Chapitre 4

Le Langage Assembleur (80×86)

Introduction

Le langage machine se compose d'instructions binaire tel qu'on les trouve en mémoire au moment de l'exécution d'un programme. En effet, les premiers programmes étaient écrit en binaire, c'était une tache difficile et exposé aux erreurs car il fallait aligner les séquences de bits dont la signification n'est pas évidente. Donc si le langage machine est parfaitement adaptés aux ordinateurs il ne convient pas aux programmeurs. C'est pour cela qu'il a été abandonné depuis longtemps.

Pour faciliter la programmation, les programmes ont été écrits en donnant directement les noms abrégés des opérations, Ce sont des codes mnémoniques qu'on pouvait facilement mémoriser.

Contrairement aux langages évolués, telque le C, Pascal… l'assembleur, ou « langage d'assemblage » est constitué d'instructions directement compréhensibles par le microprocesseur : c'est ce qu'on appelle un langage de *bas niveau*. Il est donc intimement lié au fonctionnement de la machine.

4.1. Les Registres du 80×86

Un processeur réel a toutefois trop de registres et d'instructions pour pouvoir les étudier en détail. C'est pour cette raison que seuls les registres et les instructions d'un processeur simple (Intel 80x86 16 bits) seront étudiés dans ce cours.
Les registres du processeur 80×86 se classent par les catégories suivantes :

- *les registres **généraux** (16 bits)*
- *les registres **de segment** (16 bits)*
- *les registres **d'offset** (16 bits)*
- Le registre **FLAG** *(16 bits)*

4.1.1. Les registres généraux

Ils ne sont pas réservés à un usage très précis (d'où l'appellation registres généraux ou encore registres à usage générale), aussi les utilise-t-on pour manipuler des données diverses. Ce sont en quelque sorte des registres à tout faire. Chacun de ces quatre registres peut servir pour la plupart des opérations, mais ils ont tous une fonction principale qui les caractérise.

16bits	*8bits (High)*	*8bits (Low)*
AX	*AH*	*AL*
BX	*BH*	*BL*
CX	*CH*	*CL*
DX	*DH*	*DL*

- Le registre AX : « *accumulateur* » sert souvent pour les nombreuses opérations arithmétiques. il sert aussi de registre d'entrée-sortie : on lui donne des paramètres avant d'appeler une fonction ou une procédure.
- Le registre BX peut servir de *base d'adresse*.
- Le registre CX est utilisé comme compteur dans les boucles.
- Le registre DX sert pour stocker des données et aussi comme extension à AX.

4.1.2. Les registres de segment

Ils sont utilisés pour stocker l'adresse de début d'un segment. Il peut s'agir de l'adresse du début des instructions du programme, du début des données ou du début de la pile.

Nom	Nom complet	Traduction
CS	*Code segment*	Segment de code
DS	*Data segment*	Segment de données
ES	*Extra Segment*	Segment des extras
SS	*Stack segment*	Segment de pile

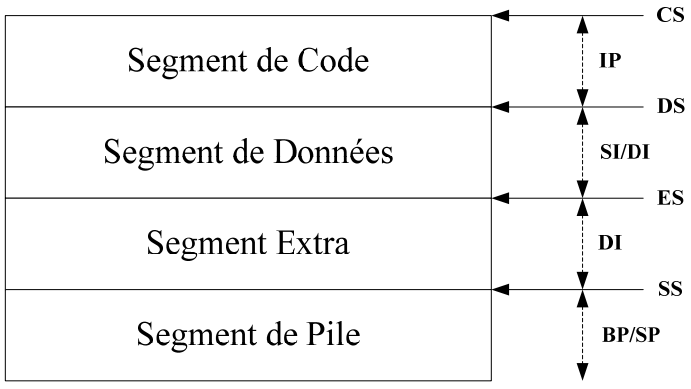
- le registre CS stocke l'adresse du segment code.
- Le registre DS contient l'adresse du segment des données du programme en cours.
- Le registre ES contient l'adresse du segment supplémentaire.
- Le registre SS adresse le segment de pile.

4.1.3. Les registres pointeurs et d'offset

Ces registres contiennent une valeur représentant un offset à combiner avec une adresse de segment, ils sont au nombre de cinq**:**

Nom	Nom complet	Traduction
SI	*Source index*	Index de source
DI	*Destination index*	Index de destination
SP	*Stack pointer*	Pointeur de pile
IP	*Instruction pointer*	Pointeur d'instruction
BP	*Base pointer*	Pointeur de base

- **Le registre SI** est principalement utilisé lors d'opérations sur des chaînes de caractères; il est associé au registre de segment DS.
- **Le registre DI** est normalement associé au registre de segment DS; dans le cas de manipulation de chaînes de caractères, il est associé à ES.
- **Le registre IP** est associé au registre de segment CS (CS**:**IP) pour indiquer la prochaine instruction à exécuter.
- **Le registre BP** est associé au registre de segment SS (SS**:**BP) pour accèder aux données de la pile lors d'appels de sous-programmes (*CALL*)
- **Le registre SP** est associé au registre de segment SS (SS**:**SP) pour indiquer le dernier élément de la pile.



4.1.4. Le registre FLAG

Le registre FLAG (registre d'état) est un ensemble de 16 bits organisé comme suit :

--	--	--	OF	DF	IF	TF	SF	ZF	--	AF	--	PF	--	CF

La valeur représentée par ce nombre de 16 bits n'a **aucune** signification en tant qu'ensemble**,** ce registre est manipulé **bit par bit**, certains d'entre eux influencent le comportement du programme.

Les Flags modifiés par les instructions arithmétiques, logiques et de comparaison sont :

CF (« *Carry Flag* ») est l'indicateur de retenue. Il est positionné à 1 si et seulement si l'opération précédemment effectuée a produit une retenue.

PF (« *Parity Flag* ») renseigne sur la parité du résultat. Il vaut 1 ssi ce dernier contient un nombre pair de bits 1.

ZF (« *Zero Flag* ») passe à 1 ssi le résultat d'une opération est égal à zéro.

SF (« *Sign Flag* ») passe à 1 ssi le résultat d'une opération sur des nombres signés est négatif (bit de poids fort =1).

OF (« *Overflow Flag* ») indique qu'un débordement s'est produit, c'est-à-dire que la capacité de stockage a été dépassée. Il est utile en arithmétique *signée*. Avec des nombres non signés, il faut utiliser ZF et SF.

4.2. La structure générale d'un programme assembleur

```
Title   exemple

pile            SEGMENT STACK ; pile est le nom du segment de pile
        DB        256  Dup (?)
pile            ENDS ; fin du segment de pile


données         SEGMENT ; données est le nom du segment de données
                        ; directives de declaration de données
données         ENDS    ; fin du segment de données


code            SEGMENT     ; code est le nom du segment d'instructions
    ASSUME      DS:données, CS:code, SS:pile
                MovAX,données
                Mov DS, AX
    debut :     Suite d'instructions
                .
                .
                Mov ah,4ch
                int 21h
code            ENDS
  END       debut ; fin du programme avec l'étiquette de la première instruction.
```

4.3. Définitions des données

- ***DB : Define Byte***

Permet de réserver un emplacement mémoire de 1octet

Exemple :

Nom-var DB ?

→ Permet de déclarer une variable Nom-var de 1 octet non initialisée

Nom-var DB 23

→ Permet de déclarer une variable Nom-var de 1 octet initialisée à 23

Nom-tab DB 4, 3, 10, 15

→ Permet de déclarer un tableau Nom-tab contenant les valeurs 4, 3, 10, 15

- ***DW : Define word***

Permet de réserver un emplacement mémoire de 1mot (16 bits pour le processeur 80×86).

- ***DD : Define double word***

Permet de réserver un emplacement mémoire sur un double mot (32 bits).

4.4. Le jeux d'instructions du 80×86

4.4.1. Le transfert des données

- ***MOV***

Syntaxe : MOV *Destination*, *Source*

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

Mouvements autorisés : MOV Registre général, Registre quelconque

MOV *Mémoire*, Registre quelconque. Exemple : MOV [100h], BX

MOV Registre général, *Mémoire*. Exemple : MOV CX,[100h]

MOV Registre général, *Constante*. Exemple : MOV AX, 100h

MOV *Mémoire*, *Constante*. Exemple : MOV [100h], 100 h

Remarques : *Source* et *Destination* doivent avoir la même taille.

LEA (« Load effective address »)

Syntaxe : LEA *Destination*, *Source*

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

Exemples : LEA BX, variable ; équivalent à MOV BX, OFFSET variable *met dans BX l'adresse de la variable*

4.4.2. Les Instructions Arithmétiques

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

Syntaxe : INC *Destination*

Description : Incrémente *Destination*.

Exemple : INC CX ;

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

Syntaxe : DEC *Destination*

Description : Décrémente *Destination*.

Exemple : DEC AX ;

- **l'instruction ADD (« Addition »)**

Syntaxe : ADD *Destination*, *Source*

Description : Ajoute *Source* à *Destination*

Exemple : ADD AX,*BX* ; AX= AX+BX

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

Syntaxe : SUB *Destination*, *Source*

Description : Soustrait *Source* à *Destination*.

- **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.

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

- **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.

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

4.4.3. Les instructions logiques

- **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).

- **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*.

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.

- **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*.

- **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*.

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.

4.4.3. L'opérateur de Comparaison

- **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.

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 *branchements conditionnels* (voir suite).

4.4.4. Les instructions de branchement

4.4.4.1 Les instructions de branchement inconditionnel

- **L'instruction JMP (« *Jump* »)**

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

4.4.4.2 Les instructions de branchement conditionnel

Les sauts conditionnels sont 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).
Certaines mnémoniques de sauts conditionnels sont totalement équivalentes, c'est-à-dire qu'ils représentent le même opcode hexadécimal.

- **JE** (« *Jump if Equal* ») fait un saut au label spécifié si et seulement si ZF = 1.

- **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.
  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* »)

- **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. Cette mnémonique correspond au même opcode que JNE.

4.4.5. Les instructions de Boucle

- **l'instruction LOOP**

Syntaxe : LOOP *MonLabel*
Description : Décrémente CX, puis, si CX <> 0, fait un saut à *MonLabel*.
Exemple : exécution d'un bloc d instructions 4 fois

```
MOV cx, 4
Etiquette : ensemble d'instructions
LOOP  Etiquette
```

4.5. Le code opération de quelques instructions du 80×86

Symbole	Code Op	Octets	Opération
MOV AX, *valeur*	B8	3	AX←*valeur*
MOV AX, [*adr*]	A1	3	AX←contenu de l'adresse *adr*
MOV [*adr*], AX	A3	3	[*adr*] ←AX
ADD AX, v*aleur*	05	3	AX←AX+*valeur*
ADD AX, [*adr*]	0306	4	AX←AX+contenu de *adr*
SUB AX, v*aleur*	2D	3	AX←AX-*valeur*
SUB AX, [*adr*]	2B06	4	AX←AX-contenu de *adr*
INC AX	40	1	AX←AX+1
DEC AX	48	1	AX←AX-1
CMP AX, v*aleur*	3D	3	Compare AX et *valeur*
CMP AX, [*adr*]	3B06	4	Compare AX et contenu de *adr*
JMP *adr*	EB	2	Saut inconditionnel (adr. Relatif)
JE *adr*	74	2	Saut si =
JNE *adr*	75	2	Saut si ≠
JG *adr*	7F	2	Saut si >
JLE *adr*	7E	2	Saut si ≤