Apprendre l'assembleur

(INTEL - DOS 16 bits)

Voici un petit exemple en assembleur qui écrit la lettre ‘A’ à l’écran :

```
;(...)

mov dl, 'A'
mov ah, 02
int 21h

;(...)
```

Examinons-le ligne par ligne :

l’instruction “MOV DL, ‘*A*’” demande au processeur de mettre dans le registre DL le code ASCII de la lettre ‘A’, c’est-à-dire 65, ou 41h.

“MOV AH, *02”* : mettre le nombre 2 dans AH.

Enfin, la dernière instruction appelle l’interruption numéro *21h*. Il existe 256 interruptions. *Toutes sont notées en base hexadécimale.*

Explications :

Comme vous aurez très vite l’occasion de vous en rendre compte, l’interruption 21h est l’interruption du DOS par excellence. Elle permet d’appeler de nombreuses fonctions très diverses. Pour cela, il suffit de mentionner leur numéro dans le registre AH.

Il est très difficile de mémoriser le rôle de chaque interruption, et a fortiori de chaque fonction ou sous-fonction, d’autant plus d’elles sont désignées par des numéros hexadécimaux et qu’elles attendent des paramètres dans des registres précis. C’est pourquoi tout programmeur se doit d’avoir à sa disposition une liste des interruptions pour travailler. Celle de Ralph Brown est très connue, et vous la trouverez sur l’Internet.

Revenons à notre exemple. La fonction numéro 2 de l’interruption 21h sert à écrire un caractère à l’écran. Il faut pour cela écrire le code ASCII du caractère dans le registre DL et bien sûr placer le nombre 2 dans AH.

Une fois que AH et DL ont été ajustés, l’interruption 21h peut être appelée à l’aide de l’instruction INT.

D’autres interruptions ne remplissent qu’une seule tâche. Vous n’avez donc pas besoin de mettre un numéro de fonction dans AH. L’appel de l’interruption suffit.

II. SAUTS INCONDITIONNELS

1. Les sauts inconditionnels

Ce paragraphe répond à la question : « *Mais comment fait-on un GOTO en assembleur* ? ».

L'instruction équivalente est JMP qui signifie « *jump* ».

La syntaxe est

JMP *MonLabel*

La référence à l'étiquette *MonLabel* sera remplacée lors de la compilation par la *distance* (signée) en octets qui sépare l'instruction qui suit immédiatement le JMP de l'instruction adressée par *MonLabel*. En pratique, le JMP s'utilise exactement comme un GOTO en BASIC.

Cette instruction permet de faire un *saut inconditionnel* : le programme fera ce saut quoiqu'il arrive.

Exemple :

```
;(...)

jmp short COUCOU

;tout ceci ne sera pas exécuté
;(...)
;(...)

COUCOU:

;le saut amène l'exécution ici
;(...)
;(...)

end start
```

Le mot "short" ajouté après "jmp" indique au compilateur que le label "*COUCOU*" se trouve à une distance (signée) qui peut être stockée sur un seul octet. Le compilateur ne le sait pas encore lorsqu'il essaie de compiler l'instruction de saut car le label se trouve en deçà de cette instruction. C'est pourquoi il prévoit deux octets pour écrire le saut, au cas où l'adresse d'arrivée soit éloignée de plus de 128 octets. Lorsqu'il effectue une deuxième « passe », s'il s'aperçoit qu'un seul octet aurait suffi il remplit alors l'octet inutile avec l'instruction NOP (« *No Operation* ») qui ne fait rien du tout. Le programme marchera parfaitement (encore heureux !), mais cela gaspille un octet. Le programmeur a la possibilité d'aider le compilateur en ajoutant le mot "short", ainsi un seul octet est prévu pour la distance de saut. Si le label est placé plus haut que l'instruction de saut, il est inutile de l'écrire.

Remarque : Ce mot peut également être utilisé avec les instructions de saut conditionnel que nous étudierons plus loin.

III. LES PRINCIPALES 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.

7. 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

9. 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*.

Mnémonique équivalent : LOOPNZ

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.

k) l'instruction XLAT (« *Translate* »)

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.

L'assembleur 80x86

3.1 L'assembleur

3.1.1 Pourquoi l'assembleur ?

Lorsque l'on doit lire ou écrire un programme en langage machine, il est difficile d'utiliser la notation hexadécimale (voir l'exemple page pageref). On écrit les programmes à l'aide de symboles[8] comme MOV, ADD, etc. Les concepteurs de processeur, comme Intel, fournissent toujours une documentation avec les codes des instructions de leur processeur, et les symboles correspondant.

Nous avons déja utilisé un programme, *debug*, très utile pour traduire automatiquement les symboles des instructions en code machine. Cependant, *debug* n'est utilisable que pour mettre au point de petits programmes. En effet, le programmeur doit spécifier lui même les adresses des données et des instructions. Soit par exemple le programme suivant, qui multiplie une donnée en mémoire par 8 :

```
0100    MOV BX, [0112]  ; charge la donnee
0103    MOV AX, 3
0106    SHL BX          ; decale a gauche
0108    DEC AX
0109    JNE  0106       ; recommence 3 fois
010B    MOV [0111], BX  ; range le resultat
010E    MOV AH, 4C
0110    INT 21H
0112    ; on range ici la donnee
```

Nous avons spécifié que la donnée était rangée à l'adresse 0111H, et que l'instruction de branchement JE allait en 0106H. Si l'on désire modifier légèrement ce programme, par exemple ajouter une instruction avant`MOV BX, [0111]`, il va falloir modifier ces deux adresses. On conçoit aisément que ce travail devienne très difficile si le programme manipule beaucoup de variables.

L'utilisation d'un *assembleur* résout ces problèmes. L'assembleur permet en particulier de nommer les variables (un peu comme en langage C) et de repérer par des *étiquettes* certaines instructions sur lesquelles on va effectuer des branchements.

3.1.2 De l'écriture du programme à son exécution

L'assembleur est un utilitaire qui n'est pas interactif, contrairement à l'utilitaire *debug*. Le programme que l'on désire traduire en langage machine (on dit *assembler*) doit être placé dans un fichier texte (avec l'extension `.ASM` sous DOS).

La saisie du programme source au clavier nécessite un programme appelé *éditeur de texte*.

L'opération d'assemblage traduit chaque instruction du programme source en une instruction machine. Le résultat de l'assemblage est enregistré dans un fichier avec l'extension `.OBJ` (*fichier objet*).

Le fichier `.OBJ` n'est pas directement exécutable. En effet, il arrive fréquemment que l'on construise un programme exécutable à partir de plusieurs fichiers sources. Il faut ``relier'' les fichiers objets à l'aide d'un utilitaire nommé *éditeur de lien* (même si l'on en a qu'un seul). L'éditeur de liens fabrique un fichier exécutable , avec l'extension `.EXE`.

Le fichier `.EXE` est directement exécutable. Un utilitaire spécial du système d'exploitation (DOS ici), le *chargeur* est responsable de la lecture du fichier exécutable, de son implantation en mémoire principale, puis du lancement du programme.

3.1.3 Structure du programme source

La structure générale d'un programme assembleur est représentée figure .

data SEGMENT ; data est le nom du segment de donnees

; directives de declaration de donnees

data ENDS ; fin du segment de donnees

ASSUME DS:data, CS:code

code SEGMENT ; code est le nom du segment d'instructions

debut: ; 1ere instruction, avec l'etiquette debut

; suite d'instructions

code ENDS

END debut ; fin du programme, avec l'etiquette ; de la premiere instruction.

Figure 3.1: Structure d'un programme en assembleur (fichier .ASM).

Comme tout programme, un programme écrit en assembleur comprend des définitions de données et des instructions, qui s'écrivent chacune sur une ligne de texte.

Les données sont déclarées par des *directives*, mots clef spéciaux que comprend l'assembleur. Les directives qui déclarent des données sont regroupées dans le *segment de données*, qui est délimité par les directives `SEGMENT` et `ENDS`.

Les instructions sont placées dans un autre segment, le *segment de code*.

La directive `ASSUME` est toujours présente et sera expliquée plus loin (section).

La première instruction du programme (dans le segment d'instruction) doit toujours être repérée par une étiquette. Le fichier doit se terminer par la directive `END` avec le nom de l'étiquette de la première instruction (ceci permet d'indiquer à l'éditeur de liens quelle est la première instruction à exécuter lorsque l'on lance le programme).

Les points-virgules indiquent des commentaires.

3.1.4 Déclaration de variables

On déclare les variables à l'aide de directives. L'assembleur attribue a chaque variable une adresse. Dans le programme, on repère les variables grâce à leur nom.

Les noms des variables (comme les étiquettes) sont composés d'une suite de 31 caractères au maximum, commençant obligatoirement par une lettre. Le nom peut comporter des majuscules, des minuscules, des chiffres, plus les caractères `@`, `?` et `_`.

Lors de la déclaration d'une variable, on peut lui affecter une valeur initiale.

Variables de 8 ou 16 bits

Les directives DB (*Define Byte*) et DW (*Define Word*) permettent de déclarer des variables de respectivement 1 ou 2 octets.

Exemple d'utilisation :

```
data    SEGMENT
entree  DW  15     ; 2 octets initialises a 15
sortie  DW  ?      ; 2 octets non initialises
cle     DB  ?      ; 1 octet non initialise
nega    DB  -1     ; 1 octet initialise a -1
data    ENDS
```

Les valeurs initiales peuvent être données en hexadécimal (constante terminée par `H`) ou en binaire (terminée par `b`) :

```
 data    SEGMENT
```

```
truc    DW  0F0AH     ; en hexa
masque  DB  01110000b ; en binaire
data    ENDS
```

Les variables s'utilisent dans le programme en les désignant par leur nom. Après la déclaration précédente, on peut écrire par exemple :

```
MOV  AX, truc
AND  AL, masque
MOV  truc, AX
```

L'assembleur se charge de remplacer les noms de variable par les adresses correspondantes.

Tableaux

Il est aussi possible de déclarer des tableaux , c'est à dire des suite d'octets ou de mots consécutifs.

Pour cela, utiliser plusieurs valeurs initiales :

```
data    SEGMENT
machin  db  10, 0FH  ; 2 fois 1 octet
chose   db  -2, 'ALORS'
data    ENDS
```

Remarquez la déclaration de la variable `chose` : un octet à -2 (=FEH), suivi d'une suite de caractères. L'assembleur n'impose aucune convention pour la représentation des chaînes de caractères : c'est à l'utilisateur d'ajouter si nécessaire un octet nul pour marquer la fin de la chaîne.

Après chargement de ce programme, la mémoire aura le contenu suivant :

Début du segment `data` → 0AH ← machin
OFH ← machin + 1
FEH ← chose
41H ← chose + 1
4CH ← chose + 2
4FH ← chose + 3
52H ← chose + 4
53H ← chose + 5

Si l'on veut écrire un caractère `X` à la place du `O` de `ALORS`, on pourra écrire :

```
MOV  AL, 'X'
MOV  chose+1, AL
```

Notons que `chose+1` est une constante (valeur connue au moment de l'assemblage) : l'instruction générée par l'assembleur pour
`MOV chose+1, AL`
est `MOV [adr], AL` .

Directive dup

Lorsque l'on veut déclarer un tableau de n cases, toutes initialisées à la même valeur, on utilise la directive `dup` :

```
tab   DB  100 dup (15) ; 100 octets valant 15
zzz   DW  10  dup (?)   ; 10 mots de 16 bits non initialises
```

3.3 Adressage indirect

Nous introduisons ici un nouveau mode d'adressage, l'adressage indirect , qui est très utile par exemple pour traiter des tableaux[10].

L'adressage indirect utilise le registre `BX` pour stocker l'adresse d'une donnée.

En adressage direct, on note l'adresse de la donnée entre crochets :

```
MOV  AX, [130]   ; adressage direct
```

De façon similaire, on notera en adressage indirect :

```
MOV  AX, [BX]   ; adressage direct
```

Ici, BX contient l'adressage de la donnée. L'avantage de cette technique est que l'on peut modifier l'adresse en BX, par exemple pour accéder à la case suivante d'un tableau.

Avant d'utiliser un adressage indirect, il faut charger BX avec l' *adresse* d'une donnée. Pour cela, on utilise une nouvelle directive de l'assembleur, `offset`.

```
data         SEGMENT
truc         DW   1996
data         ENDS

             ...
             MOV  BX, offset truc
             ...
```

Si l'on avait employé la forme

```
             MOV  BX, truc
```

on aurait chargé dans BX la *valeur* stockée en `truc` (ici 1996), et non son adresse[11].

3.3.1 Exemple : parcours d'un tableau

Voici un exemple plus complet utilisant l'adressage indirect. Ce programme passe un chaîne de caractères en majuscules. La fin de la chaîne est repérée par un caractère `$`. On utilise un ET logique pour masquer le bit 5 du caractère et le passer en majuscule (voir le code ASCII).

```
data         SEGMENT
tab          DB 'Un boeuf Bourguignon', '$'
data         ENDS

code         SEGMENT
             ASSUME DS:data, CS:code

debut:       MOV  AX, data
             MOV  DS, AX

             MOV  BX, offset tab ; adresse debut tableau

repet:       MOV  AL, [BX]       ; lis 1 caractere
             AND  AL, 11011111b  ; force bit 5 a zero
             MOV  [BX], AL       ; range le caractere
             INC  BX             ; passe au suivant
             CMP  AL, '$'        ; arrive au $ final ?
             JNE  repet          ; sinon recommencer

             MOV  AH, 4CH
             INT  21H            ; Retour au DOS
code         ENDS
             END debut
```

3.3.2 Spécification de la taille des données

Dans certains cas, l'adressage indirect est ambigu. Par exemple, si l'on écrit

```
MOV [BX],  0      ; range 0 a l'adresse specifiee par BX
```

l'assembleur ne sait pas si l'instruction concerne 1, 2 ou 4 octets consécutifs.

Afin de lever l'ambiguïté, on doit utiliser une directive spécifiant la taille de la donnée à transférer :

```
MOV  byte ptr [BX], val  ; concerne 1 octet
MOV  word ptr [BX], val  ; concerne 1 mot de 2 octets
```