1 Programmation en langage d'assemblage∇du CPU 8086

I. Pourquoi le langage d'assemblage?

Le langage d'assemblage permet de contrôler directement la CPU. Cela permet d'avoir une total maîtrise du système et surtout permet de faire des programmes rapides par rapport aux langages de haut niveau (C++, Basic, Pascal ...). En effet, bien que ces langages permettent de faire des programmes facilement et rapidement, ils n'optimisent pas l'exécution du code. Notons que l'on peut insérer des programmes d'assemblage dans certain langage (Pascal et C par exemple) pour accélérer l'exécution du programme.

II. Organisation d'une ligne de programme d'assemblage

Le format d'une ligne de programme d'assemblage est :

Etiquette : mnémonique opérandes ; commentaire

- Etiquette : (optionnel) représente l'adresse de l'instruction. Elle permet d'y faire référence ailleurs dans le programme (dans les instructions de branchement).
- Mnémonique: Souvent c'est la compression d'un mot ou d'une expression en anglais présentant l'action de l'instruction. Par exemple l'instruction MUL permet la multiplication (MULtiply). Elles sont constituées de 2à 6 lettres comme JZ et LOOPNZ.
- Opérandes sur lesquelles on veut agir. Ils peuvent être des registres, des emplacements de mémoire ou des constantes (valeurs immédiates).

En générale dans une instruction : **Mnémonique op1, op2 .**

op1 : représente la destination. C'est l'endroit ou le résultat sera stocké.

op2 : représente la source. C'est la donnée qui sera éventuellement combinée avec op1 pour calculer le résultat de l'instruction.

Dans la plupart des instructions, la source et la destination doivent être de même taille.
On peut trouver des instructions à opérande implicite : MUL BH.

Les types d'opérandes
On ne peut pas utiliser tous les types d'opérandes n'importe où. Les types autorisés selon l'instruction et nombre d'opérandes sont donnés sur le tableau suivant.

∇ Noter bien que c'est fréquent d'utiliser le mots "assembleur" pour désigner le "langage d'assemblage". Mais la définition exacte est que l'assembleur est le programme qui permet de transformer le langage d'assemblage en code machine.

	2 opérandes	1 opérande
Avec tous les opérateurs	R, R R, I R, M M, R M, I	R M
Pour MOV, on a aussi	R, S M, S S, R S, M	
Pour PUSH et POP, on a		R(16bits)

Où :

S un registre de segment (CS, DS, ES, SS)
R un registre ordinaire (yX, yH, yL, SI, DI, BP, SP)
I un immédiat (78h, 54h, ...)
M un des modes d'adressage de la mémoire

II.1. Les modes d'adressage de la mémoire

–En adressage direct, on indique l'adresse d'un emplacement en mémoire principale en hexadécimal entre crochets.
Exemple : MOV AX, [A340]

–En adressage relatif, on indique simplement l'adresse (hexa). L'assembleur traduit automatiquement cette adresse en un déplacement (relatif sur un octet).
Exemple: JNE 0108

–En adressage indirect, on indique une expression - en fonction du contenu des registres- indiquant l'adresse où se trouve l'adresse d'un emplacement en mémoire principale.
Pour l'adressage indirect, on peut utiliser exclusivement les registres : BX, BP, SI et DI.
La forme générale de l'adressage indirect est :
[registre de base +/- registre index +/- déplacement]
tel que : registre de base = BX, BP
registre d'index = SI, DI
déplacement = une constante

Les modes d'adressage qui en découlent sont :
[déplacement]
[reg. base] : [BX],[BP] ;
[reg. index] : [SI], [DI];
[reg. Base +/- reg. index] : [BX +/- SI], [BX +/- DI], [BP +/- SI], [BP +/- DI] ;
[reg. Base +/- déplacement] : [BX +/- déplacement], [BP +/- déplacement] ;
[reg. Index +/- déplacement] : [SI +/- déplacement], [DI +/- déplacement] ;
[reg. Base +/- reg. Index +/- déplacement] : [BX +/- SI+/- déplacement], [BX +/- DI +/- déplacement], [BP +/- SI +/- déplacement], [BP +/- DI +/- déplacement].

Noter bien que
Les expressions utilisant BX pointent sur la RAM et celles utilisant BP pointent sur la pile. BX est différent de [BX]: BX correspond au contenu du registre qui peut être une adresse ou autre, alors que [BX] est le contenu de l'emplacement mémoire dont l'adresse est dans BX

Exemple:
MOV SI, 4
MOV AL, [SI]
……
MOV AH, [BX+5]
……
MOV Al, [BX+SI]

III. Jeu d'instruction

III.1 Instructions de transfert

1) Entre registre et mémoire,
MOV a, b a := b
2) Entre registre et la pile
PUSH a pile := a
PUSHF pile := registre d'état
POP a a := pile
POPF registre de drapeaux := pile
3) Echange entre registres
XCHG a, b temp := b b := a a := temp
4) Entre l'unité entrée/sortie et microprocesseur
OUT port, a port := (port entre 0 et FFFF)
IN a, port a := port (port entre 0 et FFFF)
Les adresses sont dans DX. Seuls les registres AL et AX sont autorisés pour ces transferts.

```
MOV DX, 100h        MOV DX, 100h
IN AL, DX           MOV AL, 0Fh
                    OUT DX, AL
```

Remarque : pas de transfert entre mémoire et mémoire.

III.2. Instructions arithmétiques

ADD a, b a := a + b sans retenue
ADC a, b a := a + b avec retenue
SUB a, b a := a - b sans retenue
SBB a, b a := a - b avec retenue
INC a a := a + 1

DEC	a	a := a - 1	
NEG	a	a := -a	
MUL	a	AX := AL* a	si a est un octet
		DX:AX := AX * a	si a est un mot
IMUL	a	idem à MUL, mais en signé	
DIV	a	AL := AX(16bits) / a(8bits) AH := reste	si a est un octet
		AX := DX:AX (32bits) / a(16bits)DX := reste	si a est un mot
IDIV	a	idem à DIV, mais en signé	

III.3. Instructions logiques

AND	a, b	a := a et b
OR	a, b	a := a ou b
XOR	a, b	a := a xor b
NOT	a	a := complément à 1 de (a)
NEG	a	a := complément à 2 de (a)

Remarque :
1) L'instruction AND permet de masquer des valeurs :
MOV AL, 5Dh
AND AL, 0Fh; ne retient que les 4bits de poids faible de AL

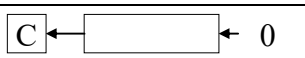
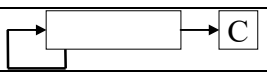
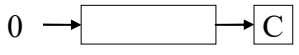
2) L'instruction OR peut être utilisée pour forcer des bits à 1
MOV AL, 5Dh
OR AL, 0Fh

3) L'instruction XOR permet d'inverser les bits d'un registre ou de les mettre plus rapidement à zéro que ne le fait un MOV.

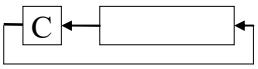
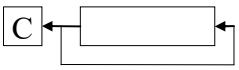
Exemple:
XOR AX, AX; remise à zéro rapide
MOV BL, 10110001b
NOT BL; BL=01001110b
XOR BL, 0FFh; BL reprend sa valeur initiale

III.4. Instructions de décalages, de rotations

a) Décalage:

SAL	a, b	déplacement à gauche de b bits de a	
SAR	a, b	déplacement à droite de b bits de a	
SHL	a, b	déplacement à gauche de b bits de a	
SHR	a, b	déplacement à droite de b bits de a	

b) Rotation :

RCL	a, b	rotation à gauche de b bits de a en passant par CF	
RCR	a, b	rotation à droite de b bits de a en passant par CF	
ROL	a, b	rotation à gauche de b bits de a sans passer par CF	
ROR	a, b	rotation à droite de b bits de a sans passer par CF	

III.5. Instructions de test et de comparaison

CMP a, b effectue a - b
TEST a,b effectue a and b
SCASB AL – [ES:DI]
SCASW AX – [ES:DI]

III.6. Instructions de branchement (saut)

III.6.a. Branchement inconditionnel

JMP a saut à l'adresse a (saut libre)
LOOP a saut à l'adresse a (antérieur)

Exemple :

```
MOV   CX, 100          ; boucle de 100
eti:
      ici, passe 100 fois
      LOOP eti         ; décrémente CX et boucle si CX ≠ 0
```

III.6.b. Branchement conditionnel

JO a saut à l'adresse a si OF = 1
JC a saut à l'adresse a si CF = 1
JZ a saut à l'adresse a si ZF = 1
JS a saut à l'adresse a si SF = 1
JP a saut à l'adresse a si PF = 1
JNO a saut à l'adresse a si OF = 0
JNC a saut à l'adresse a si CF = 0
JNZ a saut à l'adresse a si ZF = 0
JNS a saut à l'adresse a si SF = 0
JNP a saut à l'adresse a si PF = 0
JE a saut à l'adresse a si a = b (idem à JZ)
JNE a saut à l'adresse a si a ≠ b (idem à JNZ)
JG a saut à l'adresse a si a > b (idem à JNLE)(en non signé: JA, JNBE)
JGE a saut à l'adresse a si a >= b (idem à JNL)(en non signé: JAE, JNB)
JL a saut à l'adresse a si a < b (idem à JNGE)(en non signé: JB, JNAE)
JLE a saut à l'adresse a si a <= b (idem à JNG)(en non signé: JBE, JNA)
JCXZ a saut à l'adresse a si CX=0
LOOPZ a saut à l'adresse a si ZF=0

LOOPNZ a saut à l'adresse a si ZF=1

III.7. Instructions de traitement de chaîne caractères

STD Met à 1 DF
CLD clear direction_flag, Met à 0 DF
REP instruction répète CX fois l'instruction (décrémente CX à chaque passage)
Les instructions qu'elle répète sont MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW.

Exemple :

```
MOV CX, 7
REP MOVSB     ; exécute movsb 7 fois
```

REPZ instruction répète CX fois l'instruction tant que le ZF vaut un.

Exemple:

```
MOV CX, 7
REPZ CMPSB   ; répète 7 fois
```

MOVSB, MOVSW Copie de string:

```
[ES:DI] := [DS:SI]
if direction_flag = 0 then
  SI := SI + size;       (size = Byte=octet, Word=2octets)
  DI := DI + size;
Else
  SI := SI - size;
  DI := DI - size;
endif;
```

CMPSB, CMPSW Compare chaîne de caractères:

```
CMP [DS:SI], [ES:DI]
if direction_flag = 0 then
  SI := SI + size;       (size = B, W)
  DI := DI + size;
Else
  SI := SI - size;
  DI := DI - size;
endif;
```

Exemple:

```
CLD
MOV CX, 7       ; longueur à comparer
REPZ CMPSB
JNZ   pas_egal
...
pas_egal:
...
```

LODSB, LODSW
```
AL/AX := [DS:SI]
if direction_flag = 0 then
  SI := SI + size;        (size =1 si B, =2 si W)
Else
  SI := SI - size;
endif;
```

STOSB, STOSW
```
[ES:DI] := AL/AX
if direction_flag = 0 then
  DI := DI + size;        (size = B, W ou D)
Else
  DI := DI - size;
endif;
```

SCASB, SCASW cherche Ax/AL dans string définit par l'@ ES:DI
```
CMP AX/AL, [ES:DI]
if direction_flag = 0 then
  DI := DI + size;        (size =1 si B, =2 si W)
Else
  DI := DI - size;
endif;
```

Exemple:

```
CLD
MOV CX, 7       ; longueur du string
MOV AL, 'H'     ; on cherche H
REPNE  SCASB
JNZ   pas_trouve
```

Instructions Divers

Conversion : CBW AX := AL "étendu" (converti en Byte en Word)

III.8. Autres instructions spécifiques au microprocesseur

NOP pas d'opération

IV. Gestion de la pile

La programmation d'un CPU nécessite l'usage (d'au moins) une pile surtout pour le passage des paramètres aux sous-programmes, en sauvegardant temporairement le contenu des registres.
La pile est une zone mémoire, définie dans un segment de mémoire particulier (segment de pile) fonctionnant en mode LIFO (Last In, First Out: dernier entré premier sorti). C'est pour cela que tout ce qu'on déposera sur la pile via l'instruction push devra obligatoirement en être retiré dès que possible dans l'ordre inverse à l'ordre de dépôt via l'instruction pop.
Exemple:

```
push ax
push bx
push cx
[...]
pop cx
pop bx
pop ax
```

Le CPU possède deux registres dédiés à la gestion de la pile, SS et SP. SS est normalement initialisé au début du programme et reste fixé par la suite. Le registre SP contient le déplacement du sommet de la pile. Donc, l'adresse logique du dernier élément (sommet de la pile) est posée dans SS:SP. Lorsque l'on ajoute un élément à la pile, l'adresse contenue dans SP est décrémentée de 2 octets (car un emplacement de la pile fait 16 bits de longueur). La pile et le code (programme) croissent au sens inverse pour diminuer le risque de collision entre code et pile dans le cas où celle-ci est placée dans le même segment que le code (SS=SC).

Attention :
- Un usage maladroit de la pile peut provoquer des erreurs difficiles à trouver, comme sur une pile d'assiettes : vouloir retirer une assiette du milieu conduit à la catastrophe. Mais l'exemple des assiettes est un peu trompeur tout de même ; en effet, on peut très bien accéder à n'importe quel élément de la pile en y faisant référence de manière indirecte via sp, comme nous le verrons.

V. Procédures (sous-programme)

Pour éviter la répétition d'une même séquences d'instructions plusieurs fois dans un programme, on rédige cette séquence une seul fois à une adresse et on l'appelle lorsqu'on a besoin. Le programme appelant est le **programme principal**. La séquence d'instructions appelée est le **sous-programme** ou **procédure (subroutine).**
L'appelle d'une procédure revient à effectuer un saut -comparable à celui effectué par un JMP- vers un sous-programme dont l'adresse est donné en opérande. La différence réside dans le fait qu'il est possible après l'exécution du sous-programme de revenir par

l'instruction RET à l'instruction qui suivait le CALL. Pour ce faire, l'instruction CALL empile sur le sommet de la pile l'adresse de l'instruction suivant le CALL afin que l'instruction RET puisse dépiler cette adresse, puis y sauter.

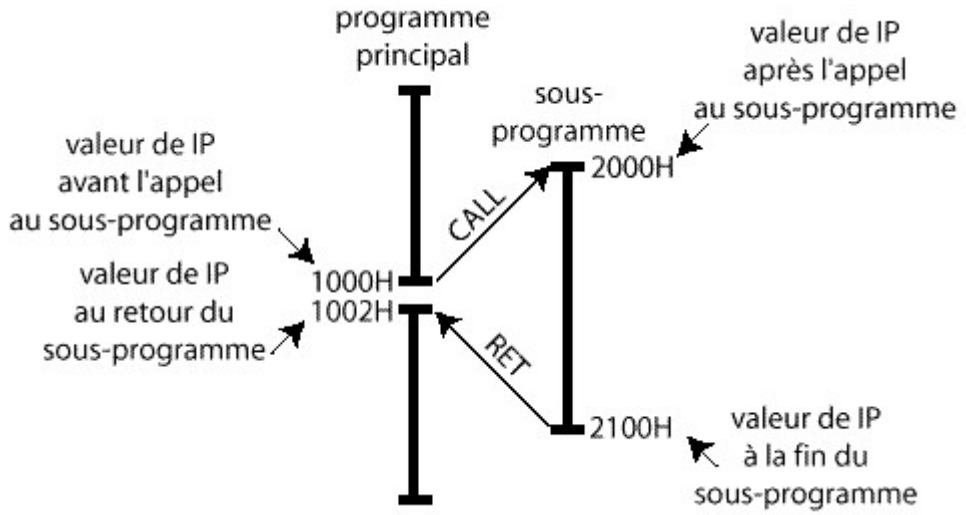


Figure 1. principe de l'appelle d'un sous-programme.

Remarque:
- Lors de l'appel à un sous-programme, Il est très important de remettre la pile dans l'état où on l'avait trouvé avant que ne soit atteinte une instruction RET car les instructions CALL et RET utilisent la pile pour mémoriser l'adresse à laquelle devra se faire le RETour.

V.1. Passage de paramètres

Une procédure effectue généralement des traitements sur des données (paramètres) qu'on lui fournit. Dans les langages évolués en place les paramètres entre parenthèses séparés par des virgules à coté du nom de la procédure. En langage d'assemblage, en passe les paramètres à une procédure par le programme principale de deux façons.

a- Passage des paramètres par pile
b- Passage des paramètres par registre

V.1.a. Passage des paramètres par registre

Cette méthode consiste à mettre les valeurs des paramètres dans les registres juste avant l'appel de la procédure. Avant le retour la procédure appelée doit à son tour ranger ses résultats dans les registres.
Le passage de paramètres par registre est simple à mettre en œuvre mais elle est très limitée, car on ne peut pas passer autant de paramètres que l'on désire, à cause du nombre limité de registres.

Exemple: soit une procédure effectuant la somme de deux nombres placés dans les registres AX et BX et retournant le résultat dans le registre AX :

```
programme principal :
        MOV AX,200
        MOV BX,300
        CALL somme ; appel de la procédure somme

procédure somme :
somme   ADD AX,BX ; addition des paramètres passés dans les registres AX et BX
        RET  ; retour au programme principale
```

V.1.b. Passage des paramètres par pile

Pour transmettre des paramètres à une procédure, on peut les placer sur la pile avant l'appel de la procédure, puis celle-ci les récupère en effectuant un adressage basé de la pile en utilisant le registre BP.

Exemple : soit une procédure effectuant la somme de deux nombres et retournant le résultat dans le registre AX :

```
programme principal :
        MOV AX,200
        PUSH  AX ; empilement du premier paramètre
        MOV AX,300
        PUSH AX ; empilement du deuxième paramètre
        CALL somme ; appel de la procédure somme(Pile ←IP, JMP somme)

procédure somme :
somme
        PUSH BP ; sauvegarde de BP
        MOV BP,SP ; faire pointer BP sur le sommet de la pile
        MOV AX,[BP+4] ; récupération du deuxième paramètre
        ADD AX,[BP+6] ; addition au premier paramètre
        POP BP ; restauration de l'ancienne valeur de BP
        RET 4; retour et dépilage des paramètres
somme endp
```

L'instruction RET 4 permet de retourner au programme principal et d'incrémenter le pointeur de pile de 4 unités pour dépiler les paramètres afin de remettre la pile dans son état initial, figure 2.

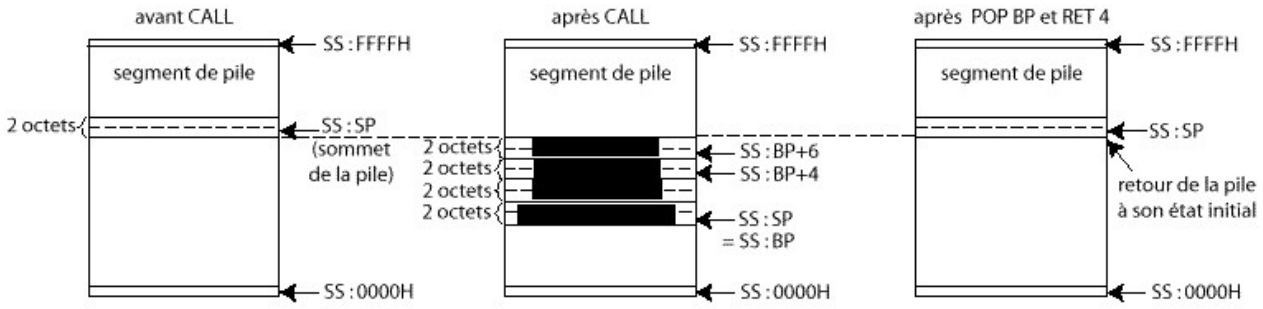


Figure 2. L'état de la pile.

Remarque:

- Gardez bien à l'esprit tout de même que, s'il est vrai que les sous-programmes permettent une meilleure lisibilité et une meilleure maintenance du code, elles le ralentissent également. Si c'est donc la plus grande rapidité possible que vous recherchez et que votre code n'est après tout pas si long que ca, n'hésitez pas à écrire le même code trois ou quatre fois au lieu de l'encapsuler dans un sous-programme. Le programme généré sera bien sûr plus long, mais aussi plus rapide. **A vous de voir**.
- Toute opération concernant la pile est plus lourde que les opérations avec les registres, c'est pourquoi il est préférable lorsque cela est possible de sauvegarder les valeurs dans des registres au lieu de les empiler/dépiler.
- Il existe deux conventions pour l'empilement/dépilement des arguments passés à une procédure :
    - la convention C (c'est à l'appelant de nettoyer la pile — cela permet une gestion plus aisée des arguments variables);
    - la convention Pascal (c'est à la fonction appelée de nettoyer la pile).
- Les instructions CALL, RET, ainsi que toutes les opérations impliquant la pile sont très coûteuses en cycles processeur.

Temps d'exécution d'une instruction

Chaque instruction nécessite un certain nombre de cycles d'horloges pour être exécuter. Le nombre de cycles dépend de la complexité de l'instruction et aussi du mode d'adressage : il est plus long d'accéder à la mémoire principale qu'à un registre du processeur.
La durée d'un cycle dépend bien sur de la fréquence d'horloge du microordinateur. Plus l'horloge bat rapidement, plus un cycle est court et plus on exécute un grand nombre d'instructions par seconde.