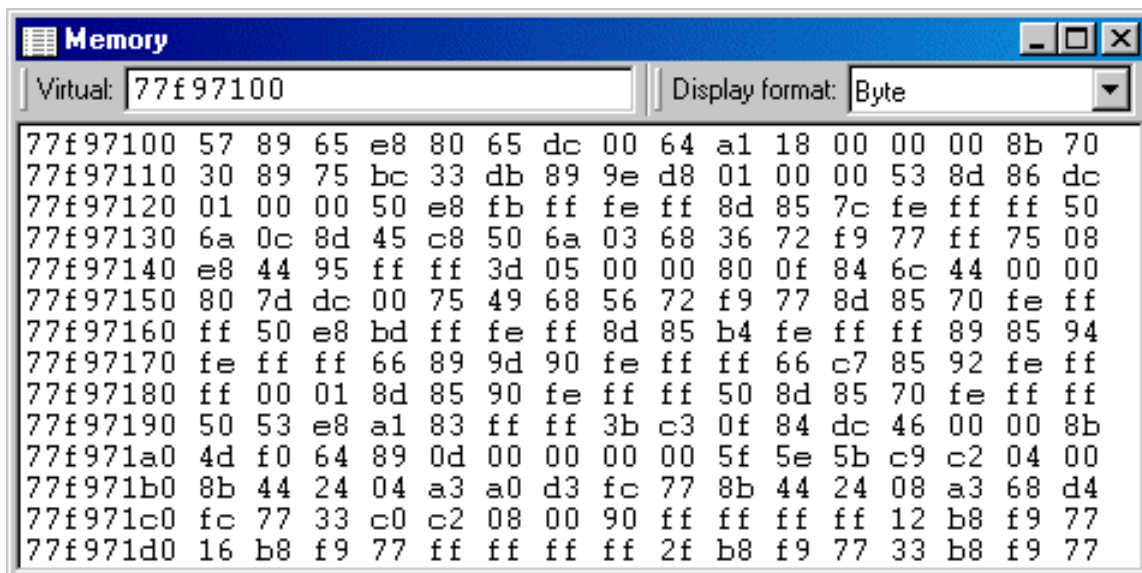



Figure 12

Pour visualiser du code désassemblé, cliquez sur l'icône . Vous verrez alors apparaître une fenêtre présentant, pour chaque instruction du code source en assembleur, en C ou en C++, le numéro de ligne, la traduction en assembleur, qui peut consister de plusieurs instructions, avec l'adresse où chacune est implantée et le codage en langage machine.

```

Disassembly
77f970d5 c9          leave
77f970d6 c20400      ret     0x4
77f970d9 55          push    ebp
77f970da 8bec        mov     ebp, esp
77f970dc 6aff        push    0xff
77f970de 68c871f977  push    0x77f971c8
77f970e3 68db80fb77  push    0x77fb80db
77f970e8 64a100000000 mov     eax, fs:[00000000]
77f970ee 50          push    eax
77f970ef 64892500000000 mov     fs:[00000000], esp
77f970f6 51          push    ecx
77f970f7 51          push    ecx
77f970f8 81ec8c010000 sub     esp, 0x18c
77f970fe 53          push    ebx
77f970ff 56          push    esi
77f97100 57          push    edi
77f97101 8965e8      mov     [ebp-0x18], esp
77f97104 8065dc00    and     byte ptr [ebp-0x24], 0x1
77f97108 64a118000000 mov     eax, fs:[00000018]
77f9710e 8b7030      mov     esi, [eax+0x30]
77f97111 8975bc      mov     [ebp-0x44], esi
77f97114 33db        xor     ebx, ebx
77f97116 899ed8010000 mov     [esi+0x1d8], ebx

```

Figure 13

## Exercices

1. Quel est le contenu du registre bl dans l'illustration de la Figure 11 ? Quel est celui de du registre bx ? Quel est celui du registre ebx ?
2. En vous basant sur la Figure 11, quelle est l'adresse de la prochaine instruction qui sera exécutée ?
3. La valeur du registre FLAGS est 0x0202 à la Figure 12. Quelle sont les valeurs de OF, SF, ZF et CF ?
4. Quelle est la valeur du registre EFLAGS en hexadécimal quand tous les indicateurs sont 0 ?

## Assembleur



## 4. Types de données

### 4.1 Entiers

Les types de données rencontrés dans le traitement d'entiers avec le Pentium sont les suivants :

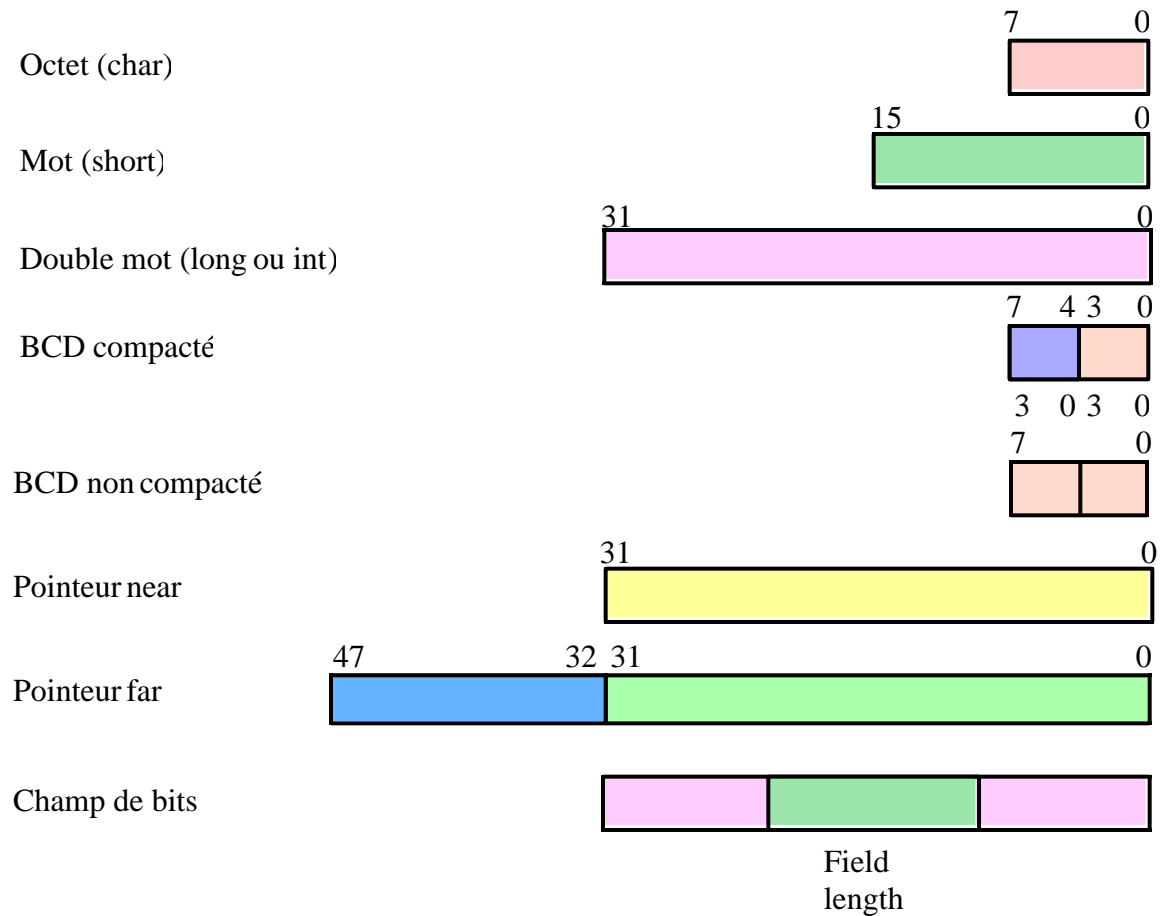


Figure 14

## 4.2 Virgule flottante

La famille Intel reconnaît aussi les quatre types de données suivants pour les calculs en virgule flottante.

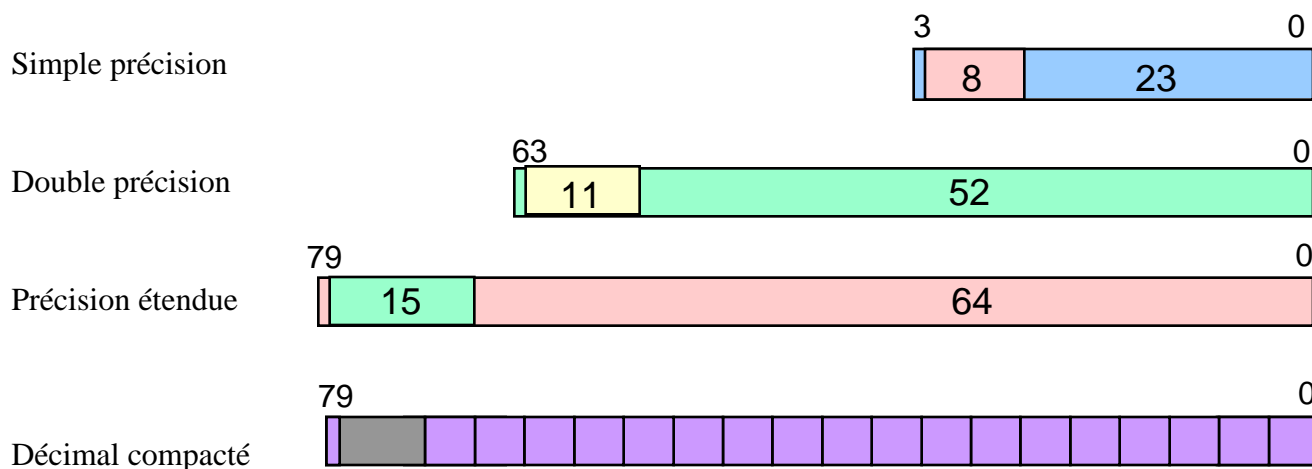


Figure 15

## 4.3 MMX

Pour les processeurs supportant la programmation MMX, on a les types suivants :

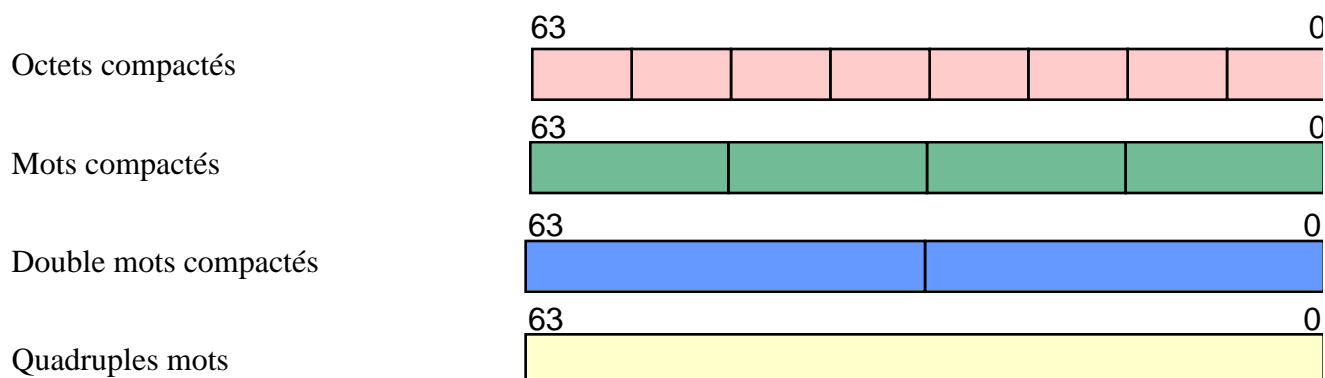


Figure 16

## 4.4 SIMD

Pour les processeurs supportant la programmation SIMD, on a le type suivant (packed single):

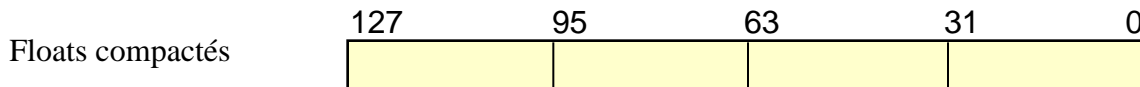
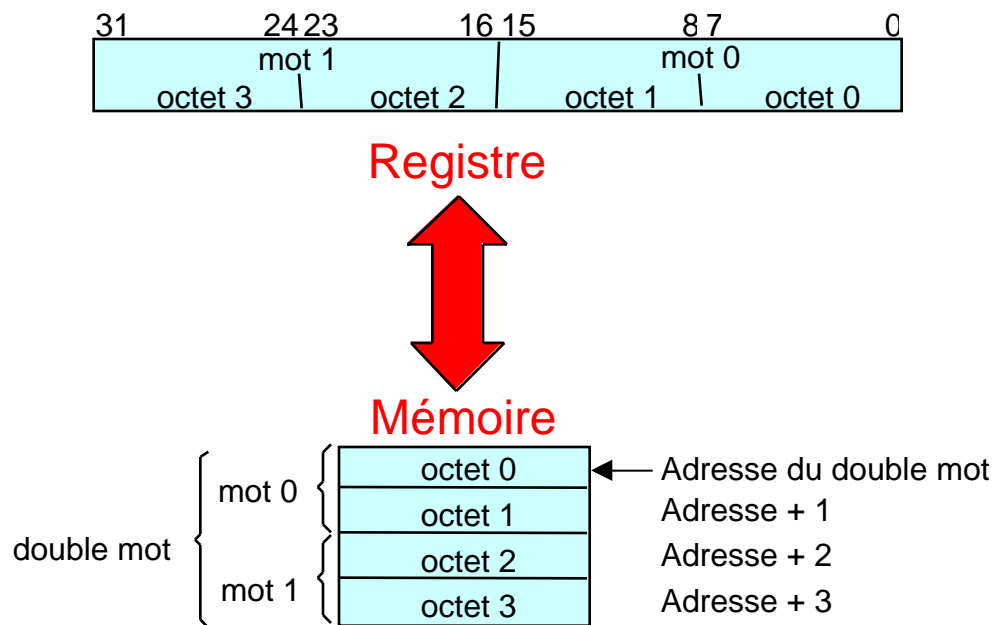


Figure 17

## 4.5 Rangement des données en mémoire

Comme la mémoire est organisée en octets, il y a deux façons de ranger les différents octets d'un registre de plus de 8 bits lors de l'écriture de son contenu en mémoire. On peut mettre l'octet de poids fort à l'adresse basse (big Endian) ou à l'adresse haute (little Endian). Voir Zanella et Ligier, p. 33, *Endianisme*. Intel a choisi la seconde solution :



{ mot 1 {

octet 2

Adresse + 2

octet 3

Adresse + 3

Figure 18



Ainsi, si le registre `eax` contient `0x12345678`, si on fait `mov 0x1000, eax`, alors on lira en mémoire aux adresses `0x1000`, `0x1001`, `0x1002`, `0x1003` les contenus : `0x78`, `0x56`, `0x34`, `0x12`.

## 4.6 Temps d'exécution des instructions

Le temps d'exécution du Pentium pour la plupart des instructions est de 1 à 5 cycles d'horloge. Des exceptions notoires sont :

<b>IMUL</b> et <b>MUL</b>	11
<b>IDIV</b> et <b>DIV</b>	22 à 46
<b>FDIV</b>	39 à 42

Il faudra donc les éviter dans la mesure du possible si on veut du code performant.

## Exercices

1. Supposons qu'on écrive en mémoire le contenu de ebx dans l'illustration de la Figure 11, par exemple à l'adresse 0x40000. Quel sera le contenu des adresses 0x40000, 0x40001, 0x40002 et 0x40003 ?

## 5. Structures de contrôle

### 5.1 Principales structures de contrôle

Les langages de programmation utilisent des structures de contrôle tels que **if-else**, **for**, **while**, **switch**, etc. Ces structures permettent l'exécution conditionnelle d'un bout de code dépendant de la valeur d'une variable. Ces structures de contrôle sont implantées au moyen de branchements conditionnels et inconditionnels. Par exemple :

<b>if</b> (a > b) {	<b>if:</b>	<b>cmp</b>	a,b
...		<b>jng</b>	else
...		...	
}		...	
<b>else</b> {		<b>jmp</b>	endif
...	<b>else:</b>	...	
...		...	
}			
...	<b>endif:</b>	...	

La première instruction compare a et b et saute si a n'est pas plus grand que b (**jng** = jump if not greater). Donc, si a <= b, on saute à l'étiquette **else**, on exécute les instructions qui suivent et on aboutit à **endif**, sinon, on exécute les instructions suivant le **jng** jusqu'au branchement inconditionnel **jmp endif**.

Dans le cas de conditions multiples, on procède comme suit :

<b>if</b> (a > b) && (c <= d) {	<b>if:</b>	<b>cmp</b>	a,b
...		<b>jng</b>	endif
...		<b>cmp</b>	c,d
}		<b>jnle</b>	endif
...		...	
		...	
	<b>endif:</b>	...	

La dernière instruction **jnle** effectue un saut si c > d (**jnle** = jump if not less or equal).

Voici maintenant comment on implémente une boucle **while** :

<b>while</b> (a > b) {	<b>while:</b>	<b>cmp</b>	a,b
...		<b>jle</b>	<b>endwhile</b>
...		...	
}		...	
		<b>jmp</b>	<b>while</b>
	<b>endwhile:</b>	...	

Ici encore, on compare a à b. Si  $a \leq b$ , donc a n'est pas plus grand que b, on saute à la fin de la boucle **endwhile**. Si  $a > b$ , on exécute la boucle, puis on effectue un saut inconditionnel vers l'étiquette **while**, où on effectue de nouveau la comparaison.

Implantation d'une boucle **for** :

<b>for</b> (i = 1; i < 10; i++) {	<b>for:</b>	<b>mov</b>	i,1
...		<b>jmp</b>	<b>test</b>
...	<b>next:</b>	...	
...		...	
}		<b>inc</b>	i
...	<b>test:</b>	<b>cmp</b>	i,10
		<b>jl</b>	<b>next</b>
	<b>endfor:</b>	...	

Implantation d'un **switch** :

<b>switch</b> (i) {	<b>switch:</b>	<b>cmp</b>	i,1
case 1: ...		<b>jne</b>	<b>case2</b>
...		...	
break;		...	
case 2: ...		<b>jmp</b>	<b>endswitch</b>
...	<b>case2:</b>	<b>cmp</b>	i,2
break;		<b>jne</b>	<b>default</b>
default: ...		...	
}		...	
...		<b>jmp</b>	<b>endswitch</b>
	<b>default:</b>	...	
	<b>endswitch:</b>	...	

Nous allons maintenant examiner plus en détail comment fonctionnent ces branchements.

## 5.2 Branchements

Lors d'une comparaison, par exemple **cmp a,b**, le processeur effectue la soustraction  $a - b$  et positionne les indicateurs selon le résultat de l'opération. Les indicateurs qui nous intéressent ici sont :

ZF = zero flag = 1 si le résultat est nul, sinon ZF = 0,

CF = carry flag = 1 s'il y a une retenue de générée, sinon CF = 0,

SF = sign flag = 1 si le résultat est négatif, sinon SF = 0,

OF = overflow flag = 1 s'il y a débordement de capacité, sinon OF = 0.

Pour mieux comprendre comment ces indicateurs sont affectés, examinons ce qui se passe lors d'une addition.

	Cas	Exemple*	SF	ZF	OF	CF
1.	$P + P = P$	$20 + 30 = 50$	0	0	0	0
2.	$P + P = N *$	$40 + 50 = 90$	1	0	1	0
3.	$P + N = P$	$70 + F0 = 60$	0	0	0	1
4.	$P + N = N$	$50 + 90 = E0$	1	0	0	0
5.	$N + N = P *$	$A0 + 90 = 30$	0	0	1	1
6.	$N + N = N$	$F0 + E0 = D0$	1	0	0	1

\* Les nombres sont en hexadécimal

On suppose qu'on effectue des opérations sur 8 bits. Tout nombre dont le digit le plus significatif est supérieur à 7 est considéré comme négatif puisque son bit le plus significatif est 1. Lors de l'addition, il y a **débordement de capacité** si les deux opérandes sont de même signe et que le résultat est de signe opposé. C'est ce qui se passe dans les cas 2 et 5, qui ne peuvent se produire en arithmétique ordinaire. Il y a retenue quand le résultat de l'addition des deux digits les plus significatifs dépasse  $16_{10}$ , comme dans les cas 3, 5 et 6. En effet  $0x70 + 0xF0 = 0x160$ , donc la représentation sur 8 bits est  $0x60$  et le 1 va dans la retenue. Notez qu'on peut avoir une retenue et un débordement de capacité en même temps, comme dans le cas 5.

Voyons maintenant ce qui se passe lors d'une soustraction (comparaison) :

	Cas	Exemple*	SF	ZF	OF	CF
1.	P - P = P	40 - 20 = 20	0	0	0	0
2.	P - P = N	20 - 40 = E0	1	0	0	1
3.	P - N = P	50 - F0 = 60	0	0	0	1
4.	P - N = N *	30 - 90 = A0	1	0	1	1
5.	N - P = P *	B0 - 70 = 40	0	0	1	0
6.	N - P = N	A0 - 10 = 90	1	0	0	0
7.	N - N = P	C0 - A0 = 20	0	0	0	0
8.	N - N = N	A0 - F0 = B0	1	0	0	1

Ici encore, il y a deux cas qui ne sont pas possible en arithmétique ordinaire, les cas 4 et 5. Pour détecter les dépassements de capacité, on effectue les soustractions au moyen d'une addition du complément à 2. La même règle que pour l'addition s'applique alors. Notez que dans ce cas le bit de retenue est inversé.

Examinons maintenant comment on peut utiliser ces résultats pour générer des branchements sur des quantités signées et non signées.

Supposons qu'on veut que les nombres 00 à FF soient considérés comme tous positifs, i.e. pas en complément à 2. On dit alors que ce sont des nombres non signés. Les différents cas possibles sont les suivants :

**A > B non signé**

	Exemple	Cas	SF	ZF	OF	CF
A > 0 et B > 0	50 - 10 = 40	P - P = P	0	0	0	0
A < 0 et B > 0	90 - 20 = 70	N - P = P	0	0	1	0
A < 0 et B > 0	90 - 10 = 80	N - P = N	1	0	0	0
A < 0 et B < 0	90 - 80 = 10	N - N = P	0	0	0	0

Si on effectue la simplification booléenne, on obtient :

$$\begin{aligned}
 (A > B) &= \overline{SF}.\overline{ZF}.\overline{OF}.\overline{CF} + \overline{SF}.\overline{ZF}.OF.\overline{CF} + SF.\overline{ZF}.\overline{OF}.\overline{CF} \\
 &= \overline{ZF}.\overline{CF},
 \end{aligned}$$

c'est-à-dire **ZF=0** et **CF=0**.

On peut dériver de la sorte les conditions suivantes pour des quantités **non signées** :

**a** (above)                      ZF = 0 et CF = 0  
**ae** (above or equal)        CF = 0  
**b** (below)                      CF = 1  
**be** (below or equal)        CF = 1 ou ZF = 1

Examinons maintenant le cas de quantités signées en complément à 2.

*A > B signé*

	Exemple	Cas	SF	ZF	OF	CF
A > 0 et B > 0:	50 - 10 = 40	P - P = P	0	0	0	0
A > 0 et B < 0:	50 - F0 = 60	P - N = P	0	0	0	1
A > 0 et B < 0:	50 - 90 = C0	P - N = N	1	0	1	1
A < 0 et B < 0:	F0 - 90 = 60	N - N = P	0	0	0	0

La simplification booléenne nous donne :

$$(A > B) = \overline{SF}.\overline{ZF}.\overline{OF}.\overline{CF} + \overline{SF}.\overline{ZF}.\overline{OF}.CF + \overline{SF}.\overline{ZF}.OF.\overline{CF}$$

Comme le cas 1010 ne se produit jamais dans une soustraction, on peut ajouter le terme  $\overline{SF}.\overline{ZF}.OF.\overline{CF}$  ce qui nous permet de simplifier et d'obtenir :

$$(A > B) = \overline{ZF}.\overline{SF}.\overline{OF}$$

c'est-à-dire **ZF = 0** et **SF = OF**.

On peut ainsi dériver les conditions suivantes pour les quantités **signées** :

**g** (greater)                      ZF = 0 et SF = OF  
**ge** (greater or equal)        SF = OF  
**l** (less)                          SF    OF  
**le** (less or equal)            ZF = 1 ou SF    OF

Il faut bien prendre garde d'utiliser le bon branchement selon le type de données considéré. En C, les variables sont signées par défaut. Pour avoir des variables non signées, on utilise la déclaration **unsigned**.

Exemple :

**unsigned char** x;      0    x    255

```
unsigned short y;    0   y   65535
unsigned long  z;    0   z  4294967296
```

```
char x;              -128  x   127
short y;             -32768 y   32767
long z;              -2147483648 z  2147483647
```

En assembleur standard, il n'y a pas de distinction entre un nombre signé ou non signé lors de sa déclaration. Ce n'est que le choix du branchement qui détermine si on le considère comme signé ou non signé.

Voici maintenant la liste des branchements du Pentium :

### Branchements inconditionnels

```
call
jmp   jump
ret, retn, retf, iret    return (near, far), interrupt return
```

### Branchements conditionnels

#### L 'instructionjcc

#### Branchements conditionnels simples

<b>je</b>	jump if equal:	ZF = 1
<b>jne</b>	jump if not equal:	ZF = 0
<b>jz</b>	jump if zero:	ZF = 1
<b>jnz</b>	jump if not zero:	ZF = 0

#### Branchements conditionnels non signés

<b>ja</b>	jump above (x > y non-signé )	CF = 0 & ZF = 0
<b>jna</b>	jump not above = jbe	
<b>jae</b>	jump above or equal	CF = 0
<b>jnae</b>	jump not above or equal = jb	
<b>jb</b>	jump below (x < y non-signé)	CF = 1
<b>jnb</b>	jump not below = jae	
<b>jbe</b>	jump below or equal	CF = 1   ZF = 1
<b>jnb</b>	jump not below or equal = ja	



### Branchements conditionnels sur les indicateurs eux mêmes

<b>jc</b>	jump if carry:	CF = 1
<b>jnc</b>	jump if not carry:	CF = 0
<b>jo</b>	jump if overflow:	OF = 1
<b>jno</b>	jump if not overflow:	OF = 0
<b>jp</b>	jump if parity:	PF = 1
<b>jnp</b>	jump if not parity:	PF = 0
<b>jpo</b>	jump if parity odd:	PF = 0
<b>js</b>	jump if sign:	SF = 1 (negative)
<b>jns</b>	jump if no sign:	SF = 0

### Branchements conditionnels signés

<b>jg</b>	jump greater (x > y signé)	SF = OF & ZF = 0
<b>jng</b>	jump not greater	SF = OF & ZF = 1
<b>jge</b>	jump greater or equal	SF = OF
<b>jnge</b>	jump not greater or equal = jl	
<b>jl</b>	jump less (x < y signé)	SF = OF
<b>jnl</b>	jump not less = jge	
<b>jle</b>	jump less or equal	SF = OF
<b>jnle</b>	jump not less or equal = jg	

### Branchements conditionnels sur compteur C

<b>jcxz</b>	jump if cx = 0
<b>jecxz</b>	jump if ecx = 0

## Exercices

1. Écrive le code pour effectuer la condition if ((a > b || c > d)
2. Dérivez en détail les conditions booléennes pour le branchement signé a < b.

## 6. Modes d'adressage

L'architecture Intel supporte plusieurs modes d'adressage. En voici une description utilisant l'instruction `mov`, qui copie le contenu d'un opérande source dans un opérande destination.

### Adressage immédiat

`mov eax,0122Bh`      ou `eax, 0x0122B` : la valeur 0x122B est placée dans `eax`

### Adressage registre

`mov ebx,eax`      le contenu de `eax` est copié dans `ebx`.

### Adressage direct

`mov eax,variable`      `variable` est interprété comme une adresse

### Adressage implicite

Certaines instructions n'ont pas d'opérande explicite et la description de l'adresse est contenue dans l'instruction elle-même ou dans des registres prédéfinis :

`ret`      ; dépile l'adresse de retour  
`xlat`      ; utilise `EBX` et `AL`

### Indirection registre

`mov edx,[ebx]`      ; adresse dans `ebx`; son contenu est copié dans `edx`.

Les registres suivants peuvent servir de registre de base :

<code>eax</code>	<code>edi</code>
<code>ebx</code>	<code>esi</code>
<code>ecx</code>	<code>ebp</code>
<code>edx</code>	<code>esp</code>

Dans les cas où il y a ambiguïté, on devra utiliser la forme :

<code>mov</code>	<code>word ptr [ebx], 0</code>	; transfert de 16 bits
<code>movzx</code>	<code>eax, byte ptr [ebx]</code>	; transfert de 8 bits avec changement de type
<code>movsz</code>	<code>eax, word ptr [ebx]</code>	; transfert de 16 bits avec changement de type

### Indirection registre avec offset

```
mov    eax,[ebx + 8]    ; adresse = ebx + 8
```

### Indirection registre avec offset registre (index)

```
mov    [ebx + edi * k],eax    ; adresse = ebx + edi * k
                                ; k peut être 1, 2, 4 ou 8 seulement.
```

### Indirection registre avec index + offset

```
mov    ax,[ebx + esi * k + 2] ; adresse= ebx + esi*k + 2
```

Les registres suivants peuvent servir d'index :

eax	edi
ebx	esi
ecx	ebp
edx	

Dans ce mode d'adressage, la constante k permet d'ajuster l'instruction à la taille de l'opérande, de façon à accéder directement au  $i^{\text{ème}}$  élément d'un tableau.

Pour des octets (**char**, **byte**, **Boolean**),  $k = 1$ , pour des **short**,  $k = 2$ , pour des **long**,  $k = 4$ , pour des **double**,  $k = 8$ .

```
short tableau[ ] = {50, 75, 342, 9, ... };
long i = 3;
_asm {
    lea    esi,tableau
    mov    edx,i
    mov    ax,[esi + edx * 2]    ; ax = tableau[3]
    ...
}
```

On observe ici que pour adresser un élément du tableau, on utilise deux registres. ESI pointe vers l'élément 0 du tableau. EDX sert d'indice dans le tableau. Pour accéder au  $i^{\text{ème}}$  élément, on place  $i$  dans EDX. On le multiplie par 2 dans l'instruction parce que le tableau est constitués d'éléments **short** de 16 bits. (2 octets).

En effet, un tableau, ou encore chaîne, est une suite d'éléments de même taille (même type) rangés consécutivement en mémoire (de façon contiguë). Une chaîne de caractères est un cas particulier de chaîne qui a des propriétés particulières (voir 7.2 ci-dessous). Un tableau à deux dimensions ou plus est également une chaîne en mémoire.

En C, on peut déclarer un tableau ou chaîne de deux façons :

```
short Tableau[ ] = { 1, 2, 3, 4 };
```

ou     `short * Tableau ;`

Dans la seconde version, il faut réserver de la mémoire avec l'instruction **malloc** :

```
Tableau = (short *) malloc(4 * sizeof(short));
```

puis y inscrire des valeurs.

Dans les deux cas on peut accéder à la  $i^{\text{ème}}$  valeur du tableau au moyen de l'une des instructions suivantes :

```
x = Tableau[i];
```

ou     `x = *Tableau + i * sizeof(element);` ; dans le cas de char, on multiplie par 1.  
; dans le cas de long, on multiplie par 4.

Dans le présent exemple, les valeurs sont rangées consécutivement en mémoire comme suit à l'adresse Tableau :

```
01 00 02 00 03 00 04 00
```

Dans des tableaux à plus d'une dimension, les données sont encore rangées en mémoire de façon consécutive et contiguë.

La matrice  $3 \times 4$  suivante :

```
      1  2  3  4
M =  5  6  7  8
      9 10 11 12
```

est déclarée comme suit :

```
short M[ ][ ] = {{ 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 }};
```

Les données sont rangées en mémoire consécutivement à partir de l'adresse Tableau :

01 00 02 00 03 00 04 00 05 00 06 00 07 00 08 00 09 00 0A 00 0B 00 0C 00

De façon générale, l'adresse de l'élément  $M_{i,j}$  d'un tableau  $m \times n$  commençant à l'adresse  $M$  est donnée par l'expression :

$$\text{Adresse } (M_{i,j}) = M + (j + n \times i) \times \text{taille des éléments.}$$

En assembleur, on ne peut définir que des tableaux à une seule dimension. Leur définition s'effectue au moyen des directives de création de variables (voir section 7.1). C'est au programmeur à gérer la multidimensionnalité, puisqu'on ne peut avoir qu'un indice dans les modes d'adressage du Pentium que nous venons de voir.

La matrice  $M$  présentée ci-dessus serait définie comme suit :

M      dw      1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

## 7. Instructions de base

Vous trouverez une documentation abrégée de ces instructions dans la section 5 du présent document intitulée *Instructions de base du Pentium*.

L'instruction la plus utilisée est l'instruction **mov**, qui copie la valeur d'un opérande source dans un opérande destination. La syntaxe est la suivante :

```
mov    reg, reg
mov    reg, mem
mov    mem, reg
mov    reg, immedi
mov    mem, immedi
```

Notez qu'il n'existe pas de transfert de mémoire à mémoire.

Une autre instruction fréquemment utilisée est **lea**, qui calcule l'adresse effective de l'opérande source et place le résultat dans l'opérande destination.

```
lea    reg, mem
```

C'est la façon de mettre dans un registre l'adresse d'une variable. Par exemple, l'instruction :

```
lea    esi, toto
```

place dans esi l'adresse mémoire de la variable *toto*. On pourrait également utiliser l'instruction

```
mov    esi, offset toto
```

L'instruction **lea** permet aussi d'obtenir une multiplication et une addition en un cycle :

```
lea    esi, [edi*4+7]
```

Quand doit-on utiliser **lea** au lieu de **mov** ? Pour les paramètres d'une fonction, on utilise toujours **mov**. Pour les variables, on utilise **lea** lorsqu'il s'agit d'une variable de type tableau, mais on utilise **mov** lorsqu'il s'agit d'un pointeur ou d'un type simple. On peut aussi remplacer **lea** eax, x par **mov** eax, offset x.

Des variantes de l'instruction **mov** sont utiles pour changer le type d'une variable. Ce sont **movzx** et **movsx**. L'instruction **movzx** prend un opérande source de 8 bits ou de 16 bits et l'étend sur 16 ou 32 bits en mettant des 0 dans le reste de la destination, tandis que **movsx** remplit la destination avec le bit de signe de l'opérande source. Par exemple, si **bl** contient 0x94, sa valeur non signée est + 148 et sa valeur en complément à 2 est -108 :

<b>movzx</b>	<b>ax, bl</b>	-> ax = 0094	= + 148 sur 16 bits
<b>movsx</b>	<b>ax, bl</b>	-> ax = FF94	= -108 sur 16 bits
<b>movzx</b>	<b>eax, bl</b>	-> eax = 00000094	= + 148 sur 32 bits
<b>movsx</b>	<b>eax, bl</b>	-> eax = FFFFFFF94	= - 108 sur 32 bits

## 7.1 Directives de base

Pour programmer en assembleur, on doit utiliser, en plus des instructions assembleur, des directives ou pseudo-instructions, par exemple pour créer de l'espace mémoire pour des variables, pour définir des constantes, etc.

### Définition de variables

**.data**

<b>db</b>	<b>0</b>	; définit un octet initialisé à 0
<b>db</b>	<b>"Toto", 0</b>	; définit une chaîne de caractères terminée par un NULL
<b>dw</b>	<b>100</b>	; définit un mot initialisé à 100 (0x64)
<b>dw</b>	<b>1, 2, 3</b>	; définit un tableau de trois mots initialisés à 1, 2, 3
<b>dd</b>	<b>0F70ABCDh</b>	; définit un mot double initialisé à 0xF70ABCD
<b>dd</b>	<b>10 dup(0)</b>	; définit un tableau de 10 valeurs initialisées à 0

**.data?**

<b>db</b>	<b>?</b>	; définit un octet non initialisé
<b>dw</b>	<b>10 dup(?)</b>	; définit un tableau de 10 mots non initialisés

### Définition de constantes

**.const**

<b>dix</b>	<b>equ</b>	<b>10</b>
------------	------------	-----------

### Type de processeur

**.386, .486, .586**

**Début du programme**

.code

**Appel d'une fonction ou d'une procédure ou d'un sous-programme :**

invoke fonction a, b, c ; appelle fonction(a, b, c)

Le résultat d'une fonction est toujours dans al, ax ou eax, selon que la taille du résultat est 8, 16 ou 32 bits.

**Inclusion de fonctions de librairie**

include \masm32\include\kernel32.inc  
includelib \masm32\lib\kernel32.lib

**Fin du programme**

end

Tous vos programmes assembleur devront donc avoir la structure suivante :

```
.586 ; processeur = Pentium
.model flat, stdcall ; un seul segment e 4Go, appel standard
option casemap: none ; l'assembleur respecte les majuscules et minuscules
;-----
include \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib ; librairie où se trouve ExitProcess
;-----
maFonction PROTO: a:DWORD ; prototype de maFonction
;-----
.data ; variables globales initialisées
    ...
.data? ; variables globales non initialisées
    ...
.code
;-----
start:
    ...
    invoke ExitProcess, 0 ; retour à Windows
;-----
```



```
maFonction proc a:DWORD           ; déclaration de maFonction(a)
LOCAL b:WORD
LOCAL c[10]:BYTE
...
maFonction endp
;-----
end start                         ; fin du programme
```

Votre programme principal doit commencer à `start` et se terminer par `ExitProcess(0)`. Les sous-programmes appelés par ce programme sont définis entre `ExitProcess` et `end start`. Ces fonctions doivent avoir un prototype avant les déclarations de données `.data` et `.data?`.

On peut déclarer des variables locales dans les fonctions comme c'est le cas pour *b* et *c* ci-dessus. *b* est un mot, *c* est une chaîne de 10 octets.

Au besoin, on pourra inclure d'autres fichiers d'en-tête, telles que `windows.inc`, `user32.inc`, `gdi32.inc` et `masm32.inc` et les bibliothèques correspondantes `user32.lib`, `gdi32.lib` et `masm32.lib`.

## 7.2 Un premier exemple : addition de deux nombres

On va ici simplement additionner deux nombres.

```
.586                                ; processeur = Pentium
.model flat, stdcall                ; un seul segment e 4Go, appel standard
option casemap: none                ; l'assembleur respecte les majuscules et minuscules
;-----
include  \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib ; librairie où se trouve ExitProcess
;-----
.data                                ; variables globales initialisées
    x      dw      175
    y      dw      150
.data?                                ; variables globales non initialisées
    z      dw      ?
.code
;-----
start:
    mov     ax, x                    ; on utilise ax parce que x = word = 16 bits
    add     ax, y
    mov     z, ax                    ; z = x + y
    invoke  ExitProcess, 0
;-----
end start
```

## 7.3 Un deuxième exemple : conversion majuscules-minuscules

Il s'agit ici de convertir une chaîne de caractères en minuscules.

```
.586                                ; processeur = Pentium
.model flat, stdcall                ; un seul segment e 4Go, appel standard
option casemap: none                ; l'assembleur respecte les majuscules et minuscules
;-----
include  \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib ; librairie où se trouve ExitProcess
;-----
.data                                ; variables globales initialisées
    monTexte    db    "Chaine A CONvertIR",0
.data?          ; variables globales non initialisées
    ...
.code
;-----
start:
    lea  esi, monTexte                ; adresse du texte à convertir
While: mov  al, [esi]                  ; lire un caractère
    cmp  al, 0                        ; si c'est un NULL, terminé
    jz   endWhile
si:     cmp  al, 'A'                    ; si c'est une majuscule (entre A et Z)
    jb   finSi
    cmp  al, 'Z'
    ja   finSi
    add  al, 'a'-'A'                  ; convertir en minuscule
    mov  [esi], al                    ; récrire le caractère modifié où on l'avait pris
finSi:  inc  esi                        ; pointer sur caractère suivant
    jmp  repete
endWhile:
    invoke ExitProcess, 0              ; retour à Windows
;-----
end start                            ; fin du programme
```

## 7.4 Un autre exemple : recherche du maximum

### Recherche du maximum dans une chaîne de mots de 16 bits signés

Nous passons à un exemple un peu plus élaboré, qui comporte un programme principal qui appelle un sous-programme appelé FindMax.

Une telle fonction serait définie ainsi en C :

```
short FindMax(short * nombres, short n)
```

et serait appelée comme suit pour trouver le maximum d'un tableau de 6 nombres appelé Liste :

```
Max = FindMax(Liste, 6);
```

En assembleur, on a le programme complet suivant :

```
.586                                ; processeur = Pentium
.model flat, stdcall                ; un seul segment e 4Go, appel standard
option casemap: none                ; l'assembleur respecte les majuscules et minuscules
include  \masm32\include\kernel32.inc
includelib \masm32\lib\kernel32.lib ; librairie où se trouve ExitProcess
;
Findmax PROTO :DWORD, :WORD          ; prototype de notre fonction
;
;-----
.data                                ; données initialisées
Liste dw    100, 326, -7, 21, 4, 0    ; 6 mots initialisés = données à traiter
.data?                                ; données non initialisées
Max dw      ?                        ; espace pour résultat
.code
;-----
start: invoke Findmax, ADDR Liste, n ; programme principal
        mov    Max, ax                ; écriture du résultat
        invoke ExitProcess, 0         ; retour à Windows
;-----
FindMax proc nombres:DWORD, n:WORD
        mov    esi, nombres           ; adresse de nombres = List
        mov    cx, n                  ; n est word et doit être 0
        mov    ax,[esi]               ; max = premier élément
```

```

        add     esi,2           ; pointer sur l'élément suivant
        cmp     cx, 2
        jb      fin            ; terminé si n < 2
        dec     cx             ; on a déjà lu un élément
repeat:  cmp     ax, [esi]      ; max < élément ?
        jge     next           ; ou encore ja pour non signé
        mov     ax, [esi]      ; si oui, prendre comme nouveau maximum
next:    add     esi, 2         ; incrémenter pointeur
        dec     cx             ; décrémenter compteur
        jnz     repeat        ; répéter jusqu'à ce que compteur = 0
fin:
        ret
Findmax  endp
;-----
end start

```

Vous pouvez entrer le programme précédent dans l'éditeur QEditor de MASM32. Sauvegardez-le sous le nom FindMax.asm. Choisissez Assemble & Link dans le menu Project, puis choisissez Run Program dans le menu Project. Il ne se passe rien et c'est normal. Ce programme n'a aucune entrée/sortie. Il s'est exécuté trop rapidement pour qu'on voie quoi que ce soit. Il faut le faire exécuter pas à pas dans le débogueur ou ajouter une sortie comme on va voir dans la section suivante.

## 7.5 Entrées / sorties en mode console

Les bibliothèques fournies avec Masm32 contiennent des fonctions élémentaires pour l'entrée de données et l'affichage de résultats. La fonction **StdIn** permet de lire une chaîne de caractères au clavier. La fonction **StdOut** permet d'afficher une chaîne de caractères à l'écran. Pour pouvoir afficher des nombres, il faut les convertir en caractères avant de passer le résultat à StdOut. À cette fin, la fonction **dwtoa** convertit un nombre binaire de 32 bits en chaîne de caractères. Pour entrer des nombres, il faut convertir en nombre les caractères lus avec StdIn. On peut utiliser la fonction **atodw**, qui convertit une chaîne de caractères en nombre de 32 bits.

Ainsi, dans l'exemple précédent, on aurait pu afficher le résultat en ajoutant au début :

```

include    \masm32\include\masm32.inc
includelib \masm32\lib\masm32.lib

```

Ajouter dans la section .data?

```
monMsg    db    10 dup(?)                ; 10 octets pour caractères
```

Puis, juste avant invoke ExitProcess, 0

```
movzx     eax, word ptr Max                ; convertir Max sur 32 bits
invoke    dwtoa , eax, ADDR monMsg         ; convertir en caractères
invoke    StdOut, ADDR monMsg              ; afficher
ici:      jmp     ici                      ; attendre CTL-C avant de quitter
```

Finalement, il faut compiler au moyen de Console Assemble & Link.

## Exercices

1. Écrire un programme assembleur qui affiche le message “Bonjour le monde!” à l'écran de la console.
2. Écrire un programme assembleur utilisant StdIn et StdOut qui lit une chaîne de caractères entrée par l'utilisateur et affiche ensuite cette chaîne à l'écran. La syntaxe pour StdIn est la suivante :

```
invoke StdIn, ADDR monMsg, bLen
```

où bLen est la taille du tampon, i.e. le nombre maximum de caractères qu'on doit lire.

Par exemple :

```
invoke StdIn, ADDR monMsg, 25
```

## 7.6 Chaînes de caractères

On peut distinguer quatre types de chaînes de caractères :

- la chaîne C, qui contient les codes ASCII des caractères et se termine par un octet NULL (0x00 ou \0) ;
- la chaîne Pascal, qui débute par un octet de longueur suivi des codes ASCII des caractères ; cette chaîne est limitée à 255 caractères ;
- la chaîne PC, qui combine la chaîne Pascal avec la chaîne C : un octet de longueur au début et un octet NULL à la fin ; cette chaîne est également limitée à 255 caractères ;
- la chaîne ASIS ou telle quelle, qui n'a ni octet de longueur, ni caractère de fin de chaîne ; l'information sur sa longueur doit être conservée ailleurs.

L'architecture Intel comporte plusieurs instructions pour manipuler les chaînes de caractères ou autres, c'est à dire des chaînes de **short** ou de **long**..

L'instruction **scas** parcourt une chaîne pour trouver une valeur spécifiée dans l'accumulateur A. S'il s'agit d'un scan de 8 bits, la valeur est dans al, d'un scan de 16 bits, dans ax, d'un scan de 32 bits, dans eax. On l'utilise habituellement avec le préfixe **repne** (ou **repnz**) pour trouver la première valeur qui corresponde à l'accumulateur, ou **repe** (ou **repz**), pour trouver la première valeur qui en diffère. Le nombre maximum d'éléments de la chaîne à parcourir doit être contenu dans ecx. L'adresse de la chaîne à parcourir doit se trouver dans edi. Pour chaque élément de la chaîne, l'élément destination est soustrait de la valeur de l'accumulateur et les indicateurs sont mis à jour pour refléter le résultat, bien que ce dernier ne soit pas enregistré. Edi est ajusté en fonction de la taille des opérandes et l'état de l'indicateur de direction DF. Edi est incrémenté si l'indicateur de direction a été mis à zéro avec l'instruction **cld**, ou décrémenté si l'indicateur de direction a été mis à un avec **std**. Lorsqu'on utilise **scas**, il faut fournir un opérande pour indiquer la taille des opérandes à traiter. Les variantes **scasb** (octets), **scasw** (mots) et **scasd** (doubles mots) permettent de ne pas fournir d'opérande, puisque c'est l'instruction elle-même qui en détermine la taille. À la fin de l'opération, edi pointe vers le mot suivant la dernière comparaison

La syntaxe est la suivante :

```

scas    mem
scasb
scasw
scasd

```

Par exemple **repne scasw** parcourt la chaîne débutant à l'adresse edi et s'arrête soit quand le premier mot correspondant à ax est trouvé, soit quand le compteur ecx est épuisé.

L'instruction **repe scasb** parcourt la chaîne débutant à l'adresse edi et s'arrête quand elle rencontre le premier octet ne correspondant pas à al, ou quand ecx tombe à 0.

Voici une fonction pour calculer la longueur d'une chaîne de caractères C (terminée par un \0) :

```

strlen proc src:LPSTR                ; src est l'adresse de la chaîne
    xor     al,al                    ; eax = 0 ; al = caractère à trouver
    mov     edi, src
    mov     ecx, 0ffffffffh          ; ou mov ecx, -1: longueur maximale = 4294967295
    cld                                  ; direction vers le haut
    repne   scasb                    ; scan tant que pas égal à al (0)
    neg     ecx                      ; rendre positif
    lea     eax, -2 [ecx]            ; 2 de trop, résultat dans eax
    ret
strlen     endp

```

La déclaration de paramètre src : LPSTR indique qu'il s'agit d'un double mot représentant l'adresse d'une chaîne de caractères.

Une deuxième instruction de comparaison de chaînes est **cmps**, avec les variantes **cmpsb**, **cmpsw** et **cmpsd**, qui permet de comparer deux chaînes en mémoire, élément par élément, afin de trouver le premier élément différent (avec **repe**) ou encore le premier élément égal (avec **repne**). Esi doit pointer vers la chaîne source et edi vers la chaîne destination. ecx doit contenir le maximum d'éléments à comparer. À chaque comparaison, edi et esi sont incrémentés de la taille d'un élément.

Une autre instruction de chaîne est **lods** avec les variantes **lodsb**, **lodsw** et **lods**, qui charge l'accumulateur avec un élément de chaîne situé à l'adresse contenue dans esi et incrémente esi de la taille de l'élément. Elle est souvent jumelée avec l'instruction **stos** et ses variantes **stosb**, **stosw** et **stosd**, qui stocke le contenu de l'accumulateur en mémoire à l'adresse contenue dans edi et incrémente edi de la taille de l'élément. On n'utilise pas lods avec un préfixe **rep**, car on n'a aucune raison de vouloir remplir un registre avec les



valeurs d'une chaîne sans autre traitement. Par contre, on utilise **stos** avec **rep** pour remplir une partie de la mémoire avec une même valeur.

Voici une fonction qui copie une chaîne de caractères d'un emplacement mémoire à un autre :

```
strcpy proc dst:LPSTR, src:SPSTR
    mov     edi, dst           ; adresse de la destination
    mov     esi, src          ; adresse de la source
    cld                       ; direction vers le haut
While:    lodsb               ; copier un octet
    stosb
    cmp     al, 0              ; ou test al, al
    jnz     While             ; tant que le car != 0
    ret
strcpy    endp
```

Voici une fonction qui concatène deux chaînes de caractères C :

```
strcat proc dst:LPSTR, src:LPSTR
    mov     edi, dest         ; adresse de la destination
    mov     esi, src          ; adresse de la source
    xor     eax, eax          ; on va chercher un caractère NULL
    mov     ecx, 0xffffffff   ; longueur maximale
    cld                       ; direction vers le haut
    repne   scasb              ; trouver la fin de la chaîne dest
    dec     edi                ; éliminer le \0 à la fin
While:    lodsb               ; copier octets de src
    stosb                      ; vers dest
    cmp     al, 0              ; ou test al, al
    jne     While             ; tant que != 0
    ret
strcat    endp
```

Les fonctions que nous venons de voir impliquent la copie de données d'un endroit à l'autre de la mémoire. Ceci semble ne pas présenter de problèmes, mais il faut en général prendre certaines précautions. En effet, considérons une fonction que nous pouvons appeler Blockmove, qui déplace un certain nombre d'octets d'une adresse à une autre.

**void** BlockMove(Source , Destination, Octets)

Par exemple, soit 0x1000 l'adresse de la Source et 0x1003 celle de la Destination, et soit Octets = 5, comme suit :

Adresse :	1000	1001	1002	1003	1004
Contenu:	1	2	3	4	5

on voudrait avoir :

1 2 3 4 5 -> 1 2 3 1 2 3 4 5

Essayons l'implémentation suivante :

BlockMove proc source:LPBYTE, destination:LPBYTE, octets:WORD

```

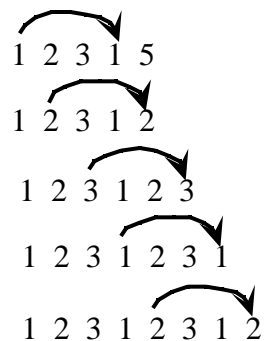
mov     esi, source
mov     edi, destination
movzx   ecx, word ptr octets    ; compteur = nb. d'octets à copier
cld                                           ; Direction vers le haut
rep     movsb                      ; copier les octets
ret

```

BlockMove endp

Ici, LPBYTE signifie que source et destination sont des adresses de 32 bits contenant chacune une donnée d'un octet.

On obtient :

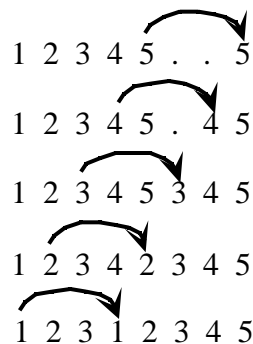


Ceci n'est pas le résultat escompté. Par contre, on n'aurait pas de problème si on partait de la fin :

```
BlockMove proc source:LPBYTE, destination:LPBYTE, octets:WORD
```

```

    mov     esi, Source
    mov     edi, Destination
    movzx   ecx, word ptr octets    ; compteur = nb. octets à copier
    mov     eax, ecx
    dec     eax
    add     esi, eax                ; ajouter octets-1 à esi et edi
    add     edi, eax                ; pour partir de la fin
    std     ; direction vers le bas
    rep     movsb                  ; copier les octets
    ret
BlockMove endp
```



Comme règle générale, si l'adresse de la destination est supérieure à celle de la source, on copie vers le bas à partir de la fin de la source vers la fin de la destination, sinon on copie vers le haut à partir du début de la source vers le début de la destination.

Tels quels, ces deux programmes seraient trop lents. Il vaut mieux transférer les données en blocs plus grands, au moyen, par exemple, de l'instruction **movsd**.

La première version deviendrait ainsi :

```
BlockMove proc source:LPBYTE, destination:LPBYTE, octets:WORD
```

```

    mov     esi, source
    mov     edi, destination
    movzx   ecx, word ptr octets    ; compteur = nb. d'octets à copier
    cld     ; direction vers le haut
    mov     edx, ecx
    and     edx, 3                  ; octets modulo 4
    shr     ecx, 2                  ; octets / 4
```

```

                jz      bytes      ; si 0 doubles mots, copier par octets
                rep     movsd      ; copier les doubles mots
bytes:          mov     ecx, edx    ; copier les 1, 2 ou 3 derniers octets
                test    ecx, ecx    ; s'il y en a 0, on a fini
                jz      fin
                rep     movsb
fin:            ret
BlockMove      endp

```

## 7.7 Comparaisons de chaînes

Comparaison de deux chaînes de caractères string1 et string2 et retourne dans ax :

```

-1, si string1 < string2,
0, si string1 = string2
1, si string1 > string2

```

StrCmp proc, string1:LPSTR, string2:LPSTR

```

    cld                                ; on compare vers le haut
    mov     edi, string2
    mov     ecx, -1
    mov     al, 0                      ; on va déterminer la longueur de string2
    repne   scasb
    neg     ecx                        ; rendre positif
    lea     ecx, [ecx-1]               ; résultat dans ecx, pointe après le null.
    mov     esi, string1
    repe    cmpsb                      ; comparer tant qu'il y a égalité
    je      egal
    ja      sup
inf:  mov     ax, -1                   ; string1 < string2
    ret
egl:  xor     ax, ax                   ; string1 = string2
    ret
sup:  mov     ax, 1                    ; string1 > string2
    ret
StrCmp endp

```

## 7.8 Recherche d'une chaîne dans une autre

Il s'agit de trouver d'une chaîne de caractères dans une autre, par exemple, trouver un mot ou une phrase dans un texte.

Soit *source* la chaîne à trouver et *destination* le texte à parcourir. On doit se fixer une longueur maximum ou arrêter lors d'un caractère de fin de chaîne. Notre programme sera plus général si on donne une longueur. Le programme sera donc appelé comme suit :

invoke StrSrch, source, longueur, destination, plage

où *longueur* est la longueur de la chaîne source et *plage* est la longueur du texte à parcourir. Si la chaîne source est trouvée dans le texte, la fonction retournera dans *eax* l'adresse où elle débute, sinon elle retournera -1.

StrSrch proc, source:LPSTR, longueur:WORD, destination:LPSTR, plage:WORD

```

    cld                                ; comparer vers le haut
    xor     ecx, ecx
    mov     esi, source
    mov     edi, destination
    mov     bx, plage
    sub     bx, longueur
    jl      pasla                      ; si longueur > plage, pas trouvé
    inc     bx                          ; compteur
    mov     edx, edi                   ; sauver edi, car cmps le modifie
    mov     eax, esi                   ; idem pour esi
lp:   mov     cx, longueur              ; comparer sur longueur caractères max
    repe    cmpsb                      ; et tant qu'il y a égalité
    je      trouv                      ; si toujours égalité, source trouvée dans dest
    inc     edx                        ; sinon se déplacer dans destination
    mov     edi, edx                   ; restaurer edi
    mov     esi, eax                   ; restaurer esi à chaque tour
    dec     bx                          ; tant que compteur > 0
    jne     lp
pasla: mov     eax, -1                 ; pas trouvé, retourner -1
    ret
trouv: xor     eax, edx                ; trouvé, retourner adresse
    ret
StrSrch endp

```

## 7.9 Transposée d'une matrice

Soit  $m$  une matrice de  $r$  rangées et  $c$  colonnes. Avec  $r = 3$  et  $c = 4$ , on aurait, par exemple :

$$m = \begin{matrix} & 1 & 2 & 3 & 4 \\ 5 & 5 & 6 & 7 & 8 \\ 9 & 9 & 10 & 11 & 12 \end{matrix}$$

Sa transposée  $n$  est :

$$n = \begin{matrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{matrix}$$

En assembleur, la matrice est unidimensionnelle et est définie comme suit :

```
m      dw      1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
ou
m      dw      1, 2, 3, 4
      dw      5, 6, 7, 8
      dw      9, 10, 11, 12
```

Le résultat désiré est :

1, 5, 9, 2, 6, 10, 3, 7, 11, 4, 8, 12

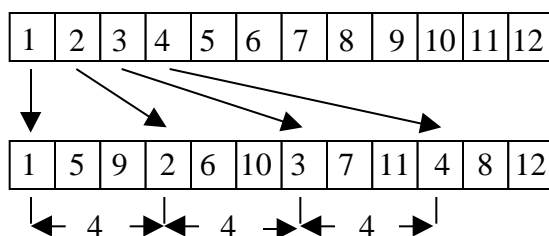
qu'on placera dans la matrice  $n$  :

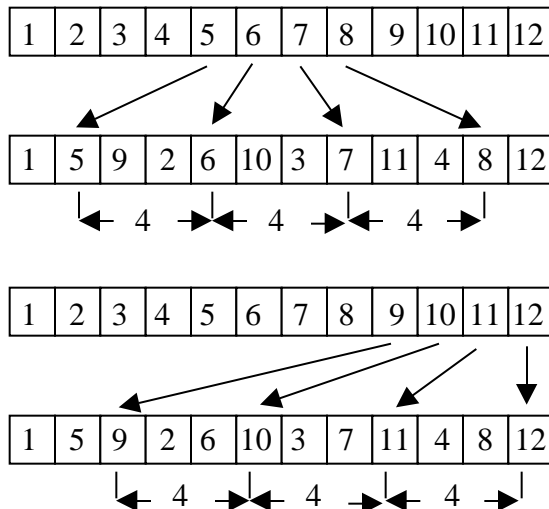
```
n      dw      12 dup(?)
```

La fonction Transpose aura la syntaxe suivante :

```
invoke Transpose, 3, 4, m, n
```

Une réalisation possible est la suivante :





Transpose proc r:WORD, c:WORD, m:LPWORD, n:LPWORD

```

    mov     esi, m
    mov     edi, n
    mov     cx, c                ; compteur = nombre de colonnes
    mov     bx, r
    xor     edx, edx
    movzx   ebx, cx
lp:    lodsw                       ; charger un élément de m
    mov     [edi], ax             ; stocker dans n
    lea     edi, [edi+ebx*2]      ; destination += 2 * c (word = 2 octets)
    dec     cx
    jnz     lp                   ; répéter c fois

    mov     cx, c                ; restaurer compteur
    mov     edi, n               ; repartir au début
    inc     edx                  ; décaler de 1 élément
    lea     edi, [edi+edx*2]
    cmp     dx, r                ; répéter r fois
    jne     lp
    ret

```

## Exercices

1. Quelle modification devrait-on apporter à la fonction FindMax pour chercher le maximum d'un tableau d'octets non signés ?
2. Quelles modifications devrait-on apporter à la fonction FindMax pour chercher le maximum d'un tableau de doubles mots signés ?
3. Écrivez une fonction FindMin qui retourne le minimum d'un tableau de mots signés.
4. Écrire une fonction **void** Strins(**char** \* dest, **char** \* src, **short** pos) qui insère une chaîne de caractère *src* à la position *pos* dans la chaîne de caractères *dest*.
5. Écrire une fonction **short** StrCmp(**char** \*dest, **char** \* src, **short** cnt) qui compare deux chaînes de **short** et retourne dans ax la position du premier élément qui soit le même dans les deux chaînes.



## 8. Instructions arithmétiques

Nous examinons ensuite les instructions arithmétiques usuelles. Ce sont **add**, **addc**, **sub**, **subc**, **mul**, **imul**, **div**, **idiv** et **neg**.

### 8.1 Addition

L'instruction **add** fait l'addition des deux opérandes et enregistre le résultat dans l'opérande destination. Les indicateurs sont positionnés selon le résultat de l'opération.

L'instruction **addc** fait l'addition de deux opérandes plus le bit de retenue CF provenant d'une opération antérieure. Elle sert à additionner les octets/mots/doubles mots de poids fort à la suite de l'addition des octets/mots/doubles mots de poids plus faible dans une addition portant sur plusieurs octets/mots/doubles mots comme dans l'addition de précision qui suit.

#### Addition de précision

Supposons qu'on veut additionner deux nombres en mémoire A et B de 128 bits chacun. Chacun occupe donc 4 doubles mots de 32 bits  $A_3$  à  $A_0$  et  $B_3$  à  $B_0$ . On commence par additionner les deux doubles mots de poids faible  $A_0$  et  $B_0$  avec l'instruction **addc** en ayant pris soin de mettre CF à 0 et on place le résultat dans  $S_0$ . Ensuite, puisque cette addition peut avoir généré une retenue, on utilise **addc** avec  $A_1$  et  $B_1$ , puis  $A_2$  et  $B_2$ , et  $A_3$  et  $B_3$ .

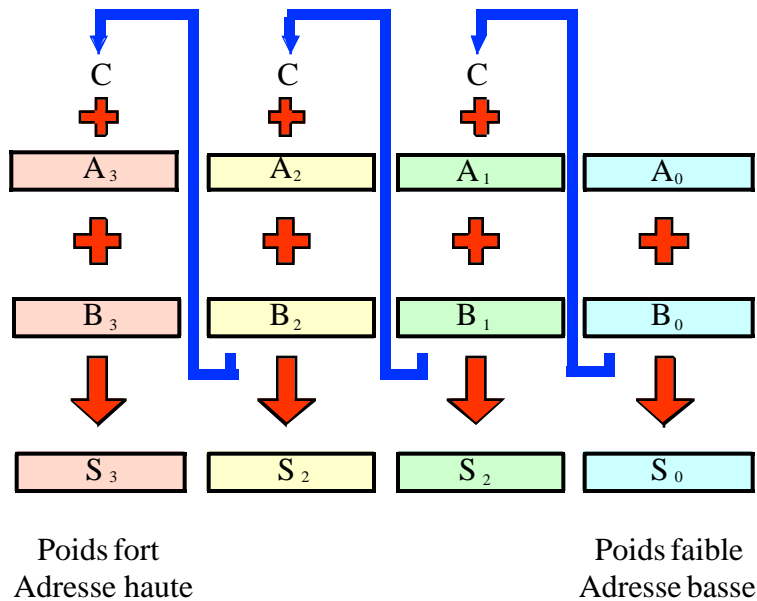


Figure 19. Addition de précision

Voici comment on pourrait coder une telle opération :

```

lea    esi, A      ; charger adresse des opérandes
lea    edi, B
lea    ebx, S
clc                      ; CF = 0
mov     ecx, n      ; nombre de mots
lp:    mov     eax, [esi]
       adc     eax, [edi]
       mov     [ebx], eax
       add     esi, 4      ; on ajoute 4 à chaque pointeur car 32 bits = 4 octets
       add     edi, 4
       add     ebx, 4
       dec     ecx        ; ou loop ecx (déconseillée parce que plus lente)
       jne     lp

```

## 8.2 Soustraction

L'instruction **sub** effectue la soustraction de l'opérande source de l'opérande destination et place le résultat dans l'opérande destination. Si un emprunt est nécessaire, le bit CF est mis à 1. L'instruction **sbb** tient compte de cet emprunt dans une soustraction de précision.

## 8.3 Multiplication

Il y a deux instructions pour la multiplication: **mul**, qui considère les opérandes comme des quantités non signées (toujours positives) et **imul**, qui considère les opérandes comme des quantités signées en complément à 2. L'instruction **mul** considère qu'il y a un opérande destination implicite qui est la paire de registres `edx:eax` pour les opérations sur 32 bits, `dx:ax` pour les opérations sur 16 bits et `ax` pour les opérations sur 8 bits. Il en va de même pour l'instruction **imul** dans sa forme à un opérande.

La syntaxe est la suivante :

```
mul    reg
mul    mem

imul   reg
imul   mem
imul   reg, immed
imul   reg, reg, immed
imul   reg, reg
```

Supposons que `eax = 0x80000001`. `eax` peut être interprété comme:

	2 147 483 649	(non signé)
ou	-2 147 483 647	(signé)

Si `ebx = 0x00000003`, alors:

```
mul ebx          eax = 0x80000003
                  edx = 0x00000001
```

La réponse est le nombre `0x00000001 80000003`, qui vaut  $6\,442\,450\,947_{10}$ .

Par contre, avec les mêmes opérandes,

```
imul ebx         eax = 0x80000003
                  edx = 0xFFFFFFFF
```

et la réponse est le nombre `$FFFFFFFFE 80000003`, qui vaut  $-6\,442\,450\,941_{10}$ .

Remarquez que dans les deux cas, le double mot de poids faible est le même.

## Multiplication de précision

Supposons qu'on multiplie en décimal deux nombres de deux chiffres chacun, par exemple  $45 \times 67$ . On peut effectuer le calcul au moyen de quatre multiplications d'un chiffre par l'autre comme suit :

$$\begin{array}{r}
 45 \\
 \times 67 \\
 \hline
 35 \\
 28 \phantom{0} \\
 30 \phantom{00} \\
 \hline
 24 \phantom{000} \\
 \hline
 3015
 \end{array}$$

On obtient un produit de quatre chiffres.

De la même façon, multiplions deux nombres de 64 bits X et Y constitués chacun de deux doubles mots de 32 bits XH, XL et YH, YL respectivement. L'instruction **mul** peut effectuer le produit 32 bits par 32 bits et donner le résultat de 64 bits dans edx:eax. On effectue quatre produits comme dans le cas décimal ci-dessus, on effectue l'addition de quatre nombres décalés, et on obtient le résultat de 128 bits.

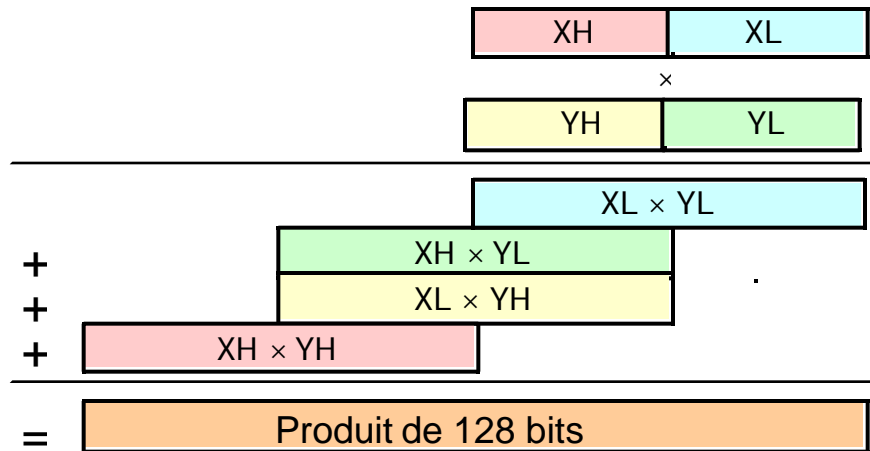


Figure 20. Multiplication de précision

La principale difficulté consiste à effectuer l'addition sur 128 bits des nombres décalés en faisant attention de bien propager les retenues s'il y en a.

On a le programme suivant pour la multiplication non signée :

## Assembleur

X	dd	0FFFFFFFFh, 0FFFFFFFFh
Y	dd	0FFFFFFFFh, 0FFFFFFFFh
Produit	dd	4 dup(?)

```

lea    esi, X                ; chargement de l'adresse des opérandes
lea    edi, Y
lea    ebx, Produit
mov     eax, [esi]            ; XL
mul     dword ptr [edi]       ; x YL
mov     [ebx], eax           ; (XL x YL)L
mov     [ebx + 4], edx        ; (XL x YL)H
mov     eax, [esi + 4]       ; XH
mul     dword ptr [edi]       ; x YL
add     [ebx+4], eax          ; (XH x YL)L
adc     edx, 0                ; propager la retenue
mov     [ebx + 8], edx        ; (XH x YL)H
mov     dword ptr [ebx + 12], 0
adc     dword ptr [ebx + 12], 0
mov     eax, [esi]            ; XL
mul     dword ptr [edi + 4]    ; x YH
add     [ebx + 4], eax        ; (XL x YH)L
adc     [ebx + 8], edx        ; (XL x YH)H
adc     dword ptr [ebx + 12], 0
mov     eax, [esi + 4]       ; XH
mul     dword ptr [edi + 4]    ; x YH
add     [ebx + 8], eax        ; (XH x YH)L
adc     [ebx + 12], edx       ; (XH x YH)H

```

Le résultat en mémoire est :

Produit = 0x01000000, 0x00000000, 0xFEFFFFFF, 0xFFFFFFFF, qui représente le nombre de 128 bits 0xFFFFFFFF FFFFFFFF 00000000 00000001.

La multiplication est une opération plutôt lente. Rappelons que la multiplication par une puissance de 2 peut être effectuée par un décalage de 1 ou plusieurs bits vers la gauche. Comme un décalage ne prend qu'un cycle, il est toujours avantageux de remplacer la multiplication par un décalage quand c'est possible.

$$00000011_2 \times 2 = 00000110_2$$

$$00000011_2 \times 4 = 00001100_2.$$

## 8.4 Division

Comme pour la multiplication, il y a deux instructions pour la division, **div** pour la division non signée, et **idiv** pour la division signée. Comme pour la multiplication, il y a un opérande destination implicite qui est la paire de registres `edx:eax` pour les opérations sur 32 bits, `dx:ax` pour les opérations sur 16 bits et `ax` pour les opérations sur 8 bits.

La syntaxe est la suivante :

```
div    reg
div    mem
```

```
idiv   reg
idiv   mem
```

Supposons que `eax = 0xFFFFFFFF8`

`eax` peut être interprété comme:

```
      4 294 967 28810    (non signé)
ou      -810    (signé)
```

Si `ebx = 0x00000003` et `edx = 0x00000000`, alors:

```
div    ebx          eax = 0x55555552
                        edx = 0x00000002
qui vaut 1 431 655 76210, reste 2.
```

Mais si `ebx = 0x00000003` et `edx = 0xFFFFFFFF`

```
idiv   ebx          eax = 0xFFFFFFFFE,
                        edx = 0xFFFFFFFFE
qui vaut -210, reste -2.
```

Par contre,

Si `ebx = 0x00000003` et `edx = 0xFFFFFFFF`

```
div    ebx          débordement de capacité
                        parce que le quotient est > 32 bits.
```

Avant une division de 32 bits non signée, on mettra `edx` à 0 au moyen de `xor edx, edx`. Avant une division signée, on utilisera plutôt l'instruction **cdq**, qui met `edx` à 0 si `eax` est positif et `edx` à `0xFFFFFFFF` si `eax` est négatif.

Il n'est pas possible d'effectuer la division de précision en utilisant plusieurs divisions sur des entités plus petites comme c'était le cas pour la multiplication. Si on veut effectuer une division sur des quantités de plus de 64 bits, il faut utiliser les décalages.

La division est une opération très lente. Rappelons que la division par une puissance de 2 peut être effectuée par un décalage de 1 ou plusieurs bits vers la droite. Comme un décalage ne prend qu'un cycle, il est toujours avantageux de remplacer la division par un décalage dans ce cas.

$$00001001_2 / 2 = 00000100_2$$

$$00001001_2 / 4 = 00000010_2$$

## 8.5 Décalages et rotations

L'architecture Intel supporte plusieurs instructions de décalage. On peut distinguer entre les décalages arithmétiques, **sal**, décalage arithmétique vers la gauche, et **sar**, décalage arithmétique vers la droite, et les décalages logiques, **shl**, décalage logique vers la gauche et **shr**, décalage logique vers la droite. Pour des décalages multiples, le nombre de bits à décaler est placé dans le registre de 8 bits CL. Il existe deux autres instructions de décalage, **shld** et **shrd** pour passer les bits d'un registre à un autre.

Le décalage arithmétique vers la gauche et le décalage logique vers la gauche sont identiques. Tous les bits sont déplacés de 1 bit vers la gauche, et la position laissée libre à droite est remplacée par un 0. Le bit sortant à gauche est placé dans l'indicateur de retenue CF.

Le décalage logique vers la droite déplace tous les bits de 1 bit vers la droite, et la position laissée libre à gauche est remplacée par un 0. Le bit sortant à droite est placé dans l'indicateur de retenue CF.

Le décalage arithmétique vers la droite déplace tous les bits de 1 bit vers la droite, et la position laissée libre à gauche est remplacée par le bit le plus significatif de l'opérande, le bit de signe. Ce décalage maintient donc le signe de l'opérande.

La syntaxe est la suivante :

sal/shl/sar/shr	reg, 1
sal/shl/sar/shr	mem, 1
sal/shl/sar/shr	reg, CL

sal/shl/sar/shr      mem, CL

Quant aux deux autres instructions de décalage **shld** et **shrd**, elles décalent les bits du deuxième opérande dans le premier opérande. Le nombre de bits à décaler est spécifié par le troisième opérande.

**shld** décale le premier opérande de  $n$  bits vers la gauche. Les positions laissées vacantes sont remplies par les  $n$  bits les plus significatifs du deuxième opérande.

**shrd** décale le premier opérande de  $n$  bits vers la droite. Les positions laissées vacantes sont remplies par les  $n$  bits les moins significatifs du deuxième opérande.

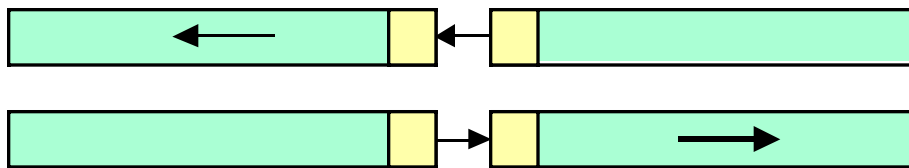


Figure 21

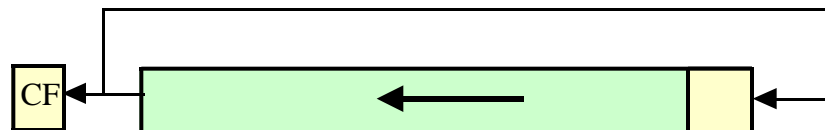
Syntaxe :

shld/shrd	reg, reg, immed
shld/shrd	reg, reg, CL
shld/shrd	mem, reg, immed
shld/shrd	mem, reg, CL

## Rotations

Mentionnons au passage les instructions de rotation **ROL**, **RCL**, **ROR** et **RCR**, qui se comportent comme des décalages, mais où le bit sortant, en plus d'être envoyé dans CF est appliqué à l'autre bout du registre. Les formes RCL et RCR sont des rotations sur 33 bits et sont souvent utilisées pour passer un bit d'un registre à l'autre

ROL





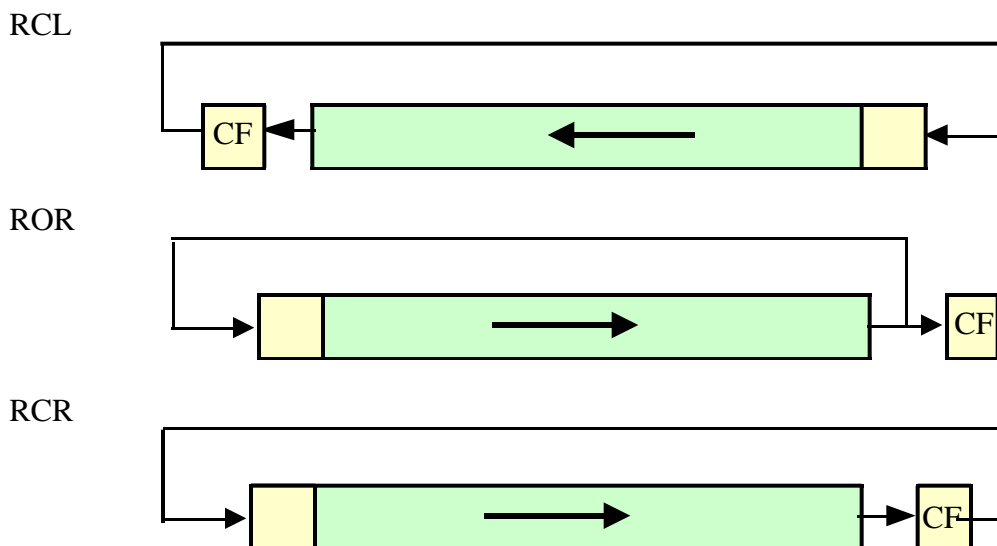


Figure 22 Rotations

## Division au moyen de décalages

Comme application de ces instructions, voyons comment effectuer la division au moyen de décalages. Ceci nous permettra par la suite d'effectuer une division sur plus de 64 bits.

Considérons d'abord la division binaire manuelle suivante :

$$\begin{array}{cccccc}
 1 & 0 & 0 & 1 & 1 & 1 \\
 -1 & 1 & 1 & \downarrow & \downarrow & \downarrow \\
 -1 & 1 & 1 & & & \\
 \hline
 0 & 0 & 1 & 0 & 1 & 1 \\
 & -1 & 1 & 1 & & \\
 & & -1 & 1 & 1 & \\
 \hline
 0 & 0 & 0 & 1 & 0 & 0
 \end{array}
 \quad
 \begin{array}{ccc}
 / & 1 & 1 & 1 \\
 \hline
 0 & & & \\
 & 1 & & \\
 & & 0 & \\
 & & & 1
 \end{array}$$

Donc,  $39/7 = 5$ , reste 4.

Cette méthode peut être appliquée telle quelle, sauf qu'il faut trouver une façon commode pour l'ordinateur de comparer au diviseur les bits les plus significatifs du dividende. On effectue cette opération au moins de décalages d'un registre Q contenant le dividende D à un autre registre R qui contiendra éventuellement le reste.

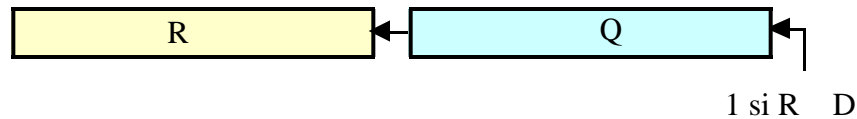


Figure 23. Division binaire

Initialement,  $R = 0$  et  $Q = \text{dividende}$ . On décale  $Q$  vers  $R$  bit par bit. À chaque décalage, si  $R > D$ , on remplace  $R$  par  $R - D$  et on insère un 1 à droite du quotient, sinon on insère un 0. Comme à chaque décalage une position est libérée à droite de  $Q$ , on peut économiser un registre en utilisant ces positions pour insérer les bits du quotient. On répète cette opération tant qu'il y a de bits dans le registre  $Q$ , par exemple 16 ou 32 fois. Finalement, on décale  $Q$  une dernière fois à gauche pour y insérer le dernier bit du quotient. À la fin de l'opération  $R$  contient le reste et  $Q$ , le quotient.

Avec un exemple numérique, vous observerez que les opérations sont exactement les mêmes qu'avec la méthode manuelle.

Pour illustrer le principe, voici un algorithme pour la division d'un registre de 32 bits par un registre de 32 bits.

```

mov     edx,dividende    ; edx = Q
mov     ebx,diviseur     ; ebx = D
mov     ecx,32           ; 32 décalages en tout
mov     esi,0            ; bit quotient = 0
xor     eax,eax          ; eax = R = 0
decal:  rcl     edx,1      ; décaler dividende dans eax
        rcl     eax,1      ; décaler dividende
        clc           ; bit quotient (dans CF) = 0
        cmp     eax,ebx    ; R > D ?
        jnb     suivt     ; si oui, soustraire diviseur, sinon bit suivant
        sub     eax,ebx    ; soustraire diviseur
        stc           ; bit quotient (dans CF) = 1
suivt:  loop    decal
        rcl     edx,1      ; dernier décalage quotient
                        ; quotient dans edx
        ret           ; reste dans eax

```

## Division de précision

Supposons qu'on veuille effectuer la division d'un nombre de 128 bits par un nombre de 64 bits. L'opération se fera plutôt sur des opérandes mémoire que dans des registres. Chaque rectangle dans l'illustration suivante représente un double mot de 32 bits.

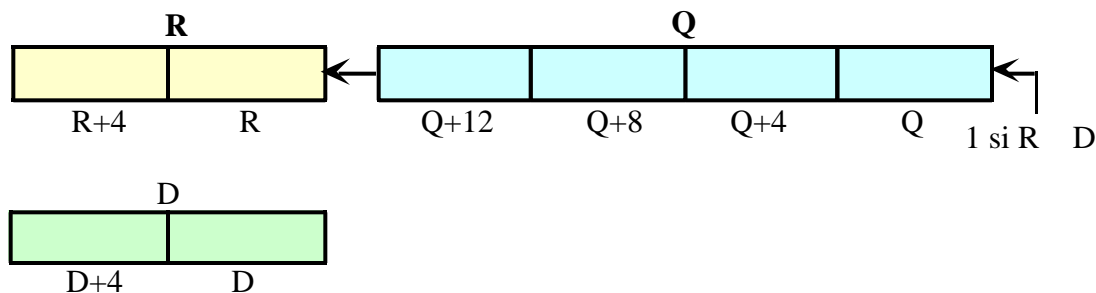


Figure 24. Division de précision

La légende sous chaque rectangle représente son adresse.

Pour effectuer cette opération, on a besoin des fonctionnalités suivantes (fonctions ou macros) :

```
longShiftLeft(src, dwords, shifts)
longCompare(dst, src, dwords)
longSub(dst, src, dwords)
```

Voici comment on pourrait l'implanter en pseudo code. Les trois fonctions ci-dessus vous sont laissées comme exercice.

On suppose que Q contient le dividende de 128 bits (4 doubles mots) et que R est 0 sur 64 bits. L'algorithme est exactement le même que pour la division binaire au moyen de décalages présentée plus haut.

```
q = 0
decal: longShiftLeft(R, 2, 1)           ; décalage de R de 1 vers la gauche
      shld      R, Q+12, 1             ; passage du bit le plus significatif de Q
      longShiftLeft(Q, 4, 1)          ; décalage de Q de 1 vers la gauche
      add      Q, q                    ; bit de quotient dans le bit le moins
      q = 0                            ; significatif de Q
      longCompare(R, D, 2)             ; R > D?
      jl      suivt
      longSub(R, D, 2)                 ; si oui, R = R - D
```

```
q = 1
suivt: loop    decal
        longShiftLeft(Q, r, 1)      ; dernier décalage de Q
        add     Q, q                ; dernier bit de quotient
```

## 8.6 Exemple : racine carrée par la méthode de Newton

La méthode de Newton obtient la racine carrée d'un nombre  $n$  par approximations successives comme suit :

$$x' = (x + n / x) / 2$$

On répète jusqu'à ce que la différence entre  $x$  et  $x'$  soit négligeable. Comme la méthode converge très rapidement, on peut aussi utiliser un nombre fixe d'itérations, par exemple 5 ou 10.

Racine proc n:DWORD

```
        mov     ebx, 10                ; valeur de départ pour x
        mov     ecx, 10                ; nombre d'itérations
Lp:      xor     edx, edx                ; doit être 0 pour la division 32 bits
        mov     eax, n
        div     ebx                    ; eax = n / x
        add     ebx, eax                ; x + n / x
        shr     eax, 1                 ; / 2
        dec     ecx                    ; décrémenter compteur
        jne     lp
        mov     eax, ebx                ; résultat dans eax
        ret
Racine  endp
```

## Exercices

1. Écrivez une fonction `longShiftLeft(long * src, short dwords, char shifts)` qui décale de *shifts* bits vers la gauche une chaîne de *dwords* éléments de type `long` situés à l'adresse *src*.
2. Écrivez une fonction `char longCompare(long * dest, long * src, char dwords)` qui compare *dwords* éléments de type `long` situés aux adresses *dest* et *src*. La valeur de type `char` retournée dans *al* contient 1 si *dest* > *src*, 0 si *dest* = *src* et -1 si *dest* < *src*.
3. Écrivez une fonction `void longSub(long *dst, long * src, char dwords)` qui soustrait un nombre de *dwords* doubles mots situés à l'adresse *src* d'un nombre de même longueur situé à l'adresse *dest*, et place le résultat à l'adresse *dest*.
4. Dans la division au moyen de décalages, que se passe-t-il si le diviseur est 0 ?

## 9. Conversions

On effectue souvent en assembleur les conversions élémentaires de binaire à décimal et vice-versa, parce que ces fonctions sont appelées très souvent et ont donc intérêt à être performantes.

La conversion entre le décimal et le binaire s'effectue habituellement comme suit :

Décimal (BCD) à binaire (voir supplément au volume, section 3.3.1) :

- $R = d + 10 \times R$

Pour la conversion binaire à décimal, trois méthodes sont fréquemment utilisées :

- $R = b + 2 \times R$  en arithmétique BCD
- Divisions répétées par 10 -> séquence de restes
- Tables

Celle qu'on choisit dépend du processeur utilisé, du jeu d'instructions dont il dispose, et du temps d'exécution de chacune.

### 9.1 Conversion décimal à binaire

Effectuons la conversion d'une chaîne de caractères décimaux représentant un nombre positif vers un nombre binaire de 32 bits non signé. On suppose que tous les caractères sont des caractères décimaux valides.

dec2Long proc decString:LPSTR

	mov	esi, decString	
	xor	eax, eax	; résultat = 0
While:	movzx	ebx, byte ptr[esi]	; while (*decString++)
	inc	esi	
	test	bl, bl	; terminé si nul
	je	fin	
	sub	bl, '0'	; conversion ASCII-binaire
	imul	eax, 10	; résultat *= 10
	add	eax, ebx	; résultat += digit
	jmp	While	
	ret		; résultat dans eax

Nous avons intentionnellement omis la détection du débordement de capacité, que nous laissons en exercice.

Ce même programme peut être utilisé pour la conversion de n'importe quelle base vers le binaire, en remplaçant la multiplication par 10 par la multiplication par cette autre base. Dans ce cas, il faut ajouter un test pour convertir en binaire les nombres > 9, représentés par des caractères alphabétiques :

```

                sub     bl, '0'           ; conversion ASCII-binaire
                cmp     bl, 9             ; si c'était une lettre
                jle     suite
                sub     bl, 7             ; 'A'-'9'+1)
suite:          imul    eax, 10

```

## 9.2 Conversion binaire à décimal

### 1<sup>re</sup> méthode : divisions par 10

```
temp      db      12 dup(0)           ; espace pour restes = 12 zéros
```

bin2String proc n:DWORD, String:LPSTR

```

                mov     eax, n           ; nombre à convertir
                lea     esi, temp
                add     esi, 10           ; pointer sur fin de temp
                mov     ecx, 10          ; 10 pour la division
divis:         mov     edx, 0            ; eax = quotient précédent
                div     ecx              ; diviser par 10
                add     dl, '0'          ; convertir reste en ASCII
                mov     [esi], dl        ; stocker dans temp
                dec     esi              ; écrire le suivant à gauche
                test    eax, eax
                jne     divis            ; tant que quotient != 0
                inc     esi              ; pointer sur dernier reste
                mov     edi, decString   ; adresse destination
copy:          lodsb                    ; reste suivant
                stosb
                test    al, al           ; tant que reste != 0
                jne     copy
                ret

```

Ce même programme peut être utilisé pour passer du binaire à n'importe quelle base en divisant par cette base au lieu de diviser par 10. Il faut ajouter un test après add dl,'0' :

```

    cmp    dl, '9'                ; si reste est > 9, base > 10 et c'est une lettre
    jle    suite                  ; on ajoute alors 'A' - ('9'+1)
    add    dl, 7                  ; ou add dl, 'a'-( '9'+1) si on veut des minuscules
suite: mov    [esi], dl

```

## 2<sup>e</sup> méthode : $R = b + 2 * R$

```
temp    db    5 dup(0)
```

```
bin2decString proc n:DWORD, decString:LPSTR
```

```

    mov     ebx, n
    mov     edx, 32                ; 32 bits à traiter
    lea     edi, temp
decal:    rcl     ebx, 1            ; mettre le msb dans CF
    mov     esi, 4
    mov     ecx, 5
abcd:     mov     al, byte ptr [edi+esi] ; addition bcd sur 5 octets
    adc     al, al                 ; b + 2 * R
    daa                                ; ajustement BCD
    mov     byte ptr [edi+esi], al   ; résultat dans temp
    dec     esi
    loop    abcd                  ; fin abcd
    dec     edx
    jne     decal                 ; fin décal
    mov     ecx, 5
    mov     esi, str
copy:     mov     al, [edi]         ; copier dans destination et
                                         ; convertir le BCD compacté
    mov     ah, al                ; en BCD non compacté
    and     ax, 0xF00F            ; masquer lsb et msb
    shr     ah, 4                 ; décaler msb en place
    add     ax, 0x3030            ; convertir en ASCII
    xchg    ah, al                ; little endian
    mov     word ptr [esi], ax     ; stocker deux car
    inc     edi                   ; incrémenter source
    add     esi, 2                 ; incrémenter destination
    dec     ecx
    jne     copy
    mov     byte ptr[edi], 0       ; car de fin de chaîne
    ret

```



### 3<sup>e</sup> méthode : à l'aide d'une table

Table dd 1000000000, 100000000, 10000000, 1000000, 100000, 10000, 1000, 100,  
10, 1

bin2decString proc n:DWORD, decString:LPSTR

```

    mov     eax, n
    lea     esi, Table
    xor     ebx, ebx
    mov     edi, decString
    mov     ecx, 10                ; 10 nombres dans Table
digit:    mov     byte ptr [edi+ebx], '0'    ; résultat = ' 0 '
    mov     edx, dword ptr [esi+ebx*4]    ; Table[i]
comp:     cmp     eax, edx                ; peut-on soustraire?
    jb     suiv
    sub     eax, edx                    ; soustraire Table[i]
    add     byte ptr [edi+ebx], 1        ; incrémenter résultat
    jmp     comp                    ; tant que eax > Table[i]
suiv:     inc     ebx                    ; élément suivant
    loop    digit
    mov     byte ptr [edi+ebx], 0        ; car de fin de chaîne
    ret

```

Certains raffinements pourraient s'avérer souhaitables dans les programmes présentés dans cette section. Notamment :

Conversion décimal-binaire

- Traitement des nombres négatifs.
- Traitement des espaces au début.

Conversion binaire-décimal

- Traitement des nombres négatifs.
- Éliminer les zéros initiaux, par exemple, retourner 3 au lieu de 0000000003.

## 9.3 Conversion binaire à chaîne hexadécimale

Supposons qu'on veuille convertir un nombre binaire de 32 bits en chaîne hexadécimale qui pourra être affichée à l'écran. On peut convertir en base 16 avec le programme bin2String ci-dessus (9.2 1<sup>e</sup> méthode) ou encore :

```
Long2HexString proc num:DWORD, HexString:LPSTR) {
    mov     edi, HexString
    mov     eax, num
    mov     ecx, 8                ; 8 car. pour représenter un long
lp:        rol     eax, 4          ; 4 bits suivants
    mov     bl, al
    and     bl, 0x0F              ; masquer les 4 bits de droite
    add     bl, '0'               ; convertir en ASCII
    cmp     bl, '9'
    jbe     ok                    ; si c'est un chiffre 0-9
    add     bl, 7                  ; si c'est une lettre A-F
ok:        mov     [edi],bl        ; placer dans la chaîne
    inc     edi
    dec     ecx
    jnz     lp
    mov     byte ptr [edi], 0      ; caractère de fin de chaîne
    ret
}
```

Le programme principal appelant cette fonction pourrait ressembler à ceci :

```
HexString db 10 dup(?)
n         dd 1234h

invoke    HexString, n, ADDR HexString
invoke    StdOut, HexString
}
```

Vous devriez voir 00001234 à l'écran.

Pour cette conversion, on pourrait également utiliser une table et l'instruction **xlat** (translate) :

---

```
table db '0123456789ABCDEF'
```

```
Long2HexString proc num:DWORD, HexString:LPSTR
```

```
    mov     edi, HexString
    lea     ebx, table           ; table de traduction dans ebx
    mov     ecx, 8              ; toujours 8 car.
    mov     edx, num            ; nombre à convertir
lp:    rol     edx, 4            ; digit suivant
    mov     al, dl
    and     al, 0x0F            ; garder 4 bits seulement
    xlat                                ; traduire
    mov     [edi], al           ; écrire dans destination
    inc     edi
    dec     ecx
    jne     lp
    move    byte ptr [edi], 0    ; caractère de fin de chaîne
    ret
```

## Exercices

D'autres conversions utiles sont les suivantes.

1. Écrire une fonction **long** `hexString2Long (const char * str)` effectuant la conversion d'une chaîne hexadécimale située à l'adresse `str` en un nombre binaire de 32 bits dans `eax`.
2. Écrire une fonction `void long2BinString (const long n, char * str)` qui convertit un nombre binaire de 32 bits en une chaîne de caractères binaire non signée située à l'adresse `str`.
3. Écrire une fonction **long** `binString2Long (const char * str)` qui convertit une chaîne de caractères binaires non signée située à l'adresse `str` en un nombre de 32 bits retourné dans `eax`.
4. Ajouter dans le programme *dec2Long* la détection des débordements de capacité qui peuvent survenir lors de l'addition et lors de la multiplication.
5. Modifiez le programme *dec2Long* pour qu'il effectue la conversion d'une chaîne décimale signée vers un nombre binaire en complément à 2.

## 10. La Pile

### 10.1 Empilement et dépilement

La plupart des microprocesseurs CISC utilisent abondamment la pile, surtout pour le passage des paramètres aux sous-programmes. Le Pentium ne fait pas exception. Il dispose de deux instructions pour empiler et dépiler les données :

L'instruction **push** décrémente le pointeur de pile ESP de la taille de l'opérande, puis écrit l'opérande en mémoire à l'adresse contenue dans ESP. C'est l'empilement.

L'instruction **pop** lit l'opérande à l'adresse ESP puis incrémente ESP de la taille de l'opérande. C'est le dépilement.

En mode 32 bits, l'instruction **push** a l'effet suivant :

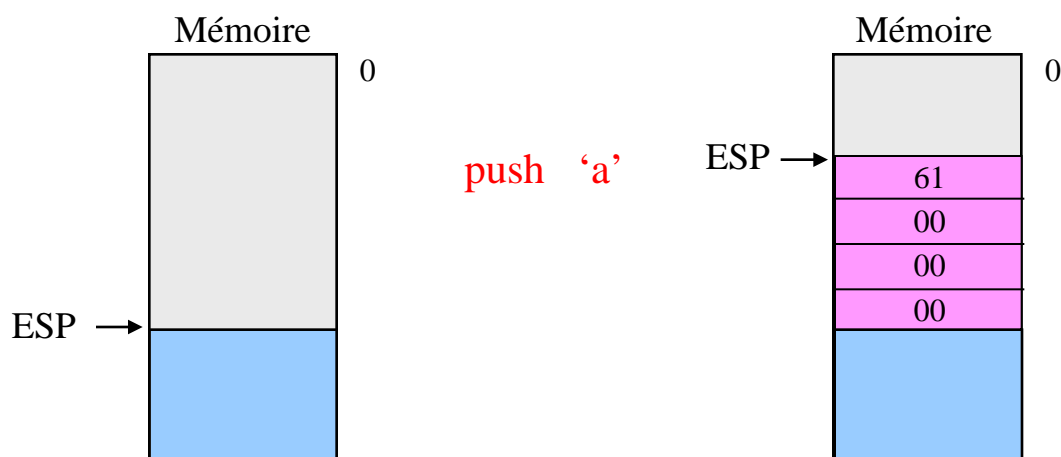


Figure 25

L'instruction **push** tient compte de la taille des opérandes s'il y a lieu, mais le nombre d'octets empilés est toujours pair.

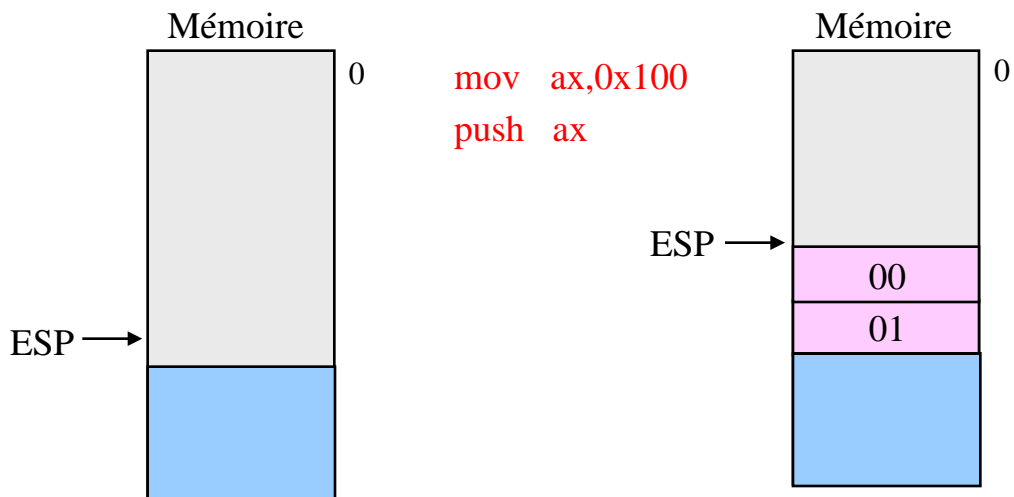


Figure 26

L'instruction **pop** a l'effet contraire :

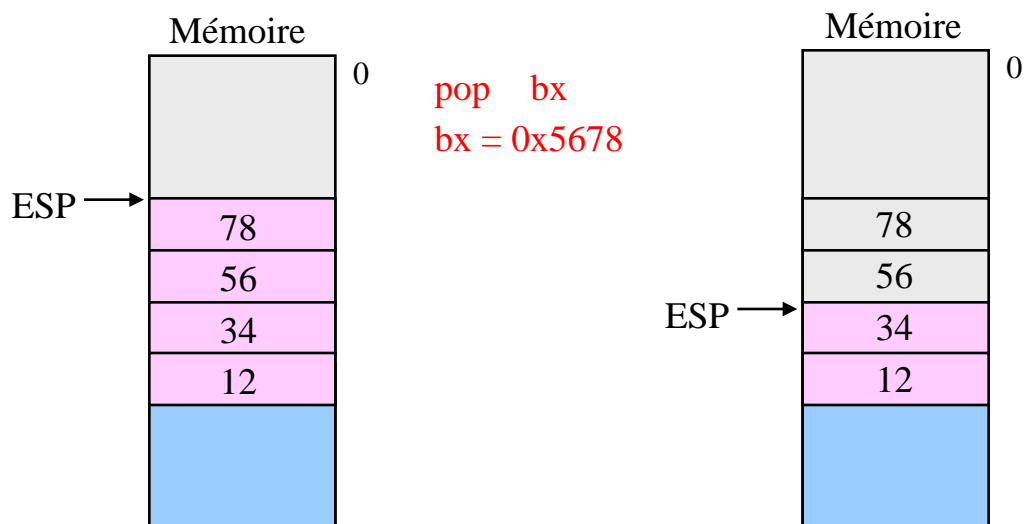


Figure 27

On peut utiliser la pile pour passer une donnée d'un emplacement mémoire à un autre sans utiliser de registre :

```
push  a
pop   b
```

On peut de cette façon échanger la valeur de deux variables :

```
push  a
push  b
pop   a
pop   b
```

## 10.2 Sous-programmes

L'architecture Intel dispose d'instructions pour l'appel de sous-programmes et pour le retour de sous-programmes. Jusqu'ici, nous avons utilisé la directive **invoke** qui est traduite par l'assembleur en instructions **push** et **call**.

L'instruction **call** pousse l'adresse de l'instruction suivante sur la pile et saute à l'adresse spécifiée par l'opérande. Sa syntaxe est la suivante :

```
call  etiquette
call  eax
call  dword ptr [ebx]
call  table[esi]
```

L'instruction **ret** dépile l'adresse de retour au sommet de la pile, la place dans EIP. L'exécution se poursuit donc à cette adresse.

```
ret
ret    32    ; dépile 32 octets de plus
```

### Passage des paramètre aux sous-programmes

#### Passage par valeur

La valeur numérique du paramètre est passée au sous-programme.

#### Exemple : sqrt (100)

Le sous-programme ne peut pas modifier le paramètre en mémoire, car il ne connaît pas son adresse.

C'est le cas pour les données de 8, 16 et 32 bits (64 bits en virgule flottante).

### Passage par référence, par adresse ou par pointeur

C'est l'adresse du paramètre qui est passée au sous-programme. C'est le cas pour les structures, les tableaux, les chaînes.

Le sous-programme connaît l'adresse mémoire du paramètre et peut en principe le modifier.

Par exemple, l'instruction

```
scanf("%d", &num);
```

lit une information au clavier et l'inscrit en mémoire à l'adresse de la variable num.

### Passage par registres

On dispose des six registres `eax`, `ebx`, `ecx`, `edx`, `edi` et `esi` pour le passage des paramètres. Cette méthode est la plus rapide puisqu'elle minimise les accès à la mémoire.

#### Appel :

```
mov    ebx, param1
mov    ecx, param2
call   sous-programme
```

Le sous-programme se termine simplement par `ret`.

C'est la méthode utilisée par les processeurs RISC. Dans ces processeurs, même l'adresse de retour d'un sous-programme est placée dans un registre pour éviter l'utilisation de la pile (accès mémoire).

Avec le Pentium, étant donné le petit nombre de registres disponibles, cette technique n'est utilisable que si on n'a qu'un ou deux paramètres.

### Passage par la pile

Lors d'un passage par la pile, le programme appelant empile d'abord les paramètres. Il exécute ensuite l'instruction `call`, qui empile l'adresse de retour. Au moment d'entrer dans le sous-programme, la pile a donc l'allure suivante :



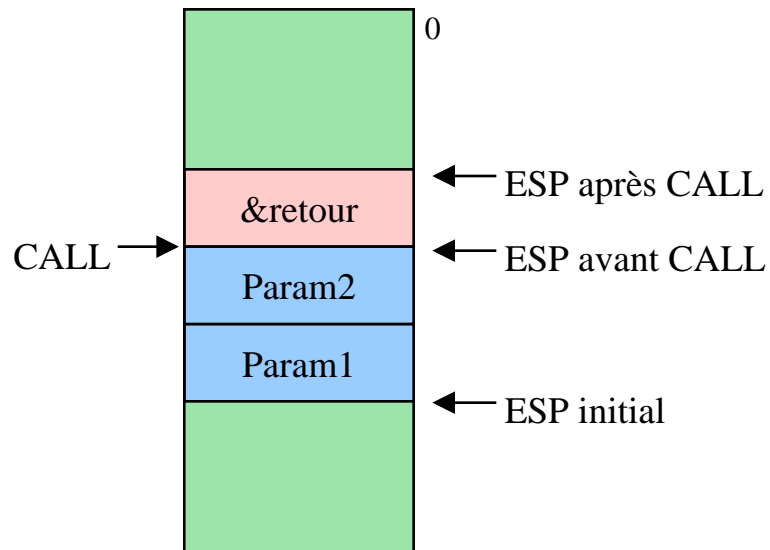


Figure 28

L'ordre d'empilement des paramètres dépend du langage. Le langage C empile les paramètres dans l'ordre inverse de leur déclaration, tandis que le Pascal les empile dans l'ordre de leur déclaration. En assembleur, on fait comme on veut.

D'autre part, il faut décider lequel, du programme principal ou du sous-programme, aura la responsabilité de restaurer la pile. Ceci aussi dépend du langage. Dans le cas du C, c'est le programme appelant qui a la responsabilité de nettoyer la pile.

Soit le sous-programme `Fonction(a, b, c)`. Ce sous-programme sera appelé comme suit si on utilise la convention C :

```
push  c
push  b
push  a
call  Fonction
add   esp, 12
```

Donc le sous-programme remet la pile inchangée, sauf pour l'adresse de retour qui a été dépilée par l'instruction **ret**, et c'est le programme principal qui ajuste le pointeur de pile à sa valeur initiale d'avant l'empilement des paramètres.

À l'intérieur du sous-programme, on accède aux paramètres en y référant par leur position relativement à `esp`. Dans l'exemple ci-dessus, on accède à `Param1` et `Param2` avec les instructions :

## Assembleur

```
mov    eax, [esp+4]      ; eax = Param2
mov    ebx, [esp+8]      ; eax = Param1
```

### Valeurs retournées

Si la valeur retournée est un type simple **char**, **short**, **long**, **ptr** ou **float**, alors cette valeur est placée dans **e(ax)** par le sous-programme. Un quadword est retourné dans **edx:eax**.

## 10.3 Blocs de pile

La méthode que nous venons de voir consistant à adresser les paramètres par rapport au pointeur de pile manque de flexibilité. En effet, le sous-programme doit habituellement empiler les registres qu'il utilise. On doit donc tenir compte des registres enregistrés pour déterminer l'adresse des paramètres. Si, par la suite, on change le nombre de registres empilés, il faut changer tous les offsets.

De plus, on souhaite disposer d'une façon élégante de faire de l'allocation dynamique pour les variables locales.

On atteint ces objectifs en créant pour le sous-programme un **bloc de pile** (stack frame).

```
prologue: push    ebp      ; sauvegarde ancien ebp
           mov     ebp, esp ; copie esp dans ebp
           push    registres ; on empile nreg registres
           sub     esp, n    ; n octets pour variables locales
           ...
           corps du sous-programme
           ...
épilogue:  add     esp, n    ; ou lea esp, dword ptr [ebp-4*nreg]
           pop     registres ; on dépile les registres
           pop     ebp      ; on dépile l'ancien ebp
           ret
```

Voici l'allure de la pile en présence d'un tel bloc de pile :

0

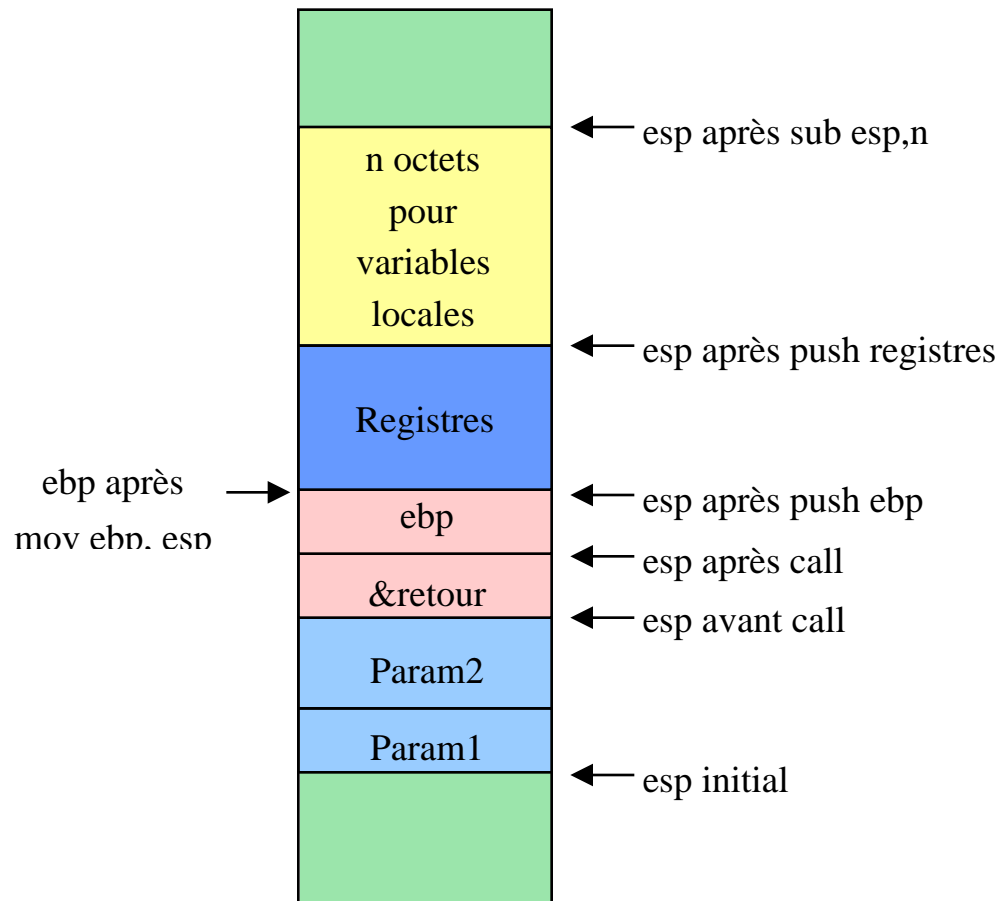


Figure 29

De cette façon, ebp fournit un point fixe par rapport auquel on peut référer aux paramètres. On réfère aux variables locales par rapport au pointeur de pile esp ou par rapport à ebp.

### Accès aux paramètres

```
mov  ebx,[ebp+8] ; charge param2
mov  ecx,[ebp+12] ; charge param1
...
```

Stockage dans une variable locale

```
mov  dword ptr [esp], eax
mov  word ptr [esp+4], ax
```

ou encore :

```
mov  dword ptr [ebp - 24]
```

Dans le fragment de code suivant, on déclare deux variables locales *i* et *j*. La figure suivante indique où elles sont placées et le code indique comment on y accède :

```
sousProgramme proc
    LOCAL i:WORD
    LOCAL j:DWORD
    ...
    mov     i, 3                ; i = 3;
    mov     j, 4                ; j = 4;
    ...
    ret
```

Sera traduit par :

```
SousProgramme:
prologue:      push     ebp
                mov      ebp, esp
                sub      esp, 6
                mov      word ptr [ebp-2], 3    ; ou encore [esp+4], 3
                mov      dword ptr [ebp-6], 4    ; ou encore [esp], 4
```

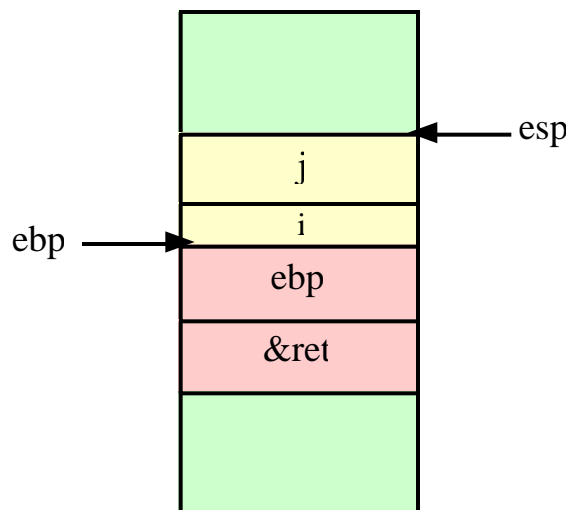


Figure 30

Quand on utilise **invoke** au lieu de **call**, l'assembleur crée automatiquement les blocs de pile appropriés. Pour déterminer quels registres seront empilés dans le bloc de pile, on utilise **uses** :

SousProgramme proc uses ebx ecx, param:DWORD

Dans ce cas, on est assuré que ebx et ecx seront préservés en plus de ebp et esp. Sinon, seuls esp et ebp sont sauvegardés.

## 10.4 Récursivité

Un programme récursif est un programme qui s'appelle lui-même. Un exemple simple est celui de la fonction factorielle, même si ce n'est pas une façon très efficace d'évaluer cette fonction:

```
unsigned long factorial(unsigned long n)
{
    if (n == 1)
        return (1);
    else
        return (n * factorial(n - 1));
}
```

En assembleur du Pentium, on aura:

```
factorial proc n:DWORD
    mov     eax, n                ; n = [esp + 4]
    cmp     eax, 1                ; if (n == 1)
    jne     suite
    mov     eax, 1                ; return 1
    ret
suite:    dec     eax
    push    eax                  ; factorial (n - 1)
    call    factorial
    add     esp, 4                ; nettoyer la pile
    mov     ebx, n                ; eax contient factorial(n-1)
    mul     ebx                  ; n * factorial (n - 1) dans eax
    ret
```

Voici à quoi ressemblera la pile lors de l'exécution :

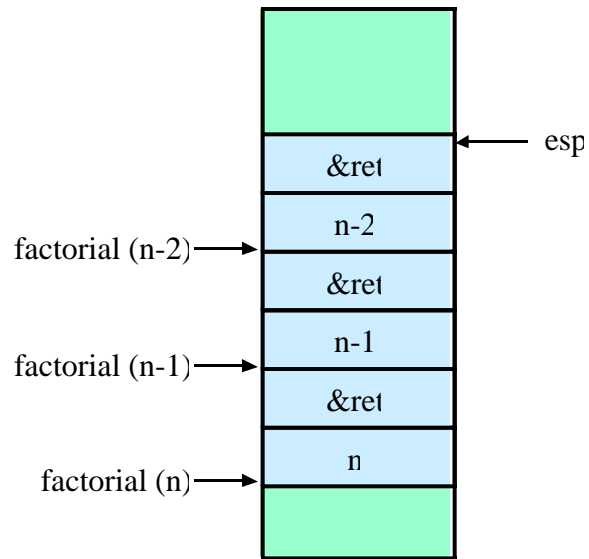


Figure 31

## 10.5 Tables de sauts

Dans les systèmes à base de menus, il arrive souvent qu'on ait un énoncé **switch** comme le suivant, où une fonction différente est exécutée suivant le choix que l'utilisateur fait dans un menu :

```
switch (choix) {
    case 0: fonction1(choix);
        break;
    case 1: fonction2(choix);
        break;
    case 2: fonction3(choix);
}
```

En C, on peut obtenir une meilleure performance, surtout si la liste est longue, à l'aide d'une table de pointeurs de fonctions :

```
void (* mesFonctions [3]) (int) = { fonction1, fonction2, fonction3 };
ou
int (* mesFonctions [3]) (int) = { fonction1, fonction2, fonction2 };
```

Ensuite, on peut appeler une de ces fonctions comme suit :

```
(* mesFonctions [choix] ) ( choix );
```

On saute directement à la bonne fonction sans avoir à parcourir le **switch**.

En assembleur, on peut réaliser une table de saut comme suit :

```
mov    eax, choix
push   eax
mov     esi, offset Table      ; ou lea esi, Table
call    dword ptr [esi+eax*4]
add     esp, 4
jmp     finTable
Table: dd    offset fonction1
        dd    offset fonction2
        dd    offset fonction3
finTable:
```

La table peut être placée n'importe où. Ici, on l'a placée directement dans le code.

## Exercices

1. Écrivez le prologue et l'épilogue du sous-programme suivant :

```
maFonction(short a, long b) {
    long i;
    short j, k;
    char b[12];
    ...
}
```

2. Réécrivez la fonction *factorial* ci-dessus en utilisant la directive **invoke**.
3. Codez en assembleur la fonction suivante pour calculer les nombres de Fibonacci sans utiliser la directive **invoke** :

```
int fibonacci (int n) {
    if (n <= 1)
        return 1;
    else
        return (fibonacci(n-1) + fibonacci(n-2));
}
```

# 11. Interruptions

On peut distinguer deux sortes d'interruptions. Les interruptions matérielles et les interruptions logicielles. Les deux sont traitées de la même façon. La différence tient à ce que les interruptions matérielles peuvent se produire n'importe quand, tandis que les interruptions logicielles se produisent à un endroit précis du code où se trouve une des instructions **int** ou **into**. De plus, les interruptions matérielles peuvent être masquées (interdites ou autorisées) à l'aide de l'indicateur IF.

## 11.1 Interruptions logicielles

Les interruptions logicielles, sont aussi appelées trappes ou déroutements. Elles incluent aussi les fautes et les arrêts. Une faute se produit quand le processeur détecte une erreur durant le traitement d'une instruction. Par exemple, division par 0, opcode invalide, etc. Quand une erreur est grave au point qu'une partie du contexte d'exécution est perdue, le résultat est un arrêt.

**En mode réel**, quand le processeur rencontre une instruction telle que **int immed8**, il va lire la table des vecteurs d'interruption **IVT** (Interrupt Vector Table). Cette table de 1 Ko est située à l'adresse 0000:0000. Chaque entrée de la table contient le numéro de segment de 16 bits et l'offset de 16 bits pour l'adresse d'un sous-programme de traitement d'interruption (Interrupt Service Routine ou **ISR**). Immed8 est utilisé comme indice dans cette table (on doit le multiplier par 4 pour avoir l'adresse physique correspondante).

De plus, les indicateurs FLAGS, CS et IP sont empilés dans cet ordre exact, puis les indicateurs TF et IF sont mis à 0. Le sous-programme d'interruption devra se terminer par l'instruction **iret**, pour dépiler correctement ces paramètres.

L'exécution se poursuit ensuite à l'adresse contenue dans la table. Par exemple, **int 8** ira lire le vecteur situé à l'adresse 32 (0x20) et branchera à l'adresse qui y est contenue. À cette adresse doit débiter un sous-programme de traitement d'interruption. Certains ISR font partie du système d'exploitation et sont déjà définis, comme ceux qui correspondent aux INT 0x21 de MS-DOS. Cette trappe y servait de mécanisme d'appel au système d'exploitation. Par contre, si vous voulez définir votre propre sous-programme de traitement d'interruption, il suffit d'aller écrire son adresse à l'endroit approprié dans la table des vecteurs d'interruption avant de l'utiliser.



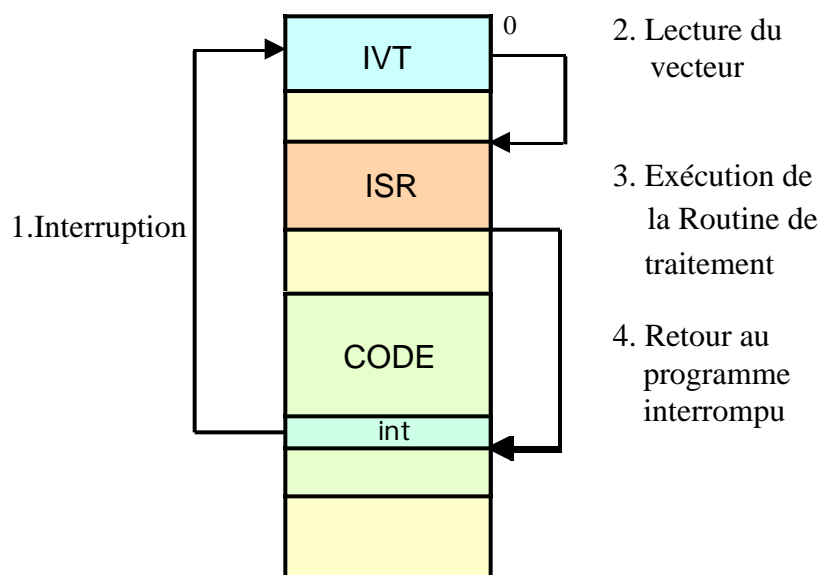


Figure 32. Déroulement d'une interruption logicielle en mode réel

Adresse		Interruption
0000	IP	INT 00 Division par 0
	CS	
0004	IP	INT 01 Exécution pas à pas
	CS	
0008	IP	INT 02 NMI
	CS	
000C	IP	INT 03 Point d'arrêt
	CS	
0010	IP	INT 04 Débordement de capacité (INTO)
	CS	
03FF	IP	INT FF
	CS	

Table de vecteurs d'interruption

**En mode protégé**, le processeur va plutôt lire la table des descripteurs d'interruption **IDT** (Interrupt Dispatch Table). L'adresse de cette table est contenue dans le registre IDTR. Elle contient des descripteurs de 64 bits pour les sous-programmes de traitement d'interruption. Ces descripteurs sont appelés Trap Gates dans le cas d'interruptions logicielles.

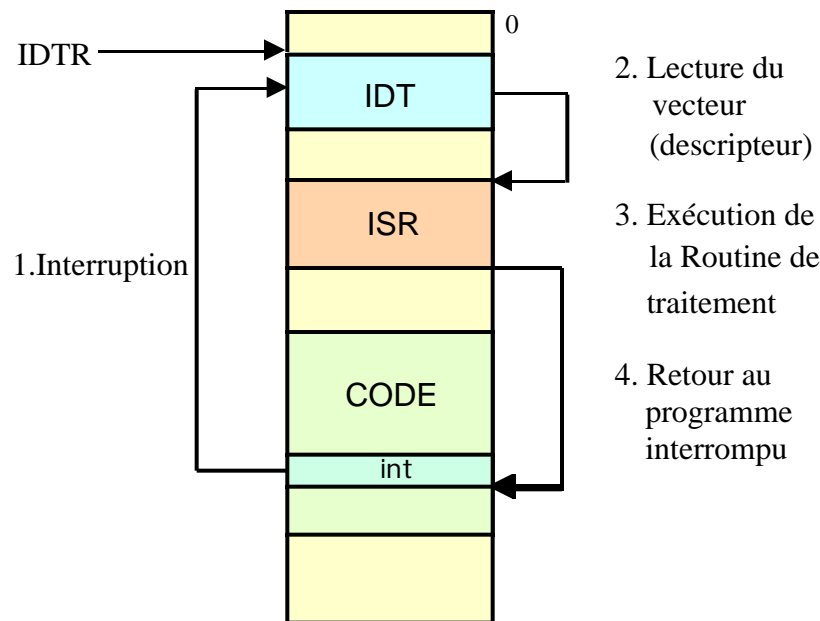


Figure 33. Déroulement d'une interruption logicielle en mode protégé

La procédure qui veut utiliser ces instructions doit avoir un niveau de privilège lui permettant d'accéder au descripteur d'interruption de la table IDT pour le numéro de trappe en question. Ainsi, si une application de niveau 3 veut exécuter **int 47**, il faut que le descripteur à IDT(47) ait DPL=3.

Dans un système d'exploitation tel que Windows NT, la IDT se trouve en mémoire protégée, de sorte qu'il ne vous est pas possible d'aller écrire un descripteur directement dans la IDT. Windows NT ne supporte pas non plus les INT 0x21 de MS-DOS. Toutefois, vous pouvez les essayer en écrivant un programme assembleur et en l'exécutant dans la fenêtre «Invite de commande» (DOS prompt).

Vous trouverez sur le site Web du cours un document décrivant un très grand nombre d'interruptions logicielles de MS-DOS. En particulier, si vous désirez implanter votre propre sous-programme de traitement d'interruption logicielle, l'appel système **int 21h** (33) fonction 25 vous permet de le faire facilement. Vous pouvez aussi créer des programmes résidents (Terminate process and Stay Resident ou TSR) grâce à l'appel système **int 21h** fonction 31.

## Exemples d'interruptions logicielles sous MS-DOS :

Écriture d'un caractère à l'écran.

Pour écrire un caractère *car* à l'écran, on peut utiliser l'interruption **int 0x10** (16) :

```
mov    al, car
mov    bx, 0
mov    ah, 14
int     16
```

On peut aussi utiliser l'interruption **int 0x21** (33) fonction 09 comme suit pour imprimer une chaîne de caractères :

```
mov    ah,09
mov    dx, offset string
int     33
```

où on a défini la chaîne de caractères *string* comme suit ;

```
string db    'Toto','$'           ; le car. de fin de chaîne sous DOS est $.
```

## 11.2 Interruptions matérielles

Les interruptions matérielles sont générées par les périphériques : souris, clavier, disque, horloge temps réel, etc. À la différence des interruptions logicielles, elles peuvent être autorisées ou interdites au moyen de l'indicateur IF du registre EFLAGS.

Comme le Pentium n'a que deux entrées d'interruption matérielle, NMI et INTR, on doit lui adjoindre un contrôleur d'interruptions programmable afin de pouvoir disposer de plusieurs sources d'interruption avec une gestion de priorité. C'est le PIC (Programmable Interrupt Controller 8259A).

Le 8259A peut accepter les interruptions de 8 sources externes, et on peut gérer jusqu'à 64 sources différentes en cascadeant plusieurs 8259A. Il gère la priorité entre les interruptions simultanées, interrompt le processeur et lui passe un code pour identifier la source d'interruption.

Une source d'interruption est connectée à chacune des 8 entrées **IR0** à **IR7**. Selon sa priorité, et s'il n'y a pas d'autre interruption en cours, le PIC décide s'il peut transmettre l'interruption au CPU.

Si oui, il affirme la ligne INT, qui est connectée à l'entrée **INTR** (Interrupt Request) du CPU. Si le CPU est prêt à accepter l'interruption, il répond au PIC via la ligne **INTA** (Interrupt Acknowledge).

Le PIC répond à son tour en envoyant le numéro d'interruption sur les lignes **D0** à **D7**. Ce numéro est un index dans la table des vecteurs d'interruption.

Le CPU est maintenant prêt à appeler le sous-programme de traitement d'interruption approprié.

Quand le sous-programme de traitement d'interruption a terminé son exécution, il en avertit le PIC pour qu'il puisse permettre à d'autres interruptions d'atteindre le CPU.

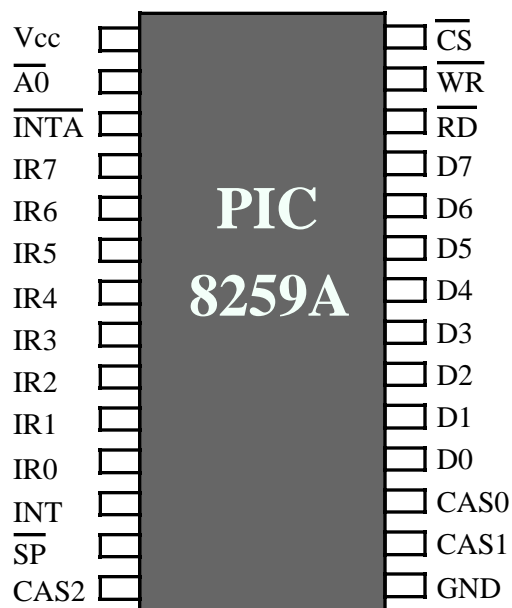


Figure 34. Le contrôleur d'interruptions 8259A

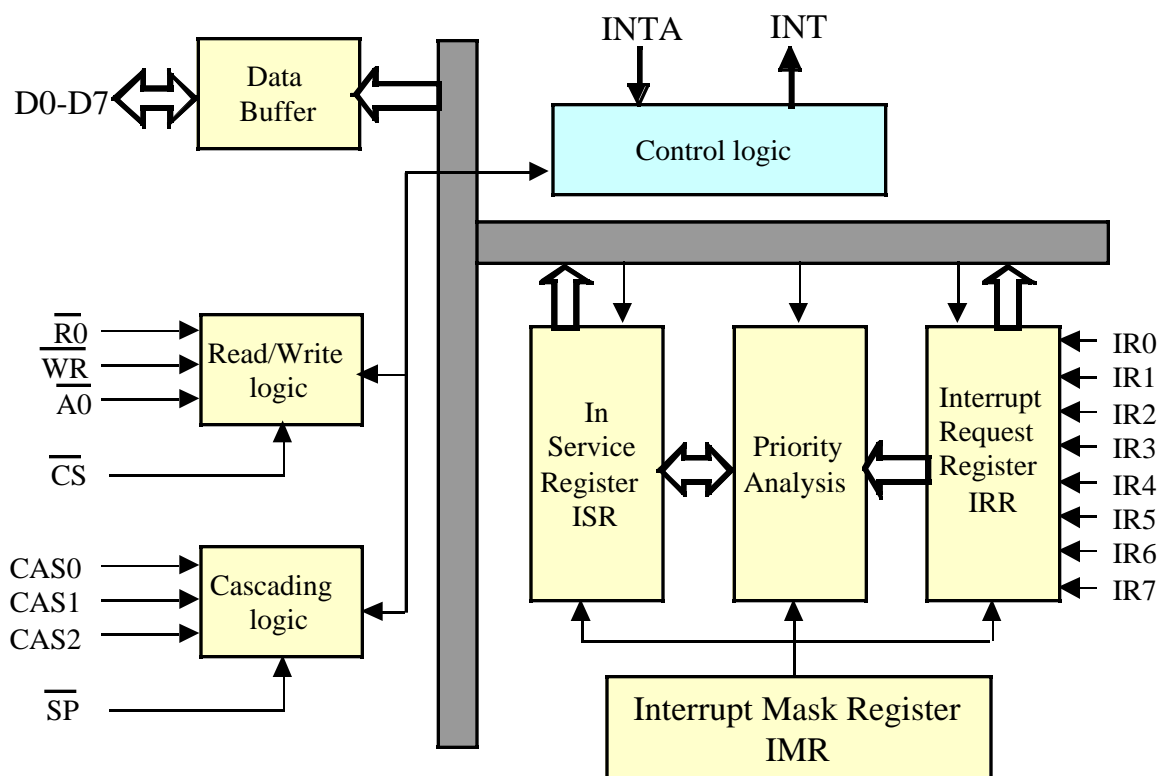


Figure 35. Organisation interne du 8259A

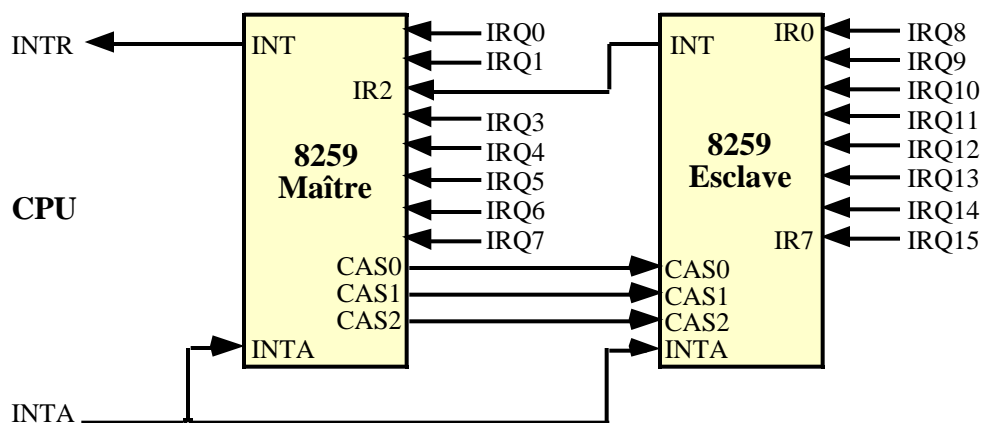


Figure 36. Façon dont sont connectés les 8259 en cascade dans le PC

Le PIC comporte deux sortes de commandes : les commandes d'initialisation (ICW) et les commandes opérationnelles (OCW). Ces commandes sont appliquées à l'un des deux registres internes. Par exemple, la commande OCW1 permet d'autoriser ou d'interdire les interruptions sélectivement. Le PIC maître occupe les adresses 0x20 et 0x21, tandis que le PIC esclave occupe les adresses 0xA0 et 0xA1. Ces adresses sont dans l'espace mémoire d'entrée-sortie et on ne peut y accéder qu'avec les instructions **in** et **out**. Une autre commande permet de programmer le PIC pour qu'il connaisse l'interruption de base

appliquée à ses entrées. Dans le PC, les entrées IRQ0 et IRQ1 du PIC maître sont reliées respectivement au temporisateur 8254 et au clavier et sont affectées respectivement aux vecteurs d'interruptions 08 et 09. Donc l'interruption de base du PIC maître est 8. L'interruption de base du PIC esclave est 0x70.

Attribution des interruptions matérielles dans un PC ISA :

IRQ	No. INT	Fonction
IRQ0	INT08	Sortie du temporisateur 8254
IRQ1	INT09	Clavier
IRQ2	INT0Ah	Interruption du PIC esclave
IRQ3	INT0Bh	Ports série COM2 et COM4
IRQ4	INT0Ch	Ports série COM1 et COM3
IRQ5	INT0Dh	Port parallèle 2 (LPT2)
IRQ6	INT0Eh	Contrôleur de disque souple
IRQ7	INT0Fh	Port parallèle 1 (LPT1)
IRQ8	INT70h	Horloge temps réel
IRQ9	INT71h	Disponible
IRQ10	INT72h	Disponible
IRQ11	INT73h	Disponible
IRQ12	INT74h	Souris
IRQ13	INT75h	Coprocasseur arithmétique
IRQ14	INT76h	Disque dur
IRQ15	INT77h	Disponible

### Utilité des interruptions matérielles

Les interruptions matérielles servent à une gestion efficace des périphériques d'entrée /sortie. Reportez-vous à la section 10.6 du livre de Zanella et Ligier. Dans un ordinateur moderne, il y a continuellement des interruptions matérielles. Le temporisateur, l'horloge temps réel, les touches du clavier, les mouvements et les clics de la souris, le modem, l'imprimante, les disques durs et souples, le cédérom, sont tous des sources d'interruptions.

Les circuits contrôleurs de périphériques contiennent plusieurs registres d'interface avec le CPU. Il y a habituellement un registre de contrôle, un registre d'état, et un ou plusieurs registres de données. Pour connaître l'état d'un périphérique, le CPU peut interroger le registre d'état. L'approche des drapeaux, mentionnée à la p 264 du livre, appelée encore

scrutation ou «polling», consiste à interroger de façon répétitive le registre d'état, pour savoir où le périphérique est rendu dans le transfert des données. A-t-il reçu une nouvelle donnée, a-t-il terminé la transmission de la dernière donnée envoyée, etc. Cette approche consomme trop de temps de la part du processeur.

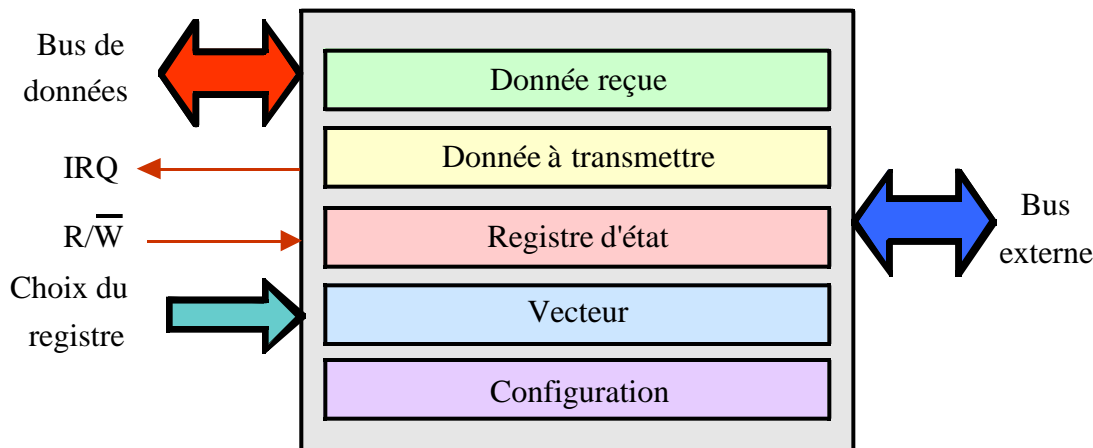


Figure 37. Schéma général d'une interface de périphérique

L'approche interruption est beaucoup plus performante. Le périphérique envoie une interruption matérielle au processeur quand il a quelque chose à signaler. Le processeur interrompt alors la tâche en cours, enregistre en mémoire l'état de la machine, et vient interroger le registre d'état du périphérique, pour connaître la cause de l'interruption. Il effectue ensuite le traitement approprié et élimine la source de l'interruption. Ce traitement consiste, par exemple, à lire la donnée reçue dans le registre de réception et à l'inscrire en mémoire, ou à lire en mémoire la prochaine donnée à transmettre et à l'inscrire dans le registre de transmission. Le processeur retourne ensuite à la tâche interrompue après avoir restauré l'état de la machine qu'il avait enregistré au moment de l'interruption.

Le sous-programme de traitement a donc 4 tâches à exécuter :

- Sauvegarder l'état de la machine en empilant les registres susceptibles d'être modifiés dans le sous-programme de traitement d'interruption (ISR).
- Interroger le registre d'état du périphérique pour savoir quelle opération effectuer.
- Éliminer la source de l'interruption en effectuant l'opération d'entrée-sortie.
- Restaurer l'état de la machine et retourner à la tâche interrompue en dépilant les registres empilés.

Comme Windows NT, que nous utilisons dans notre laboratoire, est un système d'exploitation pleinement protégé, les applications en mode usager ne peuvent accéder au matériel directement et doivent passer par un pilote de périphérique fonctionnant en mode noyau. C'est la couche HAL (Hardware Abstraction Layer) de NT qui gère les interruptions. La réalisation de tels pilotes de périphériques dépasse toutefois le cadre de ce cours.

Les VDD (Virtual Device Drivers) émulent les applications de 16 bits de MS-DOS. Ils piègent ce que l'application MS-DOS croit être des références à des ports d'entrée-sortie et les traduisent en fonctions natives d'entrée-sortie Win32. L'application intitulée «Invite de commande» ou, en anglais, «DOS prompt», est un tel VDD. On peut y exécuter la plupart des programmes écrits pour MS-DOS, y compris certains qui utilisent des interruptions matérielles. C'est le cas pour les exemples qui suivent que vous pouvez compiler avec un assembleur tel que A86, Masm32 ou TASM.

### Exemples d'interruptions matérielles sous MS-DOS :

Lecture de l'horloge système.

L'horloge système interrompt le processeur 18,2 fois par seconde si l'indicateur d'interruption IF est 1 (interruption matérielle 08). Chaque fois, le compteur de 32 bits situé aux adresses 0040:006C et 0040:006E est incrémenté de 1. Pour lire ce compteur, on peut utiliser l'interruption logicielle **int 0x1A** (26).

```
mov    ah, 0           ; mode lecture de l'horloge
int     26              ; appel à l'interruption d'horloge de MS-DOS
```

Après l'instruction **int 26**,

```
cx = TIMER_HIGH = mot de poids fort du compteur
dx = TIMER_LOW  = mot de poids faible du compteur
Si al = 0, on n'a pas dépassé 24h depuis la dernière lecture
Si al > 0, on a dépassé 24h. depuis la dernière lecture.
```

En écriture, c'est l'inverse, i.e. cx est copié dans TIMER\_HIGH et dx est copié dans TIMER\_LOW.

Pour mesurer le temps d'exécution d'un programme appelé *monProg*, on pourrait utiliser le programme qui suit :

### Assembleur



```

sti                      ; autoriser interruptions matérielles
mov  cx, 0              ; on va mettre l'horloge à 0
mov  dx, 0
mov  ah, 1              ; mode écriture de l'horloge
int  26                ; appel à l'interruption d'horloge de MS-DOS
call monProg
mov  ah, 0              ; lire l'horloge
int  26

```

La résolution de ce compteur est plutôt faible (54,94 ms par tic) pour des processeurs rapides comme, par exemple, le Pentium III à plus de 500 MHz. Il faudra donc que *monProg* soit passablement long pour arriver à mesurer son temps d'exécution avec ce temporisateur. Pour une meilleure résolution, utilisez plutôt l'instruction **rdtsc**.

Programme résident et déviation de l'interruption d'horloge temps réel

Voici un exemple d'un programme résident qui utilise l'horloge temps réel et émet un bip chaque 10 sec. La procédure de traitement de l'interruption 8 est déviée pour exécuter notre sous-programme, puis continue avec l'ancienne procédure. Cette technique s'appelle un «hook».<sup>1</sup>

```

start      segment
           org      100h
main:      jmp      short install

oldint8    dd        ?                ; espace pour ancien vecteur
counter    dw        182              ; 182 * 54,94 ms = 10 sec.

newint8    proc
           dec       cs:counter        ; décrémenter compteur
           jnz       done              ; on ne fait rien tant que count > 0
           mov       cs:counter, 182   ; réinitialiser compteur

           mov       ah, 0Eh           ; émettre un bip
           mov       al, 7
           int       10h
newint8    endp

```

<sup>1</sup> En programmation avec un assembleur standard, on utilise le suffixe h ou H pour représenter un nombre hexadécimal.

```
done:    jmp      cs:oldint8      ; continuer dans l'ancien vecteur
newint8  endp
```

; la partie suivante ne sera exécutée qu'une fois, lors de l'installation

```
install  proc      near

          mov      ah, 35h        ; charger CS:IP pour interruption 08
          mov      al, 08h        ; IP va dans BX et CS dans ES
          int      21h

          mov      word ptr oldint8, bx ; enregistrer dans espace prévu
          mov      word ptr oldint8+2, es

          mov      dx, offset newint8 ; ou lea dx, newint8
          mov      ah, 25h        ; définir CS:IP pour nouvelle
          mov      al, 08h        ; interruption 8
          int      21h            ; DX = IP, DS = CS par défaut

          mov      dx, (offset install - offset start) ; taille du programme résident
          add      dx, 15          ; arrondir à multiple de 16
          shr      dx, 4           ; division par 16
          mov      ah, 31h        ; rendre résident
          int      21h

install  endp
start    ends
end      main
```

## 12. Directives d'assemblage et macros

Les assembleurs comportent un bon nombre de directives ou pseudo-instructions pour contrôler le processus d'assemblage. Ils comportent aussi des directives d'assemblage conditionnel, et souvent des directives pour créer des structures de contrôle **if**, **while**, etc. automatiquement, ainsi que des commandes permettant de créer des macros.

### 12.1 Directives d'assemblage

Ces directives ne génèrent pas de code comme tel mais contrôlent le processus d'assemblage. Les crochets [[]] indiquent un paramètre optionnel.

#### **ALIGN** [[nombre]]

Aligne la variable ou l'instruction qui suit sur une adresse qui est un multiple de *nombre*. Ceci est particulièrement important avec le Pentium, dont la performance est beaucoup supérieure si les structures de données sont alignées sur des frontières multiples de 8 octets, parce qu'il lit la mémoire 64 bits à la fois.

#### **EVEN**

Aligne la variable ou l'instruction qui sur une adresse paire. Voir **ALIGN**.

#### *nom* **LABEL** *type*

#### *nom* **LABEL** [[**NEAR** | **FAR** | **PROC**]] **PTR** [*type*]]

Crée une nouvelle étiquette en attribuant à *nom* le type précisé et la valeur courante du compteur de position (distance en octets à partir du début du code).

#### **ORG** *expression*

Donne au compteur de position la valeur donnée par *expression*.

**.186**

Active l'assemblage du jeu d'instructions du 80186, et désactive l'assemblage des instructions introduites dans les processeurs subséquents de la famille 80x86. Active aussi les instructions du 8087.

**.286, .386 .486 et .586**

Active l'assemblage du jeu d'instructions non privilégiées du 80286, du 80386, du 80486 ou du Pentium respectivement, et désactive l'assemblage des instructions introduites dans les processeurs subséquents. Active aussi les instructions du 80287 ou du 80387 respectivement dans le cas de .286 et de .386. Le 80486 et le Pentium comportent une unité de virgule flottante intégrée.

**.286P, .386P, .486P et .586P**

Active l'assemblage de toutes les instructions (y compris les instructions privilégiées) du 80286, du 80386, du 80486 ou du Pentium respectivement, et désactive l'assemblage des instructions introduites dans les processeurs subséquents. Active aussi les instructions du 80287 ou du 80387 respectivement dans le cas de .286 et .386. Le 486 et le Pentium comportent une unité de virgule flottante intégrée.

**.287 et .387**

Active l'assemblage des instructions du coprocesseur 80287 ou du 80387 respectivement; désactive les instructions des coprocesseurs subséquents.

**.8086**

Active l'assemblage des instructions du 8086 (et du 8088) et désactive l'assemblage des instructions introduites dans les processeurs subséquents. Active aussi les instructions du 8087.

**.8087**

Active l'assemblage des instructions du coprocesseur 8087 et désactive les instructions des coprocesseurs subséquents.

**.NO87**

Désactive l'assemblage des instructions de virgule flottante.

**.MODEL modèlemémoire [[,typelangage]] [[,optionpile]]**

Initialise le modèle mémoire du programme. *modèlemémoire* peut être TINY, SMALL, COMPACT, MEDIUM, LARGE, HUGE, ou FLAT. *typelangage* peut être C, BASIC, FORTRAN, PASCAL, SYSCALL, ou STDCALL. *optionpile* peut être NEARSTACK ou FARSTACK.

**.CODE [[nom]]**

Utilisé avec .MODEL, il indique le début d'un segment de code appelé *nom* (le nom de segment par défaut est \_TEXT pour les modèles tiny, small, compact et flat, et module\_TEXT pour les autres).

**.DATA**

Utilisé avec .MODEL, il indique le début d'un segment de données *near* pour des données initialisées dont le nom sera \_DATA.

**.DATA?**

Utilisé avec .MODEL, il indique le début d'un segment de données *near* pour des données non initialisées dont le nom sera \_BSS.

**INCLUDE nomfichier**

Insère le code source du fichier donné par *nomfichier* dans le fichier source courant durant l'assemblage.

**DB**

Utilisé pour créer des données de 8 bits (bytes).

**DD**

Utilisé pour définir des données de 32 bits (dwords).

**DW**

Utilisé pour définir des données de 16 bits (words).

**DQ**

Utilisé pour définir des données de 64 bits (qwords).

## DUP

Permet de dupliquer une valeur lors de la définition de variables au moyen de db, dw, dd ou dq. Par exemple, dd 10 dup(0) crée dix mots doubles initialisés à 0, et dw 10 dup(?) crée 10 mots non initialisés.

## nom EQU expression

Attribue la valeur numérique de *expression* à *nom*. Le nom ne peut pas être redéfini plus tard. C'est l'équivalent de **#define** en C. On peut également utiliser EQU pour définir une chaîne de caractères : nom EQU "texte".

## END

Cette directive indique la fin d'un programme.

### [[nom]] BYTE [[initialiseur]] [[,initialiseur]] ...

Alloue et initialise si désiré un octet en mémoire pour chaque initialiseur.

### [[nom]] WORD [[initialiseur]] [[,initialiseur]] ...

Alloue et initialise si désiré un mot en mémoire pour chaque initialiseur.

### [[nom]] DWORD [[initialiseur]] [[,initialiseur]] ...

Alloue et initialise si désiré un double mot en mémoire pour chaque initialiseur.

### [[nom]] QWORD [[initialiseur]] [[,initialiseur]] ...

Alloue et initialise si désiré un quadruple mot en mémoire pour chaque initialiseur.

## 12.2 Directives d'assemblage conditionnel

L'assemblage conditionnel permet de compiler un programme de façon différente selon certaines conditions. Ceci permet de créer des versions différentes d'un même logiciel pour s'adapter à différentes plates-formes ou à différentes circonstances. Par exemple, le processeur utilisé a-t-il un coprocesseur arithmétique ? Si oui, on utilisera les fonctions du programme contenant les instructions de virgule flottante. Sinon, on utilisera une autre

version de ces fonctions qui ne fait pas appel à ces instructions. Il en irait de même pour la programmation MMX, par exemple.

La structure d'un tel programme a alors l'allure suivante :

```

IF      condition
    Code et directives
ELSE
    Code et directives
ENDIF

```

ou encore :

```

IF      condition1
    Code et directives
ELSE
    IF condition2
        Code et directives
    ELSE
        Code et directives
    ENDIF
ENDIF

```

Les conditions peuvent être passées comme paramètre sur la ligne de commande de l'assembleur. On peut également définir une variable et lui donner la valeur désirée, 1 pour TRUE et 0 pour FALSE.

```

IF expression1
    code1
ELSEIF expression2
    code2
ELSE
    code3
ENDIF

```

L'assembleur assemblera *code1* si *expression1* retourne TRUE, sinon, il assemblera *code2* si *expression2* retourne TRUE, sinon il assemblera *code3*.

ELSEIF peut être remplacé par ELSEIFB, ELSEIFDEF, ELSEIFDIF, ELSEIFE, ELSEIFIDN, ELSEIFIDNI, ELSEIFNB, SLESIFNDEF.

**IFDEF/IFNDEF *nom***

Assemble le code qui suit si une variable *nom* est définie (IFDEF) ou n'est pas définie (IFNDEF).

**IFIDN/IFDIF *texte1*, *texte2***

Assemble le code qui suit si *texte1* est identique (IFIDN) à *texte2* ou différent (IFDIF) de *texte2*

**IFE *expression***

Assemble le code qui suit si *expression* est fausse (0).

**IFB/IFINB *texte***

Assemble le code qui suit si *texte* est vide (IFB) ou n'est pas vide (IFNB).

## 12.3 Macros

Une macro-instruction ou macro est une séquence d'instructions et de directives que l'assembleur traite quand il rencontre le nom de la macro dans un programme. Il crée alors habituellement du nouveau code source. Cette facilité peut être utilisée comme une sorte de sténographie pour le programmeur. On remplace du code fréquemment utilisé par une macro. Ceci rend le code source plus rapide à écrire et plus compact. Le code objet n'est cependant pas plus court que si l'on n'utilisait pas de macros, contrairement à un sous-programme utilisé plusieurs fois dans un programme.

***nom* MACRO [[*paramètre* [[.REQ | :=default | :VARARG]] ]]**...

Indique le début de la définition d'une macro appelée *nom*.

**ENDM [[*valeur*]]**

Indique la fin de la définition d'une macro. Retourne *valeur* à l'énoncé appelant.

**EXITM**

Branche à l'énoncé qui suit la macro.



**LOCAL nom**

Utilisée pour créer une étiquette *nom* à l'intérieur d'une macro.

**Exemple de macro :**

Macro pour imprimer une chaîne de caractères à l'écran. On peut prendre le fragment de code utilisé dans la section précédente et en faire une macro :

```
affiche macro string
    mov     ah, 09
    lea     dx, string
    int     33
endm
```

On appelle cette macro comme suit :

```
affiche string
```

Avantages des macros : le code source est plus court et favorise la réutilisation de fragments de code débogués.

Inconvénients des macros : à moins de lire la documentation de chaque macro, il y a danger d'effets secondaires possibles, par exemple, la modification imprévue de registres ou de variables. Une surutilisation des macros peut également rendre le code difficile à lire.

## 12.4 Directives pour structures de contrôle et programmation structurée

Ces directives génèrent automatiquement du code au moyen de commandes ressemblant à celles des langages de haut niveau.

**INVOKE expression [[,arguments]]**

Appelle un sous-programme à l'adresse donnée par expression, en passant les paramètres sur la pile ou dans des registres, selon les conventions d'appel standard du type de langage (voir .MODEL).

**étiquette PROC** **[[distance]]** **[[langtype]]** **[[visibilité]]** **[[USES reglist]]** **[[,paramètre]]** ...

Marque le début d'une procédure appelée *étiquette*. La procédure peut être appelée au moyen de l'instruction CALL ou de la directive INVOKE.

**ENDP**

Indique la fin d'une procédure.

**étiquette PROTO** **[[distance]]** **[[langtype]]** **[[, paramètre]]** ...

Définit un prototype pour une fonction.

**.IF condition1**

**code1**

**.ELSEIF condition2**

**code2**

**.ELSE**

**code3**

**.ENDIF**

Génère le code qui teste *condition1* et qui exécute *code1* si elle est vraie. Sinon, il générera le code pour tester *condition2*. Si cette condition est vraie, il *code2* sera exécuté, sinon, *code3*.

Exemple :

```
.if    eax == 3
    mov    ebx, 0
.else
    mov    ebx, 1
```

Il faut prendre garde que dans les comparaisons > et <, il s'agit de comparaisons non signées. Si on veut des comparaisons signées, il faut utiliser *sbyte ptr*, *sword ptr* ou *sdword ptr* pour qualifier les opérandes. Ainsi, pour un test signé, on écrira :

```
.if    sdword ptr eax > 1
    ...
```

**.WHILE condition**

**code**

**.ENDW**

Génère le code pour tester *condition* et pour exécuter *code* tant que *condition* est vraie.

**.BREAK [[IF condition]]**

Génère le code pour quitter une boucle **while** ou **repeat** si *condition* est vraie.

**.CONTINUE [[IF condition]]**

Génère le code pour retourner au début d'une boucle **while** ou **repeat** si *condition* est vraie.

**.REPEAT**

**code**

**.UNTIL condition**

Génère le code pour répéter *code* tant que *condition* est fausse. On peut remplacer .UNTIL par .UNTILCXZ qui deviendra vraie quand CX atteindra 0. Dans ce cas, la condition est optionnelle.

**nom STRUCT [[alignement]] [[,NONUNIQUE]]**

déclaration de champs

nom **ENDS**

Déclare une structure consistant des champs indiqués. Chaque champ doit être d'une définition valide de donnée.

## La librairie MASM32

### **Mode console**

StdIn : lecture d'une chaîne de caractères

StdOut : écriture d'une chaîne de caractères

ArgCIC : extraction des arguments de ligne de commande

ClearScreen : effacer l'écran

locate : positionner le curseur dans l'écran

### **Mode Windows**

circle : tracé d'un cercle

ArgCl : extraction des arguments de ligne de commande

Frame3D : tracé d'un cadre 3D

FrameGrp : tracé d'un cadre autour d'un groupe

FrameWindow : tracé d'un cadre de fenêtre

FrameCtrl : tracé d'un cadre autour d'un contrôle

line : tracé d'une droite

### **Conversions**

ucase, lcase : conversion de minuscule à majuscule et inversement.

atodw : conversion d'une chaîne de caractères en binaire

dwtoa : conversion d'un nombre binaire en chaîne décimale

StrToFloat : conversion d'une chaîne de caractères en nombre de virgule flottante

FloatToStr : conversion d'un nombre de virgule flottante en chaîne de caractères

htodw : conversion d'une chaîne hexadécimale en binaire

## **Chaînes de caractères**

InString : recherche d'une chaîne dans une autre

Instr : calcule la longueur d'une chaîne de caractères

Istr : copie d'une chaîne de caractères

Itrim : enlève les espaces (sp, tab) au début d'une chaîne

midstr : extrait une sous-chaîne de caractères d'une chaîne

revstr : copie une chaîne source inversée vers une chaîne destination

rstr : copie les n derniers caractères d'une chaîne vers une chaîne destination

rtrim : enlève les espaces (sp, tab) à la fin d'une chaîne

StripRangeI : enlève les caractères d'une chaîne situés entre deux caractères donnés, incluant les caractères en question.

StripRangeX : enlève les caractères d'une chaîne situés entre deux caractères donnés

strlen : calcule la longueur d'une longue chaîne de caractères

## **Utilitaires**

GetAppPath : retourne le chemin d'une application

exist : trouve le premier fichier à porter le nom spécifié

filesize : retourne la taille d'un fichier

GetCl : retourne la ligne de commande

MemCopy : copie un bloc de données d'un endroit à l'autre.

