Assembleur

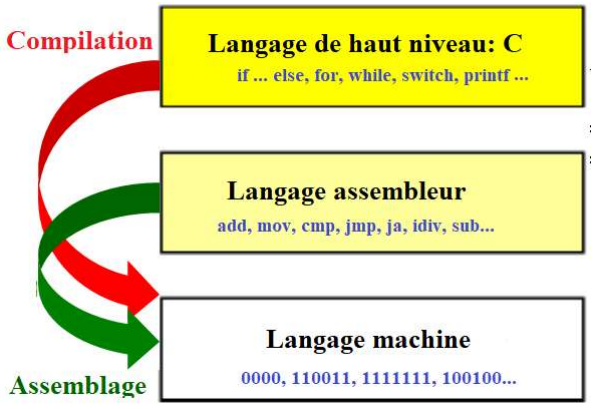
Langages

- Le **langage machine** est le langage **compris** par le **microprocesseur**.
- Le **langage assembleur** ou **ASM** est le langage de **bas niveau** **plus proche** du **langage machine**.
  - Il représente sous forme **lisible**, pour un être humain, le **code binaire exécutable** ou **code machine**.
  - Il est composé par des **instructions** en général assez **rudimentaires** que l'on appelle des **mnémoniques**.

348

Assembleur

Langages



349

Assembleur

Langage assembleur

- Il décrit **l'ensemble** des **opérations élémentaires** que le **microprocesseur** pourra exécuter.
- Les **instructions** que l'on retrouve dans chaque **microprocesseur** peuvent être classées en **4 groupes** :
  - **Transfert de données** pour charger ou **sauver** en mémoire, **effectuer** des **transferts** de registre à registre, etc…
  - **Opérations arithmétiques** : **addition**, **soustraction**, **division**, **multiplication**
  - **Opérations logiques** : **ET**, **OU**, **NON**, **comparaison**, **test**, etc…
  - **Contrôle de séquence** : **branchement**, **test**, etc…

350

Assembleur

Instruction d'assembleur

- **Chaque instruction** nécessite un certain **nombre** de **cycles d'horloges** pour s'effectuer : c'est **le temps d'exécution**. Il est dépendant de la **complexité e l'instruction**.
- Les **instructions et leurs opérandes** (paramètres) sont **stockés** en **mémoire principale**.
- **La taille totale** d'une **instruction** (**nombre de bits** nécessaires pour la représenter en mémoire) dépend **du type d'instruction** et aussi du **type d'opérande**.
- Une instruction est **composée de deux champs** :
  - Le **code instruction** qui indique au **processeur** quelle **instruction réaliser**
  - Le **champ opérande** qui contient la **donnée**, ou la **référence** à une **donnée** en mémoire (son **adresse**).

351

Assembleur

Instruction d'assembleur

- Une **instruction** est **composée de deux champs** :
  - **Le code instruction** qui indique au **processeur** quelle instruction réaliser
  - **Le champ opérande** qui contient la **donnée**, ou la **référence** à une donnée en mémoire (son adresse).
- Il existe **différents formats d'instruction** suivant le nombre de parties **réservées** aux **opérandes** (ou **adresses**), et peuvent être sur **1, 2, 3 ou 4 octects**.
  - **code_opération opérande** **(format 1 adresse)**
  - **code_opération opérande_1 opérande_2** **(format 2 adresses)**

champ code opération	champ code opérande

352

Assembleur

Instruction d'assembleur

- Une **instruction assembleur élémentaire est de la forme**

Etiquette: **Mnémonique** **OpDst[, OpSrc][;commentaire]**

code opératoire ***11001101 0101 1001***

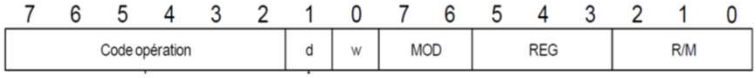
- Il existe plusieurs **convention d'écriture** d'un programme ASM. On va **adopter** celle **de l'émulateur 8086**.

353

Assembleur

Instruction d'assembleur

- Format de base d'une instruction



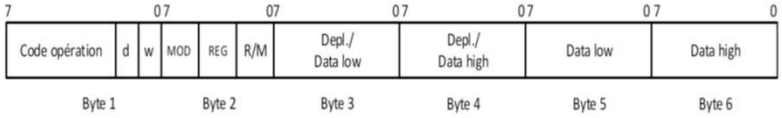
- L'instruction de base tient sur deux octets. Le premier octet définit le code de l'instruction ainsi que les bits d et w. Le second octet est composé de trois champs : MOD, REG et R/M.

354

Assembleur

Instruction d'assembleur

- Instruction sur deux opérandes



- Une instruction sur deux opérandes peut comporter plus de deux octets. Les octets supplémentaires expriment un déplacement et/ou une donnée. Lorsqu'une instruction contient à la fois un déplacement et une donnée, la donnée suit le déplacement

355

Assembleur

Instruction d'assembleur

Bits spéciaux

- Le bit ***d*** de destination : S'il existe un seul registre dans l'instruction, d spécifie si ce registre est la source ou bien la destination. d=1 : le registre est une destination, d= 0 : le registre est une source (d=1 : to REG ; d=0 : from REG).
- Le bit ***w*** (Width) : w=1 pour un opérande sur 16 bits et w=0 pour pérande sur 8 bits
- Les champ ***MOD*** et ***R/M***: indiquent le code du mode d'adressage direct ou indirect, basé ou indexé, avec ou sans déplacement, déplacement sur 8 bits ou 16 bits ainsi que le code de l'adresse mémoire (directe ou indirecte) utilisé dans l'instruction
- Le champ ***REG*** : contient un nombre binaire de 0 à 7

356

Assembleur

Instruction d'assembleur

Bits spéciaux

- **Le bit S :** Un nombre binaire sur 8 bits, selon la représentation en complément à 2, peut être étendu à un nombre sur 16 bits en complément à 2, en mettant tous les bits de l'octet haut à la même valeur que le MSB de l'octet bas. C'est ce que l'on appelle une extension du signe. Le bit S apparaît dans les instructions d'addition, de soustraction, et de comparaison sous la forme : Immédiat vers Registre/Mémoire. Il est défini comme suit :

- **S : W = 00 :** opération sur 8 bits ;
- **S : W = 01** : opération sur 16 bits ;
- **S : W = 11** : opération sur un opérande sur 8 bits avec extension du signe à 16 bits.

357

Assembleur

Instruction d'assembleur

- Bits spéciaux

- **Le bit V :** Utilisé par les instructions de décalage et de rotation, il permet de déterminer le nombre de répétition.
- **V = 0 :** Nombre de répétition = 1 ;
- **V = 1 :** Nombre de répétition contenu dans CL.
- **Le bit Z :** Utilisé par l'instruction REP dans les primitives de manipulation de chaînes pour la comparaison avec le flag Z.

358

Assembleur

Structure de programme

- Lorsque l'utilisateur **exécute un programme**, celui-ci est d'abord **chargé** en **mémoire** par le **système**.
- Le **DOS distingue deux modèles** de **programmes exécutables :**
  - **De type .COM**
  - **De type .EXE**
- Les **COM** ne peuvent pas utiliser **plus d'un segment** dans la mémoire. Leur taille est ainsi **limitée a 64 Ko**.
- Les **EXE** peuvent utilisés **plusieurs segments** et ne sont **limités que par la mémoire** disponible dans **l'ordinateur**.

359

Assembleur

Structure de programme .com

- Le **programme** suivant permet d'afficher le message "bonjour, monde".

```
1)code SEGMENT
2)   assume CS:code,DS:code,SS:code
3)   org 100h
4)   debut:
5)            mov ah, 09h
6)            mov dx, offset msg
7)            int 21h
8)            ret ;mov ah,4ch ;int 21h
9)            msg db "bonjour,monde","$"
10)  end debut
11)code ends
```

360

Assembleur

Structure de programme .com

- Le **code source commence** par des **directives** (les informations que le programmeur fournie au **compilateur**) qui ne sont pas **transformées** en **instructions assembleur**.
- **1) code SEGMENT :** Ceci permet de déclarer un segment appelé "code"
- **2) assume CS:code, DS:code, SS:code :** Ceci " informe le compilateur que CS,DS,SS pointe vers "code" afin qu'il génère des adresses correctes.
- **3) org 100h :** Ceci signifie qu'il faut ajouter 100h à tous les offsets à cause de la structure du programme COM.
- **4) debut:** La partie code du programme commence par ce label qui sert a représenter l'adresse de l'instruction qui le suit.

361

Assembleur

Structure de programme .com

- **5) mov ah, 09h 6) mov dx, offset msg 7) int 21h :** Ces lignes permettent d'écrire une chaine de caractère a l'écran l'offset de cette chaine est attendue dans "dx« . C'est la fonction "9" de l'interruption 21h qui affiche cette chaine
- **8) ret ;mov ah,4ch ;int 21h :** Ceci pour terminer un programme et revenir vers l'interpréteur de commandes DOS
- **9) msg db "bonjour,monde","$:** déclaration de la variable "msg«
- **10) end debut :** Ceci indique la fin du label
- **11) code ends** : Ceci indique la fin du segment code

362

Assembleur

Programme .com

- Lorsqu'il charge **le fichier .COM**, le Dos lui alloue **64 ko**, crée le **PSP (Program segment Prefix)** et **copie** le **programme charge** a la suite.
- **Le PSP est une zone mémoire** de **256 octets** qui contient des **informations diverses** au **sujet du programme** (chaine de caractère qui indique le **nom du fichier**….).
- Un **programme .COM** est **limité** ().
- Un **programme .COM commence** a partir **de l'offset 100h**.

363

Assembleur

Programme .exe

- Le DOS **réserve** pour un programme **.EXE**:
  - **Un segment de code**
  - **Un segment pile**
  - **1 ou 2 segments pour le programme et un segment pour le PSP.**
  - **Le programme commence à l'offset 0h.**

➔ Il est **possible** de n'utiliser **qu'un seul segment.**

364

Assembleur

Fichier .exe

```
1)  assume  CS:code,  SS:pile, DS:data
2)  data segment
3)  message db "press any key...$"
4)  data ends
5)  pile segment stack
6)      remplissage db   256  dup(?)
7)  pile ends
8)  code segment
9)  debut:
10) mov ax, data      ; set segment registers:
11) mov ds, ax
12)     mov ah, 9
13)     mov dx, offset message
14)     int 21h       ; output string at ds:dx
15)     mov ah, 1
16)     int 21h       ; wait for any key....
17)     mov ax, 4c00h ; exit to operating system.
18)     int 21h           ;remplace ret
19) code ends
20) end debut ; set entry point and stop the assembler.
```

Déclaration du segment de données

Déclaration du segment de pile

Déclaration du segment de code Avec initialisation des registres de segment

365

Assembleur

Fichier .exe

- Avec la directive **'assume'** dans la ligne 1, le compilateur est informé que **DS pointe vers data**, **SS pointe vers pile** et **CS pointe vers code** faisant le **lien** entre les segments **déclarés** et les **registres CS,DS, ES et SS** afin qu'il génère des adresses correctes.
- Le **programmeur en assembleur** doit se charger de **l'initialisation** du registre segment **DS**, de la façon suivante :

  MOV AX, nom_segment_de_donnees

  MOV DS, AX
- Il **n y a plus** de: **Org 100h**
- **Les points-virgules** indiquent des **commentaires**.

366

Assembleur

Les variables

- **L'assembleur** déclare les **variables** à l'aide de **directives** attribue à chaque **variable une adresse**.
- Dans le **programme**, les variables sont **identifiés** par des **noms**:
  - **Ils sont composés par une suite de 31 caractères maximum,**
  - **Ils commencent obligatoirement par une lettre.**
  - **Ils comportent des majuscules, des minuscules, des chiffres, plus les 3 caractères @ ? _**
- Lors de la déclaration d'une **variable**, on peut lui affecter une **valeur initiale**.

367

Assembleur

Les directives de variables

- Les **directives** permettent de déclarer des variables :
  - **DB (Define Byte)** : **1 octet** Réserver de 1 octet dans la mémoire
  - **DW (Define Word)** : **2 octets** (1 mot) Réserver de 2 octet dans la mémoire
  - Il existe aussi **DD** pour 4 octets **DQ** pour **8 et 10 octets**
- Les **valeurs initiales** peuvent être données en
  - **Hexadécimal**: se terminant par H et ayant un 0 si le nombre commence par une lettre, par exemple: 1h,12h, 0Fh
  - **Binaire**: se terminant par b , par exemple: 11b, 00101b
  - **Décimal**: 1, 2, 3, 123, 45

368

Assembleur

Les directives de variables

- Les **directives** permettent de **déclarer des variables** :
  - **DB (Define Byte)** : **1 octet** Réserver de 1 octet dans la mémoire
  - **DW (Define Word)** : **2 octets** (1 mot) Réserver de 2 octet dans la mémoire
  - Il existe aussi **DD** pour 4 octets **DQ** pour 8 et 10 octets
- **Syntaxe:** *[nomVar] DB constante1 [, constante2] […]*
  [nomVar] DW constante1 [, constante2] […]

- Réserve et initialise un ou deux octets, nomVar est un symbole permettant d'accéder à un octet ou un mot

369

Assembleur

Constante

- Il existe **la directive equ** utilisée pour la **déclaration d'une constante.**
- Cette constante **n'est pas une variable** comme les autres déclarées par les autres **directives** (DB, DW, …) il n'y a pas de **réservation** au niveau de la mémoire.
- **Syntaxe :** **nom EQU constante**
- **Exemple :**

ZERO EQU 0

Définit une constante qui s'appelle ZERO, dont la valeur est 0. Une fois cette déclaration effectuée, toute occurrence de l'identificateur ZERO sera remplacée par la valeur indiquée.

370

Assembleur

Les tableaux

- Les **tableaux** peuvent être **déclarés** de **deux façons**:
- comme **suite d'octets** ou de **mots consécutifs**, en utilisant plusieurs valeurs **initiales**.

```
tab1 db 10, 0FH ; 2 fois 1 octet
tab2 db -2, "ALORS"
```

- A l'aide de la **directive dup** pour déclarer un **tableau** de **n cases** qui sont soit **non initialisées** soit toutes **initialisées** à la **même valeur**

```
tab3 DB 10 dup (5) ; 10 octets initialisés à 5
tab4 DW 10 dup (?) ; 10 mots de 16 bits non initialises
```

371

Assembleur

Pile

- La pile est une **zone mémoire** gérée d'une façon particulière dont le registre est **SS**.
- Elle mémorise les adresses **d'appel** et **retour** de sous programmes (procédures)
- Elle est organisée comme une pile d'assiettes où le retrait se fait juste en haut ➔ principe **LIFO** (Last IN, First Out).
- Il est géré par le registre d'adresse Stack Pointer **SP**.
- **Empiler une donnée** (**PUSH**) : sauvegarder une donnée sur (le sommet) de la pile
- **Dépiler une donnée** (**POP**) : retirer une donnée (du sommet) de la pile

372

Assembleur

Registre de pile

- **Le registre SS (Stack Segment)** est le registre segment qui contient **l'adresse** du segment de pile **courant**,
- Il est **initialisé** au **début** du **programme** et reste **fixe** par la suite.
- **Le registre SP (Stack Pointer)** **pointe** sur le **dernier** bloc occupé de la **pile**
- **PUSH registre** : **Empile** le contenu du registre sur la pile.

  SP = SP- 2
- **POP registre**: **Retire** la valeur en haut de la pile et la place dans le registres spécifié.

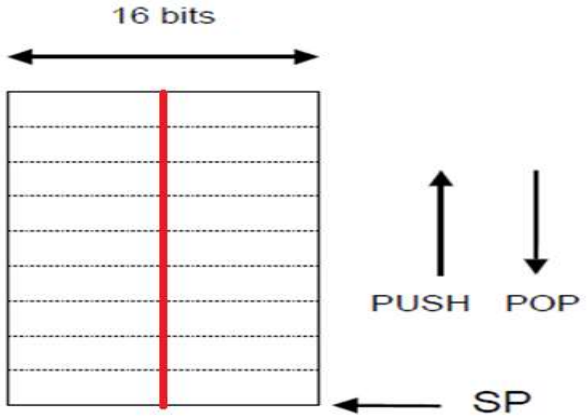
  SP = SP+ 2

373

Assembleur

Principe de pile

- Quand la pile est **vide** , SP pointe **sous** la pile



374

Assembleur

Pile: exemple

transfert de AX vers BX en passant par la pile.

MOV AX, 14 *; AX =14*

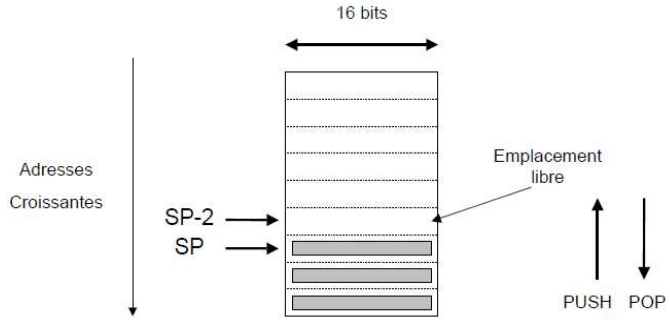
PUSH AX *; empile le contenu de AX sur la pile*

POP BX *; dépile le 1er élément dans la pile dans BX*

375

Assembleur

Illustration du fonctionnement de la pile :



376

Assembleur

Déclaration de pile

- Pour utiliser une pile en assembleur, il faut déclarer un segment de pile, et y réserver un espace suffisant. Ensuite initialiser les registres SS et SP. Ce dernier doit pointer sous le sommet de la pile.
- Exemple : déclaration d'une pile de 512 octets :

```
seg_pile SEGMENT stack ; mot clef stack car pile
DW 256 dup (?)
Base_Pile: ; étiquette base de la pile
seg_pile ENDS
```

- Le mot clé "stack" après la directive SEGMENT indique à assembleur qu'il s'agit d'un segment de pile.

377

Assembleur

Initialisation de pile

- Après, il faut **utiliser** la **séquence d'initialisation** dans le **segment de code** :

```
ASSUME SS:seg_pile
MOV AX, seg_pile
MOV SS, AX ; init Stack Segment
MOV SP, Base_Pile ; pile vide
```

- Le **registre SS s'initialise** de façon **similaire** au **registre DS**
- **SP** est **initialisé** avec **l'adresse du bas de la pile repérée** par l'étiquette **Base_Pile**

378

Assembleur

L'émulateur Emu8086

- Il se **présente sous forme** d'un **éditeur de texte classique**, avec le support d'une **colorisation syntaxique** du **code assembleur**.
- Dans cet **émulateur**, ils existent certaines **options** du **menu** '**view**' qui peuvent être **pratiques**.

379

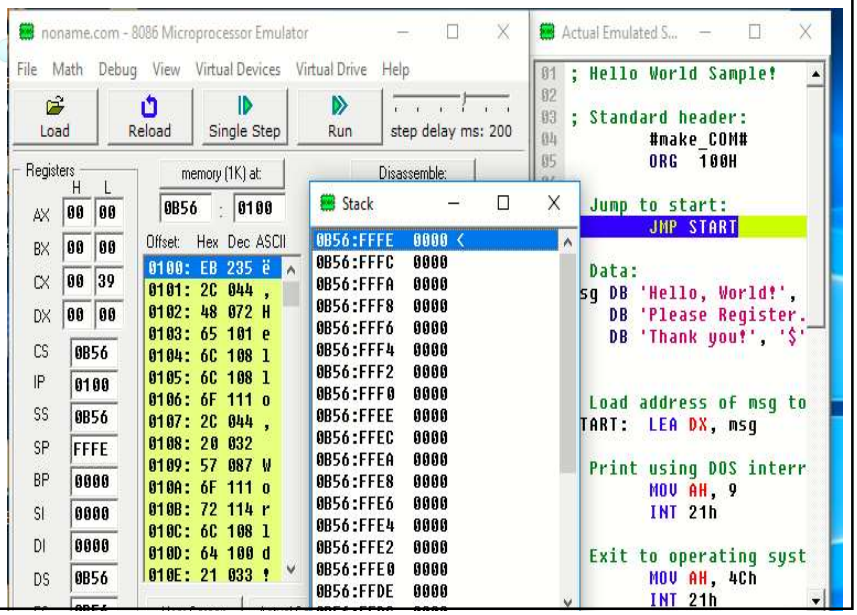
Assembleur

L'émulateur Emu8086

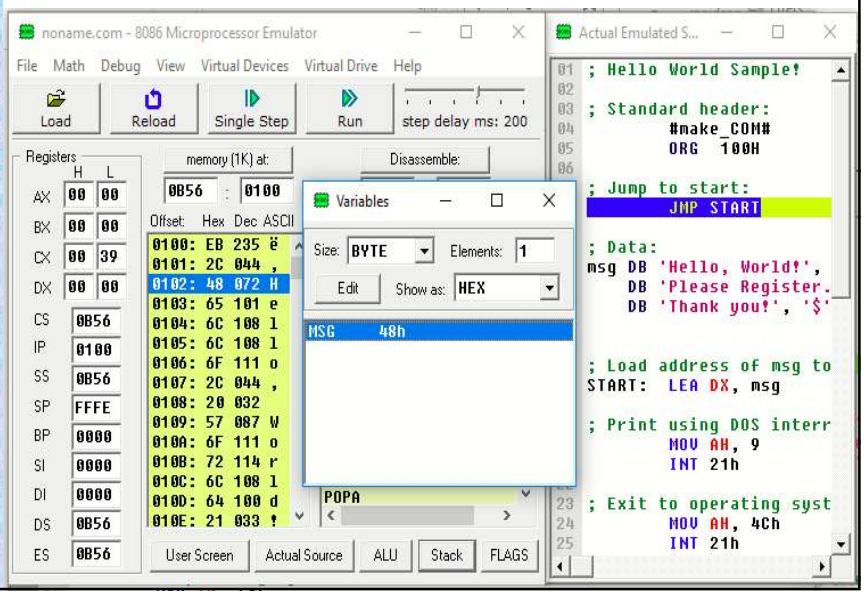
- **Les principales options disponibles:**
  - **Stack**: affiche la **pile**.
  - **Variable**: affiche la liste des **variables** et leurs **valeurs**.
  - **Memory**: affiche une partie de la **mémoire**. La visualisation en mode **table** permet plus facilement de se repérer.
  - **Flag**: affiche la valeur des différents **drapeaux**.
  - **Math**: ils existent deux outils pour les **conversions de base** et les **calculs**.

380

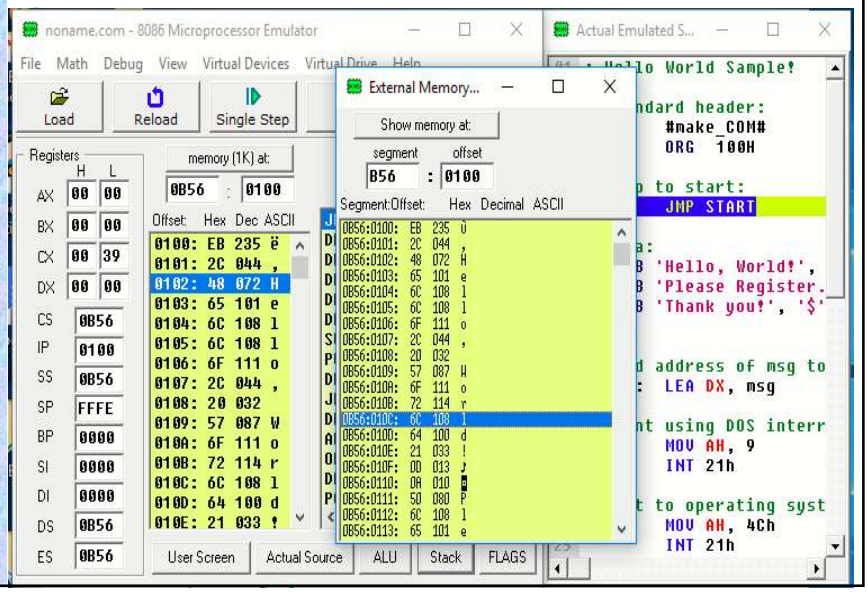
Assembleur



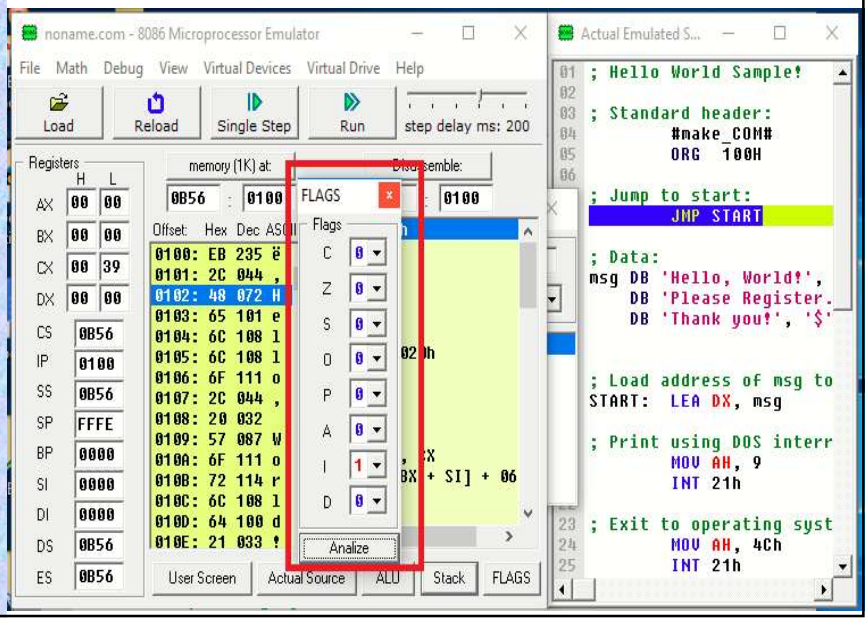
Assembleur



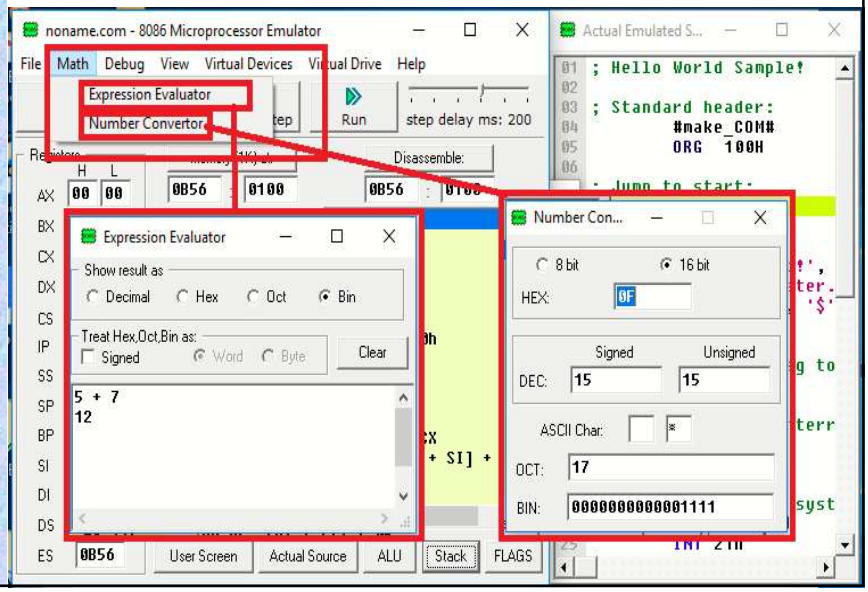
Assembleur



Assembleur

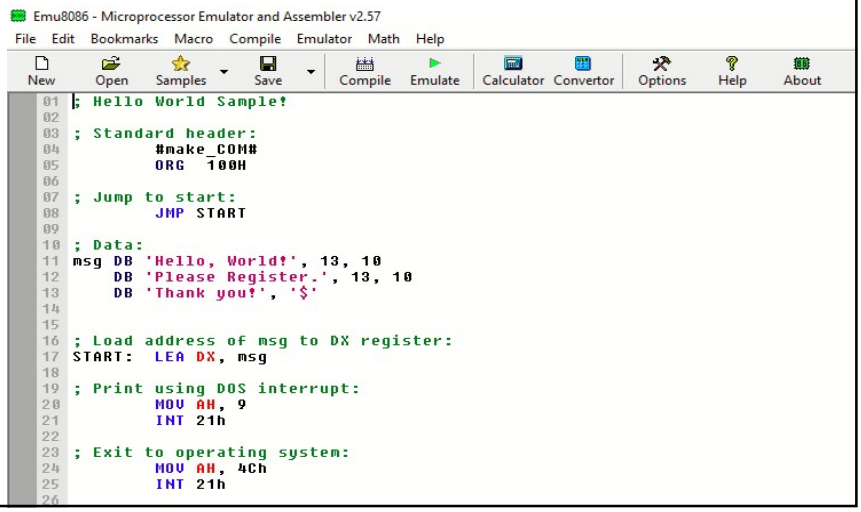


Assembleur



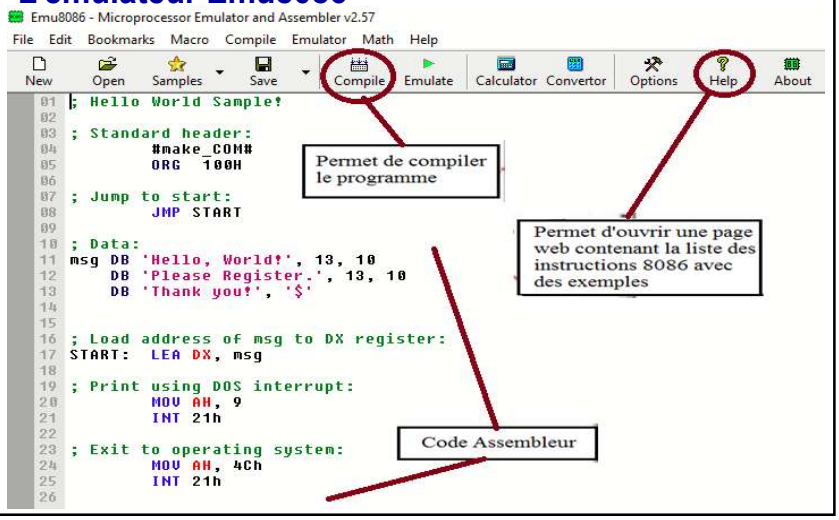
Assembleur

L'émulateur Emu8086



Assembleur

L'émulateur Emu8086



Assembleur

L'émulateur Emu8086

- Pour ouvrir un **nouveau programme**, il faut lancer **Emu8086.exe**, ensuite cliquer sur '**New**' et sélectionner **'COM Template'** ou **'EXE Template'**
- Dans les deux cas **certaines instructions** sont déjà **écrite**.
- Ensuite il suffit de **modifier ou remplacer** ce **code** afin d'écrire un **programme assembleur** réalisant un **algorithme** pour résoudre un problème

388

Assembleur

L'émulateur Emu8086

- Dans le cas **d'un fichier .exe** un fichier **squelette** apparaît sur la fenêtre de **l'émulateur**.
- Il contient les **initialisations** des **registres** de **données** (**DS**), de **pile** (**SS**) et de **code** (**CS**).
- Il contient aussi des **parties** pour déclarer les **données** et écrire le code nécessaires pour réaliser un **algorithme**

389

Assembleur

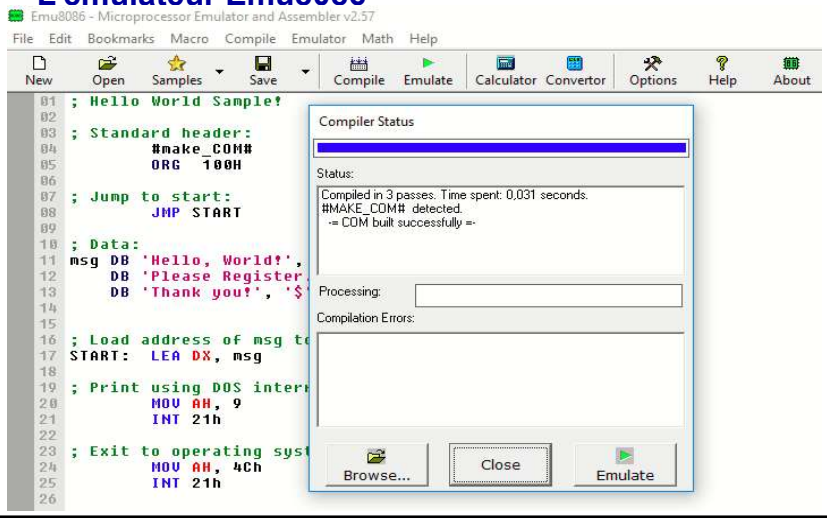
L'émulateur Emu8086

- **Une fois le code écrit** et **sauvegarder** sous un nom avec **l'extension .asm**, il faut le **compiler** en cliquant sur '**Compile**'
- **S'il ne contient pas d'erreurs**, alors il peut être **exécuter** en cliquant soit sur '**Emulate**' soit sur '**Run**'

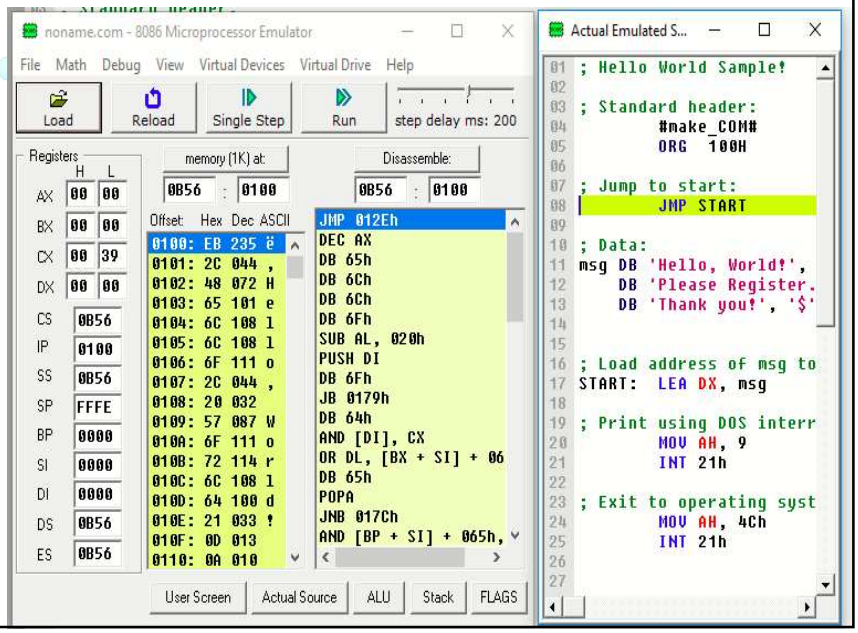
390

Assembleur

L'émulateur Emu8086



Assembleur



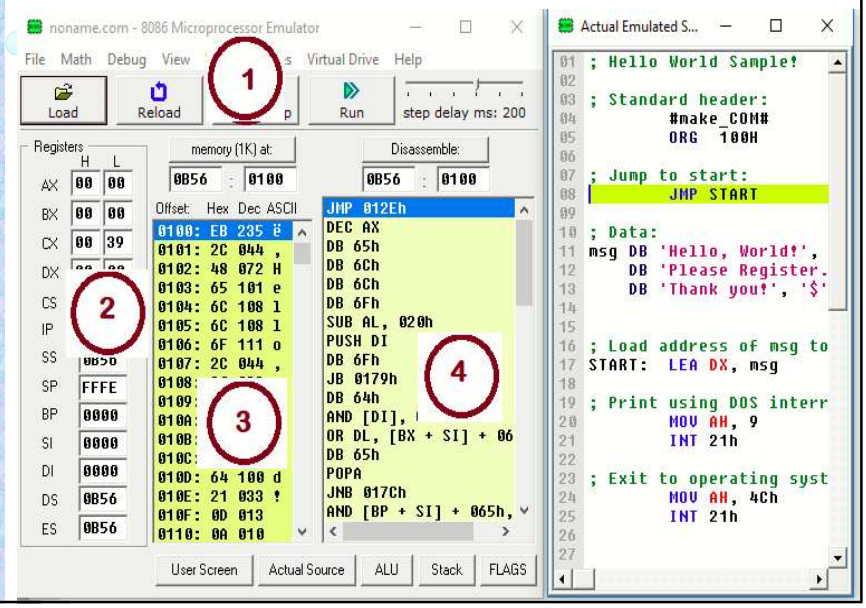
Assembleur

L'émulateur Emu8086

- **Après l'exécution** du code **deux nouvelles fenêtres** s'ouvrent définissant le **mode exécution de l'émulateur**.
- La **fenêtre *'original source code'*** contenant le **code écrit**. La **ligne surlignée en jaune** est la **prochaine** instruction qui va être exécutée. De plus, il est possible **d'interagir** avec ce code en cliquant dessus.
- La **fenêtre *'emulator'*** contenant entre autre la **zone mémoire** qui **change** pour correspondra à la partie sélectionnée par le **click**

393

Assembleur



Assembleur

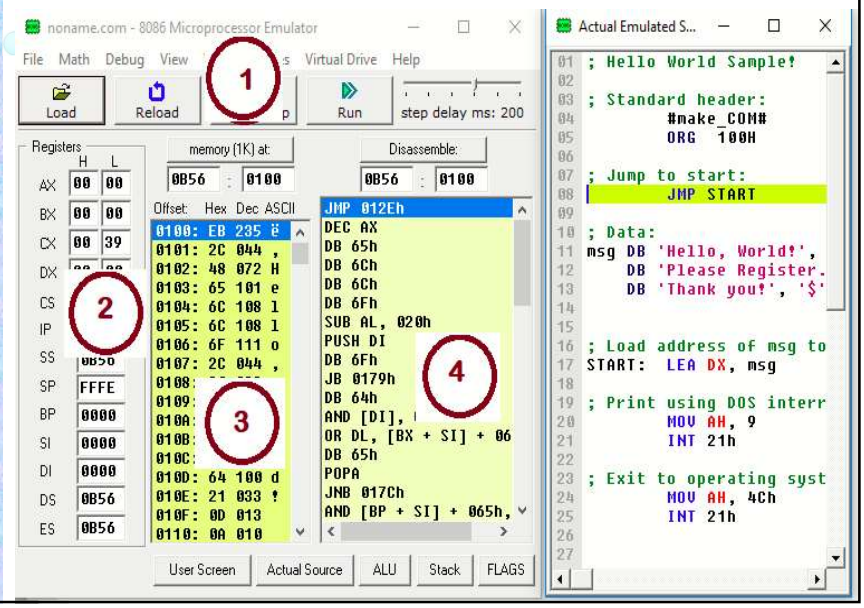
L'émulateur Emu8086

- La fenêtre '**emulator**' contient :

  1 - Les boutons qui exécute un **programme** soit **Pas à Pas**, avec la possibilité de **revenir en arrière**, soit **automatiquement** (en réglant un **délai d'attente** entre **chaque instruction**).

395

Assembleur



Assembleur

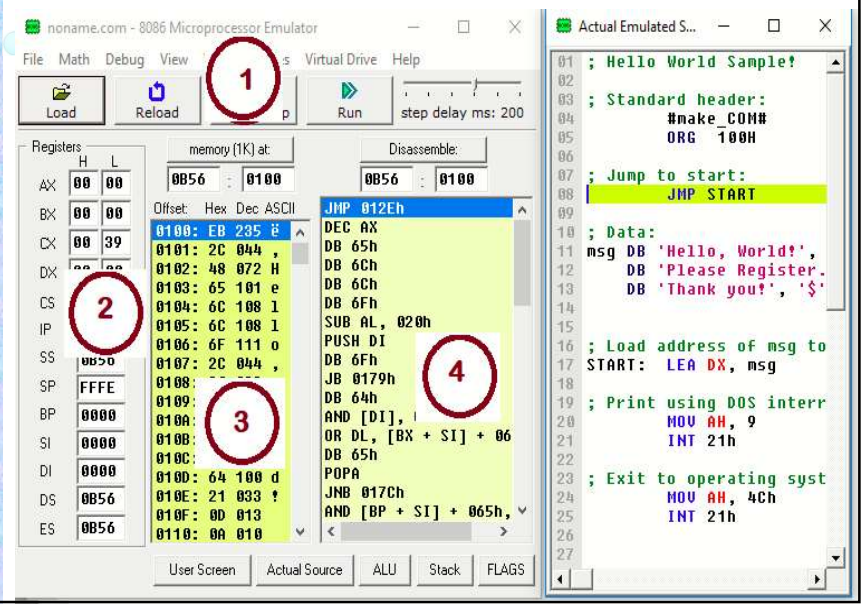
L'émulateur Emu8086

❑La fenêtre '**emulator**' contient :

2 - Les représentations des registres visualisant leur **valeur** hexadécimale tout au long de l'exécution du **programme**. Les registres **AX, BX, CX et DX** sont représenter **couper en deux**. Pour **visualiser** leur décomposition en registres **8 bits**. Il est **possible** de **modifier** la valeur des registres avec un **simple click** ou **ouvrir** une vue **détaillée** du registre avec **double click**.

397

Assembleur



Assembleur

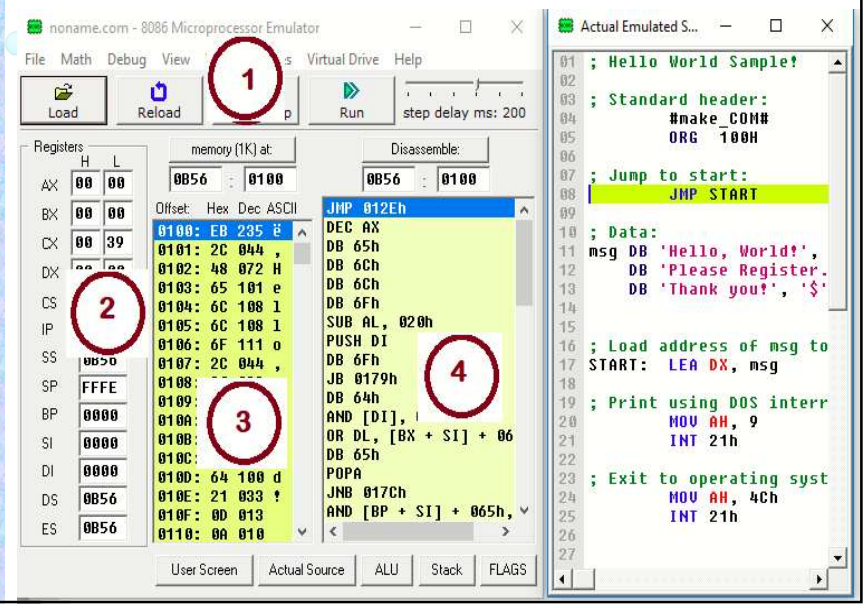
L'émulateur Emu8086

❑La fenêtre « emulator : » contient :

3 - La zone mémoire octet par octet. Sur une ligne, où une **adresse mémoire** est sur **20 bits**, la **valeur** de chaque octet est présentée en **hexadécimale**, en **décimale** et en **ASCII**.

399

Assembleur



Assembleur

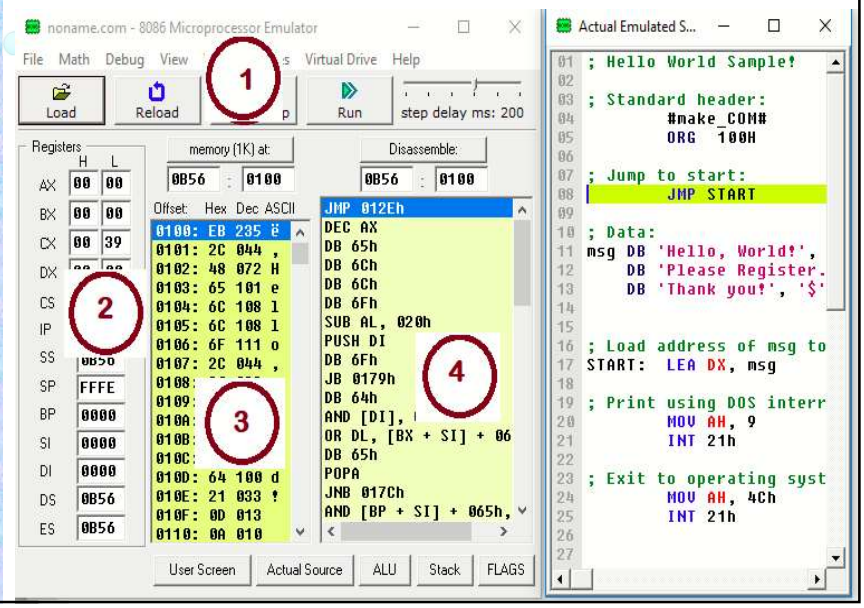
L'émulateur Emu8086

❑La fenêtre « *emulator* : » contient :

4 - Le code du programme une fois les **traductions** d'adresses **terminées**. Il n'y a plus aucune étiquette ni nom de variable. Il s'agit du **code** réellement **exécuté**.

401

Assembleur



Jeu d'instructions

Définition

- Chaque **microprocesseur** reconnait un **ensemble d'instructions machine** appelé **jeu d'instructions (Instruction Set)** fixé par le **constructeur** et supporté par le **processeur**.
- **Une instruction** est définie par son **code opératoire**, valeur numérique binaire difficile à manipuler par l'être humain.
- Une **notation symbolique est utilisée** pour représenter les **instructions** : les **mnémoniques**
- **Le jeu d'instruction** précise aussi quels sont les **registres** du **processeur** manipulable par le **programmeur**

403

Jeu d'instructions

Modes d'adressage

- **L'adressage** est la **méthode** de **localisation** des opérandes des **opérations**
- **Le mode d'adressage** est la **manière d'interpréter** les bits d'un champs d'adresse en vue de la **localisation** de l'opérande

404

Jeu d'instructions

Modes d'adressage

- Il existe **différentes façons** de spécifier l'adresse d'une case mémoire dans une instruction : ce sont **les modes d'adressage.**
  - **Adressage registre ou implicite**
  - **Adressage direct**
  - **Adressage indirect**
  - **Adressage basé**
  - **Adressage indixé**
  - **Adressage basé indixé**

405

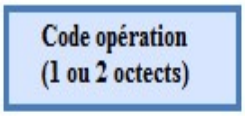
Jeu d'instructions

Adressage par registre ou implicite:

- Dans ce mode, il n'existe **aucun accès mémoire** pour les opérandes, l'instruction contient **seulement** le **code** opération, sur **1 ou 2 octets**, et l'instruction porte sur des registres ou spécifie une opération sans opérande

- **Syntaxe:** **MNQ RegDst, RegSrc**

  MNQ Reg



- **Exemples**

mov ax,bx : charge le contenu du registre BX dans le registre AX. Dans ce cas, le transfert se fait de registre à registre.

inc ax : incrémenter la valeur de ax , ax=ax+1

406

Jeu d'instructions

Adressage immédiat :

- C'est l'adressage dans lequel la valeur de l'opérande **figure directement** dans l'instruction **sans avoir besoin** de faire un nouvel **accès** mémoire.

- **Syntaxe:** **MNQ RegDst, Valeur**

- **Exemples :**

MOV AX, 2413 : charger le registre AX par le nombre décimal 2413

ADD AX, 0B24h : additionner le registre AX avec le nombre hexadécimal 2413

407

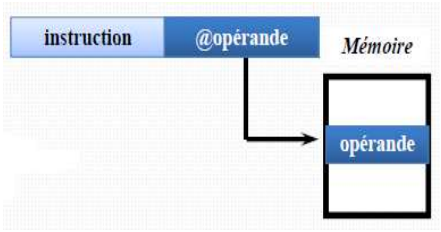
Jeu d'instructions

Adressage direct :

- Dans cet adressage un des opérandes est **l'emplacement mémoire** dont **l'adresse** figure dans l'instruction
- une fois l'instruction **lue**, il faut aller faire un **accès** mémoire pour obtenir l'opérande **désiré**
- **Syntaxe:**

MNQ RegDst, [Adresse]

MNQ [Adresse], Reg/Val



Exemples :

MOV AX, [2413h] : charger AX avec le contenu de l'@:2413

MOV [2413h] , 0B24h : Copie le nombre 0B24h dans l'@:2413

408

Jeu d'instructions

Spécificateur de format

- Avec cette **adressage**, il faut **indiquer le format** de la donnée: **1 ou 2 octets** car le microprocesseur 8086 peut manipuler **des données sur 8 ou 16 bits**
- Il faut utiliser un **spécificateur de format** :
  - **mov byte ptr [1100H],65H :** transfère la valeur 65H (sur 1 octet) dans la case mémoire d'offset 1100H ;
  - **mov word ptr [1100H],65H :** transfère la valeur 0065H (sur 2 octets) dans les cases mémoire d'offset 1100H et 1101H.

409

Jeu d'instructions

Adressage indirect :

- Un des deux **opérandes** se trouve en **mémoire**. **L'offset** de **l'adresse** n'est pas précisé **directement** dans l'instruction, il se trouve dans l'un des registres **de base ou d'index (BX, BP, SI, DI): 2 accès mémoires** à condition que le type de la donnée soit **accordé** avec le registre utilisé
- **Syntaxe:**

  MNQ R , [Rseg : Roff]

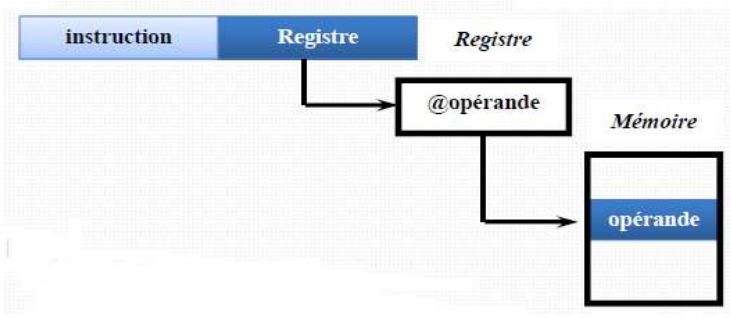
  MNQ [Rseg : Roff] , R

  MNQ table[Rseg : Roff],R
- Si **Rseg n'est pas spécifié**, le segment **par défaut** sera utilisé.

410

Jeu d'instructions

Adressage indirect :



- Il existe **trois variantes** de cette **adressage** selon le registre **d'offset** utilisé. On distingue ainsi, l'adressage **Basé**, l'adressage **indexé** et l'adressage **basé indexé**.

411

Jeu d'instructions

Exemples d'adressage indirect :

- **MOV AX, [BX] ;** Charger AX par le contenu de la mémoire d'adresse DS:BX
- **MOV AX, [BP] ;** Charger AX par le contenu de la mémoire d'adresse SS:BP
- **MOV AX, [SI] ;** Charger AX par le contenu de la mémoire d'adresse DS:SI
- **MOV AX, [DI] ;** Charger AX par le contenu de la mémoire d'adresse DS:DI
- **MOV AX, [ES:BP] ;** Charger AX par le contenu de la mémoire d'adresse ES:BP
- **MOV [DI],AX ;** Charger la mémoire d'adresse DS:DI par le contenu de AX
- **MOV [BX],5h ;** Charger la mémoire d'adresse DS:BX par la valeur 5h

412

Jeu d'instructions

Adressage basé :

- **L'offset** est contenu dans un **registre de base BX ou BP**. On peut préciser un **déplacement** (dep) qui sera **ajouté** au contenu de Rb pour **déterminer l'offset**.
- **Syntaxe**

  MNQ R , [Rseg : Rb+dep]

  MNQ [Rseg : Rb+dep] , R

  MNQ Table [Rseg : Rb+dep]

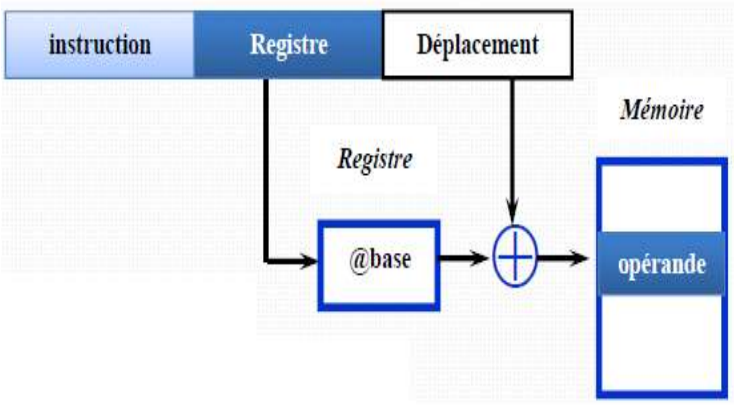
- **Exemples** :

  mov al,[bx] : transfère la donnée dont l'offset est contenu dans BX vers AL, le segment associé par défaut au registre BX est le segment de données : on dit que l'**adressage** est **basé sur DS**;

413

Jeu d'instructions

Adressage basé :



414

Jeu d'instructions

Exemples d'adressage basé :

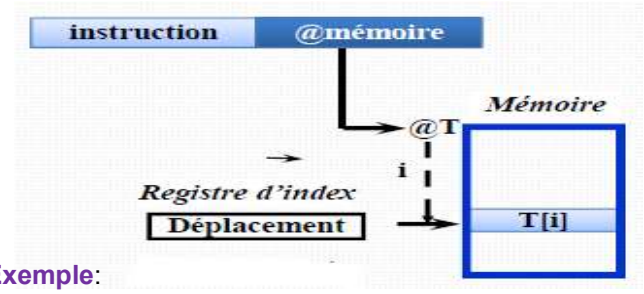
- **MOV AX, [BX]** : Charger AX par le contenu de la mémoire d'adresse DS:BX
- **MOV AX, [BX+5]** : Charger AX par le contenu de la mémoire d'adresse DS:BX+5
- **MOV AX, [BP-200]** : Charger AX par le contenu de la mémoire d'adresse SS:BP-200
- **MOV AX, [ES:BP]** : Charger AX par le contenu de la mémoire d'adresse ES:BP

415

Jeu d'instructions

Adressage indexé :

- Il utilise un **registre d'index SI ou DI** plus un **déplacement** (**L'adressage** est noté par des **crochets**). Cet adressage est **utile** pour le **parcours** de **tableaux**



- **Exemple**:
  MOV SI,2 puis **MOV AX, T[SI]**

416

Jeu d'instructions

Adressage basé et indexé :

- **l'offset de l'adresse** de **l'opérande** est obtenu en faisant la **somme** d'un **registre de base**, d'un **registre d'index** et d'une **valeur constante**.
- **Exemple** :
  - **mov ah,[bx+si+10H]; ah** est chargé par la mémoire d'@ ds:bx+si+10h
  - **MOV AX,[BX+SI]** ; AX est chargé par la mémoire d'adresse DS:BX+SI
  - **MOV AX,[BX+DI+5]** ; AX est chargé par la mémoire d'adresse DS:BX+DI+5
  - **MOV AX,[BP+SI-8]** ; AX est chargé par la mémoire d'adresse SS:BP+SI-8
  - **MOV AX,[BP+DI]** ; AX est chargé par la mémoire d'adresse SS:BP+DI

417

Jeu d'instructions

Code instruction : champs MOD et R/M

- **MOD = 00 : Adressage sans déplacement**

R/M	Adresse effective	Mode d'adressage	Segment
000	[BX+SI]	Adressage basé indexé	DS
001	[BX+DI]	Adressage basé indexé	DS
010	[BP+SI]	Adressage basé indexé	SS
011	[BP+DI]	Adressage basé indexé	SS
100	[SI]	Adressage indexé	DS
101	[DI]	Adressage indexé	DS
110	Depl.L Depl.H	Adressage direct	DS
111	[BX]	Adressage basé	DS

➔MOD =00 et R/M=110 : Adressage direct, l'adresse (offset), octet bas puis octet haut, est spécifiée sur les bytes 3 et 4 du code d'instruction.

418

Jeu d'instructions

Code instruction : champs MOD et R/M

- **MOD = 01 : Adressage avec déplacement sur 8 bits**

R/M	Adresse effective	Mode d'adressage	Segment
000	[BX+SI±Depl.8]	Adressage basé indexé avec déplacement sur 8 bits	DS
001	[BX+DI±Depl.8]	Adressage basé indexé avec déplacement sur 8 bits	DS
010	[BP+SI±Depl.8]	Adressage basé indexé avec déplacement sur 8 bits	SS
011	[BP+DI±Depl.8]	Adressage basé indexé avec déplacement sur 8 bits	SS
100	[SI±Depl.8]	Adressage indexé avec déplacement sur 8 bits	DS
101	[DI±Depl.8]	Adressage indexé avec déplacement sur 8 bits	DS
110	[BP±Depl.8]	Adressage basé avec déplacement sur 8 bits	SS
111	[BX±Depl.8]	Adressage basé avec déplacement sur 8 bits	DS

Jeu d'instructions

Code instruction : champs MOD et R/M

- **MOD = 10 : Adressage avec déplacement sur 16 bits.**

R/M	Adresse effective	Mode d'adressage
000	[BX+SI+Depl.L Depl.H]	Adressage basé indexé avec déplacement sur 16 bits
001	[BX+DI+ Depl.L Depl.H]	Adressage basé indexé avec déplacement sur 16 bits
010	[BP+SI+ Depl.L Depl.H]	Adressage basé indexé avec déplacement sur 16 bits
011	[BP+DI+ Depl.L Depl.H]	Adressage basé indexé avec déplacement sur 16 bits
100	[SI+ Depl.L Depl.H]	Adressage indexé avec déplacement sur 16 bits
101	[DI+ Depl.L Depl.H]	Adressage indexé avec déplacement sur 16 bits
110	[BP+ Depl.L Depl.H]	Adressage basé avec déplacement sur 16 bits
111	[BX+ Depl.L Depl.H]	Adressage basé avec déplacement sur 16 bits

Jeu d'instructions

Code instruction : champs MOD et R/M

- **MOD = 11 : Instructions sur deux registres ou adressage immédiat :**

MOD	REG	R/M
11	Destination	Source

421

Jeu d'instructions

Code instruction : champ REG

- Le champ REG (et R/M dans le cas où MOD=11) contient un nombre binaire de 0 à 7, il est défini par :

REG	Registre 16 (w=1)	Registre 8 (w=0)	Registre segment
000	AX	AL	ES
001	CX	CL	CS
010	DX	DL	SS
011	BX	BL	DS
100	SP	AH	
101	BP	CH	
110	SI	DH	
111	DI	BH	

422

Jeu d'instructions

Détermination du code d'une instruction

Instruction	Champs du code	Valeurs des champs	Code hex.
MOV DX, AX	100010dw mod reg r/m	100010 1 1 11 010 000	8B D0
PUSH BX	01010 reg	01010 011	53
LEA AX, [BX+0100h]	10001101 mod reg r/m	10001101 10 000 111 (00 01)h	8D 87 00 01
ADD [BX+SI], AL	000000 dw mod reg r/m	000000 0 0 00 000 000	00 00
ADD CX, 2105 h	100000 sw mod 000 r/m data data (if s:w=01)	100000 0 1 11 000 001 (05 21)h	81 C1 05 21

423

Jeu d'instructions

Les instructions de transfert

- Elles permettent de **déplacer** des **données** d'une **source** vers une **destination** :
  - **registre vers mémoire ;**
  - **registre vers registre ;**
  - **mémoire vers registre.**
- **Le microprocesseur 8086** n'autorise pas les transferts de **mémoire vers mémoire** (pour ce faire, il faut passer par un registre intermédiaire).
- **Syntaxe : MOV destination , source**
- **MOV** est l'abbréviation du verbe **« to move »**

424

Jeu d'instructions

Les instructions de transfert

- **MOV registre/variable, registre**
- **MOV registre, registre/variable**
- **MOV registre, registre de segment**
- **MOV registre de segment, registre**
- **MOV registre/variable, constante**

➔**Aucun mode d'adressage** de **MOV** ne permet de **transférer** une valeur d'un **registre de segment** vers un autre **registre de segment** ou d'une **adresse vers une autre**

425

Jeu d'instructions

Instruction de transfert pour Pile

❑ Instruction PUSH : PUSH Op

- **Empiler l'opérande Op** (Op doit être un opérande de **16 bits**)
- **Décrémente SP** de **2**
- **Copie Op** dans la **mémoire pointée** par **SP**
- **PUSH R16** ; R16=Registre sur 16 bits
- **PUSH word [adr]**
- **PUSH Val_Immédiate**

426

Jeu d'instructions

Instruction de transfert pour Pile

❑ Instruction POP : POP Op

- **Dépiler** dans **l'opérande Op** (Op doit être un opérande **16 bits**)
- **Copie** les **deux cases mémoire** pointée par **SP** dans **l'opérande Op**
- **Incrémente SP** de **2**
- **POP R16**
- **POP word M**

427

Jeu d'instructions

Exemple de transfert dans la pile

- **push ax** ; empilement du registre ax
- **push bx** ; empilement du registre bx
- **push [1100H]** ; empilement et de la case mémoire 1100H-1101H
- **pop [1100H]** ; dépilement dans l'ordre inverse de l'empilement
- **pop bx** ; dépilement dans bx
- **pop ax** ; dépilement dans ax

428

Jeu d'instructions

Instruction de transfert pour Pile

Instruction PUSHA :

- **Empile** tous les registres **généraux** et **d'adressage** dans l'ordre suivant : **AX, CX, DX, BX, SP, BP, SI, DI**

Instruction POPA :

- **Dépile** tous les registres **généraux** et **d'adressage** dans **l'ordre inverse** de **PUSHA** afin que chaque registre retrouve sa valeur. La valeur dépilée de SP est **ignorée**.

➔Les deux instructions PUSHA et POPA ne sont pas **reconnues par le 8086**. Elles ont été introduites à partir du **80186**.

429

Jeu d'instructions

Instruction d'échange

❑ **Instruction XCHG : XCHG OpDst , OpSrc**

- **Echange** l'Opérande **Source** avec l'Opérande **Destination**. **Impossible sur segment.**
- **XCHG Reg1 , Reg2**
- **XCHG Reg , [adr]**
- **XCHG [adr] , Reg**
- **XCHG [adr] , [adr]**

430

Jeu d'instructions

Instruction de chargement d'offset

❑ **Instruction LEA : LEA OpDst , OpSrc**

- Charge **l'offset** de la donnée **référencée** dans **l'opérande source** qui est, en général, une donnée déjà déclarée dans le segment de données.
- **LEA Reg , label**
- **LEA [adr] , label**

431

Jeu d'instructions

Les instructions arithmétiques

- Les instructions arithmétiques de base sont
  - **Addition,**
  - **Soustraction,**
  - **Multiplication,**
  - **Division**
- Plusieurs variantes et plusieurs modes d'adressage

432

Jeu d'instructions

Addition sans retenue: ADD opDst,opSrc

- L'opération effectuée est : **opDst = opDst + opSrc sans report de retenue**. La **retenue** est positionnée en fonction du **résultat** de l'opération
- **Syntaxe:** **ADD registre/variable, registre**
  ADD registre, registre/variable
  ADD registre/variable, constante
- **Exemple:**

mov AL , 0a9h puis **add AL , 72h** ➔ AL=1bh, C=1, A=0

mov AL , 09h puis **add AL , 3ah** ➔ AL=43h, C=0, A=1.

433

Jeu d'instructions

Autres exemples d'addition:

- **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 base) ;
- **add byte ptr [1200H],05H** : ajoute la valeur 05H au contenu de la case mémoire d'offset 1200H (adressage immediat).

434

Jeu d'instructions

Soustraction sans retenue: SUB opDst,opSrc

- L'opération effectuée est : **opDst = opDst - opSrc sans report** de **retenue**. La **retenue** est positionnée en fonction du **résultat** de l'opération
- **Syntaxe:** **SUB registre/variable, registre**
  SUB registre, registre/variable
  SUB registre/variable, constante
- **Exemple:**

mov AL , 39h puis **sub AL , 18h** ➔ AL= 21h, Z=S=C=0
mov AL , 26h puis **sub AL , 59h** ➔ AL= cdh, Z=0, C=S=A=1.

435

Jeu d'instructions

Multiplication : MUL/IMUL op8/op16

- L'instruction **IMUL** effectue la **multiplication d'opérandes signés**. L'instruction **MUL** effectue la **multiplication d'opérandes non signés**.
- Les indicateurs de **retenue** (CF) et de **débordement** (OF) sont mis à **un** si le résultat **ne peut pas** être stockée dans **l'opérande destination**.
- une **multiplication** de deux **données** de **n bits** donne un résultat sur **2n bits**
- **Syntaxe:** **MUL registre/variable**

  IMUL registre/variable

436

Jeu d'instructions

Multiplication : MUL/IMUL op8/op16

- **AX = AL * Op8 ou DX:AX = AX * Op16**
- **Multiplication** du contenu de **AL** par un opérande sur **1 octet** ou du contenu de **AX** par un opérande sur **2 octets**.
- Le **résultat** est placé dans:
  - **AX** si les données à multiplier sont sur **1 octet** car le résultat est sur 16 bits)
  - **(DX,AX)** si elles sont sur **2 octets** (résultat sur 32 bits).
- Les **drapeaux CF et OF** sont positionnés si **AH=0** pour la multiplication **8 bits** ou **DX=0** pour la multiplication **16 bits**

437

Jeu d'instructions

Multiplication : MUL/IMUL op8/op16

- **Exemple:**
  - **mov al, 4** puis **mov ah, 25** puis **imul ah** : AX=4*25=100
  - **mov BX , 435** puis **mov AX , 2372** puis **imul BX** :
    - AX=be8ch, DX=fh le résultat est donc fbe8ch
    - Les deux données étant positives, le résultat est le même que l'on utilise l'instruction IMUL ou l'instruction MUL.
  - **mov BX , -435** puis **mov AX , 2372** puis **imul BX** :
    - AX=4174h et DX=fff0h, soit la valeur négative -1031820.
    - Si l'on remplace IMUL par MUL, le résultat n'a pas de sens.

438

Jeu d'instructions

Division : DIV/IDIV op8/op16

- L'instruction **DIV** effectue la **division** sur des données **non signées**, **IDIV** sur des données **signées**.
- le **dividende** est implicite. Le **diviseur** est fourni en **opérande**.
- Le résultat se compose du **quotient** et du **reste** de la division. Le **quotient** a la même **signe** que le **dividende** et le **reste** est toujours **inférieur** au **diviseur**
- **Syntaxe:** **DIV registre/variable**

  IDIV registre/variable

439

Jeu d'instructions

Division : **DIV/IDIV op8/op16**

- **DIV op8 ; ⇔ AX/op8 , AL=Quotient et AH=Reste**
- **DIV op16 ;⇔DX:AX/op16 , AX=Quotient et DX=Reste**
- **Division** du contenu de **AX** par un opérande sur **1 octet** ou le contenu de **(DX,AX)** par un opérande sur **2 octets**.
  - Si l'opérande est sur **1 octet**, alors **AL = quotient et AH = reste** ;
  - Si l'opérande est sur **2 octets**, alors **AX = quotient et DX = reste**.
- Pour les deux instructions, si le diviseur de la division est **nul**, un **message Division par zéro** sera affiché **automatiquement**.

440

Jeu d'instructions

Division : **DIV/IDIV op8/op16**

- **Exemple:**
  - **mov AX , 37** puis **mov bx, 5** puis **div Bx :** AX = 7, DX=2.
  - **mov AX , 37** puis **mov bx, 5** puis **div bl :** AL = 7, AH=2.
  - **mov BX , 435** puis **mov AX , 2372** puis **div BX :** AX = 5 , DX=197
  - **mov BX , 435** puis **mov AX , 2372** puis **idiv BX :** AX = 5 , DX=197
  - **mov BX , -435** puis **mov AX , 2372** puis **idiv BX :** AX = -5= FFFDh , DX=197. Si l'on remplace IDIV par DIV, le résultat n'a pas de sens.

441

Jeu d'instructions

Incrémentation/Décrémentation INC/DEC op8/op16:

- **Syntaxe** : **INC registre/variable**

  DEC registre/variable
- **INC ajoute 1** au contenu de l'opérande et **DEC soustrait 1** au contenu de l'opérande,, **sans modifier** l'indicateur de **retenue**.
- Pour incrémenter ou décrémenter une case mémoire, il faut **préciser la taille** :
  - **INC byte [ ]**
  - **INC word [ ]**

442

Jeu d'instructions

Incrémentation/Décrémentation INC/DEC op8/op16:

- **Exemple** :
  - **mov al, 3fh** puis **inc al** :AL contient ensuite la valeur 40h. Le bit (Z) est mis à 0 (le résultat de l'incrémentation n'est pas nul), l'indicateur de retenue auxiliaire, le bit (A) est mis à 1 (passage de retenue entre les bits 3 et 4 lors de l'incrémentation), les bits (O) et bit (S) sont mis à 0 (pas de débordement de capacité, le signe du résultat est positif).
  - **mov al, 01h** puis **dec al** : AL contient ensuite la valeur 00. Le bit (Z) est mis à 1, les bits (O), (P), (A) et (S) mis à 0.

443

Jeu d'instructions

Autres instructions arithmétiques :

- **ADC** : **addition avec retenue** ;

  OpDst = OpDst + OpSrc + C

- **SBB** : **soustraction avec retenue** ;

  OpDst = OpDst – (OpSrc + C)

- L'indicateur **OF** permet de détecter le **débordement**

444

Jeu d'instructions

Les instructions logiques

- Ce sont des instructions qui permettent de manipuler des données au niveau des **bits**.
- Les opérations logiques de base sont :
  - **Le ET logique**
  - **Le OU logique**
  - **Le OU exclusif ;**
  - **Le complément à 1;**
  - **Le complément à 2;**
  - **Les décalages et les rotations.**
- Les différents modes d'adressage sont possibles

445

Jeu d'instructions

Le ET logique: AND op1,op2

- L'opération effectuée est **op1 = op1 ET op2**, et réalise un ET logique bit à bit entre l'opérande source et l'opérande destination
- **Syntaxe :**

AND registre/variable, registre

AND regisre, registre/variable

AND registre/variable, constante

446

Jeu d'instructions

Le ET logique: AND op1,op2

- **Exemple :**

```
mov al, 36h
and al, 5Ch
```

- Le registre AL contient ensuite la valeur 14h obtenue de la manière suivante ;

```
  36h    0011 0110
^ 5Ch    0101 1100
  14h    0001 0100
```

- Les bits (S) et (Z) sont mis à zéro et le bit (P) à 1.

447

Jeu d'instructions

Application ET logique :

- **Masquage** de bits pour mettre à zéro certains bits dans un mot.
- **Exemple** : masquage des bits 0, 1, 6 et 7 dans un octet :

```
  0011 0110
^ 0011 1100
  0011 0100
```

448

Jeu d'instructions

Le OU logique : OR op1,op2

- **L'opération** effectuée est **: op1 = op1 OU op2,** et réalise un **OU logique** bit à bit entre **l'opérande source et l'opérande destination.**
- **Syntaxe :**

OR registre/variable, registre

OR regisre, registre/variable

OR registre/variable, constante

449

Jeu d'instructions

Le OU logique: OR op1,op2

- **Exemple :**

```
mov al, 36h
or al, 5ch
```

- Le registre AL contient ensuite la valeur 7Eh obtenue de la manière suivante ;

```
 36h    0011 0110
v 5ch   0101 1100
 7eh    0111 1110
```

- Les **bits (SF) et (ZF)** sont **mis à zéro** et le **bit (PF)** à **1**.

450

Jeu d'instructions

Application OU logique :

- **Mise à 1 d'un ou plusieurs** bits dans un mot.
- Dans le mot 10110001B on veut mettre à 1 les bits 1 et 3 sans modifier les autres bits.

```
  1011 0001
v 0000 1010
  1011 1011
```

451

Jeu d'instructions

Le OU exclusif : XOR op1,op2

- **L'opération** effectuée est **: op1 = op1 OUX op2,** et réalise **un OU exclusif** bit à bit entre **l'opérande source et l'opérande destination**.
- **Syntaxe :**

XOR registre/variable, registre

XOR registre, registre/variable

XOR registre/variable, constante

452

Jeu d'instructions

Le OU exclusif : XOR op1,op2

- **Exemple :**

```
mov al, 36h
xor al, 5ch
```

- Le registre AL contient ensuite la valeur 6Ah obtenue de la manière suivante ;

	36h	0011 0110
⊕	5Ch	0101 1100
	6Ah	0110 1010

- Les bits (S) et (Z) sont mis à zéro et le bit (P) à 1.

453

Jeu d'instructions

La négation logique (Complément à 1) : NOT op

- **L'opération** effectuée est : **op = ¬ op** et **transforme** la valeur d'un registre ou d'un **opérande** mémoire en son **complément logique bit à bit**.
- **Exemple** :

```
mov al, 36h
not al
```

- Le registre AL contient ensuite la valeur C9h obtenue de la manière suivante ;

¬ 36h	0011 0110
C9h	1100 1001

- Les **bits (SF) et (PF)** sont **mis à 1**, le **bit (ZF)** à **0**.

454

Jeu d'instructions

L'opposé (Complément à 2) : NEG op

- **L'opération** effectuée est : **op = - op** et transforme la valeur d'un registre ou d'un **opérande** mémoire en **son complément à deux** .
- **Exemple** :

```
mov al, 35h
neg  al
```

- Le registre AL contient ensuite la valeur C9h obtenue de la manière suivante ;

¬ 35h	0011 0101
C8h	1100 1010
+	1
	1100 1011

455

Jeu d'instructions

Instructions de décalages et de rotations :

- Ces **instructions déplacent** d'un certain **nombre** de positions les bits d'un mot vers la **gauche** ou vers la **droite**.
- Dans les **décalages**, les bits qui sont déplacés sont **remplacés** par des **zéros ou 1**.
  - **Décalages logiques (opérations non signées)**
  - **Décalages arithmétiques (opérations signées).**
- Dans les **rotations**, les bits déplacés dans un **sens** sont **réinjectés** de **l'autre côté** du mot.

456

Jeu d'instructions

Instructions de décalages et de rotations :

- Il faut noter que pour un nombre,
  - **Un décalage** d'une position vers la **gauche** correspond à une **multiplication** de ce nombre par 2
  - **Un décalage** d'une position vers la **droite** correspond à une **division** entière par 2.
- Généralement, un **décalage de k positions vers la gauche** correspond à une **multiplication par 2^k** et un **décalage de k positions vers la droite** correspond à une **division par 2^k** .

457

Jeu d'instructions

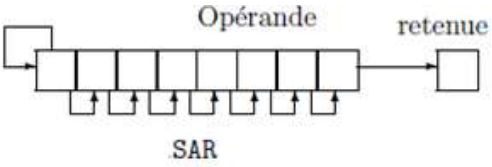
Décalage arithmétique vers la droite :

L'instruction SAR (Shift Arithmetic Right) :

SAR registre/variable, 1

SAR registre/variable, CL

Elle effectue un **décalage 1 ou CL fois vers la droite**, sauvegardant le bit de poids faible dans l'indicateur de retenue. Cependant le bit de poids fort est conservé et serais donc dupliqué.



458

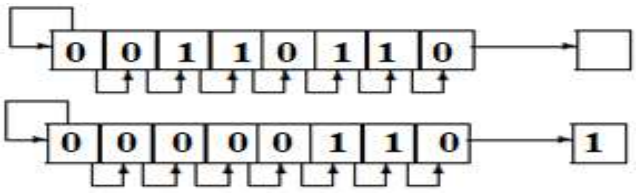
Jeu d'instructions

- **Décalage arithmétique vers la droite :**

Exemple SAR :

```
mov al, 36h
mov cl, 3
sar al, cl
```

- Le registre AL contient ensuite la valeur 06h. Le bit (C) est mis à 1.



459

Jeu d'instructions

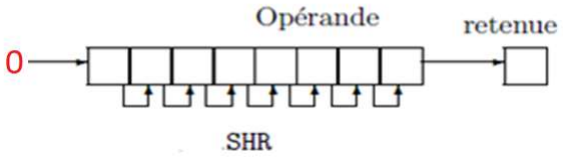
Décalage logique vers la droit :

L'instruction SHR (SHift logical Right) :

SHR registre/variable, 1

SHR registre/variable, CL

- SHR effectue un **décalage 1 ou CL fois vers la droite**. Le bit de poids faible est mis dans la retenue. Un 0 est mis dans le bit de poids fort de la donnée



460

Jeu d'instructions

Décalage vers la droite :

Différence entre SAR et SHR:

- Elle n'apparaît que si le bit de poids fort de la donnée vaut 1.
- Si al contient 70h=01110000b alors
  - **sar AL, 1** → AL contient 38h=00111000b
  - **shr AL, 1** → AL contient 38h=00111000b
- Si al contient f0=11110000b alors
  - **sar AL, 1** → AL contient f8h=11111000b
  - **shr AL, 1** → AL contient 78h=01111000b

461

Jeu d'instructions

Décalage vers la droite :

Optimisation:

- Il est préférable d'utiliser l'instruction *SHR* plutôt que *DIV* si vous effectuer des divisions par 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536,... de nombre naturel.

462

Jeu d'instructions

Décalage arithmétique ou logique vers la gauche:

Les instructions SAL (Shift Arithmetic Left) et SHL (Shift logical Left) :

SAL registre/variable, 1 ou SHL registre/variable, 1

SAL registre/variable, CL ou SHL registre/variable, CL

- Elles effectuent un décalage 1 ou CL fois vers la gauche, sauvegardant le bit de poids fort dans l'indicateur de retenue et mettant un 0 dans le bit de poids faible

463

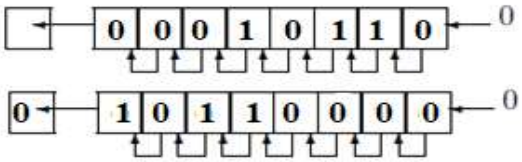
Jeu d'instructions

Décalage arithmétique ou logique vers la gauche:

Exemple SAL ou SHL :

```
mov al, 16h
mov cl, 3
sal al, cl
```

- Le registre AL contient ensuite la valeur B0h. En effet, 16h s'écrit 00010110 en binaire. Si on décale cette valeur de trois positions vers la gauche, on obtient 10110000, soit B0h Le bit (C) est mis à 0.



464

Jeu d'instructions

Applications des instructions de décalage :

- **Cadrage à droite d'un groupe de bits.**
  - **Exemple** : on veut avoir la valeur du quartet de poids fort du registre AL :

```
mov al,11001011B
mov cl,4
shr al,cl
```

- **Test de l'état d'un bit dans un mot.**
  - **Exemple** : on veut déterminer l'état du bit 5 de AL :

```
mov cl,6        mov cl,3
shr al,cl       shl al,cl
```

  - avec un décalage de 6 positions vers la droite ou 3 positions vers la gauche, le bit 5 de AL est transféré dans l'indicateur de retenue CF. Il suffit donc de tester cet indicateur.

465

Jeu d'instructions

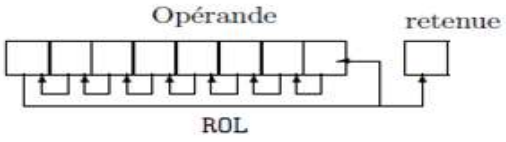
Rotation à gauche

L'instruction ROL (ROtate Left) :

ROL registre/variable, 1

ROL registre/variable, CL

- **ROL** effectue une **rotation** à **gauche** de l'opérande destination indiqué, **1 ou CL fois**, sans prendre en compte le contenu de **l'indicateur** de **retenue**. Le **bit** de **poids fort** est mis dans **l'indicateur** de **retenue** lors de la rotation ainsi que dans le **bit** de **poids faible** de **l'opérande**.



466

Jeu d'instructions

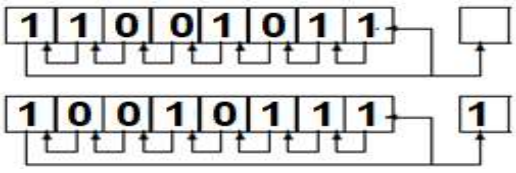
Rotation à gauche

Exemple de l'instruction ROL :

mov al,11001011B

rol al,1

- → **Réinjection** du **bit** sortant qui est **copié** dans **l'indicateur** de **retenue CF**.



467

Jeu d'instructions

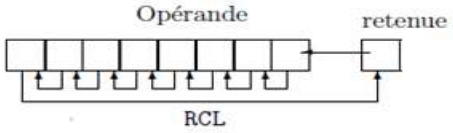
Rotation à gauche avec retenue

L'instruction RCL (Rotate Left through Carry) :

RCL registre/variable, 1

RCL registre/variable, CL

- **RCL** effectue une **rotation** à **gauche** de **l'opérande destination** indiqué, **1 ou CL fois**, en prenant en compte le contenu de **l'indicateur** de **retenue**. Le **bit** de **poids fort** de **l'opérande** destination est mis dans la **retenue**. Le contenu de la retenue est mis dans le **bit** de **poids faible** de **l'opérande destination**.



468

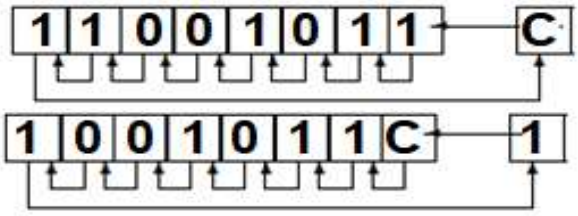
Jeu d'instructions

Rotation à gauche avec retenue

Exemple de l''instruction RCL:

```
mov al,11001011B
rcl al,1
```

- le bit **sortant** par la **gauche** est **copié** dans l'indicateur de retenue **CF** et la valeur **précédente** de CF est **réinjectée** par la **droite**.



469

Jeu d'instructions

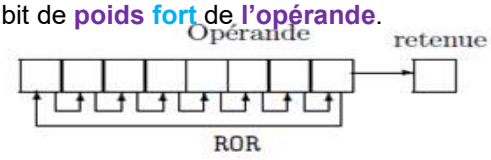
Rotation à droite

L'instruction ROR (ROtate Right) :

ROR registre/variable, 1

ROR registre/variable, CL

- **ROR** effectue une **rotation** à **droite** de **l'opérande** destination indiqué, **1 ou CL fois**, sans prendre en compte le contenu de l'indicateur de **retenue**. Le bit de **poids faible** est mis dans **l'indicateur** de **retenue** lors de la rotation ainsi que dans le bit de **poids fort** de **l'opérande**.



470

Jeu d'instructions

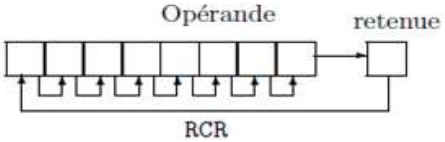
Rotation à droite avec retenue

L'instruction RCR (Rotate Right through Carry) :

RCR registre/variable, 1

RCR registre/variable, CL

- **RCR** effectue une ***rotation à droite*** de **l'opérande destination** indiqué, 1 ou CL **fois**, en prenant en compte le contenu de l'indicateur de ***retenue***. Le bit de **poids *faible*** de l'opérande destination est mis dans la ***retenue***. Le contenu de la ***retenue*** est mis dans le bit de **poids *fort*** de l'opérande **destination**.



471

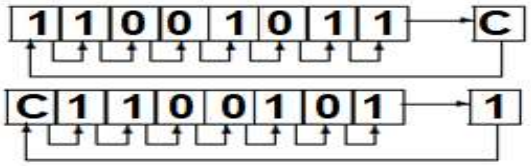
Jeu d'instructions

Rotation à droite avec retenue

Exemple de l'instruction RCR:

```
mov al,11001011B
rcr al,1
```

- Le bit ***sortant*** par la ***droite*** est ***copié*** dans l'indicateur de retenue ***CF*** et la valeur ***précédente*** de CF est ***réinjectée*** par la ***gauche***



472

Jeu d'instructions

Les instructions de branchement

- Les instructions de ***branchement*** (ou saut) permettent de **modifier l'ordre d'exécution** des instructions du programme en fonction de certaines ***conditions***.
- Il existe ***3 types*** de saut :
  - **Saut inconditionnel ;**
  - **Sauts conditionnels ;**
  - **Appel de sous-programmes.**
- **Positionnement** des bits du registre ***FLAGS*** par des instructions ***arithmétiques***, ***logiques***

473

Jeu d'instructions

Instruction de sauts conditionnels :

Comparaison CMP

- C'est l'instruction la plus utilisée pour positionner les indicateurs avant d'effectuer une instruction de saut conditionnel.

  CMP registre/variable, registre

  CMP registre, registre/variable

  CMP registre/variable, constante

- Elle permet de comparer deux valeurs et soustrait le second opérande du premier, sans modifier l'opérande destination

474

Jeu d'instructions

Instruction de saut inconditionnel :

Exemple de l'instruction CMP

```
mov al, 23
cmp al, 34
```

- C=1, car le deuxième opérande de l'instruction CMP est supérieur à la valeur du premier opérande. Z=0 car les deux données sont différentes. S=1 est également mis à 1 car 23−34 est un nombre négatif.

475

Jeu d'instructions

Instruction de saut inconditionnel :

Exemple de l'instruction JMP

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

- Le saut se fait comme l'instruction **goto en C**
- Ceci donne une boucle infinie

476

Jeu d'instructions

Instruction de saut inconditionnel :

L'instruction JMP

JMP étiquette

- Elle effectue un **saut inconditionnel** à l'étiquette (ou label) spécifiée qui est une représentation symbolique d'une instruction en mémoire. Le registre **FLAGS n'intervenant en rien** dans cette opération.
- Elle **ajoute** au **registre IP** le **nombre d'octets** (distance) qui sépare l'instruction de saut de sa destination.
- Pour un saut en **arrière**, la distance est **négative** (***codée en complément à 2***).

477

Jeu d'instructions

Instruction de saut inconditionnel :

L'instruction JMP

SI (condition vraie) ALORS **action-alors** **SINON** **action-sinon** **FSI**	**calcul de la condition** **Jcc SINON** **action-alors** **...** **JMP FSI** **SINON:** **action-sinon** **...** **FSI1:** **...**

- Où **Jcc** dénote l'une des instructions de **saut conditionnel**.

478

Jeu d'instructions

Instruction de sauts conditionnels :

Jcondition label

- Un **saut conditionnel** n'est exécuté que si une certaine **condition est satisfaite**, sinon l'exécution se **poursuit séquentiellement** à l'instruction suivante.
- La **condition** du saut porte sur **l'état** de l'un (ou plusieurs) des **indicateurs d'état** (flags) du **microprocesseur**
- **L'étiquette** référencée ne doit pas se trouver trop **loin** de l'instruction de **saut**. Sinon, une **erreur d'assemblage** est **déclenchée** et le **message** : ***Relative jump out of range by xxx bytes*** est affiché

479

Jeu d'instructions

Instruction de sauts conditionnels :

- Toutes ces instructions fonctionnent selon le **schéma suivant**: **Quand la condition est vraie, un saut est effectué à l'instruction située à l'étiquette spécifiée en opérande**.
- **Les tests d'ordre** (**inférieur**, **supérieur**, ...) se comprennent comme suit : si **CMP op1,op2** et ensuite **JG**, ***c'est-à-dire* 'saut si plus grand',** il y aura alors **saut** si la valeur du **premier opérande** de l'instruction CMP **est supérieur** à la valeur du **deuxième opérande**.

480

Jeu d'instructions

Exemple instruction de sauts conditionnels : :

```
cmp ax,bx
jg superieur
jl inferieur
superieur : ; ax>bx
.....
JMP Fin
inferieur : ; ax<bx
.....
Fin:
```

481

Jeu d'instructions

Condition de sauts

Symbole	Nb signé		Nb non signé	
=	JE label	Z=1	JE label	Z=1
≠	JNE label	Z=0	JNE label	Z=0
>	JG label JNLE label	Z=0 et S=O	JA label JNBE label	C=0 et Z=0
≥	JGE label JNL label	S=O	JAE label JNB label	C=0
<	JL label JNGE label	S≠O	JB label JNAE label	C =1
≤	JLE label JNG label	Z =1 ou S=O	JBE label JNA label	C=1 ou Z=1

482

Jeu d'instructions

Condition de sauts de flags

Opération	Syntaxe et flag	
Si retenue	JC label	C=1
Si pas de retenue	JNC label	C=0
Si overflow	JO label	O=1
Si pas d'overflow	JNO label	O=0
Si zéro	JZ label	Z=1
Si pas zéro	JNZ label	Z =0
Si parité	JP label	P=1
Si pas parité	JNP label	P=0

483

Jeu d'instructions

Les boucles en assembleur :

- Une boucle est **composée** de deux éléments :
  - Une **condition** de **sortie** pour **continuer l'exécution** des instructions en **séquence**
  - Un **corps** de boucle spécifiant **l'action** à réaliser pendant que la **condition de sortie n'est pas vérifiée**, à chaque **itération** de la boucle et entraînant le **recalcul** de la **condition** de sortie pour **la poursuite ou non des** itérations.

484

Jeu d'instructions

La boucle tant que :

TQ (condition) FAIRE **action** **FTQ**	**TQ: calcul de la condition** **Jcc FTQ** **action** **...** **JMP TQ** **FTQ:** **...**

- Une **étiquette TQ** indique le **début** de la boucle
- La boucle **commence** par une **évaluation** de la condition de sortie qui positionne les **indicateurs** du registre **FLAGS**
- En fonction de **la valeur des indicateurs** (donc du **résultat du test** de sortie), un **saut conditionnel** est effectué en **fin** de boucle (**Jcc FTQ**), pour **quitter** la boucle le moment venu.

485

Jeu d'instructions

La boucle tant que :

TQ (condition) FAIRE **action** **FTQ**	**TQ: calcul de la condition** **Jcc FTQ** **action** **...** **JMP TQ** **FTQ:** **...**

- Ensuite le **corps** de la boucle est exécuté, terminé par une instruction de **saut inconditionnel** vers le **début** de la boucle (**JMP TQ**) qui permettra de ré-évaluer la **condition d'arrêt** après chaque itération
- Une étiquette **FTQ** indiquant la **fin** de la boucle

486

Jeu d'instructions

La boucle répéter :

REPETER **action** **JUSQUA (condition vraie)**	**REPETER: action** **...** **calcul de la condition** **Jcc REPETER**

- Une étiquette **REPETER** repère le **début** de la boucle
- Ensuite **le corps de la boucle** est **exécuté**
- A l'issue de l'exécution du corps, **l'évaluation du test d'arrêt** est effectuée positionnant les **indicateurs** du registre **FLAGS**
- Après une instruction **de saut conditionnel** effectue un **branchement** pour continuer les itérations **quand la condition est toujours vérifiée (le test d'arrêt n'est pas vérifié)**

487

Jeu d'instructions

La boucle pour :

- Une **boucle pour** est généralement **codée** à l'aide d'une instruction de la famille **LOOP**.
- LOOP boucles contrôlées par un compteur: le registre CX

  LOOP étiquette : saut court si cx ≠ 0
  LOOPE étiquette : saut court si cx ≠ 0 et Z = 1
  LOOPZ étiquette : saut court si cx ≠ 0 et Z = 1
  LOOPNE étiquette : saut court si cx ≠ 0 et Z = 0
  LOOPNZ étiquette : saut court si cx ≠ 0 et Z = 0
- Elle **décrémente** le **compteur** **sans** **modifier** aucun des **indicateurs**. Si le **compteur** est **différent de 0**, un **saut à l'étiquette** opérande de l'instruction LOOP est réalisé

488

Jeu d'instructions

La boucle pour :

```
POUR indice := 1 A n FAIRE
        action
FPOUR
```

```
POUR indice := n A 1, pas := -1 FAIRE
        action
FPOUR
```

```
mov cx, n
POUR: ...
        action
loop POUR
```

489

Jeu d'instructions

Exemple :

```
Data Segment
    msg db 'hello world', 13, 10, '$'
End Segment
Code Segment
    mov AX , @data     ; initialisation de la valeur
    mov ds, AX  ; initialisation de ds
    mov cx, 10     ; initialisation du compteur de boucles
    ; corps de boucle
    boucle: mov ah, 9    ; affichage du message
            lea dx, msg
            Int  21h
            loop boucle     ; contrôle de la boucle
End Segment
```

➔ Affiche, à l'écran, **dix** fois la chaîne de caractères **'hello world'**

490

Entrée/Sortie

Interruption

- **BIOS ≈ librairie de fonctions** : Chaque fonction effectue une tache bien précise.
- **Code du BIOS** en ROM est **non modifiable** d'où l'utilisation de la **table des vecteurs d'interruptions**
- L'instruction **INT n** permet d'appeler la n-ieme vecteur de la table des **vecteurs d'interruptions**. Avec n est un entier compris entre **0 et 255 (1 octet)**

  (https://www.gladir.com/LEXIQUE/INTR/)
- **Le système DOS** (Disk Operating System) repose sur le BIOS, il appelle les **fonctions** du BIOS **pour interagir avec le matériel**
- Les **fonctions** du **DOS** s'appellent à l'aide du vecteur **21H**

491

Entrée/Sortie

Interruption

- La valeur du **registre AH** permet d'indiquer quelle est la **fonction** que l'on appelle :

```
MOV AH, numero_fonction
INT 21H
```

- **La fonction 4CH** du DOS permet de **terminer** un programme et de revenir à **l'intérpréteur** de commandes DOS:

```
MOV AH, 4CH
INT 21H
```

492

Entrée/Sortie

Fonction d'écriture à l'écran

- **Fonction "2"** : **Affiche un caractère à l'écran**

```
mov dl, 'a'
mov ah, 2
int 21h
```

➔ affiche le caractère "a" a l'ecran

- **Fonction "9"** :**Affiche une chaine de caractère à l'écran**

```
msg db 13, 10, "hello world$"
mov ah, 9
mov dx, offset msg
int 21h
```

➔ affiche la chaine de caractère " **hello world**" à l'ecran

493

Entrée/Sortie

Fonction de lecture du clavier

- **Fonction ”1”** : **Saisie d’un caractère**

```
mov ah, 1
int 21h
```

➔ renvoie dans le registre al le code du caractère lu au clavier.

- **Fonction ”7”** : **Saisie sans écho**

```
mov ah, 7
int 21h
```

➔ lit un caractère au clavier et le renvoie dans le registre al. Ce caractère n’est pas affiché a l’écran.

494

Entrée/Sortie

Fonction d’heure

- **Fonction ”2ch”** : **lit l’heure courante**, telle qu’elle est stockée dans l’ordinateur

```
mov ah, 2ch
int 21h
```

- Au retour de l’appel, les registres contiennent les information suivantes :
  - **Le registre ch: heures**
  - **Le registre cl: minutes**
  - **Le registre dh: secondes**
  - **Le registre dl: centièmes de secondes**

495

Entrée/Sortie

Fonctions de date

- **Fonction ”2ah”:** lit la date courante, telle qu’elle est stockée dans l’ordinateur

```
mov ah, 2ah
int 21h
```

- Au retour de l’appel, les registres contiennent les informations suivantes :
  - **Le registre al : jour de la semaine code (0 : dimanche, 1 : lundi, ...)**
  - **Le registre cx : année**
  - **Le registre dh : mois**
  - **Le registre dl : jour**

496