INSTRUCTIONS DU LANGAGE MACHINE

Remarques préliminaires :

- **Le principe du langage assembleur est de remplacer chaque opcode hexadécimal par un mot facile à retenir. Ce mot est appelé mnémonique. Par exemple, "INT" est le mnémonique associé à l'opcode CDh. Chaque fois que le compilateur rencontrera ce mot, il le remplacera par l'octet CDh et écrira ensuite l'opérande (ici : le numéro de l'interruption) en hexadécimal.**

- **Cette liste récapitule les instructions que nous connaissons déjà et en présente de nouvelles. Elle n'est pas exhaustive mais vous sera amplement suffisante pour la plupart de vos programmes.**

- **Certaines instructions, comme PUSHA, ne sont disponibles que pour des modèles de processeurs plus évolués que le 8086, par exemple le 286. N'oubliez pas la directive .386 si vous les utilisez.**

1. L'instruction NOP (« *No Operation* »)

Syntaxe : NOP

Description : Ne fait rien ! Mais alors RIEN ! Que dalle ! Niet !

2. L'instruction MOV (« *Move* »)

Syntaxe : MOV *Destination*, *Source*

Description : Copie le contenu de *Source* dans *Destination*.

Mouvements autorisés : MOV Registre général, Registre quelconque

MOV *Mémoire*, Registre quelconque

MOV Registre général, *Mémoire*

MOV Registre général, *Constante*

MOV *Mémoire*, *Constante*

MOV Registre de segment, Registre général

Remarques : *Source* et *Destination* doivent avoir la même taille. On ne peut charger dans un registre de segment que le contenu d'un registre général (SI, DI et BP sont considérés ici comme des registres généraux).

Exemples : MOV AX, *5*

MOV ES, DX

MOV AL, [*Variable1*] *;Copie un octet car AL contient 8 bits*

MOV [*Variable2*], DS *;Copie un word car DS contient 16 bits*

MOV word ptr [*Variable3*], *12* *;Ici, on spécifie que la variable est un word*

3. L'instruction XCHG (« *Exchange* »)

Syntaxe : XCHG *Destination*, *Source*

Description : Echange les contenus de *Source* et de *Destination*.

Mouvements autorisés : XCHG Registre général, Registre général

XCHG Registre général, *Mémoire*

XCHG *Mémoire*, Registre général

4. L'instruction JMP (« *Jump* »)

Syntaxe : JMP *MonLabel*

Description : Saute à l'instruction pointée par *MonLabel*.

5. L'opérateur CMP (« *Compare* »)

Syntaxe : CMP *Destination*, *Source*

Description : Cet opérateur sert à comparer deux nombres : *Source* et *Destination*. *C'est le registre des indicateurs qui contient les résultats de la comparaison.* Ni *Source* ni *Destination* ne sont modifiés.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Remarque : Cet opérateur effectue en fait une soustraction mais contrairement à SUB, le résultat n'est pas sauvegardé.

Le programme doit pouvoir réagir en fonction des résultats de la comparaison. Pour cela, on utilise les *sauts conditionnels* (voir ci-dessous).

6. Les instructions de saut conditionnel

Les sauts conditionnels sont terriblement importants car ils permettent au programme de faire des choix en fonction des données.

Un saut conditionnel n'est effectué qu'à certaines conditions portant sur les flags (par exemple : CF = 1 ou ZF = 0).

Certains mnémoniques de sauts conditionnels sont totalement équivalents, c'est-à-dire qu'ils représentent le même opcode hexadécimal. C'est pour aider le programmeur qu'ils existent parfois sous plusieurs formes.

a) les sauts de comparaison

- **JE** (« *Jump if Equal* ») fait un saut au label spécifié si et seulement si ZF = 1. Rappelez-vous que ce flag est à 1 si et seulement si le résultat de l'opération précédente vaut zéro. Comme CMP réalise une soustraction, on utilise généralement JE pour savoir si deux nombres sont égaux.

Exemple :

```
;(...)

cmp ah, bh ;comparaison de AH et BH (soustraction)
je egal ;saut ssi AH = BH (le processeur regarde ZF pour le savoir)

different :

;(...)

egal :

;(...)
```

Mnémonique équivalent : **JZ** (« *Jump if Zero* »)

- **JG** (« *Jump if Greater* ») fait un saut au label spécifié si et seulement si ZF = 0 et SF = OF. On l'utilise en arithmétique *signée* pour savoir si un nombre est supérieur à un autre.

Exemple :

```
;(...)

cmp ah, bh ;comparaison
jg AH_superieur_BH ;saut ssi AH > BH

AH_inferieur_ou_egal_BH :

;(...)

AH_superieur_BH :

;(...)
```

Mnémonique équivalent : **JNLE** (« *Jump if Not Less Or Equal* »)

- **JGE** (« *Jump if Greater or Equal* ») fait un saut au label spécifié si et seulement si SF = OF. On l'utilise en arithmétique *signée* pour savoir si un nombre est supérieur ou égal à un autre.

Mnémonique équivalent : **JNL** (« *Jump if Not Less* »)

- **JL** (« *Jump if Less* ») fait un saut au label spécifié si et seulement si SF <> OF. On l'utilise en arithmétique *signée* pour savoir si un nombre est inférieur à un autre.

Mnémonique équivalent : **JNGE** (« *Jump if Not Greater Or Equal* »)

- **JLE** (« *Jump if Less Or Equal* ») fait un saut au label spécifié si et seulement si SF <> OF ou ZF = 1. On l'utilise en arithmétique *signée* pour savoir si un nombre est inférieur ou égal à un autre.

Mnémonique équivalent : **JNG** (« *Jump if Not Greater* »)

- **JA** (« *Jump if Above* ») fait un saut au label spécifié si et seulement si ZF = 0 et CF = 0. On l'utilise en arithmétique *non signée* pour savoir si un nombre est supérieur à un autre.

Mnémonique équivalent : **JNBE** (« *Jump if Not Below Or Equal* »)

- **JAE** (« *Jump if Above or Equal* ») fait un saut au label spécifié si et seulement si CF = 0. On l'utilise en arithmétique *non signée* pour savoir si un nombre est supérieur ou égal à un autre.

Mnémonique équivalent : **JNB** (« *Jump if Not Below* »)

- **JB** (« *Jump if Below* ») fait un saut au label spécifié si et seulement si CF = 1. On l'utilise en arithmétique *non signée* pour savoir si un nombre est inférieur à un autre.

Mnémonique équivalent : **JNAE** (« *Jump if Not Above Or Equal* »)

- **JBE** (« *Jump if Below or Equal* ») fait un saut au label spécifié si et seulement si CF = 1 ou ZF = 1. On l'utilise en arithmétique *non signée* pour savoir si un nombre est inférieur ou égal à un autre.

Mnémonique équivalent : **JNA** (« *Jump if Not Above* »)

b) les sauts de test sur les flags

Ces instructions testent un flag unique et exécutent ou non le saut selon la valeur de ce flag.

- **JC** (« *Jump if Carry* ») fait un saut au label spécifié si et seulement si CF = 1.

Remarques : Ce mnémonique correspond au même opcode que JB. Il est souvent employé pour vérifier que l'appel d'une interruption n'a pas déclenché d'erreur.

Exemple :

```
;(...)

mov ah, 3eh
int 21h

jc short ErreurSurvenue

; sinon : pas d'erreur...
;(...)
;(...)

ErreurSurvenue :

;(...)
```

Si l'appel de la fonction 3Eh de l'interruption 21H se solde par une erreur, alors la CF vaudra 1 et le saut sera accompli. Dans le cas opposé, l'exécution continuera normalement de manière linéaire.

- **JNC** (« *Jump if not Carry* ») fait un saut au label spécifié si et seulement si CF = 0.
- **JZ** (« *Jump if Zero* ») fait un saut au label spécifié si et seulement si ZF = 1. Ce mnémonique correspond au même opcode que JE.
- **JNZ** (« *Jump if not Zero* ») fait un saut au label spécifié si et seulement si ZF = 0. Ce mnémonique correspond au même opcode que JNE.
- **JS** (« *Jump if Sign* ») fait un saut au label spécifié si et seulement si SF = 1.
- **JNS** (« *Jump if not Sign* ») fait un saut au label spécifié si et seulement si SF = 0.
- **JO** (« *Jump if Overflow* ») fait un saut au label spécifié si et seulement si OF = 1.
- **JNO** (« *Jump if not Overflow* ») fait un saut au label spécifié si et seulement si OF = 0.
- **JP** (« *Jump if Parity* ») fait un saut au label spécifié si et seulement si PF = 1.
- **JNP** (« *Jump if not Parity* ») fait un saut au label spécifié si et seulement si PF = 0.

c) le saut de test sur le registre CX

JCXZ (« *Jump if CX = Zero* ») fait un saut au label spécifié si et seulement si CX = 0.

Les instructions arithmétiques

a) l'instruction INC (« *Increment* »)

Syntaxe : INC *Destination*

Description : Incrémente *Destination*.

Indicateurs affectés : AF, OF, PF, SF, ZF

Exemple : INC CL

b) l'instruction ADD (« *Addition* »)

Syntaxe : ADD *Destination*, *Source*

Description : Ajoute *Source* à *Destination*

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Exemple : ADD byte ptr [*VARIABLE* + DI], *5*

c) l'instruction ADC (« *Add with Carry* »)

Syntaxe : ADC *Destination*, *Source*

Description : Ajoute (*Source* + CF) à *Destination*.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

d) l'instruction DEC (« *Decrement* »)

Syntaxe : DEC *Destination*

Description : Décrémente *Destination*.

Indicateurs affectés : AF, OF, PF, SF, ZF

e) l'instruction SUB (« *Substract* »)

Syntaxe : SUB *Destination*, *Source*

Description : Soustrait *Source* à *Destination*.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

f) l'instruction SBB (« *Substract with Borrow* »)

Syntaxe : SBB *Destination*, *Source*

Description : Soustrait (*Source* + CF) à *Destination*.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

g) l'instruction MUL (« *Multiply* »)

Syntaxe : MUL *Source*

Description : Effectue une multiplication d'entiers *non signés*.

- Si *Source* est un octet : AL est multiplié par *Source* et le résultat est placé dans AX.
- Si *Source* est un mot : AX est multiplié par *Source* et le résultat est placé dans DX:AX.
- Si *Source* est un double mot : EAX est multiplié par *Source* et le résultat est placé dans EDX:EAX.

Indicateurs affectés : CF, OF

Remarque : *Source ne peut être une valeur immédiate.*

Exemples : MUL CX

MUL byte ptr [*TOTO*]

h) l'instruction IMUL (« *Integer Multiply* »)

Syntaxe : IMUL *Source*

IMUL *Destination*, *Source*

IMUL *Destination*, *Source*, *Valeur*

Description : Effectue une multiplication d'entiers *signés*.

IMUL *Source* :

- Si *Source* est un octet : AL est multiplié par *Source* et le résultat est placé dans AX.
- Si *Source* est un mot : AX est multiplié par *Source* et le résultat est placé dans DX:AX.
- Si *Source* est un double mot : EAX est multiplié par *Source* et le résultat est placé dans EDX:EAX.

IMUL *Destination*, *Source* : Multiplie *Destination* par *Source* et place le résultat dans *Destination*.

IMUL *Destination*, *Source*, *Valeur* : Multiplie *Source* par *Valeur* et place le résultat dans *Destination*.

Indicateurs affectés : CF, OF

i) l'instruction DIV (« *Divide* »)

Syntaxe : DIV *Source*

Description : Effectue une division euclidienne d'entiers *non signés*.

- Si *Source* est un octet : AX est divisé par *Source*, le quotient est placé dans AL et le reste dans AH.
- Si *Source* est un mot : DX:AX est divisé par *Source*, le quotient est placé dans AX et le reste dans DX.
- Si *Source* est un double mot : EDX:EAX est divisé par *Source*, le quotient est placé dans EAX et le reste dans EDX.

Indicateurs affectés : AF, OF, PF, SF, ZF

Remarque : *Source ne peut être une valeur immédiate.*

j) l'instruction IDIV (« *Integer Divide* »)

Syntaxe : IDIV *Source*

Description : Effectue une division euclidienne d'entiers *signés*.

- Si *Source* est un octet : AX est divisé par *Source*, le quotient est placé dans AL et le reste dans AH.
- Si *Source* est un mot : DX:AX est divisé par *Source*, le quotient est placé dans AX et le reste dans DX.
- Si *Source* est un double mot : EDX:EAX est divisé par *Source*, le quotient est placé dans EAX et le reste dans EDX.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

k) l'instruction NEG (« *Negation* »)

Syntaxe : NEG *Destination*

Description : Forme le complément à 2 de *Destination*, i.e. prend l'opposé de *Destination*.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Remarque : A ne pas confondre avec NOT, qui forme le complément à 1.

8. Les instructions logiques

a) l'instruction NOT (« *Logical NOT* »)

Syntaxe : NOT *Destination*

Description : Effectue un NON logique bit à bit sur *Destination* (i.e. chaque bit de *Destination* est inversé).

b) l'instruction OR (« *Logical OR* »)

Syntaxe : OR *Destination*, *Source*

Description : Effectue un OU logique inclusif bit à bit entre *Destination* et *Source*. Le résultat est stocké dans *Destination*.

Indicateurs affectés : CF, OF, PF, SF, ZF

Remarque : Afin d'optimiser la taille et les performances du programme, on peut utiliser l'instruction "OR AX, AX" à la place de "CMP AX, 0".
En effet, un OU bit à bit entre deux nombres identiques ne modifie pas *Destination* et est exécuté « infiniment » plus rapidement qu'une soustraction. Comme les flags sont affectés, les sauts conditionnels sont possibles.

c) l'instruction AND (« *Logical AND* »)

Syntaxe : AND *Destination*, *Source*

Description : Effectue un ET logique bit à bit entre *Destination* et *Source*. Le résultat est stocké dans *Destination*.

Indicateurs affectés : CF, OF, PF, SF, ZF

d) l'instruction TEST (« *Test for bit pattern* »)

Syntaxe : TEST *Destination*, *Source*

Description : Effectue un ET logique bit à bit entre *Destination* et *Source*. Le résultat n'est pas conservé, donc *Destination* n'est pas modifié. Seuls les flags sont affectés.

Cet opérateur est souvent utilisé pour tester certains bits de *Destination*.

Indicateurs affectés : CF, OF, PF, SF, ZF

Exemple :

```
;(...)

test ax, 00000010b
jnz bit2_vaut_1 ;saut ssi le 2e bit = 1

; sinon : le 2e bit de AX vaut 0...
;(...)

bit2_vaut_1:

;(...)
```

e) l'instruction XOR (« *Exclusive logical OR* »)

Syntaxe : XOR *Destination*, *Source*

Description : Effectue un OU logique exclusif bit à bit entre *Destination* et *Source*. Le résultat est stocké dans *Destination*.

Indicateurs affectés : CF, OF, PF, SF, ZF

Remarque : Pour remettre un registre à zéro, il est préférable de faire "XOR AX, AX" que "MOV AX, 0". En effet, le résultat est le même mais la taille et surtout la vitesse d'exécution de l'instruction sont très largement optimisées.

f) l'instruction SHL (« *Shift logical Left* »)

Syntaxe : SHL *Destination*, *Source*

Description : Décale les bits de *Destination* de *Source* positions vers la gauche. Les bits les plus à droite sont remplacés par des zéros.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Exemple : SHL AX, *4 ;permet de multiplier par 16 de façon infiniment plus rapide que MUL*

Mnémonique équivalent : SAL (« *Shift Arithmetical Left* »)

g) l'instruction SHR (« *Shift logical Right* »)

Syntaxe : SHR *Destination*, *Source*

Description : Décale les bits de *Destination* de *Source* positions vers la droite. Les bits les plus à gauche sont remplacés par des zéros.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Mnémonique équivalent : SAR (« *Shift Arithmetical Right* »)

h) l'instruction ROL (« *Rotate Left* »)

Syntaxe : ROL *Destination*, *Source*

Description : Effectue une rotation des bits de *Destination* de *Source* positions vers la gauche. Le dernier bit à être sorti à gauche et à être rentré à droite est placé dans CF. OF est mis à 1 si et seulement si le signe de *Destination* a changé.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

i) l'instruction ROR (« *Rotate Right* »)

Syntaxe : ROR *Destination*, *Source*

Description : Effectue une rotation des bits de *Destination* de *Source* positions vers la droite. Le dernier bit à être sorti à droite et à être rentré à gauche est placé dans CF. OF est mis à 1 si et seulement si le signe de *Destination* a changé.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

j) l'instruction RCL (« *Rotate through Carry Left* »)

Syntaxe : RCL *Destination*, *Source*

Description : Effectue une rotation des bits de *Destination* de *Source* positions vers la gauche. CF est utilisé comme intermédiaire : chaque bit qui sort à gauche est placé dans CF, et le contenu de CF est ensuite réinséré à droite. OF est mis à 1 si et seulement si le signe de *Destination* a changé.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

k) l'instruction RCR (« *Rotate through Carry Right* »)

Syntaxe : RCR *Destination*, *Source*

Description : Effectue une rotation des bits de *Destination* de *Source* positions vers la droite. CF est utilisé comme intermédiaire : chaque bit qui sort à droite est placé dans CF, et le contenu de CF est ensuite réinséré à gauche. OF est mis à 1 si et seulement si le signe de *Destination* a changé.

Indicateurs affectés : AF, CF, OF, PF, SF, ZF

Les instructions de manipulation des flags

a) l'instruction CLC (« *Clear Carry flag* »)

Syntaxe : CLC

Description : Met CF à 0.

Indicateurs affectés : CF

b) l'instruction STC (« *Set Carry flag* »)

Syntaxe : STC

Description : Met CF à 1.

Indicateurs affectés : CF

c) l'instruction CLD (« *Clear Direction flag* »)

Syntaxe : CLD

Description : Met DF à 0.

Indicateurs affectés : DF

d) l'instruction STD (« *Set Direction flag* »)

Syntaxe : STD

Description : Met DF à 1.

Indicateurs affectés : DF

e) l'instruction CLI (« *Clear Interrupt flag* »)

Syntaxe : CLI

Description : Met IF à 0.

Indicateurs affectés : IF

f) l'instruction STI (« *Set Interrupt flag* »)

Syntaxe : STI

Description : Met IF à 1.

Indicateurs affectés : IF

g) l'instruction CMC (« *Complement Carry flag* »)

Syntaxe : CMC

Description : Inverse CF.

Indicateurs affectés : CF

h) l'instruction LAHF (« *Load AH from Flags* »)

Syntaxe : LAHF

Description : Charge dans AH l'octet de poids faible du registre des indicateurs.

i) l'instruction SAHF (« *Store AH into Flags* »)

Syntaxe : SAHF

Description : Stocke les bits de AH dans le registre des indicateurs.

10. Les instructions de gestion de la pile

a) l'instruction PUSH (« *Push Word onto Stack* »)

Syntaxe : PUSH *Source*

Description : Empile le mot *Source*. SP est décrémenté de 2.

Remarques : *Source* ne peut être une valeur immédiate. Il est possible d'abréger votre code source en écrivant par exemple "PUSH AX BX BP". Le compilateur écrira alors trois fois l'instruction PUSH du langage machine. Il est possible également d'empiler des doubles mots.

b) l'instruction POP (« *Pop Word off Stack* »)

Syntaxe : POP *Destination*

Description : Dépile le mot qui se trouve au sommet de la pile et le place dans *Destination*. SP est incrémenté de 2.

c) l'instruction PUSHF (« *Push Flags onto Stack* »)

Syntaxe : PUSHF

Description : Empile le registre des indicateurs. SP est décrémenté de 2.

Remarque : PUSHFD empile le registre des indicateurs codé sur 32 bits.

d) l'instruction POPF (« *Pop Flags off Stack* »)

Syntaxe : POPF

Description : Dépile le mot qui se trouve au sommet de la pile et le place dans le registre des indicateurs. SP est incrémenté de 2.

Indicateurs affectés : Tous

Remarque : POPFD est utilisé pour un registre des indicateurs codé sur 32 bits.

e) l'instruction PUSHA (« *Push All registers onto Stack* »)

Syntaxe : PUSHA

Description : Empile AX, BX, CX, DX, BP, SI, DI et SP.

Remarque : PUSHAD est utilisé pour des registres de 32 bits.

f) l'instruction POPA (« *Pop All registers off Stack* »)

Syntaxe : POPA

Description : Restaure AX, BX, CX, DX, BP, SI, DI et SP à partir de la pile.

Remarque : POPAD est utilisé pour des registres de 32 bits.

11. Les instructions de gestion des chaînes d'octets

a) l'instruction MOVSB (« *Move String Byte* »)

Syntaxe : MOVSB

Description : Copie l'octet adressé par DS:SI à l'adresse ES:DI. Si DF = 0, alors DI et SI sont ensuite incrémentés, sinon ils sont décrémentés.

Remarque : Pour copier plusieurs octets, faire REP MOVSB (« *Repeat Move String Byte* »).
Le nombre d'octets à copier doit être transmis dans CX de même que pour un LOOP.

Exemple :

```
;(...)

mov ax, ds ;mettre ds
mov es, ax ; dans es

mov si, offset depart ;source

mov di, offset destination ;destination

mov cx, fin – depart ;nb d'octets

cld ;vers la droite

rep movsb ;copier !

;(...)
```

b) l'instruction SCASB (« *Scan String Byte* »)

Syntaxe : SCASB

Description : Compare l'octet adressé par ES:DI avec AL. Les résultats sont placés dans le registre des indicateurs. Si DF = 0, alors DI est ensuite incrémenté, sinon il est décrémenté.

Remarques : Pour comparer plusieurs octets, faire "REP SCASB" ou "REPE SCASB" (« *Repeat until Egal* »), ou encore "REPZ SCASB" (« *Repeat until Zero* »). Ces trois préfixes sont équivalents.
Le nombre d'octets à comparer doit être transmis dans CX. La boucle ainsi créée s'arrête si CX = 0 ou si le caractère pointé par ES:DI est le même que celui contenu dans AL (i.e. si ZF = 1). On peut ainsi rechercher un caractère dans une chaîne.

Pour répéter au contraire la comparaison *jusqu'à ce que* ZF = 0, c'est-à-dire jusqu'à ce que AL et le caractère adressé par ES:DI *diffèrent*, utiliser REPNE ou REPNZ.

Exemple :

```
;(...)

mov ax, ds ;mettre ds
mov es, ax ; dans es

mov di, offset chaine ;destination

mov cx, fin – chaine ;longueur

cld ;vers la droite

mov al, 0

rep scasb ;chercher 0

;ES:DI pointe maintenant vers l'octet qui suit le premier 0 trouvé (si un 0 a
été trouvé...)

;(...)
```

c) l'instruction LODSB (« *Load String Byte* »)

Syntaxe : LODSB

Description : Charge dans AL l'octet adressé par DS:SI. Si DF = 0, alors SI est ensuite incrémenté, sinon il est décrémenté.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour MOVSB.

d) l'instruction STOSB (« *Store String Byte* »)

Syntaxe : STOSB

Description : Stocke le contenu de AL dans l'octet adressé par ES:DI. Si DF = 0, alors DI est ensuite incrémenté, sinon il est décrémenté.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour LODSB.

e) l'instruction CMPSB (« *Compare String Byte* »)

Syntaxe : CMPSB

Description : Compare l'octet adressé par DS:SI et celui adressé par ES:DI. Si DF = 0, alors SI et DI sont ensuite incrémentés, sinon ils sont décrémentés.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour SCASB.

12. Les instructions de gestion des chaînes de mots

Ce sont les mêmes que les précédentes, hormis qu'elles traitent des mots et non des octets et que leur nom se termine par un 'W' au lieu du 'B'.

a) l'instruction MOVSW (« *Move String Word* »)

Syntaxe : MOVSW

Description : Copie le mot adressé par DS:SI à l'adresse ES:DI. Si DF = 0, alors DI et SI sont ensuite incrémentés de 2, sinon ils sont décrémentés de 2.

Remarque : Pour copier plusieurs mots, faire "REP MOVSW". Le nombre de mots à copier doit être transmis dans CX.

b) l'instruction SCASW (« *Scan String Word* »)

Syntaxe : SCASW

Description : Compare le mot adressé par ES:DI avec AX. Les résultats sont placés dans le registre des indicateurs. Si DF = 0, alors DI est ensuite incrémenté de 2, sinon il est décrémenté de 2.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour SCASB.

c) l'instruction LODSW (« *Load String Word* »)

Syntaxe : LODSW

Description : Charge dans AX le mot adressé par DS:SI. Si DF = 0, alors SI est ensuite incrémenté de 2, sinon il est décrémenté de 2.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour LODSB.

d) l'instruction STOSW (« *Store String Word* »)

Syntaxe : STOSW

Description : Stocke le contenu de AX dans le mot adressé par ES:DI. Si DF = 0, alors DI est ensuite incrémenté de 2, sinon il est décrémenté de 2.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour STOSB.

e) l'instruction CMPSW (« *Compare String Word* »)

Syntaxe : CMPSW

Description : Compare le mot adressé par DS:SI et celui adressé par ES:DI. Si DF = 0, alors SI et DI sont ensuite incrémentés de 2, sinon ils sont décrémentés de 2.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour CMPSB.

13. Les instructions de gestion des chaînes de doubles mots

Elles traitent des doubles mots et leur nom se termine par un 'D'.

a) l'instruction MOVSD (« *Move String Double Word* »)

Syntaxe : MOVSD

Description : Copie le double mot adressé par DS:SI à l'adresse ES:DI. Si DF = 0, alors DI et SI sont ensuite incrémentés de 4, sinon ils sont décrémentés de 4.

Remarque : Pour copier plusieurs doubles mots, faire "REP MOVSD". Le nombre de doubles mots à copier doit être transmis dans CX.

b) l'instruction SCASD (« *Scan String Double Word* »)

Syntaxe : SCASD

Description : Compare le double mot adressé par ES:DI avec EAX. Les résultats sont placés dans le registre des indicateurs. Si DF = 0, alors DI est ensuite incrémenté de 4, sinon il est décrémenté de 4.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour SCASB.

c) l'instruction LODSD (« *Load String Double Word* »)

Syntaxe : LODSD

Description : Charge dans EAX le double mot adressé par DS:SI. Si DF = 0, alors SI est ensuite incrémenté de 4, sinon il est décrémenté de 4.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour LODSB.

d) l'instruction STOSD (« *Store String Double Word* »)

Syntaxe : STOSD

Description : Stocke le contenu de EAX dans le double mot adressé par ES:DI. Si DF = 0, alors DI est ensuite incrémenté de 4, sinon il est décrémenté de 4.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour STOSB.

e) l'instruction CMPSD (« *Compare String Double Word* »)

Syntaxe : CMPSD

Description : Compare le double mot adressé par DS:SI et celui adressé par ES:DI. Si DF = 0, alors SI et DI sont ensuite incrémentés de 4, sinon ils sont décrémentés de 4.

Remarque : Possibilité d'utiliser les préfixes de répétition, de même que pour CMPSB.

14. Les instructions d'appel

a) l'instruction CALL (« *Procedure Call* »)

Syntaxe : CALL *MaProc*

Description : Appel de procédure. Si *MaProc* se trouve dans un segment extérieur, le processeur empile CS. Ensuite, dans tous les cas, il empile IP et fait un saut à l'étiquette *MaProc*.

b) l'instruction RET (ou RETN)

Syntaxe : RET

Description : Retour de procédure se trouvant à l'intérieur du segment (NEAR). Un mot est dépilé et placé dans IP. Le contrôle retourne donc à la procédure appelante.

c) l'instruction RETF

Syntaxe : RETF

Description : Retour de procédure se trouvant à l'extérieur du segment (FAR). Deux mots sont dépilés et placés dans CS:IP. Le contrôle retourne donc à la procédure appelante.

15. Les instructions de boucle

a) l'instruction LOOP

Syntaxe : LOOP *MonLabel*

Description : Décrémente CX, puis, si CX <> 0, fait un saut à *MonLabel*.

b) l'instruction LOOPE (« *Loop while Equal* »)

Syntaxe : LOOPE *MonLabel*

Description : Décrémente CX, puis, si CX <> 0 et ZF = 1, fait un saut à *MonLabel*.

Mnémonique équivalent : LOOPZ

c) l'instruction LOOPNE (« *Loop while not Equal* »)

Syntaxe : LOOPNE *MonLabel*

Description : Décrémente CX, puis, si CX <> 0 et ZF = 0, fait un saut à *MonLabel*.

16. Les instructions d'adressage

a) l'instruction LEA (« *Load effective address* »)

Syntaxe : LEA *Destination*, *Source*

Description : Charge l'offset de la source dans le registre *Destination*.

Exemple : LEA BP, word ptr [BP + *TOTO*]

b) l'instruction LDS (« *Load pointer using DS* »)

Syntaxe : LDS *Destination*, *Source*

Description : Transfère dans DS:*Destination* le contenu de la mémoire adressée par *Source*.

c) l'instruction LES (« *Load pointer using ES* »)

Syntaxe : LES *Destination*, *Source*

Description : Transfère dans ES:*Destination* le contenu de la mémoire adressée par *Source*.

17. Les instructions de conversion arithmétique

Remarque préliminaire : Nous n'expliciterons pas toutes ces instructions.

a) l'instruction AAA (« *ASCII Adjust for Addition* »)
b) l'instruction AAD (« *ASCII Adjust for Division* »)
c) l'instruction AAM (« *ASCII Adjust for Multiplication* »)
d) l'instruction AAS (« *ASCII Adjust for Substraction* »)
e) l'instruction CBW (« *Convert Byte to Word* »)

Syntaxe : CBW

Description : Convertit l'octet *signé* stocké dans AL en un mot (signé) stocké dans AX. Ainsi, si AL est négatif, AH sera rempli de 1 binaires, sinon, AH sera mis à 0.

f) l'instruction CWD (« *Convert Word to Double Word* »)

Syntaxe : CWD

Description : Convertit le mot *signé* stocké dans AX en un double mot (signé) stocké dans DX:AX. Ainsi, si AX est négatif, DX sera rempli de 1 binaires, sinon DX sera mis à 0.

g) l'instruction DAA (« *Decimal Adjust for Addition* »)
h) l'instruction DAS (« *Decimal Adjust for Substraction* »)
i) l'instruction MOVSX (« *Move with Sign Extend* »)

Syntaxe : MOVSX *Destination*, *Source*

Description : Déplace le contenu *signé* d'un registre de 8 bits dans un registre de 16 bits, ou bien déplace le contenu signé d'un registre de 16 bits dans un registre de 32 bits. Si *Source* est négatif, la partie haute de *Destination* sera remplie de 1 binaires, sinon elle sera remplie de 0.

j) l'instruction MOVZX (« *Move with Zero Extend* »)

Syntaxe : MOVZX *Destination*, *Source*

Description : Déplace le contenu *non signé* d'un registre de 8 bits dans un registre de 16 bits, ou bien déplace le contenu signé d'un registre de 16 bits dans un registre de 32 bits. La partie haute de *Destination* sera donc mise à 0.

18. Les instructions d'entrée-sortie

a) l'instruction IN (« *Input from port* »)

Syntaxe : IN *Destination*, *Port*

Description : Charge un octet ou un mot depuis un port d'entrée-sortie dans AL ou AX. *Port* peut être DX ou bien une constante de 8 bits.

b) l'instruction OUT (« *Output to port* »)

Syntaxe : OUT *Port*, *Source*

Description : Ecrit dans Port la valeur contenue dans Source. Source ne peut être que AL ou AX. Port est une constante ou bien DX.

LECTURE ET ECRITURE DE FICHIERS AVEC LES HANDLES

Sous DOS, il existe deux méthodes pour accéder aux fichiers et pour chacune d'elles un lot d'interruptions spécifiques. La première est la méthode des FCB (« *File contol block* »). Elle est rarement utilisée, aussi ne l'aborderons nous pas. Nous étudierons le principe de la méthode des « *handles* ».

Pour lire ou écrire des données dans un fichier, il est nécessaire de l'*ouvrir*, c'est-à-dire de le charger en mémoire. Quand toutes les opérations de lecture et d'écriture auront été effectuées, le fichier devra être *refermé* afin d'enregistrer les éventuelles dernières modifications et surtout de libérer la mémoire occupée.

1. Ouverture d'un fichier

On ouvre un fichier en appelant la fonction 3dh de l'interruption 21h. Celle-ci attend comme paramètre dans DS:DX *l'adresse d'une chaîne de caractères* qui contient le chemin d'accès au fichier sur un disque, par exemple "*C:\MonDoss\MonFic.txt*".

Remarque : il n'est pas indispensable de mentionner le chemin d'accès complet : par défaut, le fichier sera cherché à partir du dossier courant.

Remarque: La chaîne doit être impérativement suivie de l'octet 00h qui sert à marquer sa fin.

Il nous faut également spécifier le *mode d'accès* en écrivant dans AL un 0 (si on veut ouvrir le fichier en *lecture seule*), un 1 (si on veut l'ouvrir en *écriture seule*) ou un 2 (*lecture* ET *écriture*).

Si l'interruption échoue, le flag CF sera mis à 1 sans que le fichier soit ouvert. Dans le cas contraire, CF est mis 0 et le registre AX contient un petit nombre entier (par exemple 5) appelé « *handle* » (ce qui signifie « *poignée* ») du fichier. *Ce handle représente le fichier. C'est lui qu'il faudra désormais invoquer pour effectuer des opérations de lecture ou d'écriture, et non pas le chemin d'accès.*
En effet, les chemins d'accès ne sont donc plus d'aucune utilité puisque les fichiers sont ouverts dans la mémoire vive.

2. Lecture dans un fichier

Une fois le fichier ouvert, on peut le lire avec la fonction 3fh. Il suffit de mentionner le handle dans BX, le nombre d'octets à lire dans CX, et l'adresse d'un *buffer* dans DS:DX.
Au cas où vous ne sauriez pas ce qu'est un *buffer* (ou *tampon*), sachez que c'est simplement une variable (généralement une chaîne de caractères) destinée à *recevoir des données* (ou à en fournir). Dans notre cas, le buffer va recevoir les octets lus dans le fichier.

Après l'appel, AX contient le nombre d'octets qui ont été effectivement lus (il peut être inférieur à la taille demandée si le fichier n'est pas assez long). En cas de problème, CF sera mis à 1.

3. Ecriture dans un fichier

Pour écrire des données, on procède de même avec la fonction 40h. Les paramètres sont les mêmes que pour la fonction 3fh. Le *buffer* contient cette fois les octets à *écrire*. Après l'appel, le nombre d'octets qui ont été effectivement écrits est stocké dans AX (il sera être inférieur à la taille spécifiée si le disque est plein).

Les données sont écrites sur le disque dès que le tampon (dans la mémoire vive) est plein.

4. Existence d'un pointeur de fichier

Une question se pose cependant : *à quel endroit du fichier les données sont-elles lues (ou écrites) ?*

Réponse : quand un fichier est ouvert, un pointeur spécial pointant vers le début du fichier est créé. La première opération de lecture (ou d'écriture) se fera donc *au début du fichier*.
Mais entre chaque opération, le pointeur est incrémenté de la taille des données que l'on a lues (ou écrites). La deuxième opération se fera donc sur les octets qui suivent ceux de la première.

Remarque : Il est possible de modifier directement le pointeur de fichier : voyez pour cela la fonction 42h...

5. Fermeture d'un fichier

Pour terminer, le fichier doit être refermé. Les modifications éventuellement apportées et non enregistrées seront écrites sur le disque, et le *handle* sera libéré. C'est la fonction 3eh qui se charge de tout cela. Elle attend simplement le handle du fichier dans BX. Et comme d'habitude, CF vaut 1 après l'appel si des erreurs ont été rencontrées.

Remarque : Attention lorsque vous laissez le fichier ouvert longtemps afin d'y ajouter progressivement des données ! Si le système plante, vous perdrez les données qui se trouvent dans le tampon à ce moment. C'est pourquoi il est conseillé de forcer régulièrement l'écriture sur le disque en refermant le fichier.

6. Conclusion

Le tableau suivant récapitule ces différentes étapes :

Fonction	Description	Paramètres
3d*h*	***Ouvrir le fichier***	***- DS:DX*** *: adresse d'une chaîne contenant le chemin d'accès* ***- AL*** *: mode d'accès*
3e*h*	***Fermer le fichier***	***- BX*** *: handle*
3f*h*	***Lire le fichier***	***- BX*** *: handle* ***- CX*** *: nombre d'octets* ***- DS:DX*** *: adresse d'un buffer*
40*h*	***Ecrire dans le fichier***	***- BX*** *: handle* ***- CX*** *: nombre d'octets* ***- DS:DX*** *: adresse d'un buffer*

II. LES FONCTIONS DE RECHERCHE DE FICHIERS

Pour rechercher un fichier (ou un dossier), on se sert des fonctions 4eh (« *Find First* ») et 4fh (« *Find Next* »).

Imaginons par exemple que nous voulions chercher dans le dossier courant tous les fichiers qui portent l'extension "*.com*" afin de les supprimer. Comment devons-nous nous y prendre ?

1. La fonction 4eh

La fonction 4eh sert à définir des critères de recherche et à trouver le *premier* fichier qui correspond à ces critères (s'il existe).

On doit passer dans DS:DX l'adresse de la chaîne de caractères qui contient le *masque de recherche* (dans notre exemple, ce masque est "**.com*"). Par défaut, les fichiers sont cherchés dans le dossier courant. Mais on peut évidemment spécifier un autre chemin dans le masque.

Remarque importante : afin que le DOS puisse connaître sa taille, le masque doit impérativement être terminé par l'octet 00h !

On écrit également dans CX les attributs des fichiers que l'on désire trouver. Si CX vaut 0, seuls les fichiers « normaux » pourront être trouvés. En fait, chaque bit de CL représente un attribut, comme le montre le tableau ci-dessous :

Bit	Signification
1	***Lecture seule***
2	***Fichier caché***
3	***Fichier système***
4	***Volume***
5	***Répertoire***
6	***Fichier***
7	*(Aucune...)*
8	*(Aucune...)*

Pour demander à la fonction 4eh de ne pas oublier les fichiers cachés, il suffit donc de charger CX avec la valeur 2 (bit numéro 2 = 1). De même, l'attribut **00000111**b (soit 7) nous permettra de trouver les fichiers en lecture seule, les fichiers cachés et les fichiers systèmes.

Remarque : Ne vous souciez pas trop des bits numéro 4 et 6. Laissez-les à 0.

Une fois que les paramètres ont été ajustés, on peut appeler la fonction 4eh. *Si aucun fichier n'a été trouvé, le flag CF est mis à 1. On doit donc fait un test sur CF pour savoir si la recherche peut continuer ou si elle doit s'arrêter.*
Si au contraire la fonction a trouvé un fichier, les caractéristiques de ce fichier (i.e. son nom, sa taille, ses

attributs,...) sont inscrits dans une zone de la mémoire appelée *DTA* (« *Disk Tranfer Area* »).

Mais où se trouve donc cette DTA et à quoi ressemble-t-elle ?

Réponse : par défaut, le DOS place la DTA dans le PSP de votre programme, à l'offset 80h.

Remarque : il vous est naturellement possible de la déplacer en faisant appel à la fonction 1ah.

Voici la structure de la DTA :

Adresse	Description	Taille (octets)
00*h*	*Lettre du lecteur (0=courant, 1=A, 2=B, ...) sur lequel se trouve le fichier*	1
01*h*	*Modèle de la recherche*	11
0C*h*	*Réservé*	9
15*h*	*Attributs du fichier*	1
16*h*	*Heure du fichier*	2
18*h*	*Date du fichier*	2
1A*h*	*Taille du fichier*	4
1E*h*	*Nom du fichier avec l'extension*	13

Puisque par défaut la DTA est située dans le PSP à l'offset 80h, le nom du fichier trouvé est écrit à l'offset 80h + 1eh = 9eh. De même, la taille se trouve à l'offset 9ah, etc...

2. La fonction 4fh

Jusqu'à présent, nous n'avons trouvé qu'un seul fichier ! Pour poursuivre la recherche, appelez la fonction 4fh sans écrire aucun paramètre dans les registres. Les caractéristiques de votre recherche ont été mémorisées dans la DTA : vous n'avez donc pas à les rappeler. De même que pour la fonction 4eh, CF est mis à 1 si aucun nouveau fichier n'a été trouvé. C'est le signe que vous pouvez arrêter votre recherche.

3. Conclusion : Résumons nous :

Fonction	Description	Paramètres
4e*h*	*Trouver le premier fichier qui correspond aux caractéristiques spécifiées*	***- DS:DX** : adresse d'une chaîne contenant le masque* ***- CX** : attributs*
4f*h*	*Trouver le prochain fichier*	*Aucun !*

A titre d'exemple, écrivons à présent le programme que nous évoquions tout à l'heure : "*trouver tous les fichiers COM du dossier courant et les effacer*".

```
.386

code segment use16

assume cs:code, ds:code

org 100h

debut :

mov ah, 4eh ;fonction 4eh: chercher le PREMIER fichier correspondant
mov dx, offset Masque ;mettre l'offset du masque dans DX
xor cx, cx ;attributs : nous ne cherchons que les fichiers normaux...

cherchons_le_fichier :

int 21h
jc recherche_terminee ;si CF=1, alors la recherche est terminée !

;à présent, les caractéristiques du fichier trouvé sont dans la DTA...

mov ah, 41h ;fonction servant à effacer un fichier
mov dx, 80h+1eh ;mettre dans DX l'adresse du nom du fichier trouvé
int 21h ;effacer le fichier !

;Remarque : ce programme lui-même sera effacé !!

mov ah, 4fh ;fonction 4fh : chercher le prochain fichier correspondant
jmp cherchons_le_fichier

recherche_terminee :

ret

Masque db "*.COM", 0 ;ne pas oublier le code ASCII 0 !

code ends

end debut
```

LES INTERRUPTIONS DU DOS RELATIVES AUX FICHIERS

Un exemple de prgramme

1. Entrée bufferisée au clavier

L'exemple que nous proposons ici fait appel à une saisie au clavier. Comme nous n'en n'avons pas encore rencontré, ce paragraphe explique brièvement comment utiliser les fonctions 0ah et 0ch.

On peut se servir de la fonction 0ah pour lire une chaîne de caractères au clavier. Le seul paramètre à fournir est *l'adresse d'un buffer*. Comme il est souvent indispensable d'effacer le buffer avant la lecture, il est préférable de ne pas utiliser la fonction 0ah directement mais d'appeler la fonction 0ch. En effet, celle-ci commence par *effacer le buffer*, puis *appelle la fonction 0ah* (ou une fonction voisine dont le numéro doit être transmis dans AL).

Voici le code à écrire :

```
;(...)

mov ah, 0ch
mov al, 0ah ;la fonction 0ch doit appeller la 0ah après avoir vidé le buffer
mov dx, offset maChaine ;offset du buffer qui recevra les caractères entrés
int 21h

;(...)
```

La seule difficulté est dans la déclaration du buffer. Voici comment vous devez vous y prendre si vous considérez que l'utilisateur pourra entrer au plus *n* caractères :

```
;(...)

maChaine db n+1, ?, n dup(?), ? ;ou bien : maChaine db n+1, n+2 dup(?), ?

;(...)
```

Le premier octet de "*maChaine*" doit contenir le nombre d'octets maximal qui pourra être entré, plus 1. Pourquoi plus 1 ? Tout simplement parce que le DOS met à la fin des caractères tapés le code ASCII 13 (retour chariot).

Pour comprendre la suite de la déclaration, il faut savoir de quelle manière le DOS transmet les caractères qui ont été lus. Après la lecture, le deuxième octet du buffer contiendra le nombre exact d'octets lus (sans compter le 13 final). La chaîne proprement dite ne commencera donc qu'au troisième octet. Puisqu'un octet est utilisé pour donner au DOS le nombre maximal de caractères autorisé, un autre pour recevoir le

comporter trois octets de plus que la taille maximale de la chaîne. C'est pourquoi on écrit :

maChaine db *n+1*, ?, *n* dup(?), ?

2. Le programme

Le programme suivant demande à l'utilisateur d'entrer le nom d'un fichier se trouvant dans le dossier courant puis crypte ce fichier en appliquant un NOT logique sur chaque octet. Naturellement, il n'est pas optimisé du tout. Ce n'est qu'un exemple !

```
.386

code segment use16

assume cs:code, ds:code

org 100h

debut :

mov ah, 09h
mov dx, offset message
int 21h ; écrire le message à l'écran...

mov ah, 0ch ; fonction d'effacement du buffer et de saisie au clavier
mov al, 0ah ; saisie au clavier d'une chaîne du caractères
mov dx, offset buffer ; buffer où sera placée la chaîne
int 21h

xor bx, bx; mettre BX à zéro
mov bl, byte ptr [buffer + 1] ; mettre dans bl le nombre d'octets lus au clavier
mov byte ptr [buffer + 2 + bx], 0 ; écrire un 0 après le nom du fichier

mov ah, 3dh; ouvrir le fichier de départ
mov dx, offset buffer + 2; nom du fichier de départ
mov al, 0; mode d'accès = lecture seule
int 21h
jc existe_pas; si CF=1, le fichier demandé n'existe pas ou est inaccessible

mov bx, ax; mettre le handle dans BX

mov ah, 3ch ; créer le fichier d'arrivée
mov dx, offset nomcrypte; nom du fichier à créer
xor cx, cx; attributs
int 21h

mov ah, 3dh ; ouvrir le fichier d'arrivée
mov dx, offset nomcrypte
mov al, 1 ; mode d'accès = écriture seule
int 21h

mov word ptr [handle_arrivee], ax ; stocker le handle

lire_10000_octets :

mov ah, 3fh ; lire des données dans le fichier de départ
mov cx, 10000 ; nombre d'octets à lire
mov dx, offset donnees; buffer pour recevoir les données lues
int 21h

or ax, ax; comparer AX avec 0
jz fin_du_fichier; si AX=0, on n'a pas lu d'octet donc le cryptage est fini !

mov cx, ax ; mettre le nombre d'octets lus dans CX (pour le boucle)
```

Voici un exemple de macro qui renvoie dans AL le nombre de chiffres d'un entier non signé de deux octets passé dans AX :

```
nbdgt macro

local boucle, termine

pushf ;sauvegarder les flags sur la pile
push bx cx dx ;sauvegarder reg.
xor cx, cx ;i.e. MOV CX, 0

boucle:

xor dx, dx ;i.e. "MOV DX, 0". Indispensable car "DIV BX" utilise DX.
mov bx, 10
div bx ;division euclidienne de DX:AX par 10. Quotient = AX, reste = DX.
inc cx
or ax, ax ;i.e. "CMP AX, 0"
jnz boucle ;boucler si AX<>0. Sinon, on a atteint le dernier chiffre...

termine:

mov al, cl ;renvoyer le nb de chiffres dans AL
pop dx cx bx ;restaurer reg.
popf ;restaurer les flags

endm
```

Remarque: Il doit être clair que la liste des interruptions du DOS est importante. Vous ne pourrez rien faire si vous n'en avez pas !