Architectures des Ordinateurs

1ère année GI

COURS 5
LE LANGAGE ASSEMBLEUR
(80×86)

MAROUA MASMOUDI KOTTI

2022-2023

Introduction

- Le langage machine se compose d'instructions binaires.
- Une instruction est définie par son code opératoire **opcode** : valeur numérique binaire difficile à manipuler par l'être humain.
- Les premiers programmes étaient écrit en binaire, c'était une tâche 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.

Introduction

- Pour faciliter la programmation, les programmes ont été écrits en donnant directement les noms abrégés des opérations,
- On utilise donc une **notation symbolique** pour représenter les instructions : les **mnémoniques.**
- Ce sont des codes qu'on pouvait facilement mémoriser.

→ Un programme constitué de mnémoniques est appelé **programme en assembleur**, ou en «langage d'assemblage»

Introduction

- **Contrairement** aux langages évolués,
  - tel que le C, Pascal,
- l'assembleur, 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.

→Dans la suite de ce cours, nous découvrons le langage assembleur du processeur **80x86**

Processeur 8086

X86: famille de microprocesseurs compatibles avec le jeu d'instructions de l'Intel 8086

- Evolution :
  - début 80 : 8086, microprocesseur 16 bits
  - Le premier microprocesseur développé par Intel
  - Constitué de 29000 transistors sur une puce de 32,7 mm^2
- Caractéristiques
  - Adressage direct de 1MO,
  - 14 registres de 16 bits
  - 24 modes d'adressage

Les Registres du 80×86

- Les 14 registres du processeur 80×86 se classent en 4 groupes :
  - *registres **généraux** (16 bits)*
  - *registres **de segment** (16 bits)*
  - *registres **d'offset** (16 bits)*
  - registre **FLAG** *(16 bits)*

Les registres généraux

- Ils ne sont pas réservés à un usage très précis (d'où l'appellation registres généraux ou encore registres à usage général),
- aussi les utilise-t-on pour manipuler des données diverses.
- Ils servent à contenir temporairement des données.
- 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.

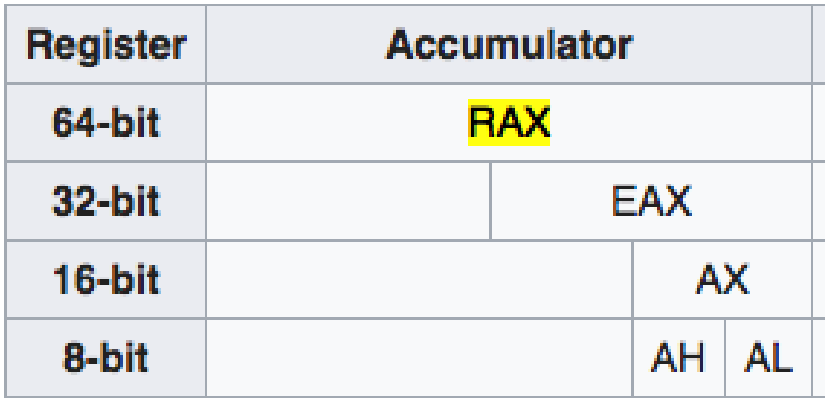
Les registres généraux

- Le registre **AX** : « *accumulateur* » .
  - sert souvent pour les 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.

Les registres généraux

→ Ils peuvent être également considérés comme 8 registres sur 8 bits.

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



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.
  - le registre **CS** pointe vers les instructions du programme (segment **code)**.
  - Le registre **DS** pointe vers les données du programme en cours (segment des **données**).
  - Le registre **ES** pointe vers les données du segment **supplémentaire**.
  - Le registre **SS** adresse le segment de **pile**.

Les registres de segment

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

Les registres pointeurs et d'offset

- Les pointeurs et les index contiennent des adresses de cases mémoire.
- Ils contiennent une valeur représentant un offset à combiner avec une adresse de segment.
- Ils sont utilisés pour les transferts de chaînes d'octets entre deux zones mémoire.

Les registres pointeurs et d'offset

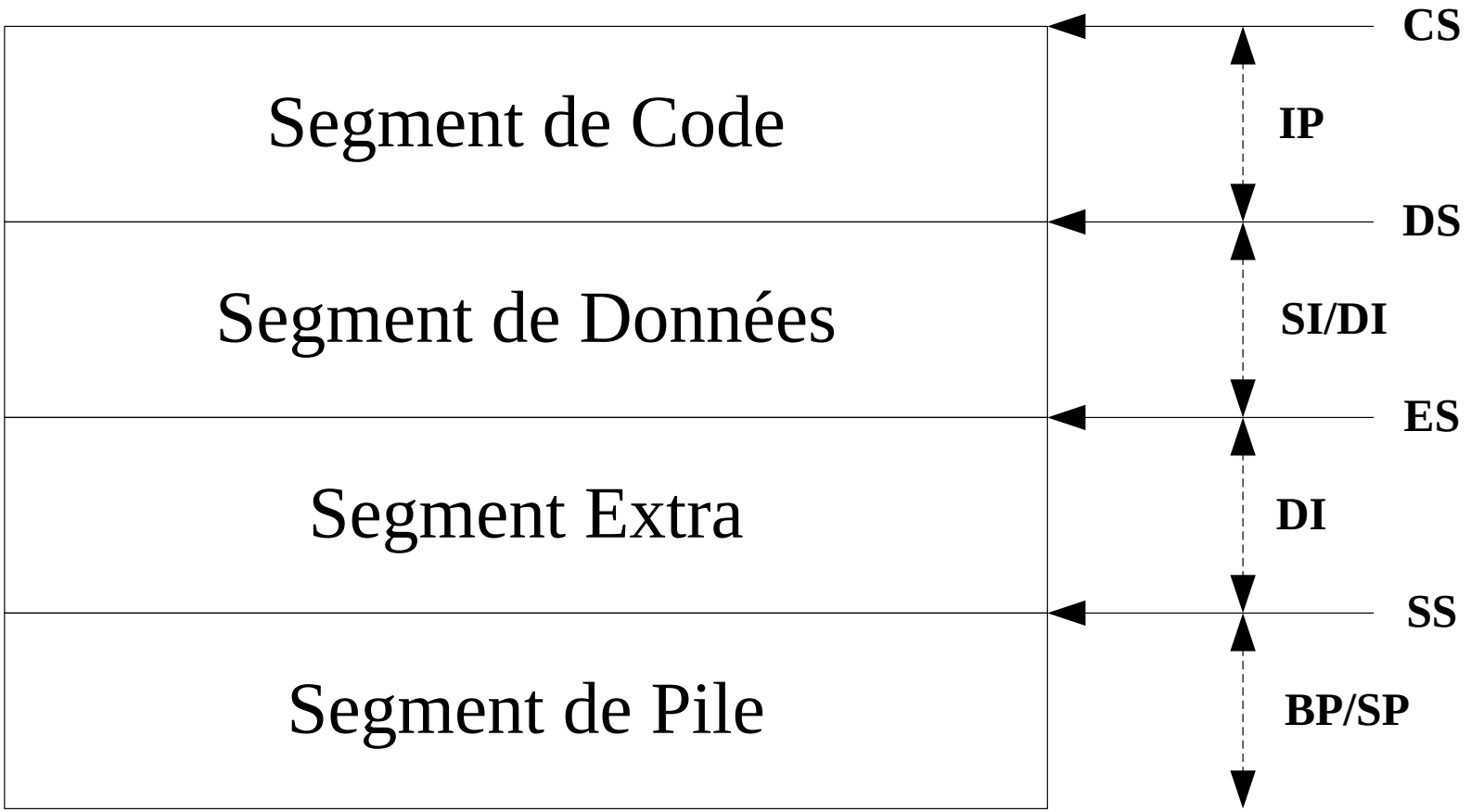
- **Le registre SI** est principalement utilisé lors d'opérations sur des chaînes de caractères; il est associé au registre de segment DS.
- **Le registre DI** est normalement associé au registre de segment DS; dans le cas de manipulation de chaînes de caractères, il est associé à ES.
- **Le registre 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 accéder 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.

Les registres pointeurs et d'offset

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

RQ: Les registres de segments, associés aux pointeurs et aux index, permettent d'adresser l'ensemble de la mémoire : **segment : offset →** est une adresse logique au sein d'un segment.

Les registres pointeurs et d'offset



Le registre FLAG

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

- 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 bits influencent le comportement du programme.

Le registre FLAG

- 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.

Le jeux d'instructions du 80×86

LES INSTRUCTIONS DU 80×86 SONT CLASSÉS EN :

- Instructions de transfert des données
- Instructions arithmétiques
- Instructions logiques
- Instruction de comparaison
- Instructions de branchement
- instructions de Boucle

Les jeux d'instructions du 80 x 86

- Le 8086 est un microprocesseur CISC: 350 instructions
- Une instruction a la forme:

  Mnémonique Destination, Source

- Les instructions du 80×86 sont classés en :
  - Instructions de transfert des données
  - Instructions arithmétiques
  - Instructions logiques
  - Instruction de comparaison
  - Instructions de branchement
  - instructions de Boucle

Instructions de transfert des données

- **Syntaxe** :
  - MOV *Destination*, *Source*
- **Description** :
  - Copie le contenu de Source dans Destination :
    - registre vers mémoire ;
    - registre vers registre ;
    - mémoire vers registre.
- **Mouvements autorisés** :
  - MOV Registre général, Registre quelconque
  - le microprocesseur 8086 n'autorise pas les transferts de mémoire vers mémoire (pour ce faire, il faut passer par un registre intermédiaire).

Instructions de transfert des données

- 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

- **Remarque** : *Source* et *Destination* doivent avoir la même taille.

Instructions de transfert des données

- **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*

Instructions de transfert des données

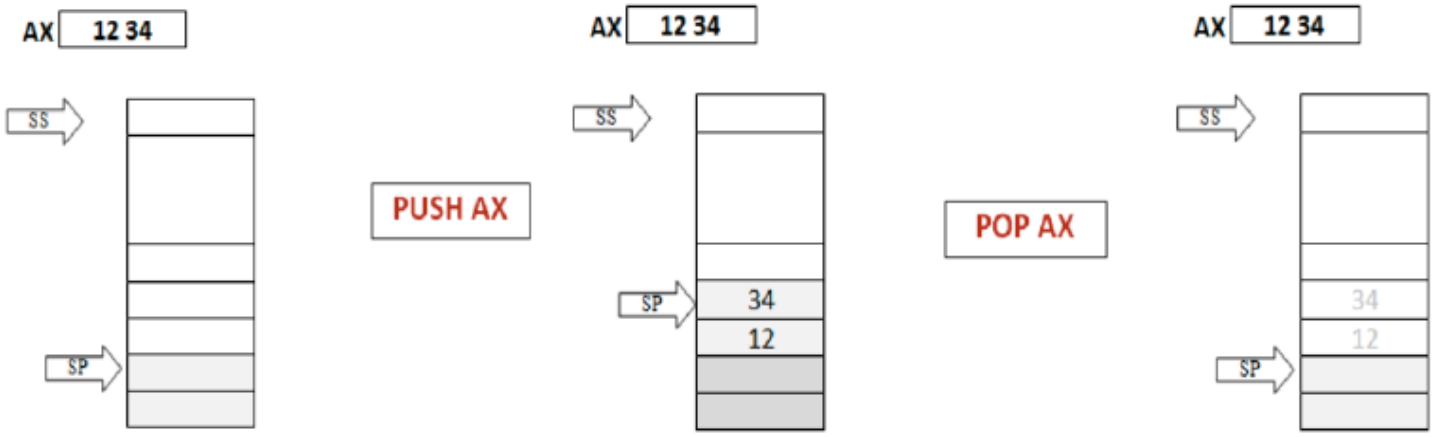
- PUSH
  - Syntaxe: PUSH source
  - Description: copie le contenu de source au sommet de la pile, commence par décrementer SP puis effectue la copie.
  - Source : adresse, constante ou register
- Exemple:
  - Push 1 ; empile la constante 1
  - Push AX ; empile le contenu de AX
  - Push [var]; empile la valeur se trouvant à l'adresse var

Instructions de transfert des données

- POP
  - Syntaxe: POP destination
  - Description: copie les octets qui se trouvent au sommet de la pile dans destination, commence par effectuer la copie puis incremente SP.
  - destination : adresse ou register
- Exemple:
  - Pop AX ; dépile dans le register AX
  - Pop [var]; dépile à l'adresse var

Instructions de transfert des données

- Exemple : Push et pop



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 ;

Instructions arithmétiques

- **l'instruction ADD (« Addition »)**
  - **Syntaxe** : ADD *Destination*, *Source*
  - **Description** : Ajoute *Source* à *Destination*
  - Exemples :
    - ADD AX, *BX* ; AX= AX+BX
    - ADD ah,[1100H] : ajoute le contenu de la case mémoire d'offset 1100H à l'accumulateur AH (adressage direct) ;
    - ADD ah,[bx] : ajoute le contenu de la case mémoire pointée par BX à l'accumulateur AH (adressage basé) ;
    - ADD [1200H],05H : ajoute la valeur 05H au contenu de la case mémoire d'offset 1200H (adressage immédiat).

Instructions arithmétiques

- **l'instruction SUB (« Substract »)**
  - **Syntaxe** : SUB *Destination*, *Source*
  - **Description** : Soustrait *Source* à *Destination*.
  - Exemple :
    - SUB AX, *BX* ; AX= AX-BX

Instructions arithmétiques

- **L'instruction MUL (« Multiply »)**
  - **Syntaxe** : MUL *Source*
    - *Avec source* un registre ou une case mémoire
  - **Description** : Effectue une multiplication d'entiers.
  - Si *Source* est un **octet** : AL est multiplié par *Source* et le résultat est placé dans AX (16 bits).
  - Si *Source* est un **mot** : AX est multiplié par *Source* et le résultat est placé dans DX:AX (32 bits).
  - Si *Source* est un **double mot** : EAX est multiplié par *Source* et le résultat est placé dans EDX:EAX (64 bits).
- Remarque : *Source ne peut être une valeur immédiate.*

Instructions arithmétiques

- **L'instruction MUL (« Multiply »)**
  - **Exemples :**

```
mov al,51
mov bl,32
mul bl
```
→ *AX = 51 × 32*

```
mov al,43
mov byte ptr [1200H],28
mul byte ptr [1200H]
```
→ *AX = 43 × 28*

```
mov ax,4253
mov bx,1689
mul bx
```
→ *(DX, AX) = 4253 × 1689*

```
mov ax,1234
mov word ptr [1200H],5678
mul word ptr [1200H]
```
→ *(DX, AX) = 1234 × 5678*

Instructions arithmétiques

- **l'instruction DIV (« *Divide* »)**
  - **Syntaxe** : DIV *Source*
    - *Avec source* un registre ou une case mémoire
  - **Description** : Effectue une division euclidienne.
  - 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.*

Instructions arithmétiques

- **l'instruction DIV (« *Divide* »)**
  - **Exemples** :

```
mov ax,35
mov bl,10
div bl
→ AL = 3 (quotient)
et AH = 5 (reste)
```

```
mov dx,0
mov ax,1234
mov bx,10
div bx
→ AX = 123 (quotient)
et DX = 4 (reste)
```

Instructions logiques

- **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*. (Destination = source ET destination)
  - **Exemple:**

```
mov al,10010110B
mov bl,11001101B      →
and al, bl
```

AL =	1	0	0	1	0	1	1	0
BL =	1	1	0	0	1	1	0	1
AL =	1	0	0	0	0	1	0	0

Instructions logiques

- **l'instruction NOT (« *Logical NOT* ») ou Complément à 1**
  - **Syntaxe** : NOT *Destination*
  - **Description** : Effectue un NON logique bit à bit sur *Destination* (i.e. chaque bit de *Destination* est inversé).

```
mov al,10010001B
not al
```

$\rightarrow \quad AL = \overline{10010001}B = 01101110B$

Instructions logiques

- **l'instruction NEG (« *Negation* ») ou Complément à 2**
  - **Syntaxe** : NEG *Destination*
  - **Description** : Effectue un complément à 2 sur *Destination* (i.e. Complément à 1 + 1).

```
mov al,25
mov bl,12
neg bl
add al,bl
```

$\rightarrow \quad \mathrm{AL} = 25 + (-12) = 13$

Instructions logiques

- **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*.
  - **Exemple** : dans le mot 10110001B on veut mettre à 1 les bits 1 et 3 sans modifier les autres bits.
  - Les instructions correspondantes peuvent s' écrire :

```
mov ah,10110001B
or ah,00001010B
```

7	6	5	4	3	2	1	0	
1	0	1	1	0	0	0	1	
0	0	0	0	1	0	1	0	← masque
1	0	1	1	1	0	1	1	

Instructions logiques

- **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*.

Rappel : XOR

•*Le résultat est VRAI si un et un seul des opérandes A et B est VRAI*

ou

•*Le résultat est VRAI si les deux opérandes A et B ont des valeurs distinctes*

Table de vérité de XOR

A	B	R = A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

Instructions logiques

- **l'instruction XOR (« Exclusive logical OR »)**
  - **Exemple** : mise à zéro d'un registre :

    mov al, 25

    xor al, al

    $\rightarrow AL = 0$
  - → 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.

Instruction 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 (FLAG) qui contient les résultats de la comparaison*. Ni *Source* ni *Destination* 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*.

Instruction de branchement inconditionnel

- **L'instruction JMP (« *Jump* »)**
  - Syntaxe : JMP *MonLabel*
  - Description : fait un Saut (jump) à l'instruction pointée par *MonLabel*.
  - Un label (ou étiquette) est une représentation symbolique d'une instruction en mémoire :

```
... ← instructions précédant le saut
jmp suite
... ← instructions suivant le saut (jamais exécutées)
suite : . . . ← instruction exécutée après le saut
```

Instruction de branchement inconditionnel

- **L'instruction JMP (« *Jump* »)**
  - Exemple :

```
boucle :    inc ax
            dec bx
            jmp boucle
```

  - → *boucle infinie*

Instructions de branchement conditionnel

- Les sauts conditionnels ***Jcondition label*** sont importants
  - car ils permettent au programme de faire des choix en fonction des données.
- Un saut conditionnel n'est exécuté que ;
  - si une certaine condition est satisfaite,
  - sinon l'exécution se poursuit séquentiellement à l'instruction suivante.

Instructions de branchement conditionnel

- Les conditions portent sur les flags, par exemple :

instruction	nom	condition
`JZ label`	Jump if Zero	saut si ZF = 1
`JNZ label`	Jump if Not Zero	saut si ZF = 0
`JE label`	Jump if Equal	saut si ZF = 1
`JNE label`	Jump if Not Equal	saut si ZF = 0
`JC label`	Jump if Carry	saut si CF = 1
`JNC label`	Jump if Not Carry	saut si CF = 0
`JS label`	Jump if Sign	saut si SF = 1
`JNS label`	Jump if Not Sign	saut si SF = 0
`JO label`	Jump if Overflow	saut si OF = 1
`JNO label`	Jump if Not Overflow	saut si OF = 0

Instructions de branchement conditionnel

- Certaines mnémoniques de sauts conditionnels sont totalement équivalentes, c'est-à-dire qu'ils représentent le même **opcode** hexadécimal.
- **JE** (« *Jump if Equal* ») et **JZ** (« *Jump if Zero* »)
  - fait un saut au label spécifié si et seulement si ZF = 1.
  - JZ correspond au même opcode que JE.
- **JNE** (« *Jump if Not Equal* ») et **JNZ** (« *Jump if not Zero* »)
  - fait un saut au label spécifié si et seulement si ZF = 0.
  - JNZ correspond au même opcode que JNE.

Instructions de branchement conditionnel

- Les sauts conditionnels sont aussi utilisés lors des **sauts arithmétiques**.
- Ils suivent en général l'instruction de comparaison : **CMP** opérande1,opérande2
- Exemple :

```
cmp ax,bx
JG superieur
JL inferieur
superieur : . . .
...
inferieur : . . .
```

Instructions de branchement conditionnel

Condition	nombres signés	nombres non signés
=	JE (*Jump if Equal*)	JE (*Jump if Equal*)
>	JG (Jump if Greater)	JA (Jump if Above)
>=	JGE (Jump if Greater or Equal)	JAE (*Jump if Above or Equal*)
<	JL (Jump if Less).	JB (Jump if Below)
<=	JLE (*Jump if Less Or Equal*)	JB (Jump if Below *Or Equal*)
<>	JNE (*Jump if Not Equal*)	JNE (*Jump if Not Equal*)

Instructions de branchement conditionnel

Condi-tion	nombres signés	Flags (saut ssi …)	Mnémonique équivalent
=	JE (*Jump if Equal*)	ZF = 1	JZ (*Jump if Zero*)
>	JG (Jump if Greater)	ZF=0 et SF=OF	JNLE (Jump if Not Less Or Equal)
>=	JGE (Jump if Greater or Equal)	SF = OF	JNL (Jump if Not Less)
<	JL (Jump if Less)	SF <> OF	JNGE (Jump if Not Greater Or Equal)
<=	JLE (*Jump if Less Or Equal*)	SF <> OF ou ZF = 1	JNG (Jump if Not Greater)
<>	JNE (*Jump if Not Equal*)	ZF = 0	JNZ (*Jump if not Zero*)

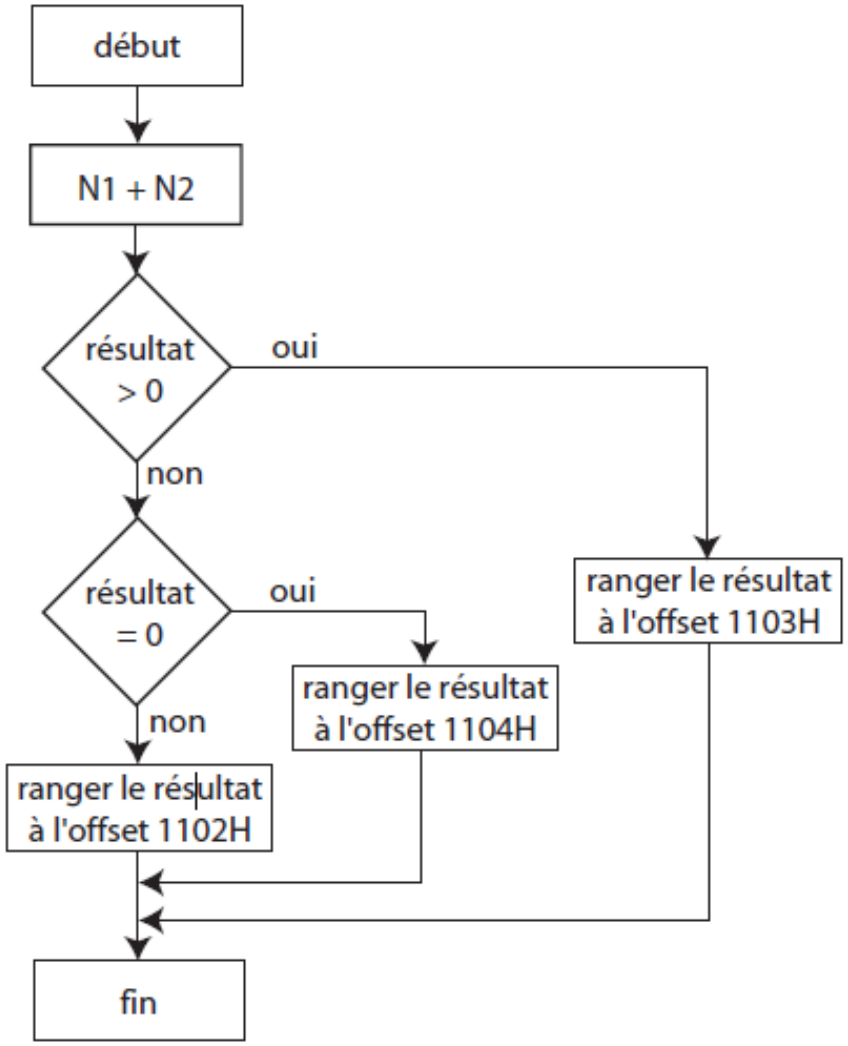
Instructions de branchement conditionnel

Condi-tion	nombres non signés	Flags (saut ssi …)	Mnémonique équivalent
=	JE (*Jump if Equal*)	ZF = 1	JZ (*Jump if Zero*)
>	JA (Jump if Above)	ZF=0 et CF=0	JNBE (Jump if Not Below Or Equal)
>=	JAE (*Jump if Above or Equal*)	CF = 0	JNB (Jump if Not Below)
<	JB (Jump if Below)	CF = 1	JNAE (Jump if Not Above Or Equal)
<=	JBE (*Jump if Below or Equal*)	CF = 1 ou ZF = 1	JNA (Jump if Not Above)
<>	JNE (*Jump if Not Equal*)	ZF = 0	JNZ (*Jump if not Zero*)

Instructions de branchement conditionnel

- Exemple d'application des instructions de sauts conditionnels :
  - on veut additionner deux nombres signés N1 et N2 se trouvant respectivement aux offsets 1100H et 1101H.
  - Le résultat est rangé à l'offset 1102H s'il est positif,
  - à l'offset 1103H s'il est négatif
  - et à l'offset 1104H s'il est nul.

Instructions de branchement conditionnel




```
mov al,[1100H]
add al,[1101H]
js negatif
jz nul
mov [1102H],al
jmp fin
negatif : mov [1103H],al
jmp fin
nul : mov [1104H],al
fin : hlt
```

Instruction 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
```

Le code operation de quelques instructions du 80x86

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

Le code operation de quelques instructions du 80x86

Symbole	Code Op	Octets	Opération
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 ≤

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
Mov AX,données
Mov DS, AX
debut :
                          Suite d'instructions …
Mov ah,4ch
int21h
code                ENDS
END debut ; fin du programme avec l'étiquette de la première instruction.
```

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

Définitions des données

- ***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).

Travaux Pratiques

- On apprendra quelques fonctions de ce langage.
- Pour cela, veuillez l'installer https://www.nasm.us/ sur vos ordinateurs.
- Sur Ubuntu:
  - sudo apt install nasm
- Un programme assembleur est un fichier texte d'extension .asm.
- Pour compiler et exécuter: helloworld.asm
  - nasm -felf64 helloworld.asm
  - ld helloworld.o -o helloworld
  - ./helloworld