Architecture des Ordinateurs

Licence fondamentale SMI : S4

Mme Soumia ZITI LABRIM

Université Mohammed V

Faculté des Sciences

RABAT

Département d'Informatique

2019/2020

Plan

- Introduction
- Unités fonctionnelles
- Processeur 80x86
- Assembleur
- Jeu d'instructions
- Entrées/sorties

Introduction

But de l'enseignement

- De quoi est composé un ordinateur ?
- Quels sont les modèles sous-jacents au fonctionnement d'une machine ?
- Comment s'exécutent les programmes ?
- Quel est le lien entre le logiciel et le matériel ?
- Comment fonctionnent les divers périphériques ?
- Comment programmer en Assembleur?

Introduction

Ordinateur

- **Besoin**: Calcul plus complexe et plus rapide => Automatisation du calcul
- **Historique**:
  - XVIIe siècle et avant : les principes fondateurs
  - XIXe siècles : les calculateurs
  - XXe siècle : théorie de l'information + machine universelle
  - ~1945 : Architecture de Von Neumann et naissance de l'ordinateur
  - ~1950 : 1ere génération : tubes a vides
  - ~1960 : 2eme génération : transistors
  - ~1970 : 3eme génération : circuits intègres
  - ~1980 : 4eme génération : puces avec des millions de transistors

Introduction

Naissance de l'ordinateur

- **Claude Shannon (1948) :** chiffres binaires pour les relations logiques et les calculs logiques et arithmétiques (Tout calcul peut être réalisé avec les 3 opérations logiques de base ET, OU, NON
- **Alan Turing :** machine universelle ou Machine de Turing décrivant un modele abstrait du fonctionnement des appareils mécaniques de calcul => Invente les concepts de programmation et de programme
- **John Von Neumann (1945) :** Enregistrer le programme en mémoire => Architecture de l'ordinateur moderne : l'architecture de Von Neumann

Introduction

Ordinateur

- Une **machine** de traitement de **l'information** (acquérir, conserver, traiter et restituer). Il est capable d'effectuer **automatiquement** des opérations arithmétiques et logiques à partir de programmes définissant la séquence de ces opérations.

- C'est un ensemble de **circuits** électroniques permettant de manipuler des **données** sous forme **binaire**, ou bits afin d' exécuter des séquences de calculs ou des traitements de tout genre.

Introduction

Information

- Un ensemble de **données** qui a un sens précis
- Des **valeurs** numériques, textes, images, son, vidéos représentés sous forme de données.
- Des **instructions** composant un programme.
- Toute information est manipulée sous forme **binaire** (ou numérique) par un système informatique.

Introduction

Informatique

- Terme employé pour la première fois en 1962 et provenant des mots « **Information** » et « **automatique** ». C'est la **science** du **traitement rationnel et automatique de l'information**, considérée comme le support des connaissances dans différents domaines .

Objectifs

- Faciliter et accélérer le calcul,
- Automatiser les traitement des données
- Contrôler et commander des processus,
- Faciliter la communication entre plusieurs composants
- Partager des informations et des ressources.

Introduction

Système informatique

- Ensemble des moyens **logiciels** et **matériels** nécessaires pour **satisfaire** les besoins informatiques des **utilisateurs**.

- ***Un système informatique est capable de:***

  - **Acquérir** des informations nécessaires pour les **calculs** et les traitement
  - **Sauvegarder** les données d'une façon **permanente** pour des traitements ultérieurs sur des **supports** de stockage
  - **Effectuer** des traitements des **données** et des **calculs** simples ou complexes
  - **Restituer** les **données** au cas de besoin

Introduction

Programmation

- A partir d'un **problème donné**, réaliser un **programme** dont l'exécution apporte **une solution satisfaisante** au problème posé suivant un **algorithme** bien précis et moins complexe
- Elle est effectuer en utilisant un **langage** de **programmation** comme le **langage machine**, **l'assembleur** ou un **langage** évolués (traduction de l'algorithme)
- Elle fait partie de **l'ingénierie de développement logiciel** (**implémentation ou code**)

Introduction

Langage de programmation

- C'est **l'intermédiaire** entre **l'humain et la machine**, il permet d'écrire, dans un langage proche de la machine mais intelligible par l'humain, toutes les opérations que l'ordinateur doit effectuer.
- il doit donc respecter une **syntaxe stricte**. Un **algorithme** peut toutefois aboutir à **plusieurs programmes**.
- Un **langage informatique** est destiné à décrire l'ensemble des **actions consécutives** qu'un ordinateur doit **exécuter**. C'est une façon pratique de **donner** des **instructions à un ordinateur**.

Introduction

Familles de langage de programmation

- **Langages fonctionnels:** (ou langage procédural) est un langage dans lequel le programme est construit par fonctions, retournant un nouvel état en sortie et prenant en entrée la sortie d'autres fonctions par exemple

=> diviser un problème complexe en sous-problèmes plus simples. Lorsqu'une fonction s'appelle elle-même, on parle alors de récursivité.

- **Langages objets:** part du principe que des choses peuvent avoir des points communs, des similarités en elles-mêmes ou en leur façon d'agir. L'idée est regrouper de tels éléments afin d'en simplifier leur utilisation.

=>Un regroupement est appelé classe, les entités qu'il regroupe sont appelées objets (définition des actions pour toute une classe et chaque objet pourra les effectuer)

Introduction

Programme

- Suite **d'instructions** dans un **langage** donnée, définissant un des actions spécifiques exécutables par un ordinateur
  - programmes systèmes (système d'exploitation gérant différents ressources machine)
  - programmes d'application (des logiciels de traitements)
- Un programme est composé de deux partie:
  - La partie contenant les données
  - La partie contenant le code des instructions à exécuter
- Les **instructions** sont des **opérations** de base que l'ordinateur peut traiter comme l'addition, la multiplication la comparaison…

Introduction

Microprocesseur

- C'est un **circuit** électronique intégré complexe et miniaturisé contenant plusieurs millions de **transistors** interconnectés(ex : le Pentium).
- C'est le **cœur** de l'ordinateur qui permet de traiter et distribuer les informations.
- Il résulte de l'intégration sur une puce de **fonctions logiques combinatoires** (logiques et/ou arithmétique) et **séquentielles** (registres, compteur, etc…).
- Il exécute les instructions élémentaires au rythme de son **horloge interne** (ex : 300 Mhz ou mégahertz => 300 millions d'instructions par seconde).

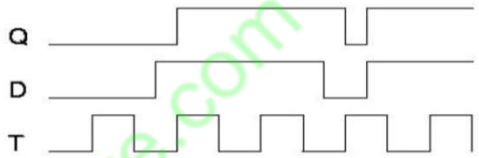
Introduction

Horloge

- Synchronisation de l'ensemble des dispositifs logiques d'un ordinateur.
- Cadencement des instructions à fréquence constante : l'horloge divise le temps en battements de même durée appelés cycles.
- une fréquence d'horloge à 500MHz: des cycles élémentaires de 2 nanosecondes.

un signal est une grandeur discrète appartenant à [0,1]

un chronogramme est la représentation graphique d'un signal évoluant dans le temps



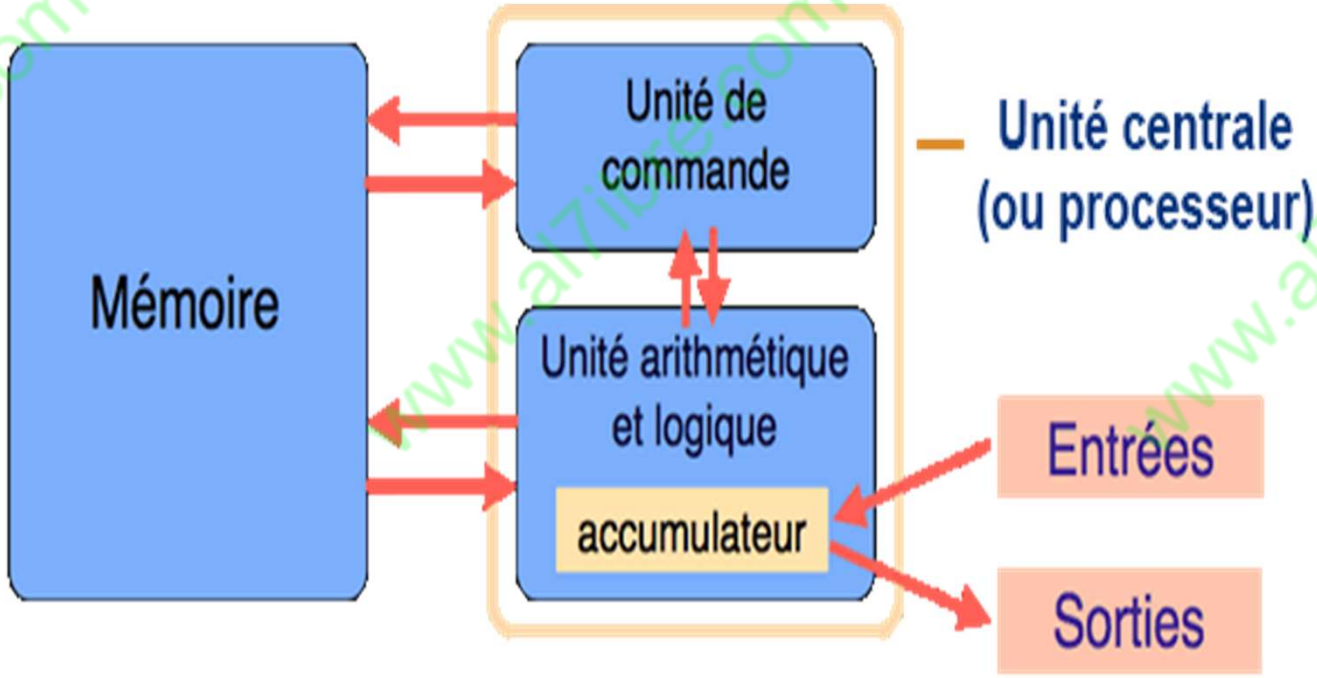
Unités fonctionnelles

Architecture de Von Neumann

- il faut insérer le microprocesseur dans un système adéquat afin de manipuler l'information.
- **John Von Neumann** a construit en 1946 un modèle de machine universelle de traitement programmé de l'information.
- Cette architecture est la base de la plupart des systèmes à microprocesseur actuel. Elle est composé de:
  - L'unité centrale de traitement (le processeur)
  - La mémoire
  - Les dispositifs d'entrée-sortie
- Ces différents éléments sont interconnectés à travers des **bus** qui constituent le support d'acheminement de l'information entre les différents composants.

Unités fonctionnelles

Architecture de Von Neumann



Unités fonctionnelles

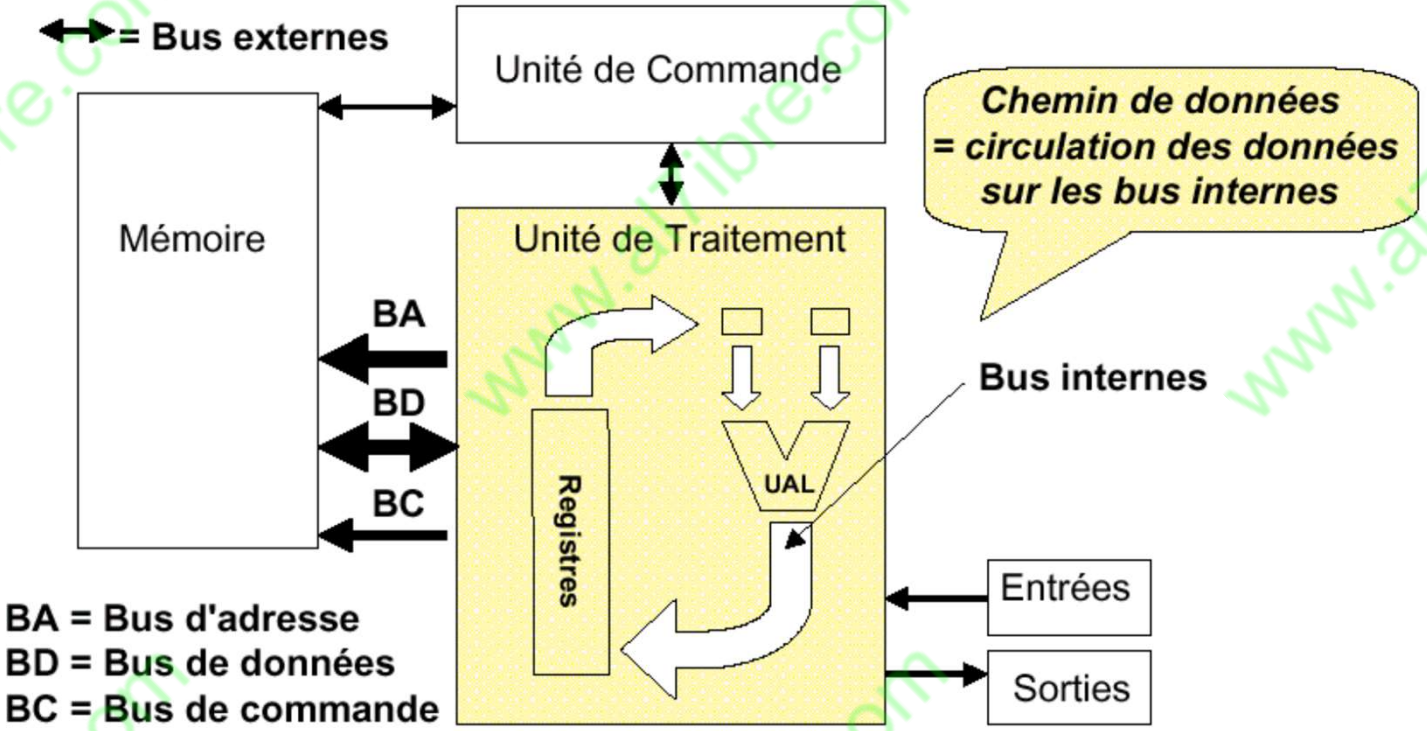
Unité centrale de traitement

Elle est chargée **d'interpréter** et **d'exécuter** les instructions d'un programme, de **lire** ou de **sauvegarder** les résultats dans la mémoire et de communiquer avec les unités d'échange.

- Elle est construite autour de deux éléments principaux :
  - Une **unité de commande** (appelée aussi Unité de commande et de contrôle (UCC)) : elle est responsable de la lecture en mémoire et du décodage des instructions,
  - Une **unité de traitement** : aussi appelée Unité Arithmétique et Logique (UAL) exécute les instructions qui manipulent les données.
- Le microprocesseur est associé à des **registres** chargées de stocker les différentes informations à traiter.

Unités fonctionnelles

Unité centrale de traitement



Unités fonctionnelles

Unité centrale de traitement

- Pour chaque instruction, le processeur effectue **schématiquement** les opérations suivantes :
  - **Lire** en mémoire principale l'instruction à exécuter
  - **Effectuer** le traitement correspondant
  - **Passer** à l'instruction suivante.

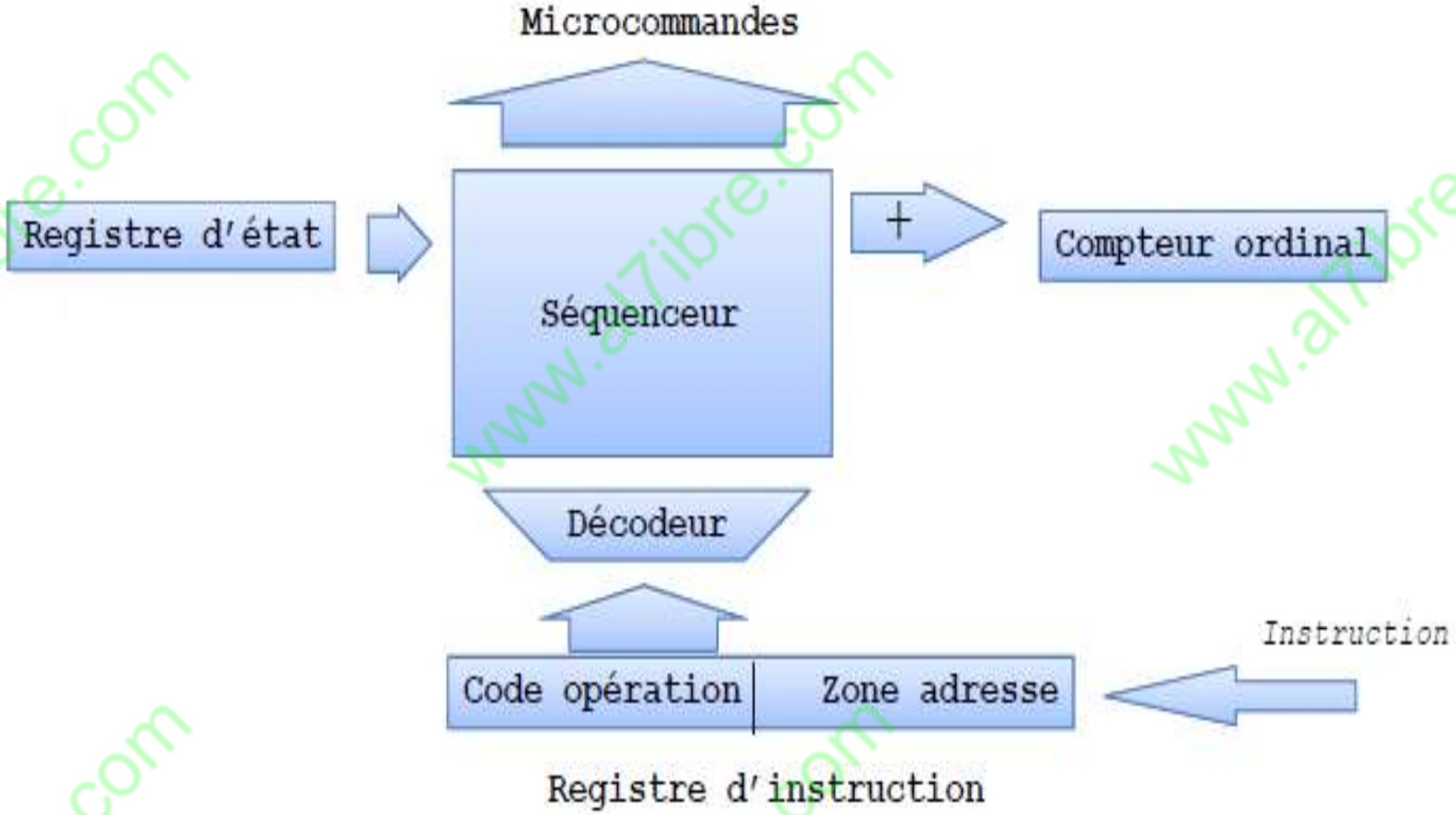
Unités fonctionnelles

Unité de commande

- Elle **contrôle le transfert** des **instructions** et des **données** entre la mémoire et l'UC et leur exécution
  - Elle permet de **séquencer** le déroulement des instructions.
  - Elle effectue la **recherche** en mémoire de l'instruction.
  - Elle assure le **décodage** de chaque instruction, codée sous forme **binaire**, pour réaliser son exécution et ensuite préparer l'instruction suivante.
- Elle gère tous les **signaux** de **synchronisation** internes ou externes (bus des commandes) au microprocesseur en **coordonnant** le **fonctionnement** des autres composants dont les activité sont **cadencées** par une **horloge** unique, de façon à ce que tous les circuits électroniques travaillent ensembles.

Unités fonctionnelles

Unité de commande



Unités fonctionnelles

Composants de l'unité de commande

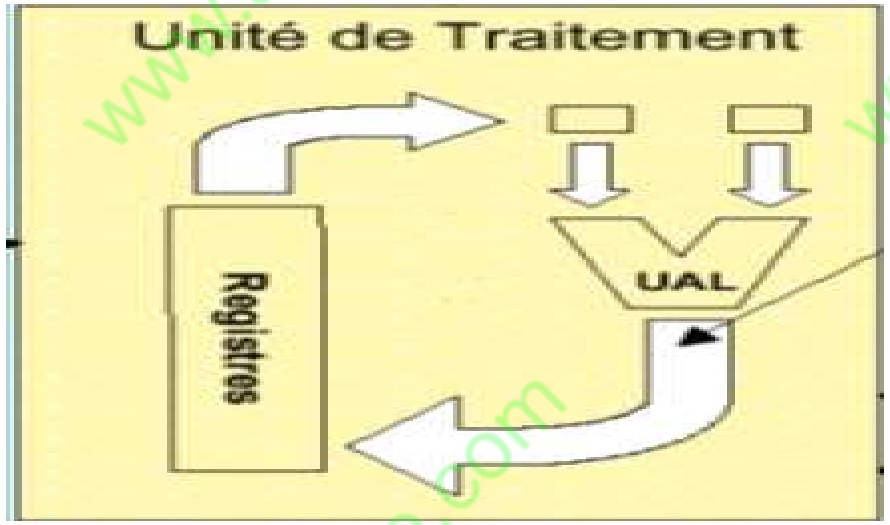
- **Le compteur ordinal de programme** constitué par un **registre** dont le contenu est initialisé avec **l'adresse** de la **première instruction** du programme. Il contient toujours l'adresse de l'instruction à **exécuter**.
- **Le registre d'instruction et le décodeur d'instruction** : chacune des instructions à exécuter est **rangée** dans le registre d'instruction ensuite elle est **décodée** par le décodeur d'instruction.
- **Bloc logique de commande (ou séquenceur)** : Il **organise** l'exécution des instructions au **rythme** d'une horloge. Il gère les signaux du **microprocesseur** en fonction des divers **signaux de commande** provenant du **décodeur d'instruction** ou du **registre d'état** par exemple.

Unités fonctionnelles

unité de traitement

Elle regroupe les **circuits** qui assurent les **traitements** nécessaires à **l'exécution** des instructions :

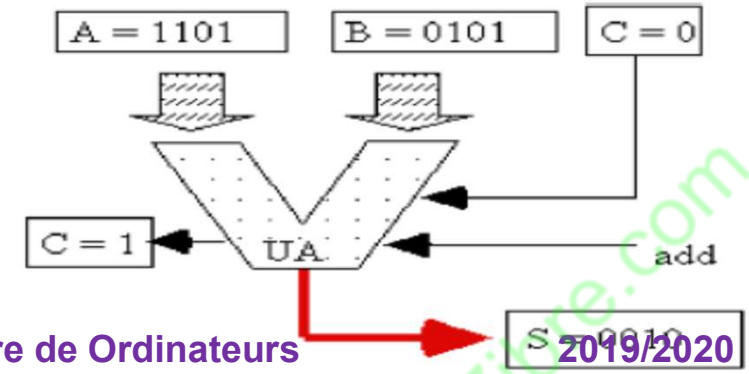
- **L'Unité Arithmétique et Logique (UAL).**
- **L'accumulateur.**
- **Le registre d'état**



Unités fonctionnelles

L'Unité Arithmétique et Logique (UAL):

- **Elle** assure les **fonctions logiques** ou **arithmétique** pour effectuer les **calculs** et les **opérations logiques** des différents **instructions** à exécuter.
- Les **données** sont prisent aux **entrées** de l'UAL, sont **traités**, ensuite le **résultat** est fourni en sortie et généralement stocké dans un **registre** dit **accumulateur**.
- Les **informations** qui concernent **l'opération** sont **envoyées** vers le **registre d'état**.



$$\begin{array}{r} A = 1101 \\ + \; B = 0101 \\ \hline S = 0010 \end{array}$$

Unités fonctionnelles

L'accumulateur

- **C'**est un **registre** de travail qui sert à **stocker** une **opérande** au début d'une **opération arithmétique** et le **résultat** à la fin de **l'opération**
- Il permet stocker les **résultats** des **opérations arithmétiques** et **logiques** en cours **d'exécution** dans l'UAL.

Unités fonctionnelles

Le registre d'état

- Il est généralement **composé** de **8 bits** à considérer individuellement.
- Chaque **bit** est un **indicateur** dont **l'état** dépend du résultat de la **dernière opération** effectuée par l'UAL. On les appelle **indicateur d'état** ou **flag** ou **drapeaux**.
- Dans un programme le **résultat du test** de leur état conditionne souvent le déroulement de la **suite** du **programme**. Par exemple la retenue, la parité….

Unités fonctionnelles

Mémoire principale

- Elle contient les **instructions** des **programmes** en cours **d'exécution** et les **données** associées à ce programme.
- Elle se décompose en :
  - Une **mémoire morte** (**ROM** = Read Only Memory) C'est une **mémoire à lecture seule**. Lors d'une coupure du courant électrique, son contenu n'est pas perdu.
  - Une **mémoire vive** (**RAM** = Random Access Memory) chargée de **stocker** les **données intermédiaires ou les résultats de calculs**. On peut lire ou écrire des données dedans, ces données sont **perdues** à la mise hors tension.

Unités fonctionnelles

Interface d'entrée / sortie

- Elles permettent d'assurer la **communication** entre le **microprocesseur** et les **périphériques** : capteur, clavier, moniteur ou afficheur, imprimante, modem, etc…..
- Sert **d'interface** avec les **périphériques**.
- Les **opérations associées** (lecture et/ou écriture) sont **fonctions du périphérique.**

Unités fonctionnelles

BUS

- Un bus est un **ensemble de fils électrique** qui assure la **transmission** des **informations** entre les différentes unités.
- **Largeur du bus** = **nombre de fils constituant le chemin** = nombre **d'impulsions** électriques pouvant être **envoyés** en **parallèle** (en même temps).
- Il existe **trois types de bus** véhiculant des informations :
  - **Bus de données**
  - **Bus d'adresses**
  - **Bus de commande (ou de contrôle)**

Unités fonctionnelles

BUS

- **Bus de données** : **bidirectionnel** qui assure le **transfert** des **informations** entre le **microprocesseur** et son **environnement**, et inversement. Son nombre de lignes est égal à la capacité de traitement du microprocesseur.
- **Bus d'adresses** : **unidirectionnel** qui permet la **sélection** des **informations** à traiter dans un **espace mémoire** (ou espace adressable) qui peut avoir 2^n emplacements, avec n = nombre de conducteurs du bus d'adresses.
- **Bus de commande** : **transporte** les **ordres** et les **signaux** de **synchronisation** en provenance de **l'unité** de **commande** et à destination de l'ensemble des **composants** matériels. Il s'agit d'un bus **bidirectionnel** dans la mesure où il transmet également les **signaux** de **réponse** des éléments **matériels**.

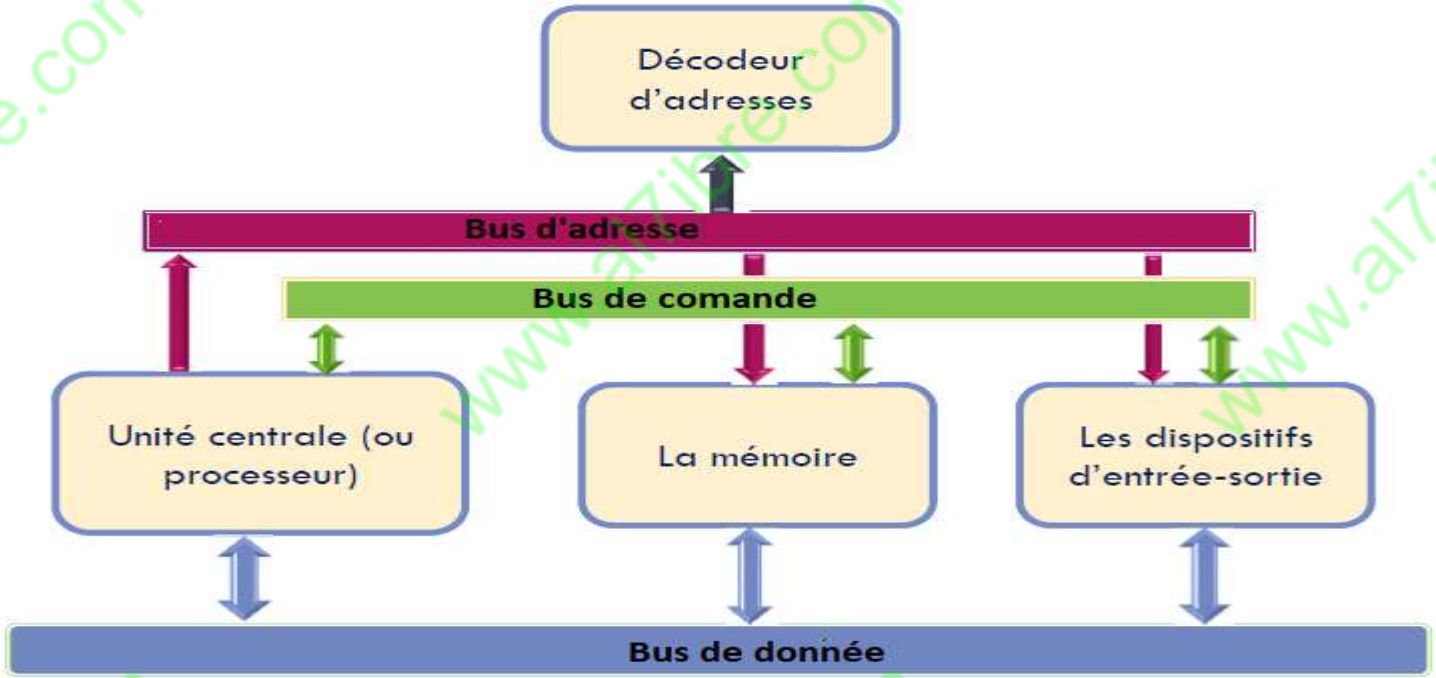
Unités fonctionnelles

Décodeur d'adresse

- Il est chargé **d'aiguiller** les **données** présentes sur le **bus de données** provenant des multiples **périphériques** .
- Le **microprocesseur** peut **communiquer** avec les différentes **mémoires** et les différents **périphériques**.
- Ces éléments sont tous reliés sur le même **bus de données** et afin d'éviter des **conflits**, un seul composant doit être **sélectionné** à la **fois**.
- Attribue à chaque **périphérique** une **zone d'adresse** et une fonction **«décodage d'adresse »** est donc nécessaire afin de fournir les **signaux** de **sélection** de chacun des **composants**.

Unités fonctionnelles

Décodage d'adresse



Unités fonctionnelles

Mémoire

- Une mémoire est un **circuit à semi-conducteur** permettant **d'enregistrer**, de **conserver** et de **restituer** des informations (instructions et variables).
- C'est cette **capacité de mémorisation** qui explique la polyvalence des **systèmes numériques** et leur adaptabilité à de nombreuses situations.
- Les **informations** peuvent être **écrites** ou **lues**. Il y a :
  - **écriture** lorsqu'on **enregistre** des informations en mémoire,
  - **lecture** lorsqu'on **récupère** des informations précédemment enregistrées.

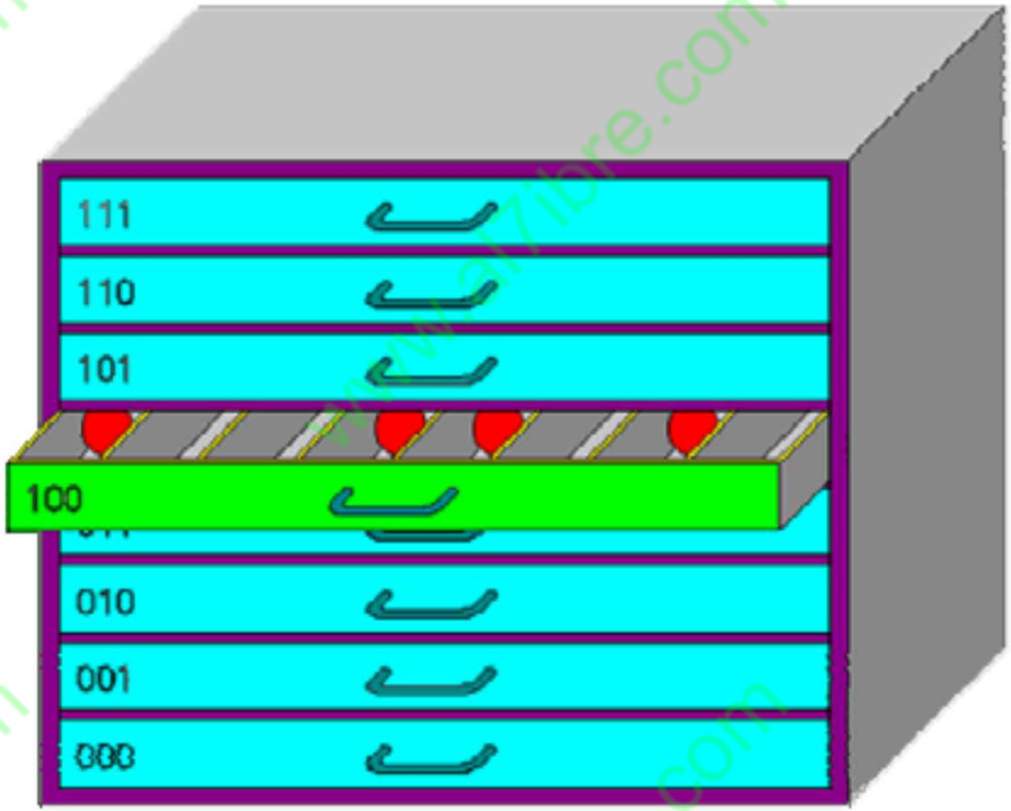
Unités fonctionnelles

Mémoire

- La **mémoire** peut être vue comme **un ensemble de cellules** ou cases contenant chacune une information :
- Chaque **case mémoire** est repérée par un **numéro d'ordre unique** : **son adresse**. Elle est exprimée généralement en **Hexadécimale**.
- Chaque case mémoire est **divisée** en **emplacements** de **taille fixe** (par exemple 8 bits)

Unités fonctionnelles

Mémoire



Unités fonctionnelles

Mémoire

- C'est **un vecteur** dont chaque **composante** est **accessible** par une **adresse**.
- Les **opérations** permises sur la **mémoires** sont les opérations de **lecture** et **d'écriture**.
- L'UC **inscrit** **l'adresse** d'une cellule dans un **registre d'adresse** (RA) et **demande** une **opération de lecture ou d'écriture**. Les **échanges** se font par **l'intermédiaire** d'un **registre de mot** (RM).
  - **Lecture**: **RA ←adresse; RM←mémoire[RA]**
  - **Écriture**: **RM←valeur; RA←adresse; mémoire[RA]←RM**

Unités fonctionnelles

Mémoire

- Si **les adresses des cases mémoires** sont codées sur **n bits**, il est possible de référencer au plus **2^n cases mémoire**.
  - **Exemple** : si les adresses des cases mémoires sont codées sur **4 bits**, il est alors possible d'avoir **2^4 = 16 cases mémoire**
- Chaque **case** est remplie par un **mot de données**
  - Un **mot est l'unité d'information** accessible en une **seule opération de lecture** (sa taille varie en fonction de la machine).
  - Un mot **représente** un **regroupement de bits**
  - Sa **longueur** est toujours une **puissance de 2**
- **Octet (byte) = 8 bits et Bit = 0/1**

Unités fonctionnelles

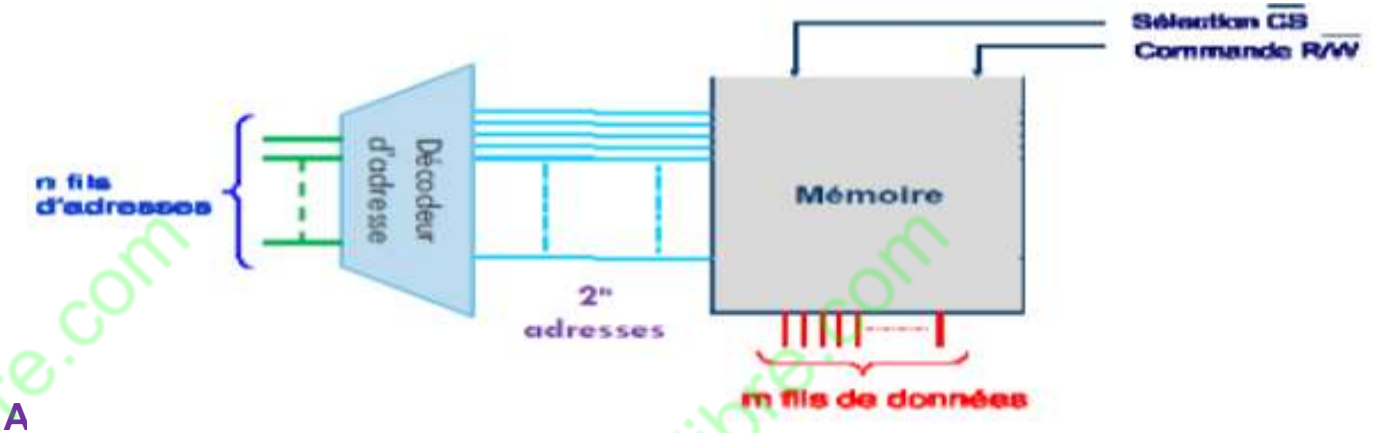
Mémoire: BIG/LITTLE ENDIAN

- La **taille minimum** d'une **donnée** stockée sur un disque est habituellement **1 byte.**
- Les données ayant **plusieurs bytes** (integer, long, float) peuvent être **emmagasinée** de 2 façons:
- **Little endian**: le byte le **moins** significatif est placé à la plus **petite adresse** dans la mémoire
- **Big endian**: le byte le **moins** significatif est placé à la plus **haute adresse** dans la mémoire

Unités fonctionnelles

Mémoire

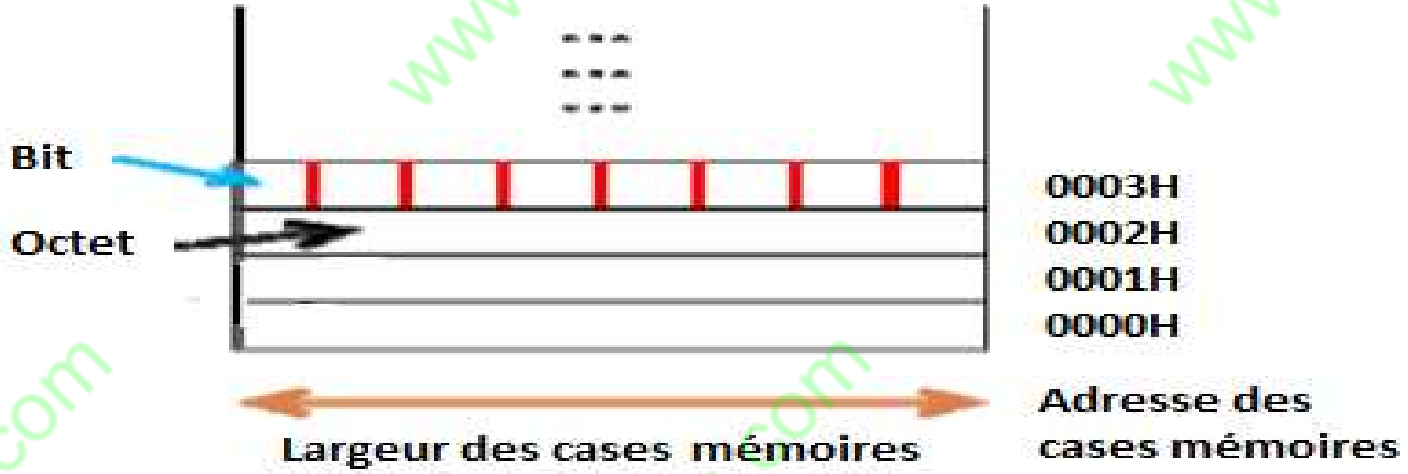
- Un **boitier mémoire** comprend en plus des **fils d'adresses et de données**:
  - Une **entrée de commande (R/W)** qui permet de définir le **type d'action** que l'on effectue avec la **mémoire** (**lecture/écriture**).
  - Une **entrée de sélection CS** qui permet de mettre les **entrées/sorties** du boitier en **haute impédance** (pour la **sélection de la mémoire correspondante**)



Unités fonctionnelles

Mémoire

- Le **nombre** de **fils d'adresses** d'une mémoire définit le **nombre de cases mémoires** qu'elle **comprend**.
- Le **nombre** de **fils de données** définit la **taille des données** que l'on peut **sauvegarder** dans chaque **case mémoire**.



Unités fonctionnelles

Fonctionnement de la mémoire

1. **Choisir l'adresse** en mémoire qui donne **l'accès** à un **emplacement mémoire** pour une **opération de lecture ou d'écriture**
2. **Choisir une opération** de **lecture** ou **d'écriture** (**R/W**)
3. **Sélectionner** le boîtier **mémoire** avec le bit **Chip Select** (CS)
4. **Valider ou invalider la mémoire** : Le **signal** de **sélection** du **boîtier** (CS) **valide** les **fonctions d'écriture et de lecture**.
   - Le boîtier est bloqué pour **CS = 1**.
   - Lorsque **CS = 0**, une fonction **d'écriture** correspond à **R/W=0** et une fonction de **lecture** à **R/W=1**.

Unités fonctionnelles

Caractéristiques de de la mémoire

- **Capacité** : c'est le **nombre total de bits** que contient la mémoire. Elle s'exprime aussi souvent en **octet** (**byte** en anglais). Si nous avons **m bits** dans chaque **case mémoire** et s'il existe **2^n cases mémoire** alors la capacité est : **$C = m*2^n$**
  - **1 K : Kilo octet = 2^{10} ~10^3**
  - **1 M : Méga octet = 2^{20} ~10^6**
  - **1 G : Giga octet = 2^{30} ~10^9**
  - **1 T : Téra octet = 2^{40} ~10^{12}**
- **Format des données** : c'est le **nombre de bits** que l'on peut **mémoriser par case mémoire**. On dit aussi que c'est la **largeur du mot** mémorisable.

Unités fonctionnelles

Caractéristiques de de la mémoire

- **Temps d'accès** : c'est le temps qui s'écoule entre l'instant où a été lancée une **opération** de **lecture/écriture** en mémoire et l'instant où la **première information est disponible** sur le bus de données.
  - **1 ms : Milli second =10^{-3}s**
  - **1 µs : Micro second =10^{-6}s**
  - **1 ns : Nano second =10^{-9}s**
  - **1 ps : Pico second =10^{-12}s**
- **Temps de cycle** : il représente **l'intervalle minimum** qui doit séparer deux **demandes successives** de **lecture ou d'écriture.**

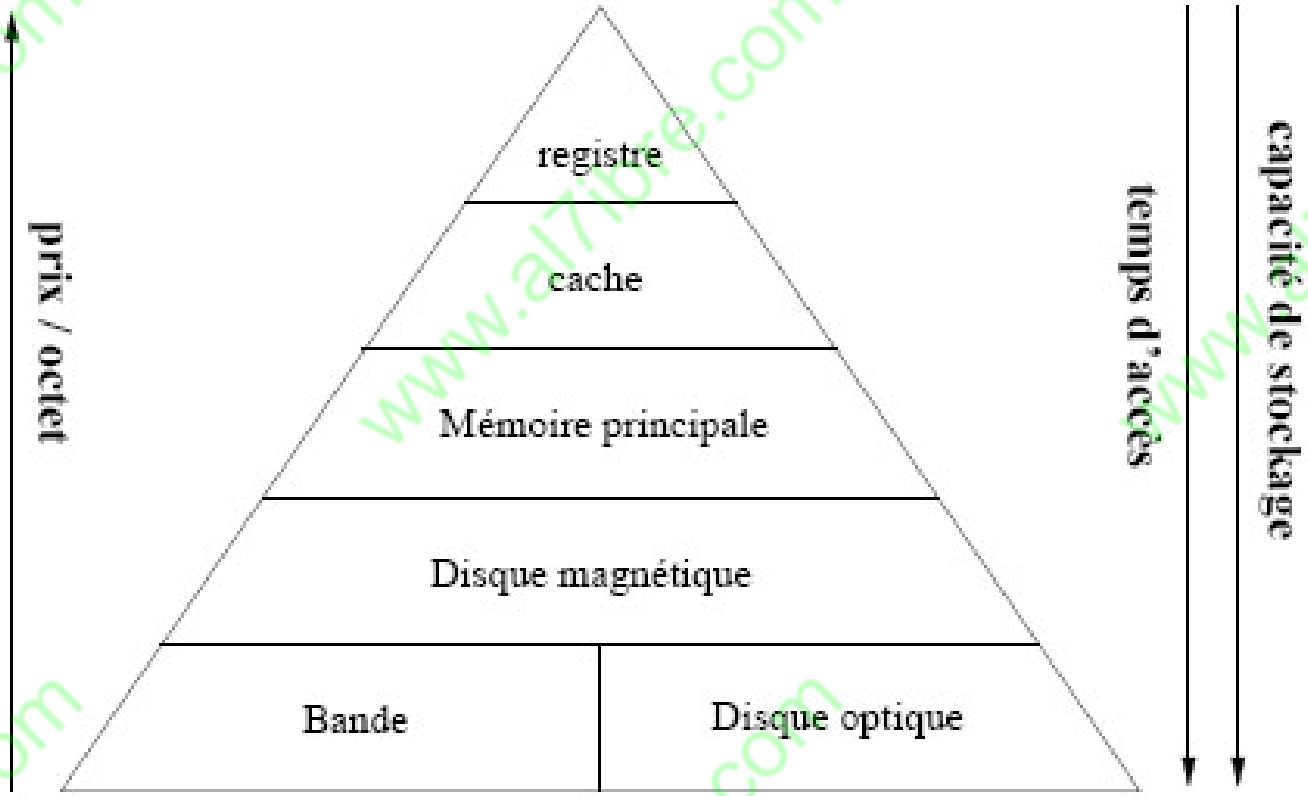
Unités fonctionnelles

Caractéristiques de de la mémoire

- **Débit** : c'est le **nombre maximum** d'informations **lues ou écrites par seconde.**
- **Volatilité** : elle caractérise la **permanence** des **informations** dans la **mémoire**. L'information stockée est **volatile** si elle risque d'être altérée (effacée) par un défaut d'alimentation électrique et **non volatile** dans le cas contraire

Unités fonctionnelles

Hiérarchie de mémoire



Unités fonctionnelles

Hiérarchie de mémoire

- **Registres** : **Mémoires très rapides** situées au niveau du **processeur**. Les registres sont utilisés pour assurer le **stockage temporaire** d'informations nécessaires à l'exécution de l'instruction en cours de traitement.
- **Mémoire cache** : une **mémoire rapide** de **faible capacité** destinée à **accélérer l'accès** à la mémoire centrale en stockant les données les plus utilisées.
- **Mémoire principale** (MP) : elle est utilisée pour le **rangement** des informations. Elle contient les **programmes** (**instructions et données**) à utiliser et est plus **lente** que les deux mémoires précédentes.

Unités fonctionnelles

Hiérarchie de mémoire

- **Mémoire d'appui** : **Mémoire tampon** qui se situe **entre** la **MP** et la **mémoire** de **masse**. (Même rôle que la mémoire cache)
- **Mémoire de masse (auxiliaires)** : est une **mémoire périphérique** de **grande capacité** utilisée pour le stockage **permanent** ou la **sauvegarde** des **informations**. Elle utilise des supports **magnétiques** (disque dur) ou optiques (CDROM, DVDROM,…)

Unités fonctionnelles

classification de mémoire

- **Mémoires de travail :**

Elles désignent les **mémoires** qui sont **actives** dans **l'exécution** d'un programme: les **registres** du **processeur**, la mémoire centrale ou **vive** (RAM), la mémoire **cache** et la mémoire **morte** (électroniques).

- **Mémoires de stockages :**

Elles permettent de **conserver** de manière **permanente** de **grandes** quantités **d'informations**. Ces informations **ne participent pas directement** à **l'exécution** d'un programme mais doivent être **chargées** en mémoire centrale pour être **exploitées** par le **processeur** (magnétique ou optique).

Unités fonctionnelles

Mémoire morte

- **ROM** est acronyme de **Read Only Memory** qui veut dire une mémoire à **lecture seule**.
- Ce genre de mémoire est **entièrement** et **définitivement** réalisé au stade de **fabrication** par le constructeur, donc son contenu **ne peut pas être modifié** (effacé) sans un appareil spécial « Programmeur de ROM ».
- Cette mémoire est composée d'une **matrice** dont la programmation s'effectue en reliant les **lignes** aux **colonnes** par des **diodes**. **L'adresse** permet de sélectionner une **ligne** de la **matrice** et les **données** sont alors reçues sur les **colonnes** (le **nombre** de **colonnes** fixant la **taille** des **mots** mémoire).

Unités fonctionnelles

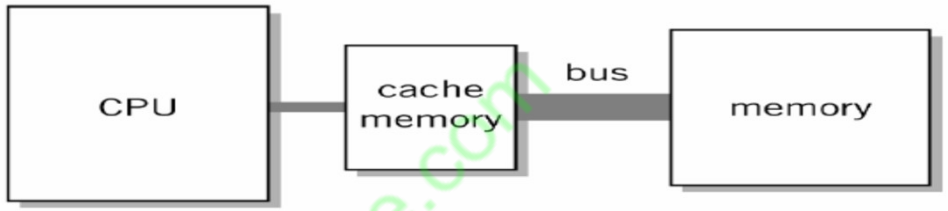
Mémoire vive

- Mémoire dite **RAM** (**Random Access Memory**), où le **temps d'accès** est **indépendant** de la **place** de **l'information** dans la **mémoire**.
- Elle sert au **stockage temporaire** de **données** et sur lesquelles les **opérations** de **lecture** et **d'écriture** sont possibles.
- Elle est **volatile** : **l'information** est **perdue** en cas de **coupure d'alimentation**. Mais le **temps d'accès** et **consommation électrique** sont **faibles** pour ne pas **ralentir** le **microprocesseur**
- Essentiellement **utilisée** en tant que mémoire **centrale** et mémoire **cache**.

Unités fonctionnelles

Mémoire cache

- **Vitesse** du **processeur** est plus **rapide** que la mémoire **principale**
- La mémoire **cache** permet de **réduire** les **délais d'attente** des **informations** stockées en mémoire **vive**.
- Cette mémoire est utilisée comme **mémoire virtuelle**, **invisible** pour le **système d'exploitation** et à **proximité** du **processeur**. Il y stocker **temporairement** les principales **données** devant être traitées par le processeur **augmentant** ainsi la **vitesse d'accès**.



Unités fonctionnelles

Augmentation de la mémoire

- **Pagination de la mémoire**
  - **Minimise** le nombre de **dépendance** d'accès au mémoire
  - **Augmente** la **vitesse** d'accès
- **Segmentation de la mémoire**:
  - **diviser** la mémoire en **plusieurs parties**
  - **Offre** la possibilité d'accès en **lecture/écriture** au même temps
  - **Augmente** la **vitesse** d'accès

Processeur 80x86

Description physique

- Le **microprocesseur Intel 8086** est un **microprocesseur 16 bits**, apparu en **1978**.
- C'est le **premier** microprocesseur de la **famille Intel 80x86** (8086, 80186, 80286, 80386, 80486, Pentium, ...) qui est devenue **l'architecture** de processeur la **plus répandue** dans le monde des **ordinateurs personnels**, **stations de travail** et **serveurs informatiques**.
- **Améliorations**
  - **Augmentation** de la **fréquence d'horloge**, de la **largeur** des **bus d'adresses** et de **données**
  - **Ajout de nouvelles instructions** et de **registres**

Processeur 80x86

Description physique

- Il est base sur des **registres 16 bits**, Il dispose de
  - Un **bus** externe **de données** de **16 bits** (D0 – D15),
  - Un **bus d'adresse** de **20 bits** (A0 – A19) qui permet d'adresser 1 Mo.
  - Un bus de **Données**/**Adresses multiplexe**
- Il contient **29 000 transistors gravés** en **3 µm**.
- Sa **puissance de calcul** varie de **0,33 MIPS** (lorsqu'il est cadencé à **4,77 MHz** comme dans **l'IBM PC**) jusqu'à **0,75 MIPS** pour la version **10 MHz**.

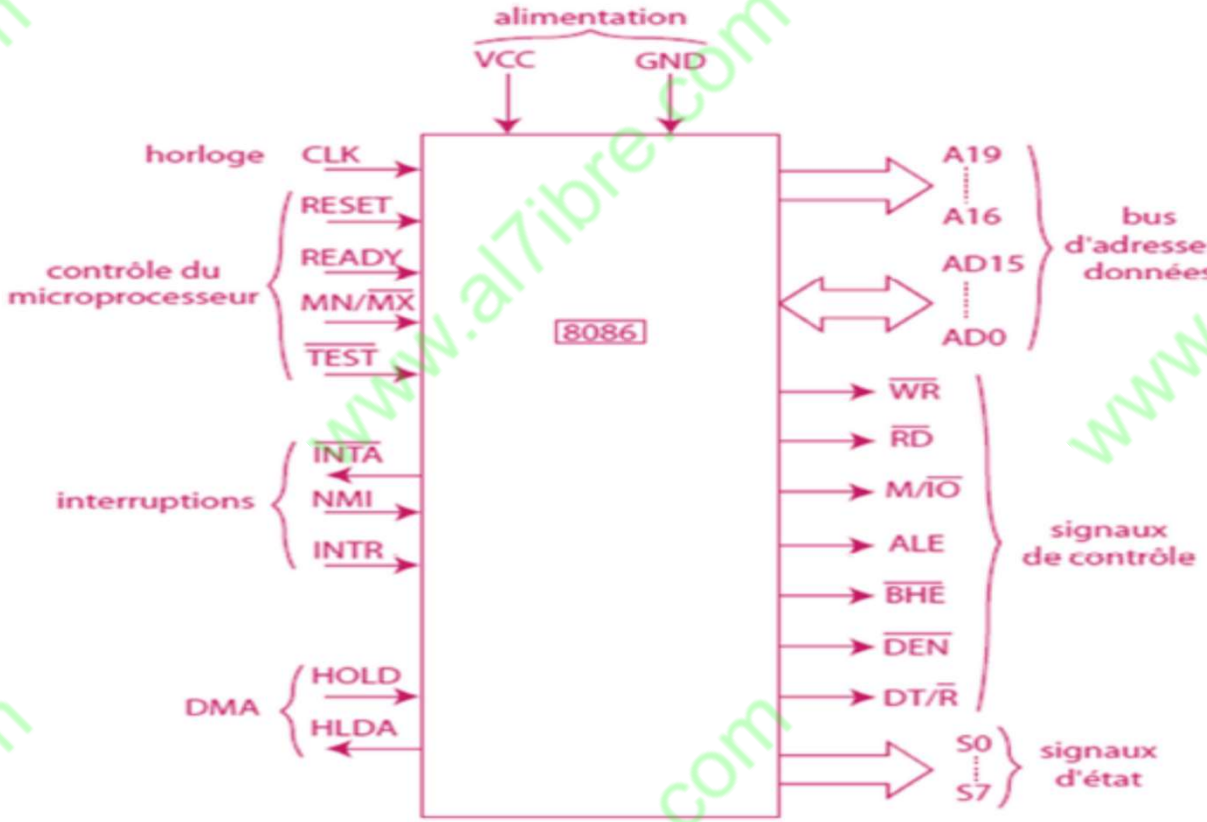
Processeur 80x86

Caractéristiques

- Le **microprocesseur 8086** se **caractérisé** par :
  - **Structure 16 bits.**
  - **Capacité d'adressage de 1 Mo.**
  - **14 registres internes de 16 bits.**
  - **7 modes d'adressage.**
  - **Operations sur des bits, des octets, des mots et des chaines de caractères.**
  - **Arithmétique signée ou non signée.**
  - **Arithmétique binaire ou BCD, sur 8 ou 16 bits.**

Processeur 80x86

Caractéristiques

- 

Processeur 80x86

Composition interne

- Le **8086** est **constitue** de deux **unités de traitement fonctionnant** en **parallèle** :
- **l'unité d'exécution** (**EU** : **Execution Unit**) ;
- **l'unité d'interface de bus** (**BIU** : **Bus Interface Unit**).

➔ **fonctionnent simultané**

➔ **une accélération du processus d'exécution**

Processeur 80x86

Unité d'interface de bus (BIU) :

- Elle **comporte** une **file d'attente d'instructions** gérée en **FIFO** (First Input First Output), les **registres de segments** (**CS, DS, SS, ES**) et le **pointeur d'instruction** (**IP**).

Rôle

- **Rechercher** les **instructions à exécuter** dans la mémoire et les **ranger** dans la file d'attente **FIFO**.
- **Calculer** les **adresses physiques** sur **20 bits**.
- **Réaliser** le **transfert** des **données** avec la mémoire.

Processeur 80x86

Unité d'exécution (EU) :

- Elle **comporte** l'**UAL**, les **registres généraux** (**AX, BX,CX, DX**), les **registres d'adressage** (**SP, BP, SI, DI**), le **registre d'état** (**Flags**) et le **décodeur d'instructions**.

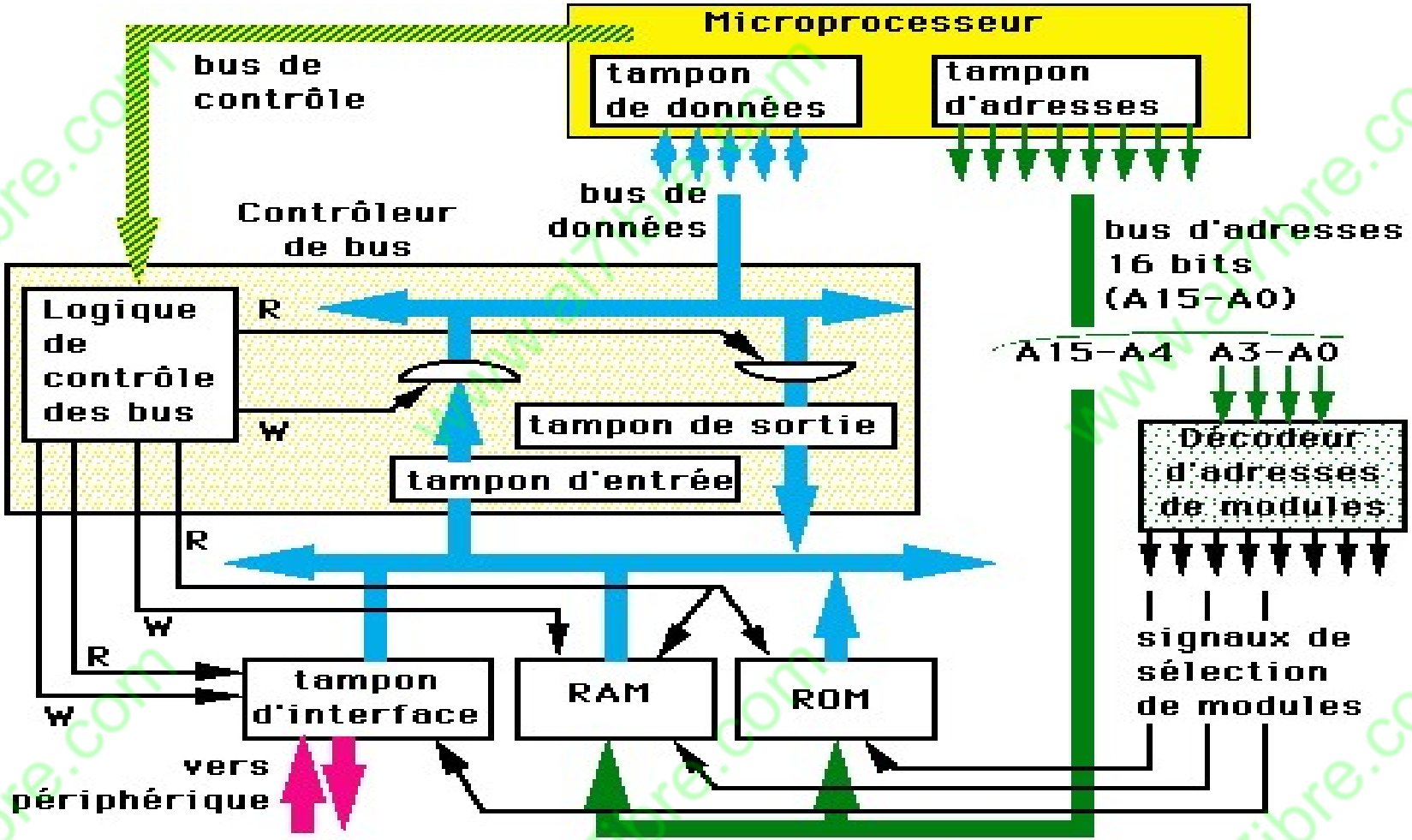
Rôle

- **Extraire** les **codes des instructions** à partir de la **file d'attente** et les **exécuter**.
- **Fournir** les **adresses des opérandes** à l'BIU en nommant le **segment** concerné et en fournissant le **déplacement** dans ce segment.

Processeur 80x86

Exécution

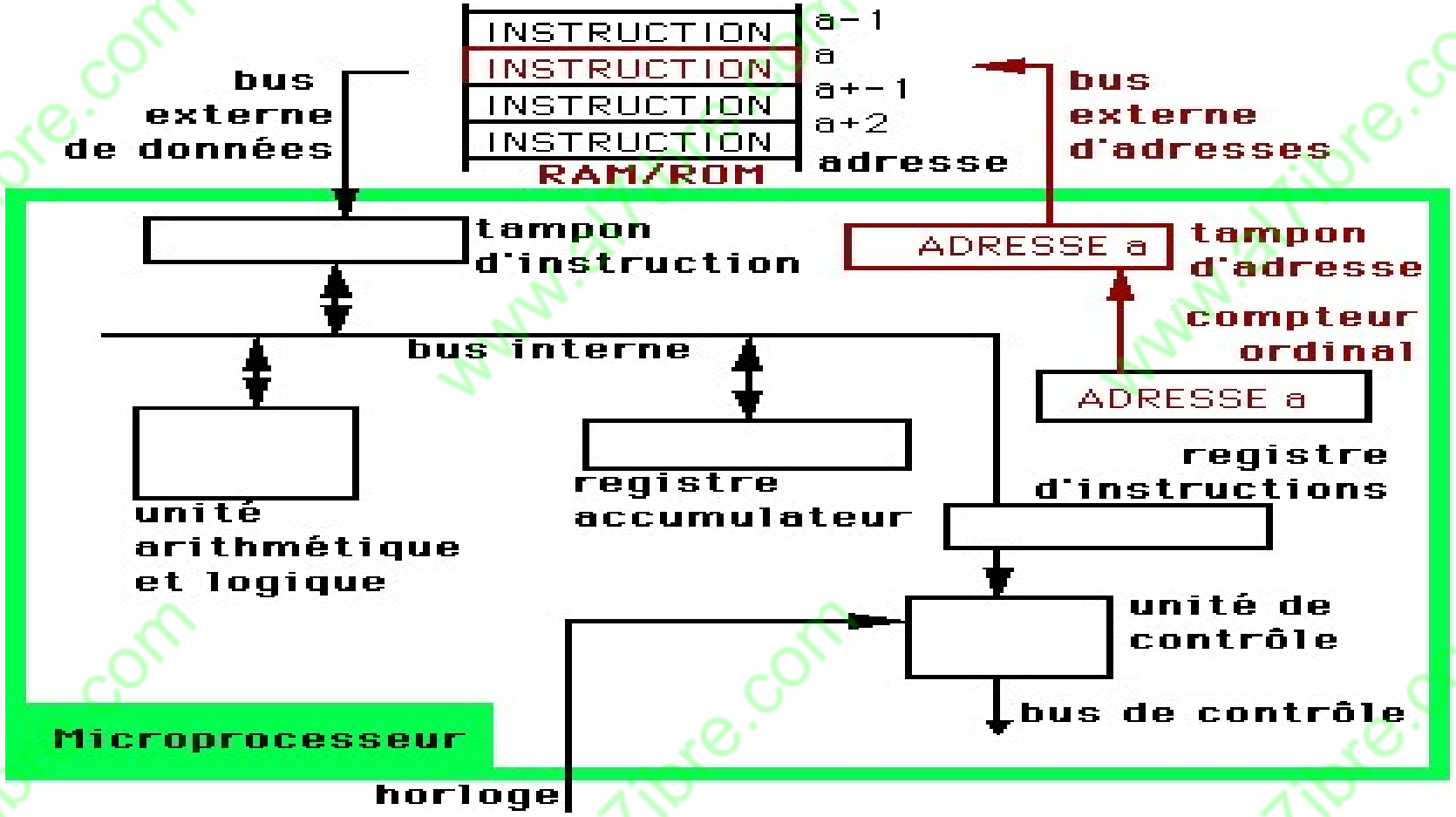
- **Micro-ordinateur** à **mots** de **16 bits** avec **adressage** sur **12 bits**



Processeur 80x86

Exécution

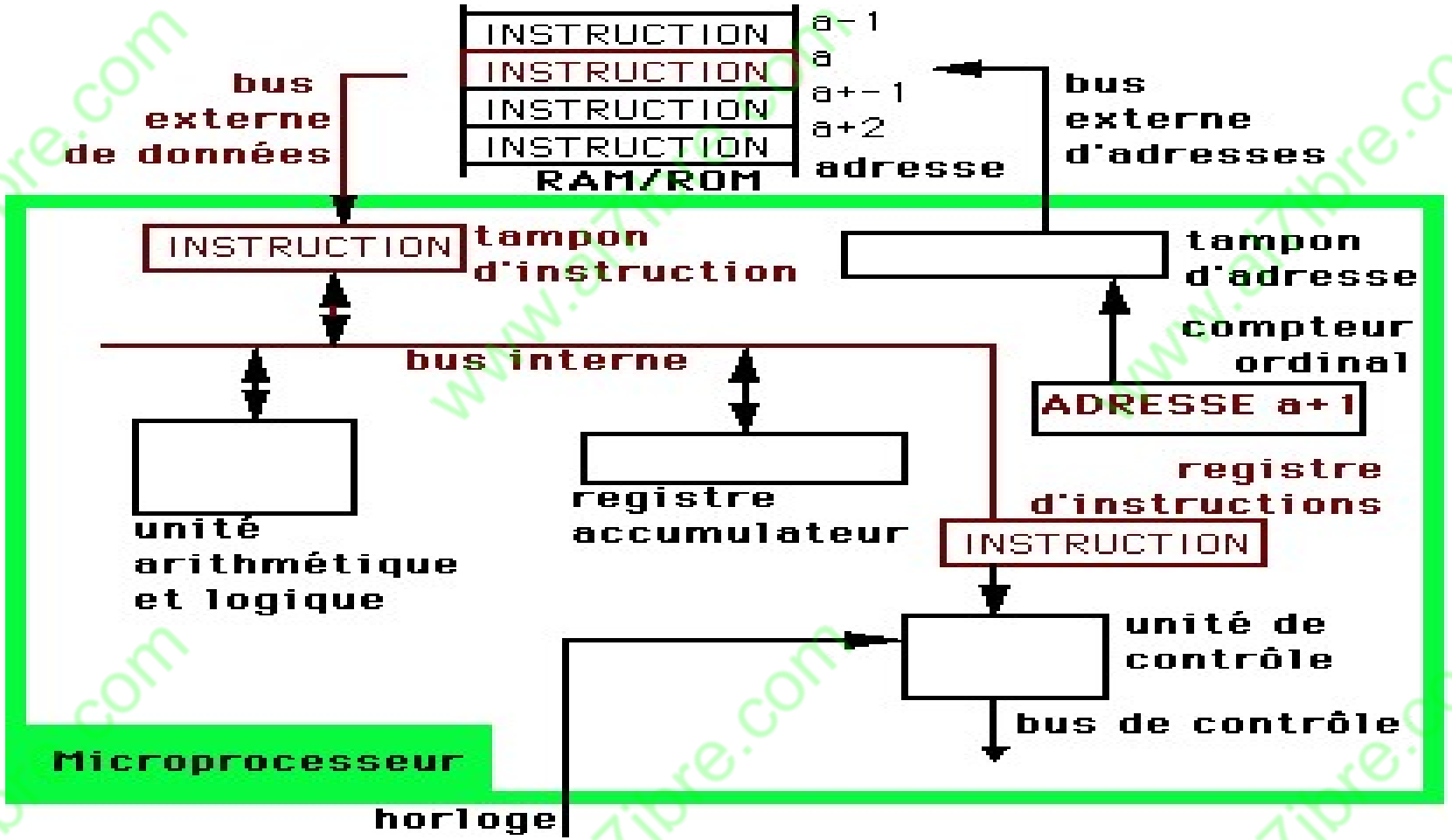
- **Le processeur** va **rechercher** en **mémoire** **l'instruction à exécuter**



Processeur 80x86

Exécution

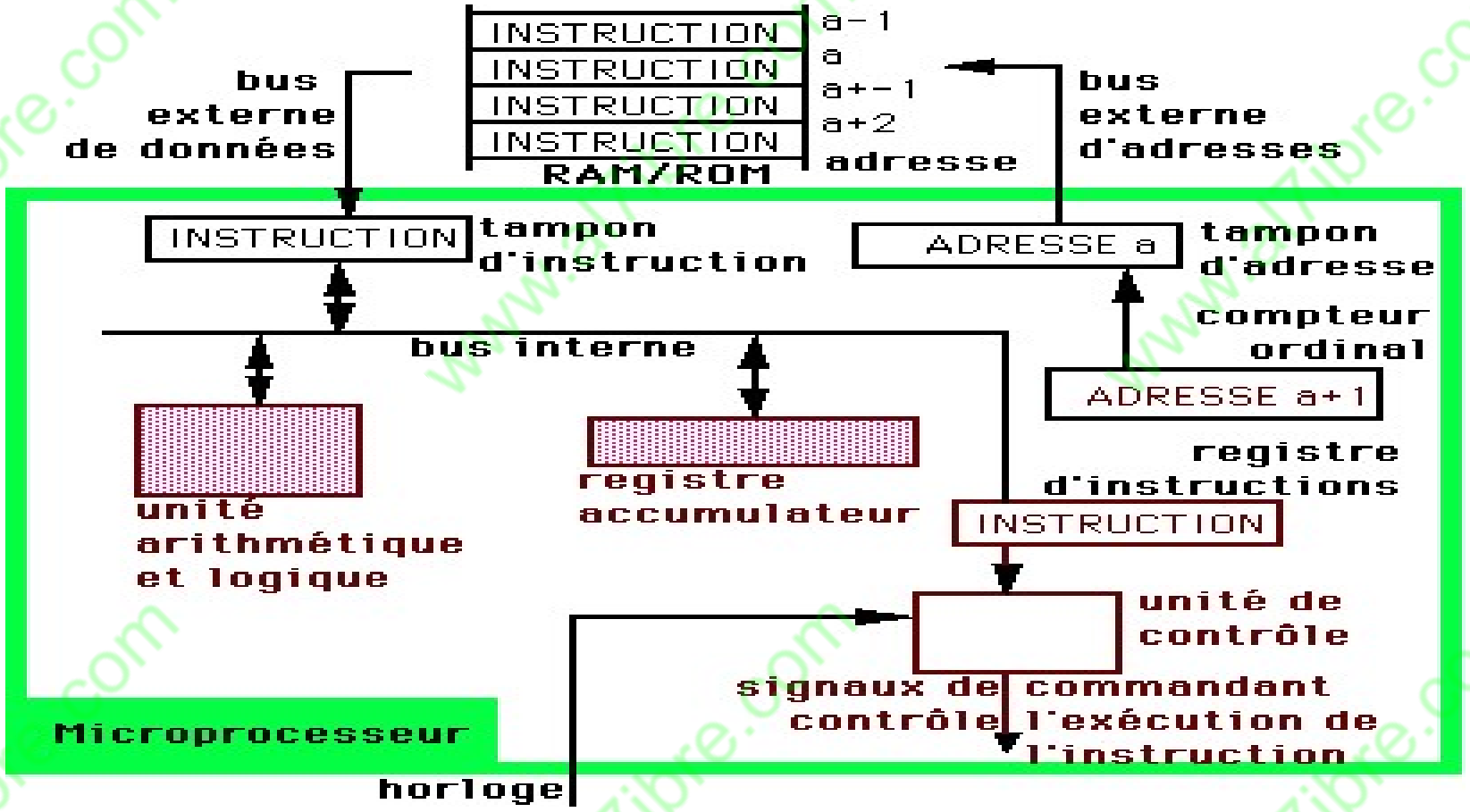
- **l'instruction à exécuter** va être **chargée** dans le **"registre instruction"** du **processeur**



Processeur 80x86

Exécution

- **L'instruction** est **décodée**, pour connaitre son **"code opération"** et ses **"parties adresses"**, puis **exécutée**



Processeur 80x86

Registres

- Le **microprocesseur 8086** contient **14 registres** repartis en **4 groupes** :
  - **Registres généraux (AX, BX, CX, DX)**
  - **Registres d'adressage de pointeurs et d'index (SP, BP, SI, DI)**
  - **Registres de segments (CS, DS, SS, ES)**
  - **Registre de Pointeur d'instruction (IP: Le compteur de programme)**
  - **Regnistre d'état (Flag)**

Processeur 80x86

Registres généraux (AX, BX, CX, DX) :

- Ils peuvent être **utilisés** dans **toutes** les **opérations arithmétiques** et **logiques** que le **programmeur** insère dans le **code assembleur**.
- Ils **servent** a contenir **temporairement** des **données**.
- Un **registre complet** est **composé** de **16 bits**, divisé en **deux** registres distincts de **8 bits**.
- Ce sont des **registres généraux** mais ils peuvent être **utilisés** pour des **opérations particulières**.

Processeur 80x86

Registres généraux (AX, BX, CX, DX) :

- **Accumulateur** (**AX**)
- **Base** (**BX**)
- **Counter** (**CX**)
- **Accumulateur auxiliaire** (**DX**)

AH	AL
BH	BL
CH	CL
DH	DL

Processeur 80x86

Le registre AX (Accumulateur) :

- Il permet de faire toutes les opérations de transfert de données avec les mémoires, le traitement des chaines de caractères et les opérations arithmétiques et logiques.
- Il permet aussi de faire les conversions en BCD du résultat d'une opération arithmétique (addition, soustraction, multiplication et division)

Processeur 80x86

Le registre BX (Registre de base) :

- Il est utilise pour **l'adressage** de **données** dans une **zone mémoire différente de la zone code**
- Il contient, en général, une **adresse** de **décalage** par rapport a une **adresse** de **référence** (**segment de données DS**).
- Il peut aussi servir pour le **stockage intermédiaire** des **données**.

Processeur 80x86

Registres de pointeurs

- **Pointeur SP (Stack Pointer) :**
  - **Pointeur de pile** (la pile est une zone de sauvegarde de **données** en **cours d'exécution** d'un programme gérée en LIFO). Il pointe sur la **tête** de la **pile** pour indiquer le **dernier élément**.
  - **Associé** par défaut au **registre de segment SS** (**SS:SP**).
- **Pointeur BP (Base Pointer) :**
  - **Pointeur de base** : utilise pour **accéder** aux **données** de la **pile** lors d'appels de **sous-programmes** (CALL).
  - **Associé** au registre de **segment de la pile SS** (**SS : BP**)

Processeur 80x86

Registres d'index

- **Registre d'Index SI (Source Index) :**
  - Il permet de **pointer** sur la **mémoire**, Il forme en général un **décalage** (un **offset**) par rapport a une **base fixe** (le registre **DS**),
  - Il est utilisé aussi dans des **instructions** de **déplacement** de **données** comme **index** de **l'opérande source**.
- **Registre d'Index DI (Destination Index) :**
  - Il permet aussi de **pointer** sur la **mémoire**, il présente un **décalage** par rapport a une **base fixe** (**DS ou ES**),
  - Il sert aussi pour les **instructions** de **chaine de caractères** en **pointant** sur la **destination**.

➔Les **pointeurs** et les **index** contiennent des **adresses** de **cases mémoire**.

Processeur 80x86

Registres segment

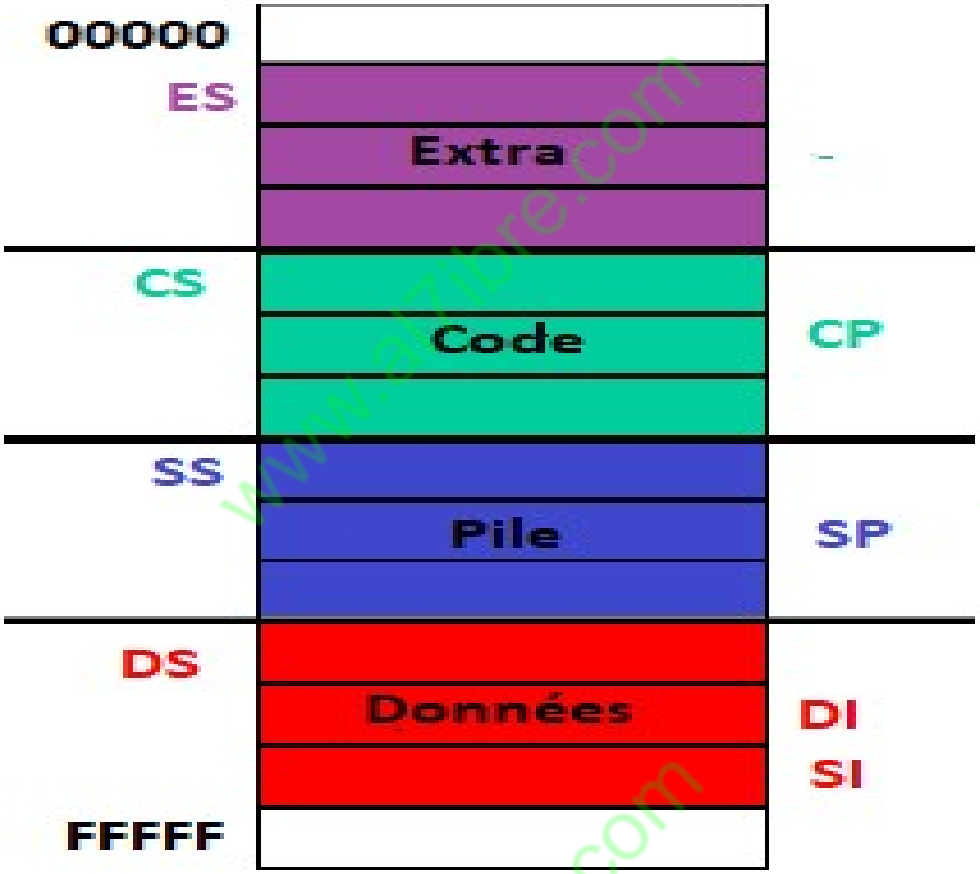
- Ces **registres segment** sont chargés de **sélectionner** les différents **segments** de la **mémoire** en pointant sur le **début** de chacun d'entre eux.
- Chaque **segment de mémoire** ne peut excéder les

  2^{16} octets = 64 Ko = 65536 o

  - **CS : Code Segment** : registre segment de **code** ;
  - **DS : Data Segment** : registre segment de **données** ;
  - **SS : Stack Segment** : registre segment de **pile** ;
  - **ES : Extra Segment** : registre segment **supplémentaire** pour les **données** ;

Processeur 80x86

Registres segment



Processeur 80x86

Registre CS (Code Segment) :

- Il pointe sur le **début** du segment qui contient les **codes** des **instructions du programme en cours**.
- Pour résoudre le **problème des programmes** qui ont une taille **supérieure** à **64 Ko**.
  - **Diviser** le code sur **plusieurs** segments **ne dépassant pas les 64 Ko**
  - **basculer** d'une partie à une autre du programme en **changeant la valeur du registre CS**

Processeur 80x86

Registre DS (Data Segment) :

- Le **registre segment de données** DS **pointe** sur le **segment** des **variables globales** du **programme**,
- **Sa taille ne peut excéder 64 K octets** (si on a des données qui **dépassent** cette limite, on utilise la **même astuce** citée précédemment mais dans ce cas on **change la valeur** de **DS**).

Processeur 80x86

Registre ES (Extra Segment) :

- Le **registre** de **données supplémentaires ES** est **utilise** par le **microprocesseur** lorsque **l'accès** aux autres **registres** est devenu **difficile** ou **impossible** pour **modifier** des **données**,
- Ce **segment** est **utilisé** aussi pour le **stockage** des **chaines de caractères.**

Processeur 80x86

Registre SS (Stack segment) :

- Le **registre SS** **pointe** sur la **pile**
- La **pile** est une **zone mémoire** où on peut **sauvegarder** les **registres** ou les **adresses** ou les **données** pour pouvoir les **récupérer** **après** **l'exécution** d'un **sous-programme** ou d'une **interruption**.

Processeur 80x86

Registre IP (Instruction Pointer) :

- Le **pointeur d'instructions** conserve l'adresse de l'emplacement mémoire de la prochaine instruction à exécuter pour l'indiquer au processeur
- Il est associé par défaut au registre de segment CS (CS : IP)
- Le processeur effectue les actions suivantes pour chaque instruction :
  - Lire et décoder l'instruction à l'adresse IP;
  - Incrémenter IP =IP + taille de l'instruction;
  - Exécuter l'instruction.

Processeur 80x86

Registre d'état (flags):

- **L'Indicateur d'état** est un **registre** sur **16 bits** qui sert à contenir **l'état** de certaines **opérations** effectuées par le **processeur**.
- Chaque **indicateur** est **manipulé individuellement** par des **instructions spécifiques**.
- Le **registre d'état** du **8086** est **formé** par les **bits** suivants :

Processeur 80x86

Registre d'état (Flag)

CF (Carry Flag : Retenue) :

- Cet indicateur est **positionne à 1** lorsqu'il y a une **retenue** du **resultat** à **8 ou 16 bits**.
- Il intervient dans les **opérations d'additions** (**retenue**) et de **soustractions** (**emprunt**) sur des **entiers naturels**.
- Il est **positionne** en particulier par les instructions **ADD**, **SUB** et **CMP** (comparaison entre deux valeurs).
- **CF = 1** s'il y a une **retenue** après **l'addition** ou la **soustraction** du **bit de poids fort des opérandes**.

PF (Parity Flag : Parité) :

- Si le **résultat de l'opération** contient un **nombre pair de 1**, cet indicateur est **positionne à 1**, sinon **zéro**.

Processeur 80x86

Registre d'état (Flag)

- **AF (Auxiliary Carry : Demie retenue) :**
  - Ce **bit est égal à 1** si on a une **retenue du quarter** (4 bits) de **poids faible** dans le **quarter** de **poids plus fort**.
- **ZF (Zero Flag : Zéro) :**
  - Cet indicateur est **positionné à 1** quand le **résultat** d'une **opération est égal à zéro**.
  - Lorsque une **soustraction** ou une **comparaison** sont effectuées, **ZF = 1** indique que les deux **opérandes** étaient **égaux**. Sinon, ZF est **mis a 0**.

Processeur 80x86

Registre d'état (flags)

- **SF (Sign Flag : signe) :**
  - **SF est positionne à 1** si le **bit de poids fort** du résultat d'une **addition** ou **soustraction** est 1 ; sinon **SF = 0**. Il est utile pour manipuler des entiers signés, car **le bit de poids fort donne alors le signe du résultat**.
- **OF (Overflow Flag : Débordement) :**
  - Si on a un **débordement arithmétique**, ce bit est **positionné à 1**, c'est a dire le **résultat** d'une **opération excède la capacité de l'opérande** (registre ou case mémoire), sinon **il est a 0**.

Processeur 80x86

Registre d'état (flags):

- **IF (Interrupt Flag : Masque d'interruption) :**
  - Pour **masquer** les **interruptions** venant de **l'extérieur**, ce bit **est mis à 0**, dans le cas contraire (**IF = 1**) le microprocesseur **reconnait l'interruption** de **l'extérieur**.
- **TF (Trap Flag : Piège) :**
  - Indique au **microprocesseur l'exécution pas à pas**.

➔Les bits **DF, IF et TF** sont des **indicateurs de contrôle** qui permettent de **modifier le comportement du microprocesseur**. Ils sont **positionnés** par le **programmeur**.

Processeur 80x86

Segmentation de la mémoire par le 8086

- **L'espace mémoire adressable** par le **8086** est de **1 Mo = 2^{20} octets = 1 048 576 octets** (**20 bits** du bus d'adresse).
- Cet **espace** est **divisé** en **segments logiques** avec accès **direct** et **simultané**.
- Un **segment est une zone mémoire** de **64 Ko** (**65 536 octets**) définie par **son adresse de départ** qui doit être un multiple de 16 où les **4 bits de poids faible sont a zéro**.
- **Le compteur programme** (**IP**) est de **16 bits** donc la possibilité **d'adressage** est de **2^{16} = 64 Ko** (ce qui ne couvre pas la **totalité** de la mémoire),

Processeur 80x86

Segmentation de la mémoire par le 8086

- **L'adresse** d'un **segment** est représenté avec seulement ses **16 bits** de **poids fort**, les **4 bits de poids faible** étant **implicitement à 0**.
- Une **valeur sur 16 bits** peut designer une **case mémoire** parmi les 2^{16}=65536 contenues dans un **segment**.
- Utilisation de **deux registres** pour indiquer une **adresse** au **processeur**.
- Chaque **segment débute** à **l'endroit spécifié** par le **registre segment**. Le **déplacement** (**offset**) à **l'intérieur** de chaque **segment** se fait par un **registre de décalage** qui permet de trouver une **information** à **l'intérieur du segment**.

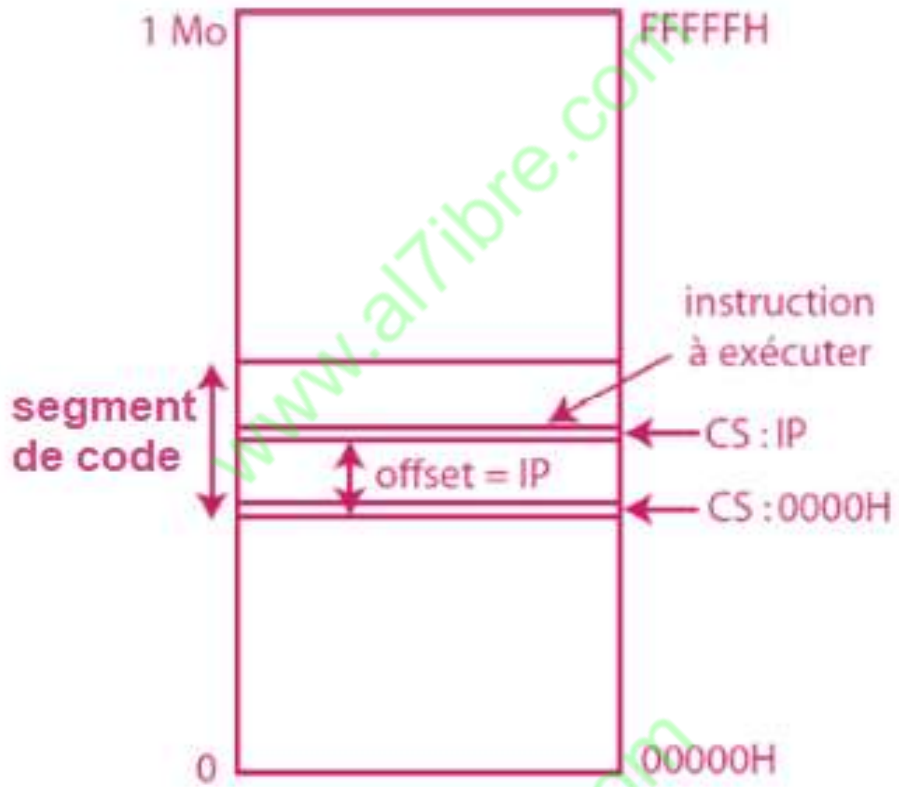
Processeur 80x86

Segmentation de la mémoire par le 8086

- Une **case mémoire** est repérée par le 8086 au moyen de **deux quantités** (registres) sur 16 bits :
  - **l'adresse d'un segment ;**
  - **un déplacement ou offset (appelé aussi adresse effective) dans ce segment.**
- **Exemple** : la paire de registres **CS : IP** : pointe sur le **code d'une instruction** (**CS registre segment** et **IP déplacement**).

Processeur 80x86

Segmentation de la mémoire par le 8086



Processeur 80x86

Segmentation de la mémoire par le 8086

- Le **couple (segment, offset)** définit une **adresse logique**, notée sous la forme :

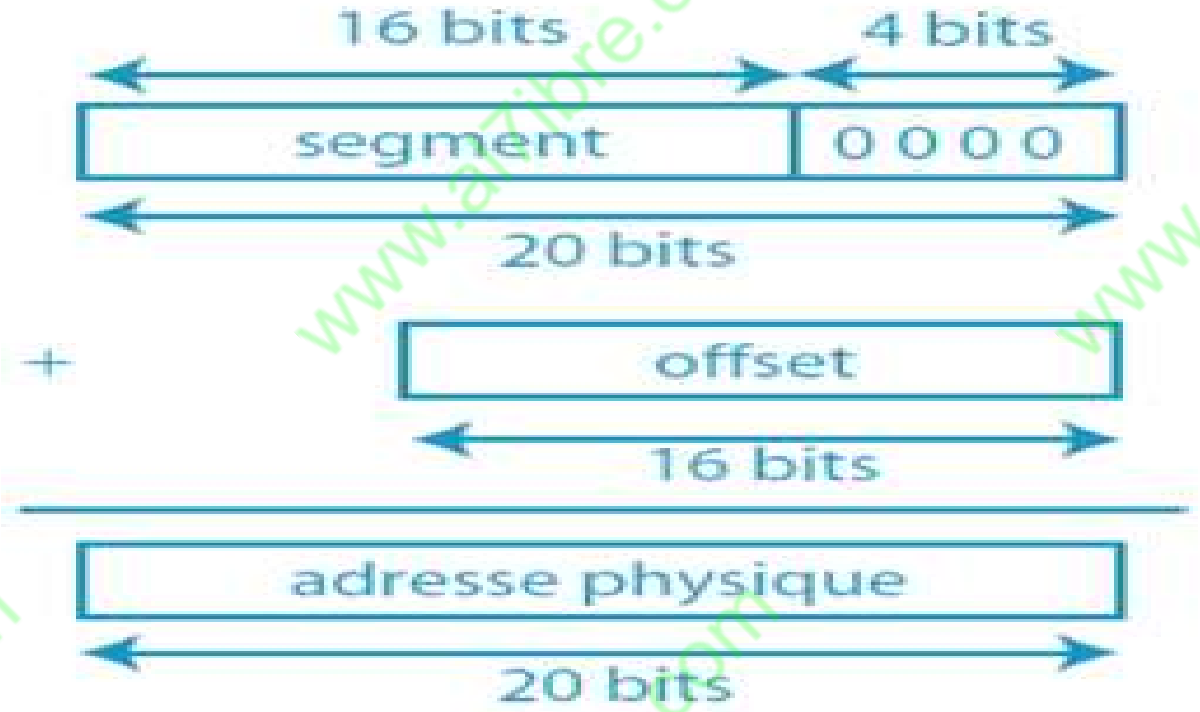
  segment : offset

- **L'adresse** d'une **case mémoire** donnée sous la forme d'une quantité sur **20 bits** (5 digits hexa) est appelée **adresse physique** car elle correspond a la **valeur envoyée** réellement sur le **bus d'adresses A0 - A19**.

Processeur 80x86

Segmentation de la mémoire par le 8086

- **Correspondance** *entre* **adresse logique** *et* **adresse physique** :



Processeur 80x86

Segmentation de la mémoire par le 8086

- **L'adresse physique** se calcule par **l'expression** :

adresse physique = segment × 10(16) + offset

- Le fait **d'injecter 4 zéros** en **poids faible** du **segment** revient a effectuer un **décalage de 4 positions vers la gauche**, c'est-a-dire une **multiplication** par 2^4 = 16 (décimal) = 10 (H).

Processeur 80x86

Segmentation de la mémoire par le 8086

- Le **registre CS** est **associe** au **pointeur d'instruction IP**, la **prochaine instruction à exécuter** se trouve a **l'adresse logique CS:IP.**
- Les **registres de segments DS et ES** peuvent être **associés** à un **registre d'index DS:SI ; ES:DI**
- Le **registre de segment de pile SS** peut être **associe** aux **registres de pointeurs SS : SP ou SS : BP**

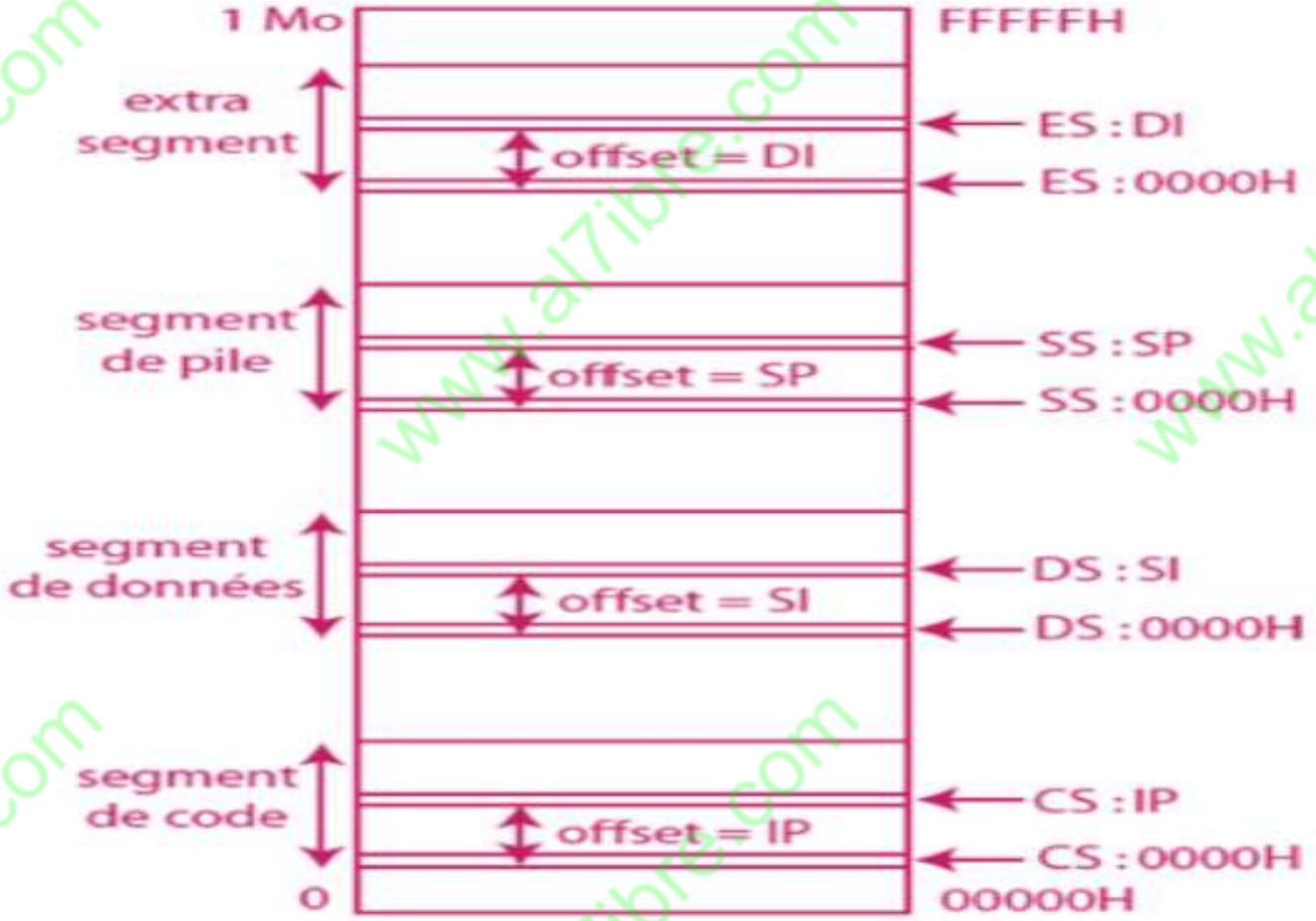
Processeur 80x86

Mémoire accessible par le 8086 :

- Le **programmeur** en **assembleur** doit se **charger** de **l'initialisation de DS**, c'est-a-dire de lui **affecter** l'adresse du **segment** de **données** à utiliser.
- Le **registre CS** sera **automatiquement initialisé** sur le **segment** contenant la **première instruction du code** au moment du chargement en mémoire du **programme**.

Processeur 80x86

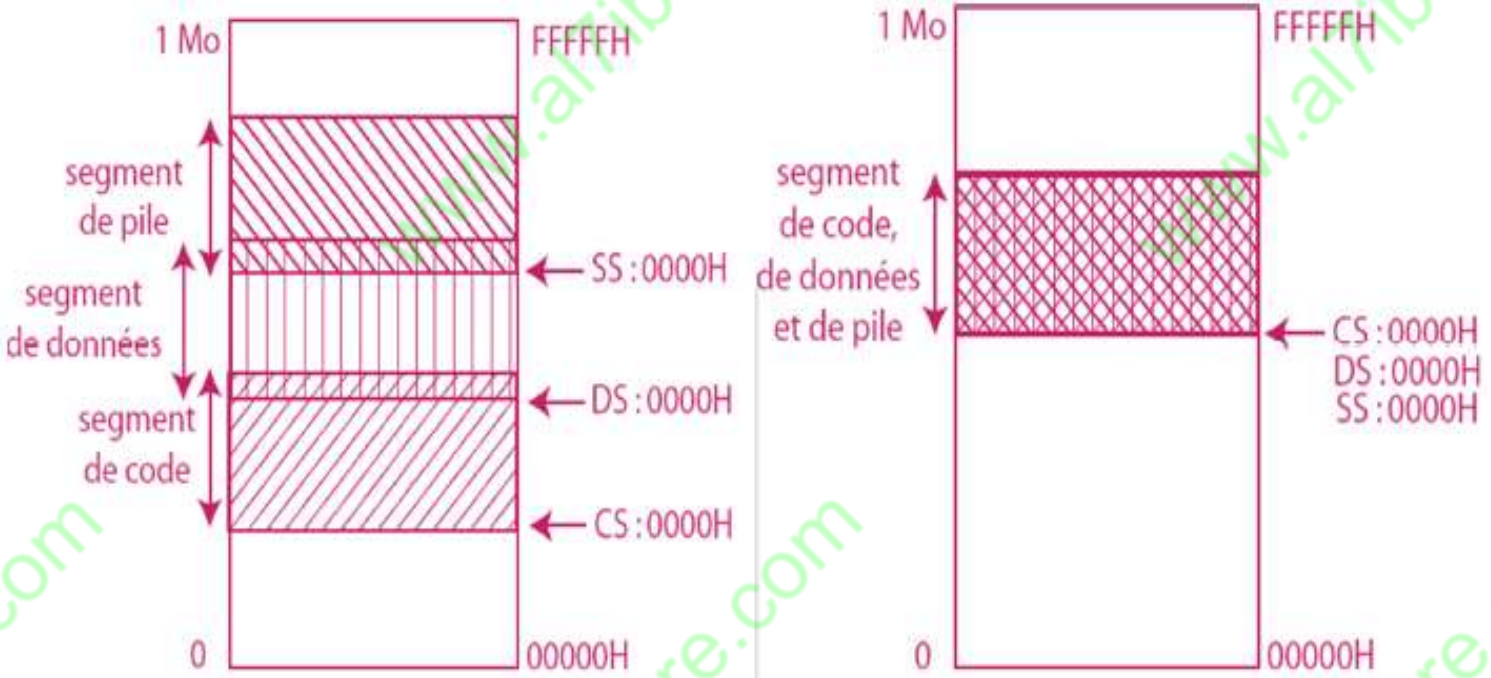
Mémoire accessible par le 8086 :



Processeur 80x86

Mémoire accessible par le 8086 :

- les **segments** ne sont pas nécessairement **distincts** les **uns des autres**, ils peuvent se **chevaucher** ou se **recouvrir complètement**.



Processeur 80x86

Mémoire accessible par le 8086 :

- Le **nombre** de **segments** utilisés définit le **modèle mémoire du programme.**
- Contenu des **registres** après un **RESET** du **microprocesseur** :
  - **IP = 0000H**
  - **CS = FFFFH**
  - **DS = 0000H**
  - **ES = 0000H**
  - **SS = 0000H**
- la **première instruction** exécutée par le 8086 se trouve donc a l'adresse **logique CS:IP = FFFFH** correspondant a l'adresse **physique FFFF0H**

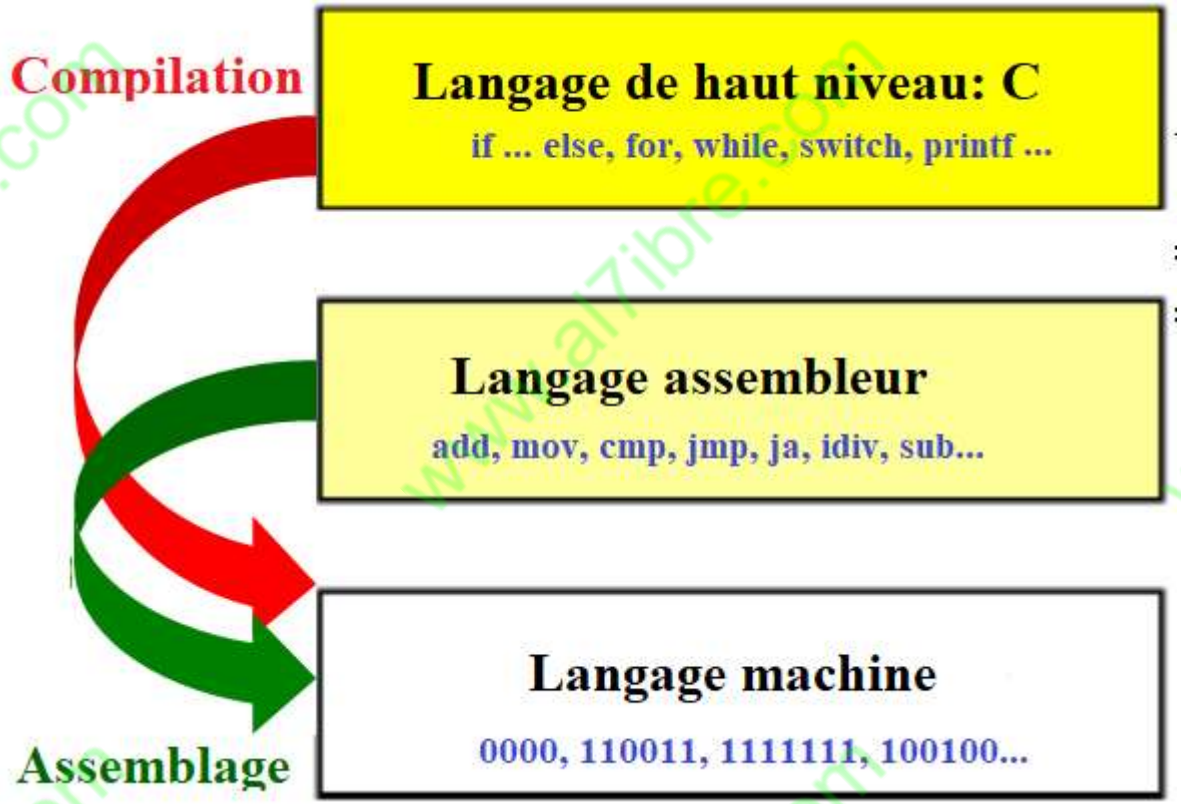
Assembleur

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**.

Assembleur

Langages



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…

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**).

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)**

Assembleur

Instruction d'assembleur

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

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

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**.

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
```

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 offset 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.

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

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** (car **64Ko=256o**).
- Un **programme .COM commence** a partir **de l'offset 100h**.

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.**

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

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**.

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**.

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

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

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.

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
```

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

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.

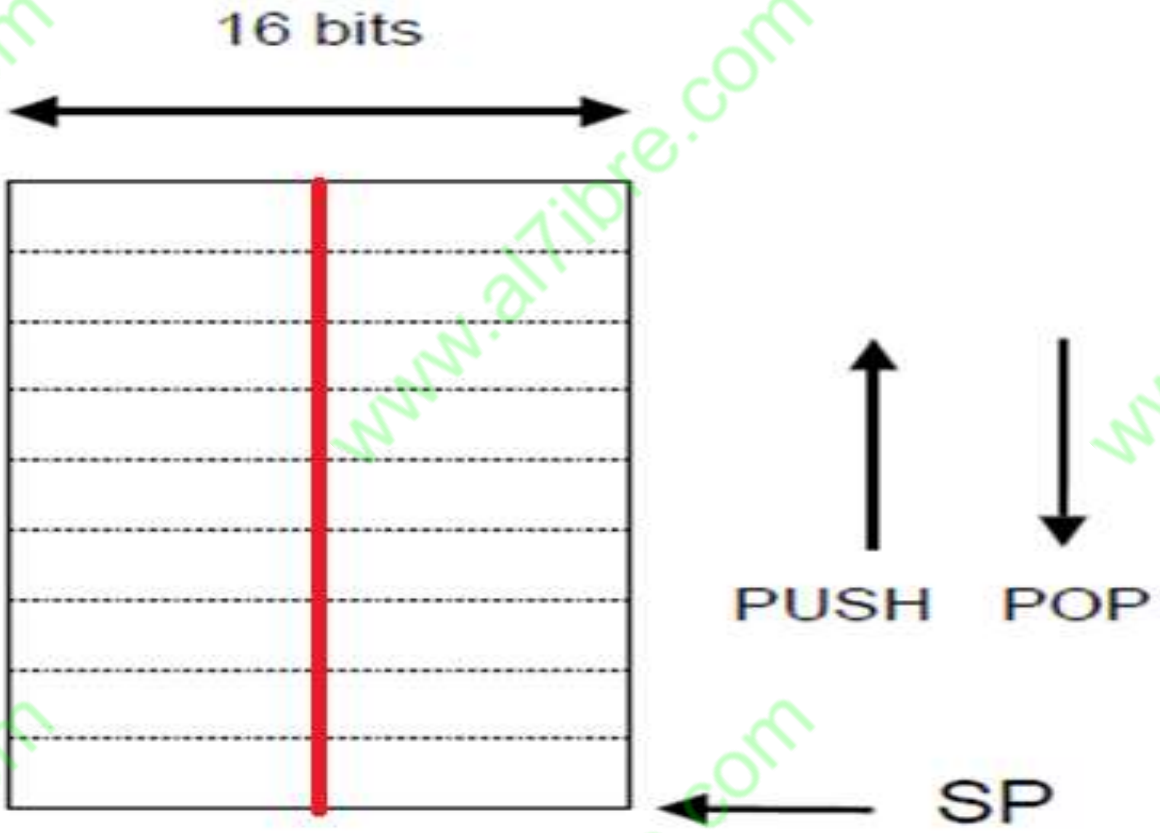
  SP = SP- 2
- **POP registre**: **Retire** la valeur en haut de la pile et la place dans le registres spécifié.

  SP = SP+ 2

Assembleur

Principe de pile

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



Assembleur

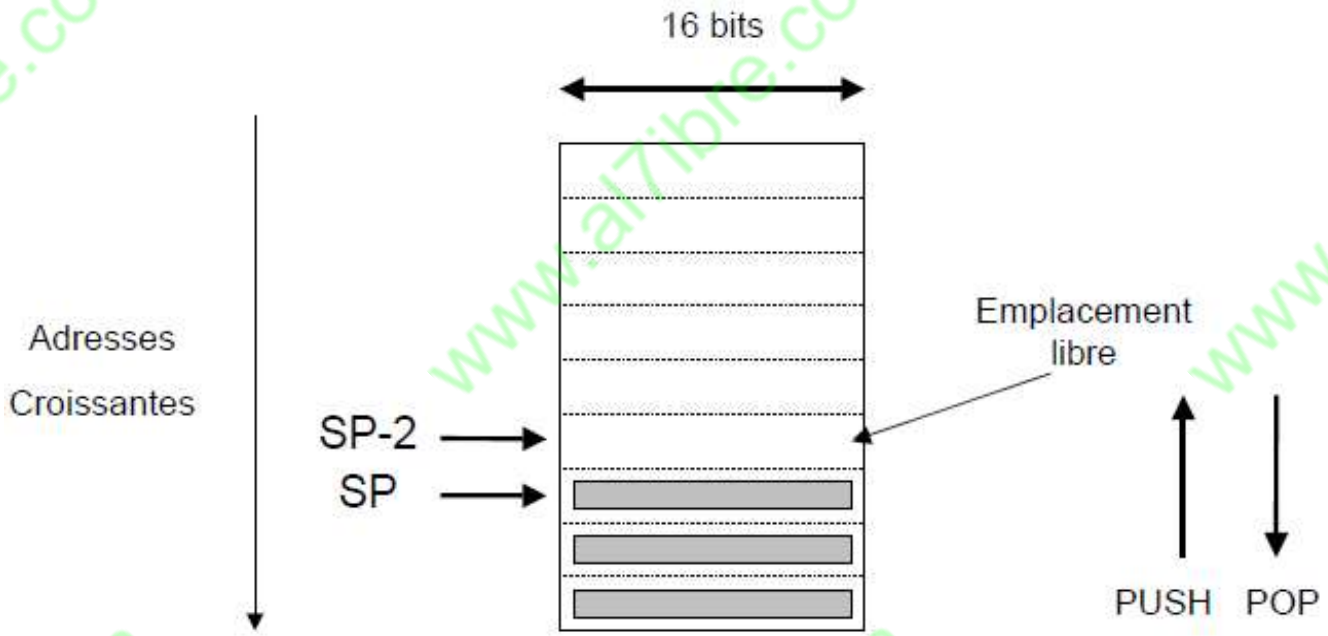
Pile: exemple

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
```

Assembleur

Illustration du fonctionnement de la pile :



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**.

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**

Assembleur

L'émulateur Emu8086

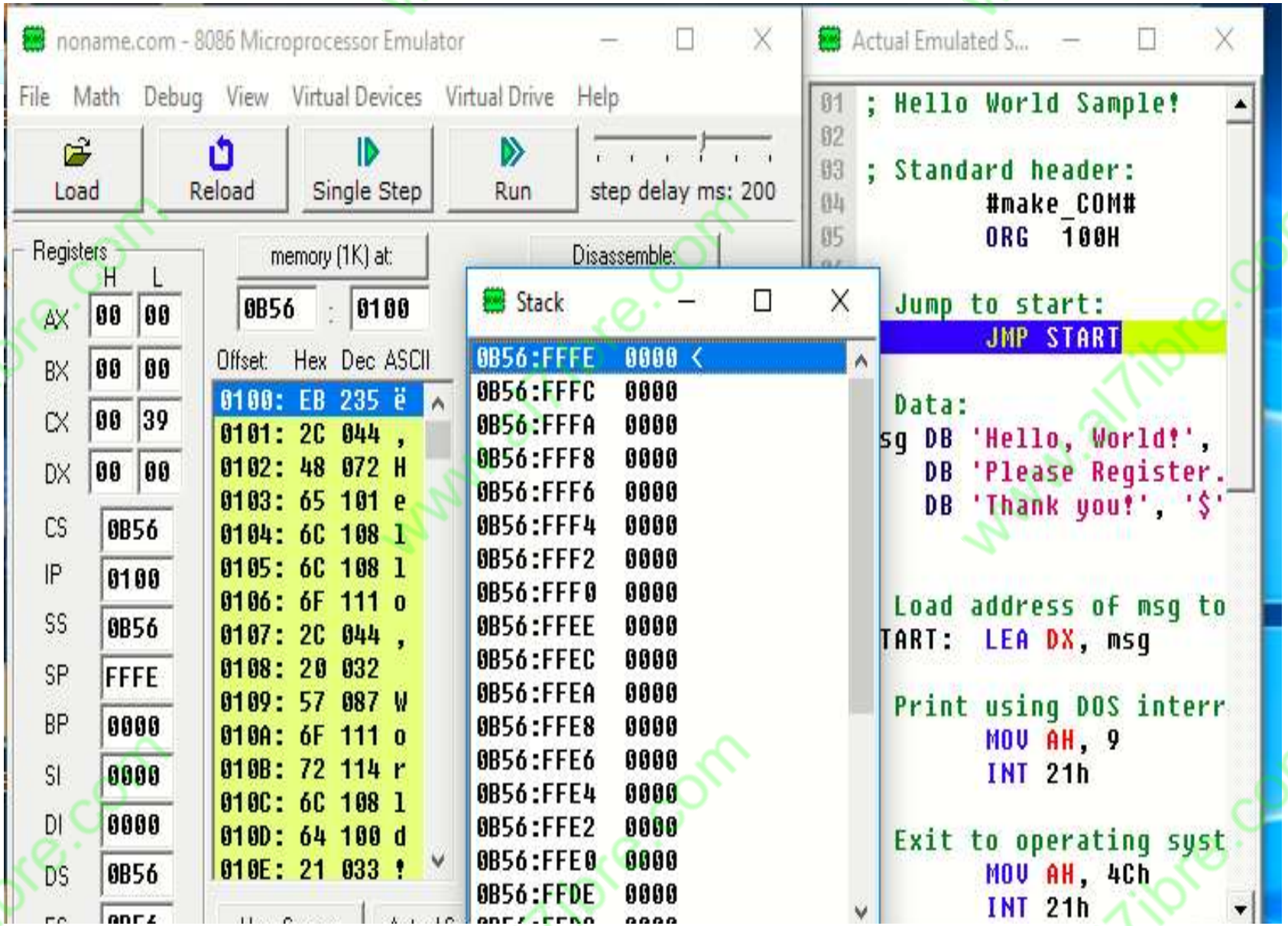
- Il se **présente sous forme** d'un **éditeur de texte classique**, avec le support d'une **colorisation syntaxique** du **code assembleur**.
- Dans cet **émulateur**, ils existent certaines **options** du **menu** '**view**' qui peuvent être **pratiques**.

Assembleur

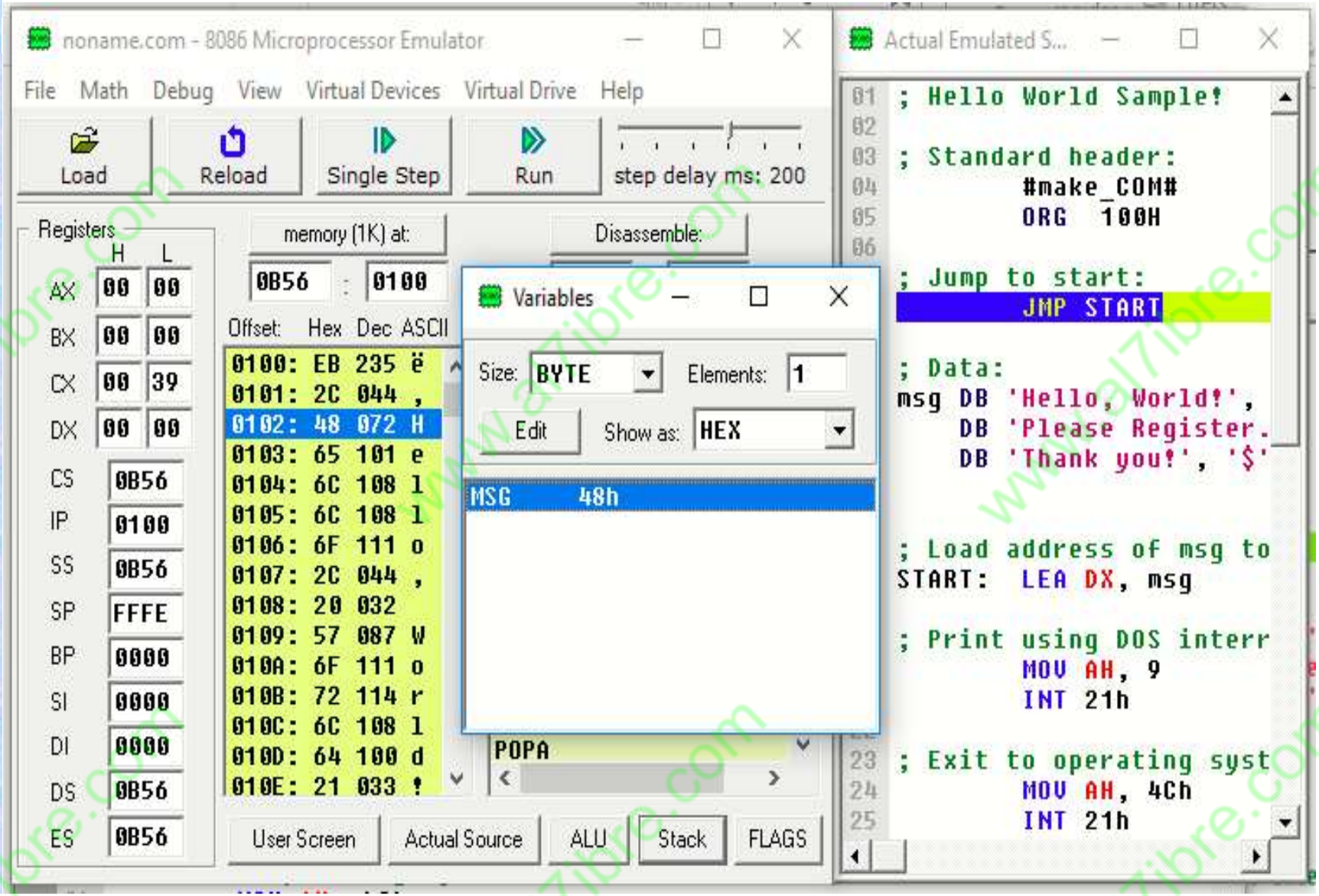
L'émulateur Emu8086

- **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**.

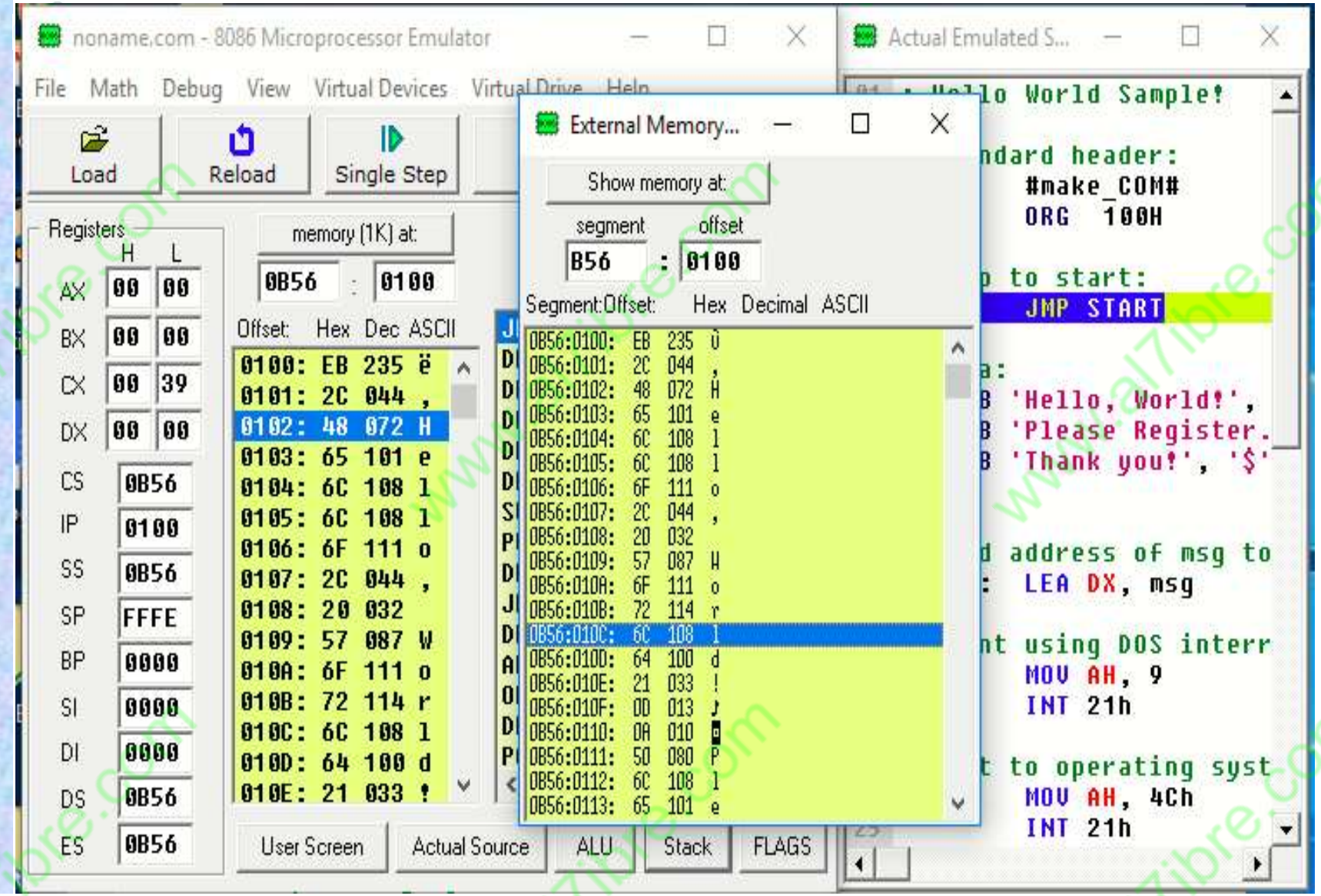
Assembleur


noname.com - 8086 Microprocessor Emulator
File Math Debug View Virtual Devices Virtual Drive Help
Load
Reload
Single Step
Run
step delay ms: 200
Registers
H L
AX 00 00
BX 00 00
CX 00 39
DX 00 00
CS 0B56
IP 0100
SS 0B56
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0B56
memory (1K) at:
0B56 : 0100
Disassemble:
Offset: Hex Dec ASCII
0100: EB 235 ë
0101: 2C 044 ,
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
0108: 20 032
0109: 57 087 W
010A: 6F 111 o
010B: 72 114 r
010C: 6C 108 l
010D: 64 100 d
010E: 21 033 !
Stack
0B56:FFFE 0000
0B56:FFFC 0000
0B56:FFFA 0000
0B56:FFF8 0000
0B56:FFF6 0000
0B56:FFF4 0000
0B56:FFF2 0000
0B56:FFF0 0000
0B56:FFEE 0000
0B56:FFEC 0000
0B56:FFEA 0000
0B56:FFE8 0000
0B56:FFE6 0000
0B56:FFE4 0000
0B56:FFE2 0000
0B56:FFE0 0000
0B56:FFDE 0000
Actual Emulated S...
01 ; Hello World Sample!
02
03 ; Standard header:
04 #make_COM#
05 ORG 100H
Jump to start:
JMP START
Data:
sg DB 'Hello, World!',
DB 'Please Register.
DB 'Thank you!', '$'
Load address of msg to
TART: LEA DX, msg
Print using DOS interr
MOV AH, 9
INT 21h
Exit to operating syst
MOV AH, 4Ch
INT 21h

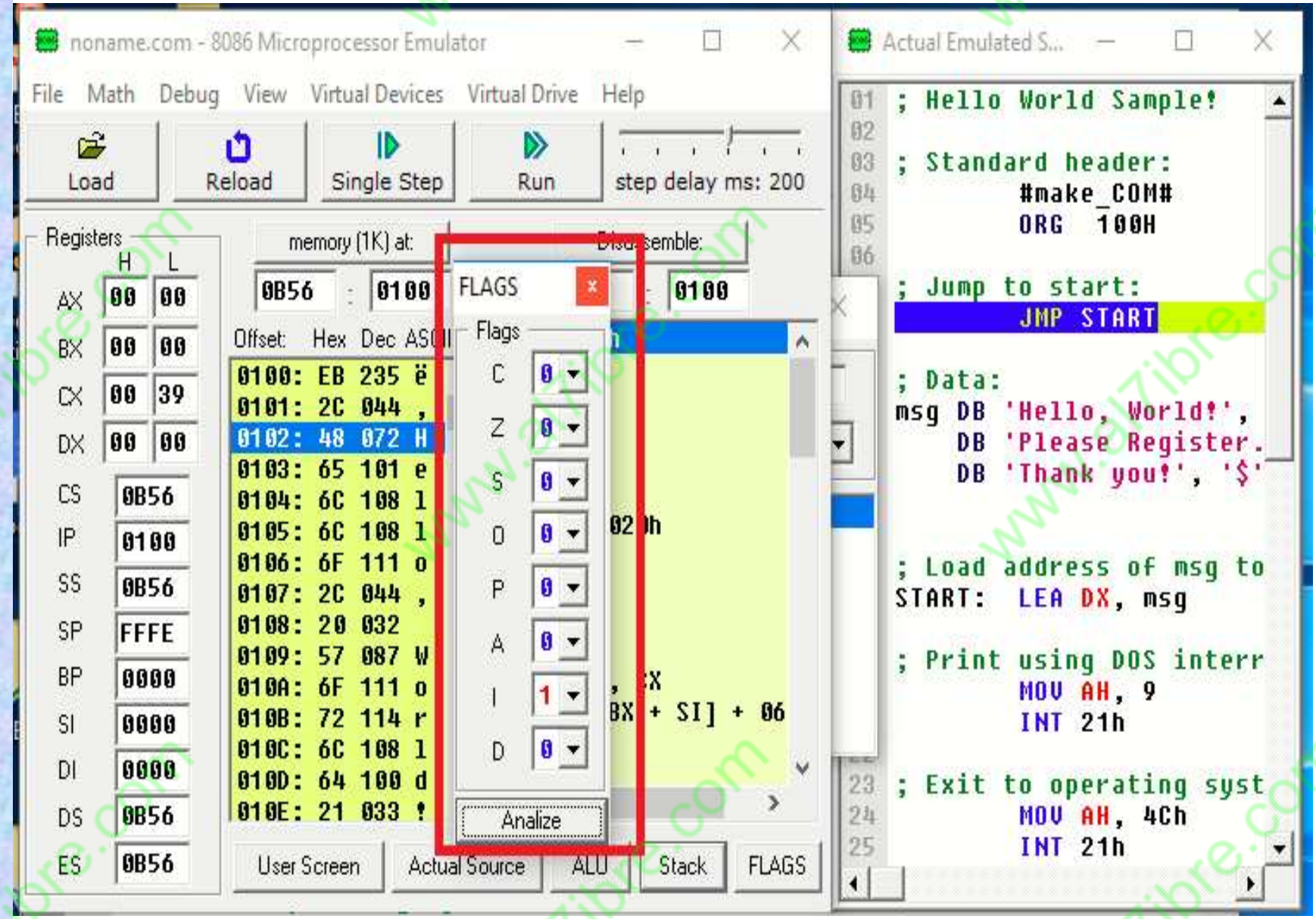
Assembleur


noname.com - 8086 Microprocessor Emulator
File Math Debug View Virtual Devices Virtual Drive Help
Load
Reload
Single Step
Run
step delay ms: 200
Registers
H L
AX 00 00
BX 00 00
CX 00 39
DX 00 00
CS 0B56
IP 0100
SS 0B56
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0B56
ES 0B56
memory (1K) at:
0B56 : 0100
Disassemble:
Offset: Hex Dec ASCII
0100: EB 235 ë
0101: 2C 044 ,
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
0108: 20 032
0109: 57 087 W
010A: 6F 111 o
010B: 72 114 r
010C: 6C 108 l
010D: 64 100 d
010E: 21 033 !
POPA
User Screen
Actual Source
ALU
Stack
FLAGS
Variables
Size: BYTE
Elements: 1
Edit
Show as: HEX
MSG 48h
Actual Emulated S...
; Hello World Sample!
; Standard header:
#make_COM#
ORG 100H
; Jump to start:
JMP START
; Data:
msg DB 'Hello, World!',
DB 'Please Register.
DB 'Thank you!', '$'
; Load address of msg to
START: LEA DX, msg
; Print using DOS interr
MOV AH, 9
INT 21h
; Exit to operating syst
MOV AH, 4Ch
INT 21h

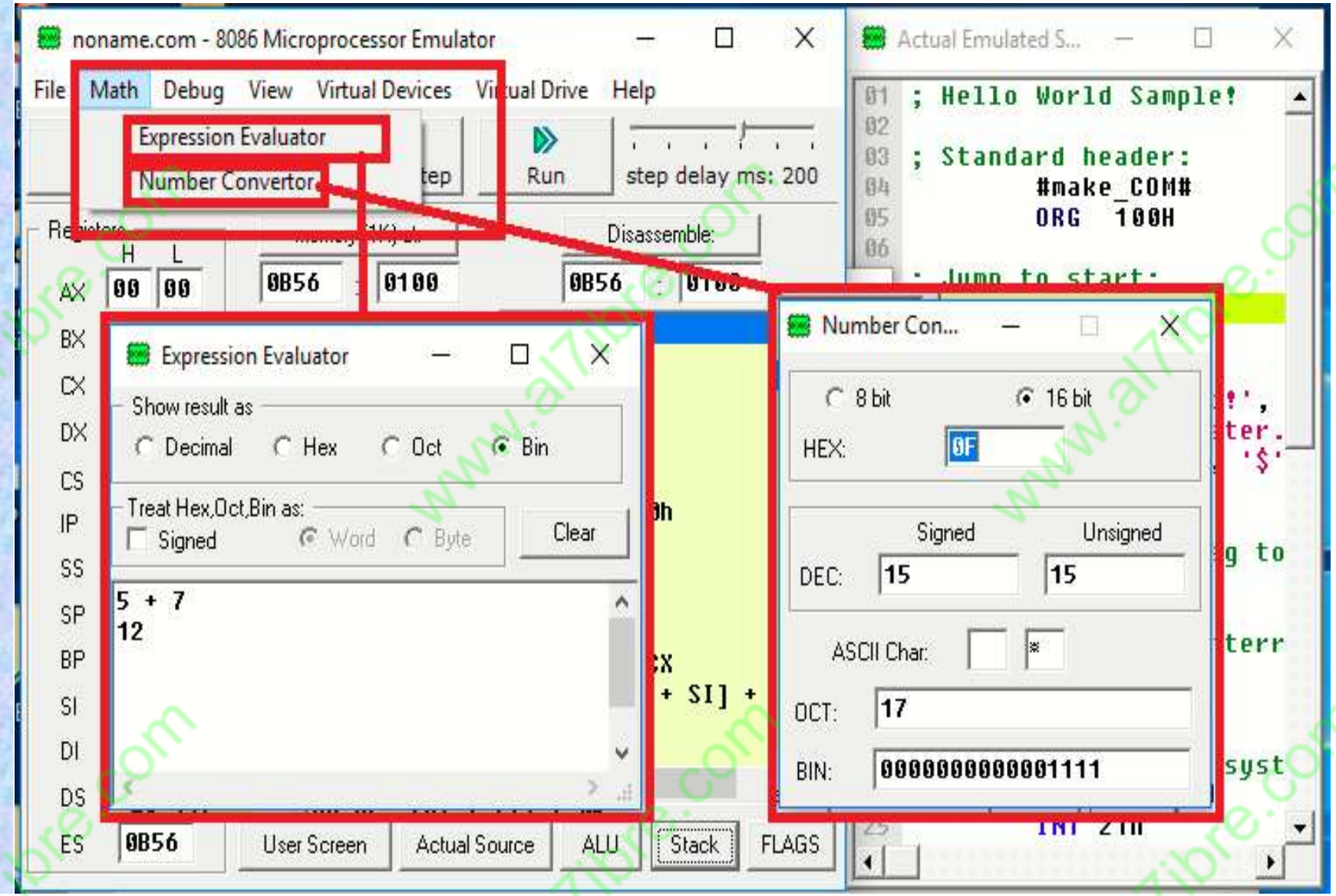
Assembleur


noname.com - 8086 Microprocessor Emulator
File Math Debug View Virtual Devices Virtual Drive Help
Load
Reload
Single Step
Registers
H L
AX 00 00
BX 00 00
CX 00 39
DX 00 00
CS 0B56
IP 0100
SS 0B56
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0B56
ES 0B56
memory (1K) at:
0B56 : 0100
Offset: Hex Dec ASCII
0100: EB 235 ë
0101: 2C 044 ,
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
0108: 20 032
0109: 57 087 W
010A: 6F 111 o
010B: 72 114 r
010C: 6C 108 l
010D: 64 100 d
010E: 21 033 !
User Screen
Actual Source
ALU
Stack
FLAGS
External Memory...
Show memory at:
segment offset
B56 : 0100
Segment:Offset: Hex Decimal ASCII
0B56:0100: EB 235 ë
0B56:0101: 2C 044 ,
0B56:0102: 48 072 H
0B56:0103: 65 101 e
0B56:0104: 6C 108 l
0B56:0105: 6C 108 l
0B56:0106: 6F 111 o
0B56:0107: 2C 044 ,
0B56:0108: 20 032
0B56:0109: 57 087 W
0B56:010A: 6F 111 o
0B56:010B: 72 114 r
0B56:010C: 6C 108 l
0B56:010D: 64 100 d
0B56:010E: 21 033 !
0B56:010F: 0D 013
0B56:0110: 0A 010
0B56:0111: 50 080 P
0B56:0112: 6C 108 l
0B56:0113: 65 101 e
Actual Emulated S...
lo World Sample!
ndard header:
#make_COM#
ORG 100H
p to start:
JMP START
a:
B 'Hello, World!',
B 'Please Register.
B 'Thank you!', '$'
d address of msg to
: LEA DX, msg
nt using DOS interr
MOV AH, 9
INT 21h
t to operating syst
MOV AH, 4Ch
INT 21h

Assembleur

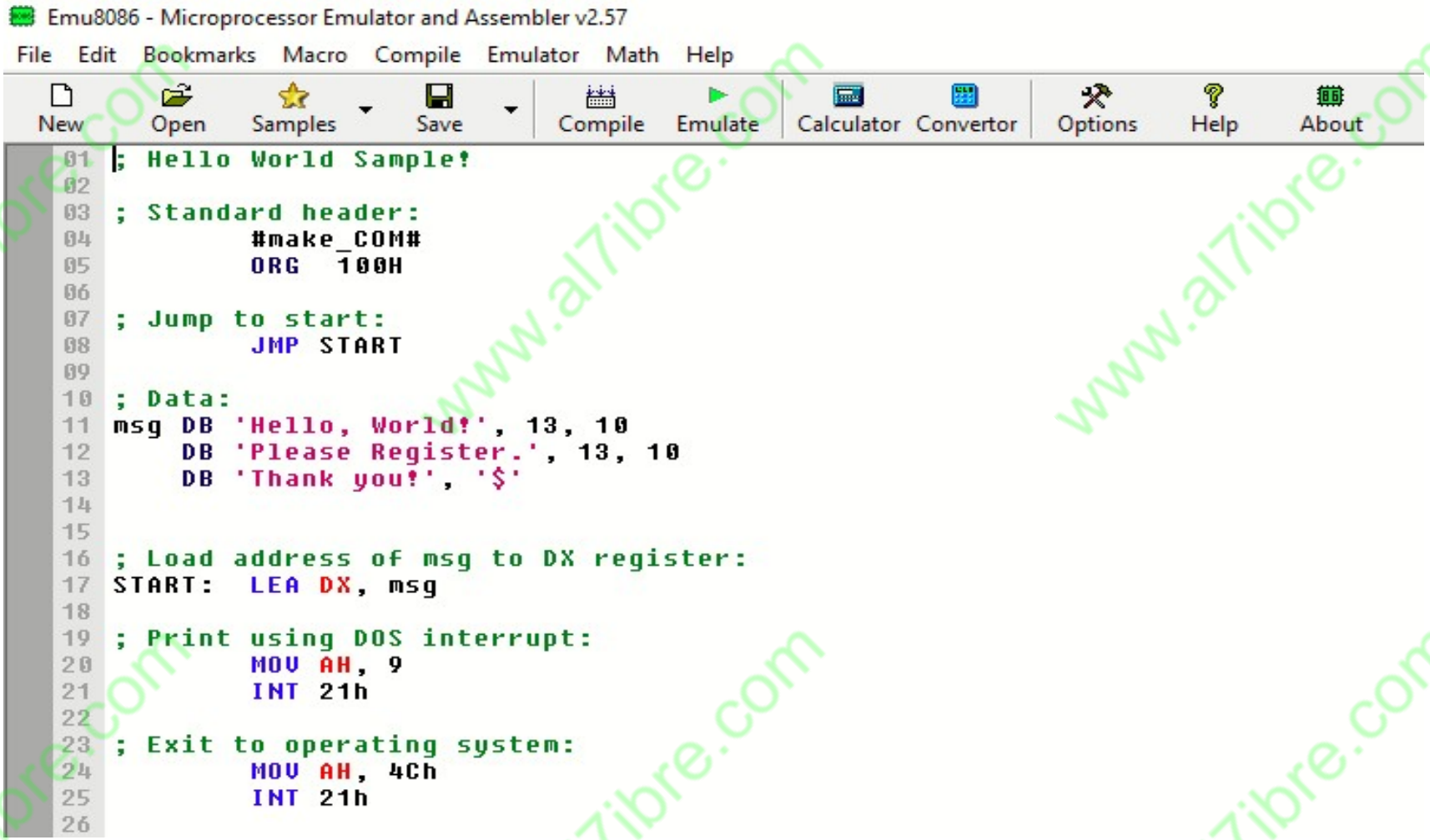

noname.com - 8086 Microprocessor Emulator
File Math Debug View Virtual Devices Virtual Drive Help
Load
Reload
Single Step
Run
step delay ms: 200
Registers
H L
AX 00 00
BX 00 00
CX 00 39
DX 00 00
CS 0B56
IP 0100
SS 0B56
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0B56
ES 0B56
memory (1K) at:
0B56 : 0100
Offset: Hex Dec ASCII
0100: EB 235 ë
0101: 2C 044 ,
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
0108: 20 032
0109: 57 087 W
010A: 6F 111 o
010B: 72 114 r
010C: 6C 108 l
010D: 64 100 d
010E: 21 033 !
FLAGS
Flags
C 0
Z 0
S 0
O 0
P 0
A 0
I 1
D 0
Analize
User Screen
Actual Source
ALU
Stack
FLAGS
Actual Emulated S...
; Hello World Sample!
; Standard header:
#make_COM#
ORG 100H
; Jump to start:
JMP START
; Data:
msg DB 'Hello, World!',
DB 'Please Register.
DB 'Thank you!', '$'
; Load address of msg to
START: LEA DX, msg
; Print using DOS interr
MOV AH, 9
INT 21h
; Exit to operating syst
MOV AH, 4Ch
INT 21h

Assembleur


noname.com - 8086 Microprocessor Emulator
File
Math
Debug
View
Virtual Devices
Help
Expression Evaluator
Number Convertor
Run
step delay ms: 200
Disassemble:
Actual Emulated S...
; Hello World Sample!
; Standard header:
#make_COM#
ORG 100H
Expression Evaluator
Show result as
Decimal
Hex
Oct
Bin
Treat Hex,Oct,Bin as:
Signed
Word
Byte
Clear
5 + 7
12
Number Con...
8 bit
16 bit
HEX:
0F
Signed
Unsigned
DEC:
15
15
ASCII Char:
OCT:
17
BIN:
0000000000001111
AX
BX
CX
DX
CS
IP
SS
SP
BP
SI
DI
DS
ES
0B56
User Screen
Actual Source
ALU
Stack
FLAGS

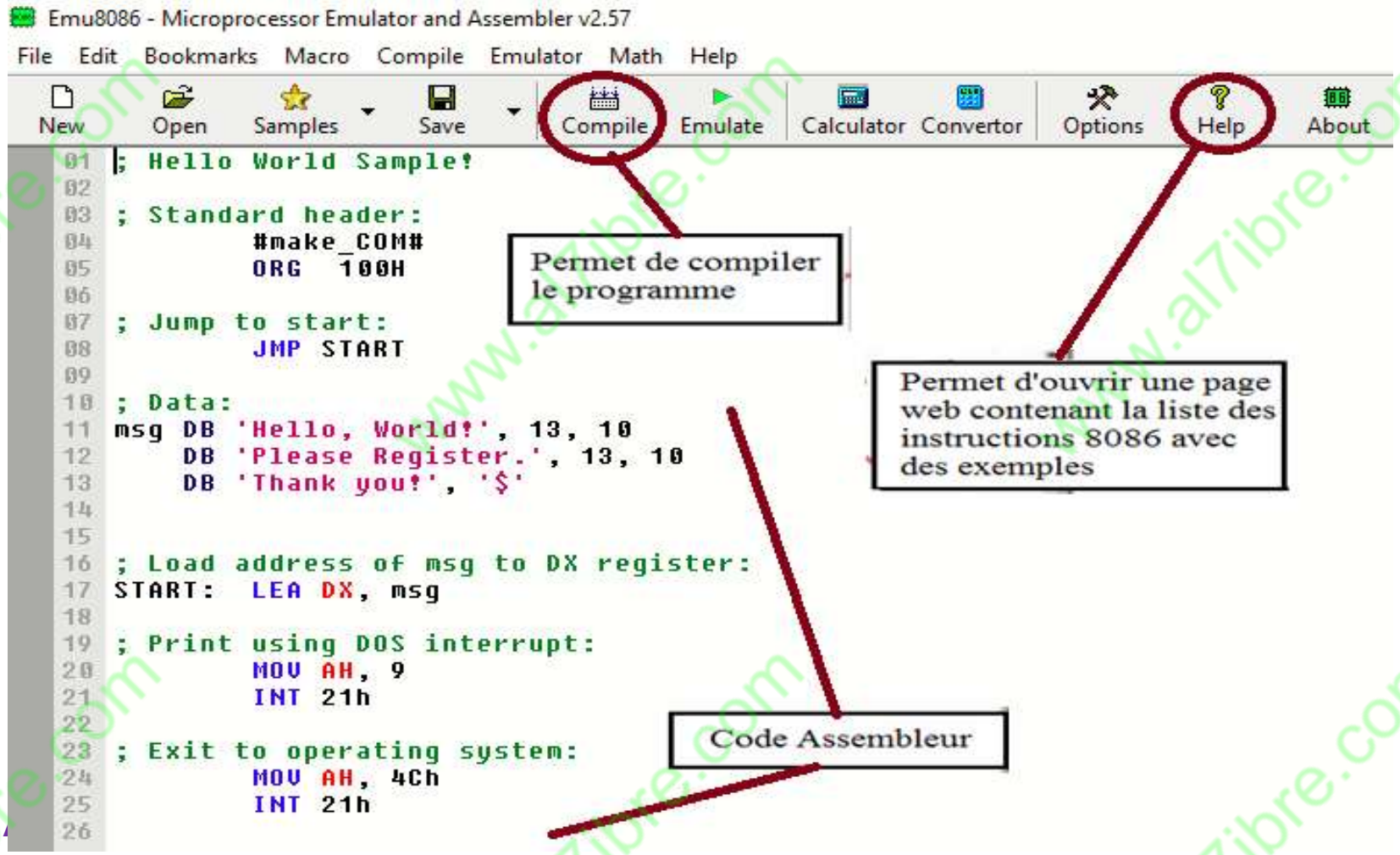
Assembleur

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

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**

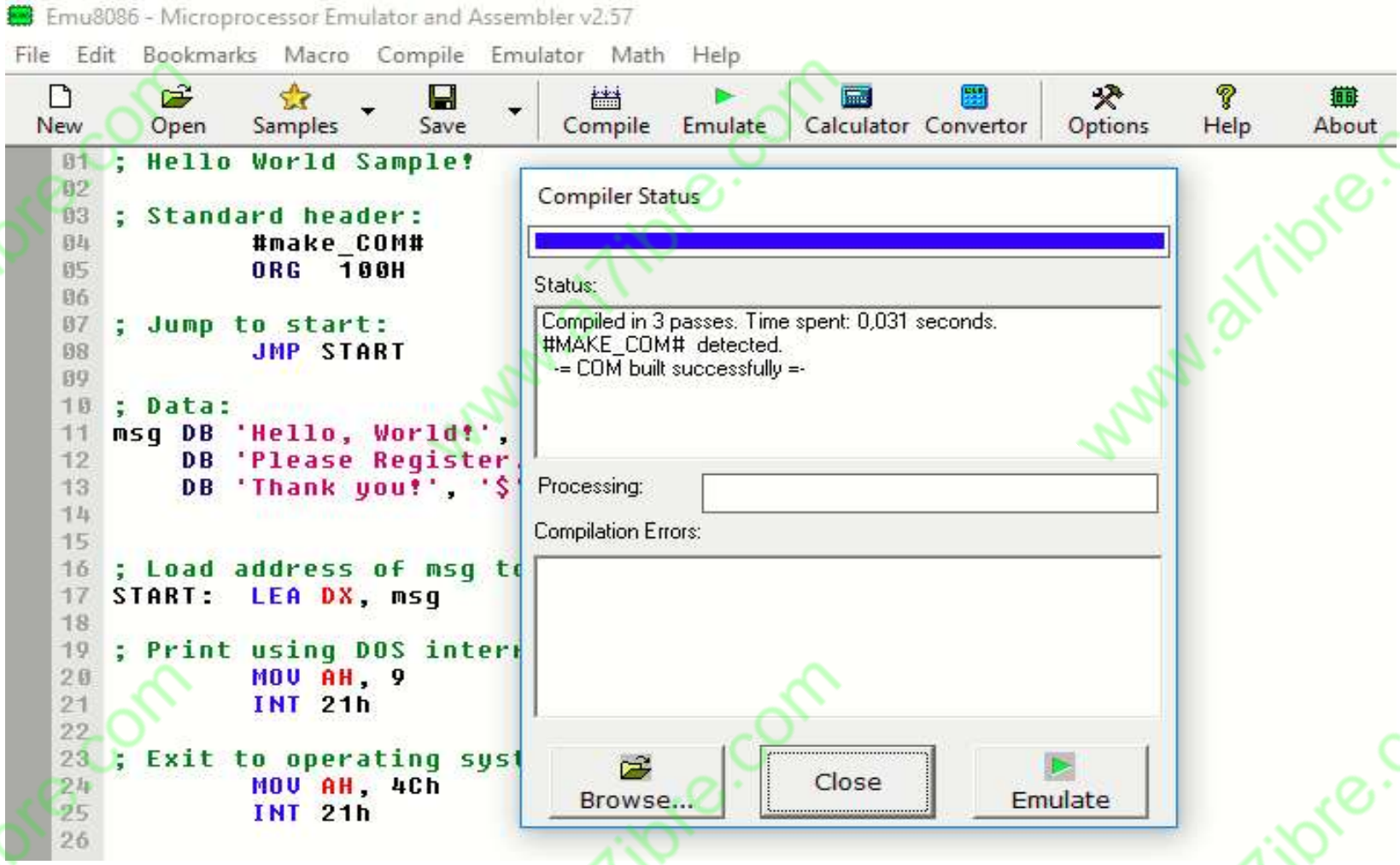
Assembleur

L'émulateur Emu8086

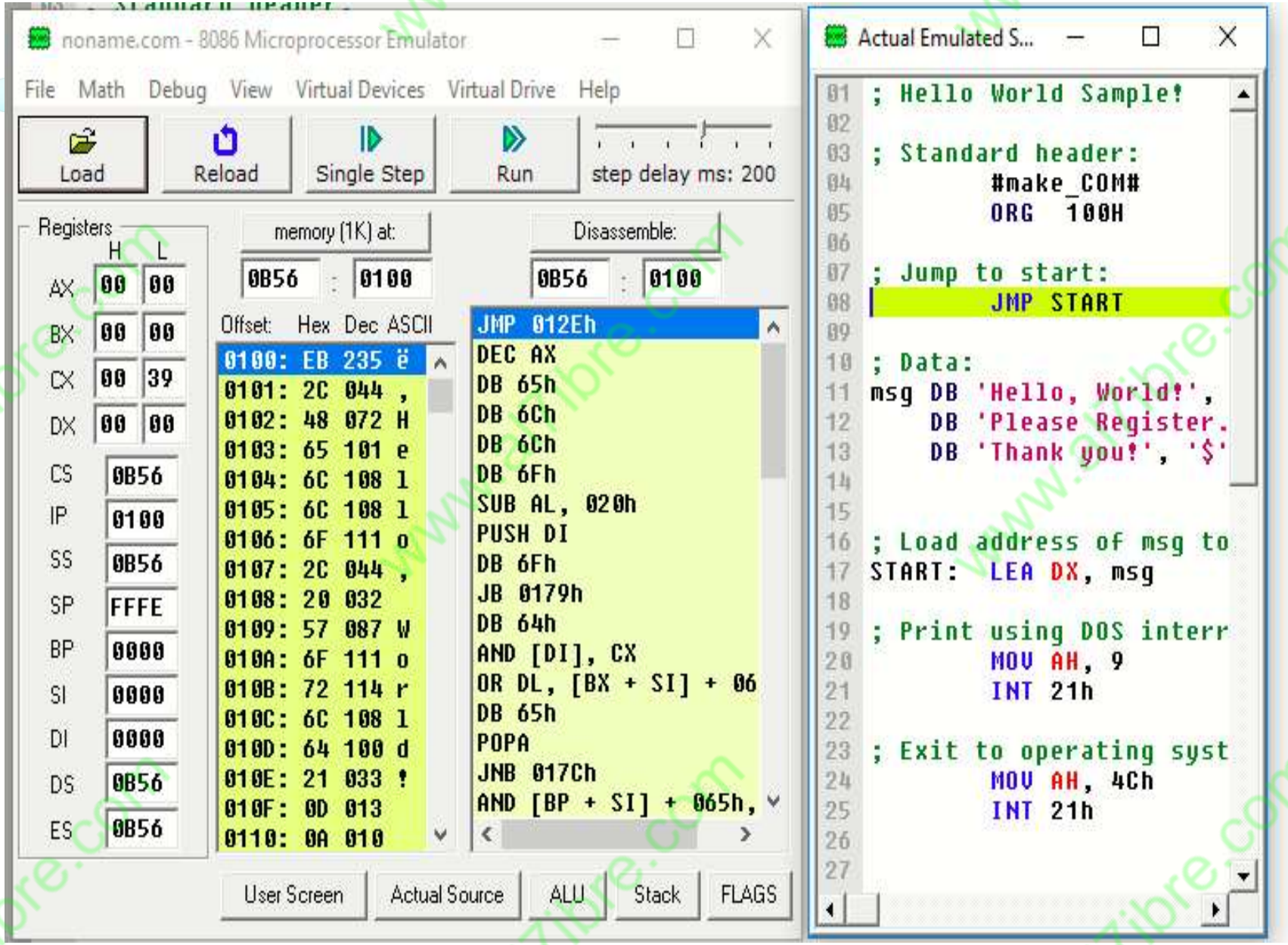
- **Une fois le code écrit** et **sauvegarder** sous un nom avec l'extension **.asm**, il faut le **compiler** en cliquant sur '**Compile**'
- **S'il ne contient pas d'erreurs**, alors il peut être **exécuter** en cliquant soit sur '**Emulate**' soit sur '**Run**'

Assembleur

L'émulateur Emu8086



Assembleur

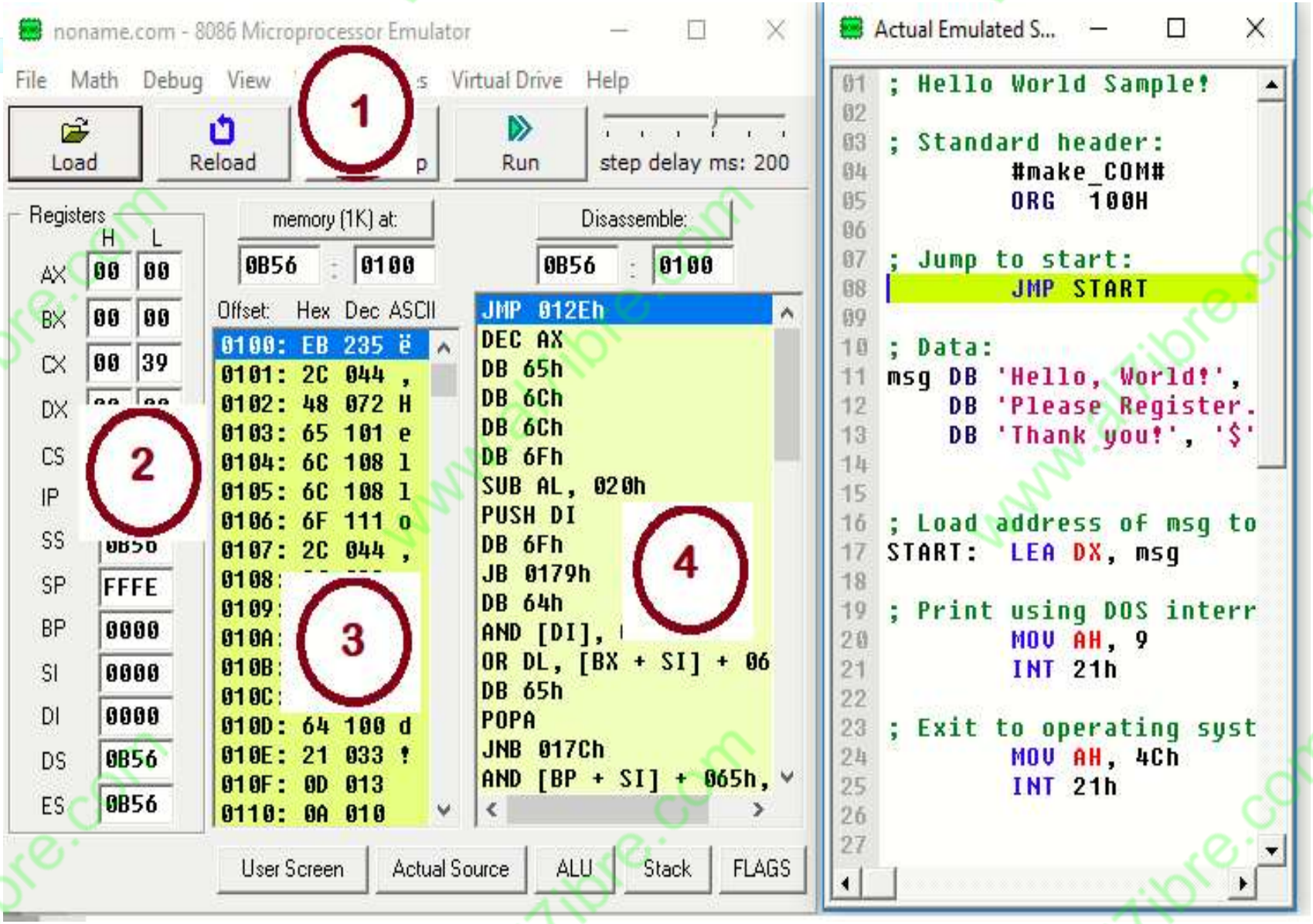

noname.com - 8086 Microprocessor Emulator
File Math Debug View Virtual Devices Virtual Drive Help
Load
Reload
Single Step
Run
step delay ms: 200
Registers
H L
AX 00 00
BX 00 00
CX 00 39
DX 00 00
CS 0B56
IP 0100
SS 0B56
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0B56
ES 0B56
memory (1K) at:
0B56 : 0100
Disassemble:
0B56 : 0100
Offset: Hex Dec ASCII
0100: EB 235 ë
0101: 2C 044 ,
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
0108: 20 032
0109: 57 087 W
010A: 6F 111 o
010B: 72 114 r
010C: 6C 108 l
010D: 64 100 d
010E: 21 033 !
010F: 0D 013
0110: 0A 010
JMP 012Eh
DEC AX
DB 65h
DB 6Ch
DB 6Ch
DB 6Fh
SUB AL, 020h
PUSH DI
DB 6Fh
JB 0179h
DB 64h
AND [DI], CX
OR DL, [BX + SI] + 06
DB 65h
POPA
JNB 017Ch
AND [BP + SI] + 065h,
User Screen
Actual Source
ALU
Stack
FLAGS
Actual Emulated S...
01 ; Hello World Sample!
02
03 ; Standard header:
04 #make_COM#
05 ORG 100H
06
07 ; Jump to start:
08 JMP START
09
10 ; Data:
11 msg DB 'Hello, World!',
12 DB 'Please Register.
13 DB 'Thank you!', '$'
14
15
16 ; Load address of msg to
17 START: LEA DX, msg
18
19 ; Print using DOS interr
20 MOV AH, 9
21 INT 21h
22
23 ; Exit to operating syst
24 MOV AH, 4Ch
25 INT 21h
26
27

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

Assembleur


noname.com - 8086 Microprocessor Emulator
File Math Debug View Virtual Drive Help
Load
Reload
Run
step delay ms: 200
Registers
H L
AX 00 00
BX 00 00
CX 00 39
DX
CS
IP
SS
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0B56
ES 0B56
memory (1K) at:
0B56 : 0100
Disassemble:
0B56 : 0100
Offset: Hex Dec ASCII
0100: EB 235 ë
0101: 2C 044 ,
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
010D: 64 100 d
010E: 21 033 !
010F: 0D 013
0110: 0A 010
JMP 012Eh
DEC AX
DB 65h
DB 6Ch
DB 6Ch
DB 6Fh
SUB AL, 020h
PUSH DI
DB 6Fh
JB 0179h
DB 64h
OR DL, [BX + SI] + 06
DB 65h
POPA
JNB 017Ch
AND [BP + SI] + 065h,
1
2
3
4
User Screen
Actual Source
ALU
Stack
FLAGS
Actual Emulated S...
; Hello World Sample!
; Standard header:
#make_COM#
ORG 100H
; Jump to start:
JMP START
; Data:
msg DB 'Hello, World!',
DB 'Please Register.
DB 'Thank you!', '$'
; Load address of msg to
START: LEA DX, msg
; Print using DOS interr
MOV AH, 9
INT 21h
; Exit to operating syst
MOV AH, 4Ch
INT 21h

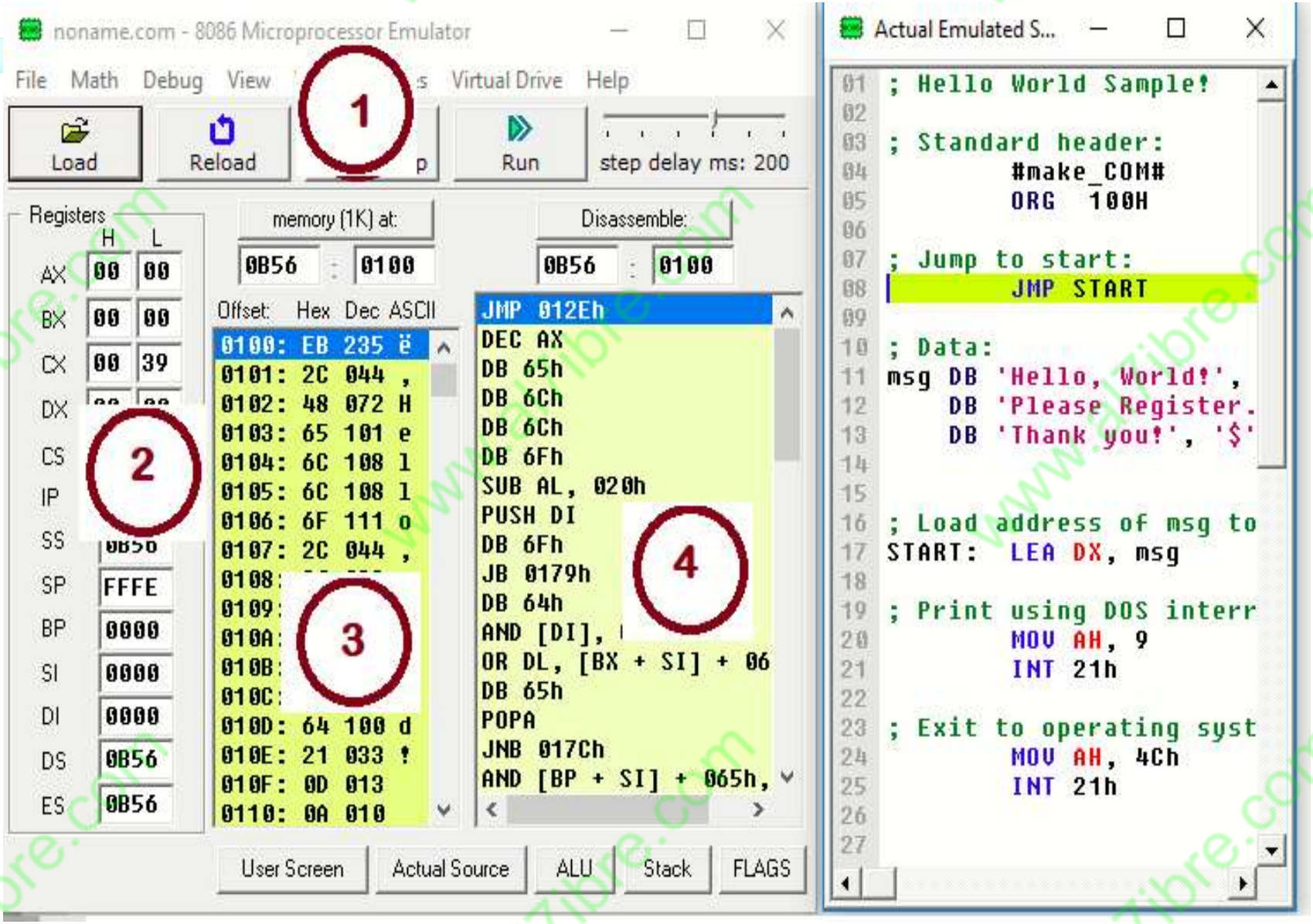
Assembleur

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**).

Assembleur


noname.com - 8086 Microprocessor Emulator
File Math Debug View Virtual Drive Help
Load
Reload
Run
step delay ms: 200
Registers
H L
AX 00 00
BX 00 00
CX 00 39
DX
CS
IP
SS
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0B56
ES 0B56
memory (1K) at:
0B56 : 0100
Disassemble:
0B56 : 0100
Offset: Hex Dec ASCII
0100: EB 235 ë
0101: 2C 044 ,
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
010D: 64 100 d
010E: 21 033 !
010F: 0D 013
0110: 0A 010
JMP 012Eh
DEC AX
DB 65h
DB 6Ch
DB 6Ch
DB 6Fh
SUB AL, 020h
PUSH DI
DB 6Fh
JB 0179h
DB 64h
OR DL, [BX + SI] + 06
DB 65h
POPA
JNB 017Ch
AND [BP + SI] + 065h,
1
2
3
4
User Screen
Actual Source
ALU
Stack
FLAGS
Actual Emulated S...
; Hello World Sample!
; Standard header:
#make_COM#
ORG 100H
; Jump to start:
JMP START
; Data:
msg DB 'Hello, World!',
DB 'Please Register.
DB 'Thank you!', '$'
; Load address of msg to
START: LEA DX, msg
; Print using DOS interr
MOV AH, 9
INT 21h
; Exit to operating syst
MOV AH, 4Ch
INT 21h

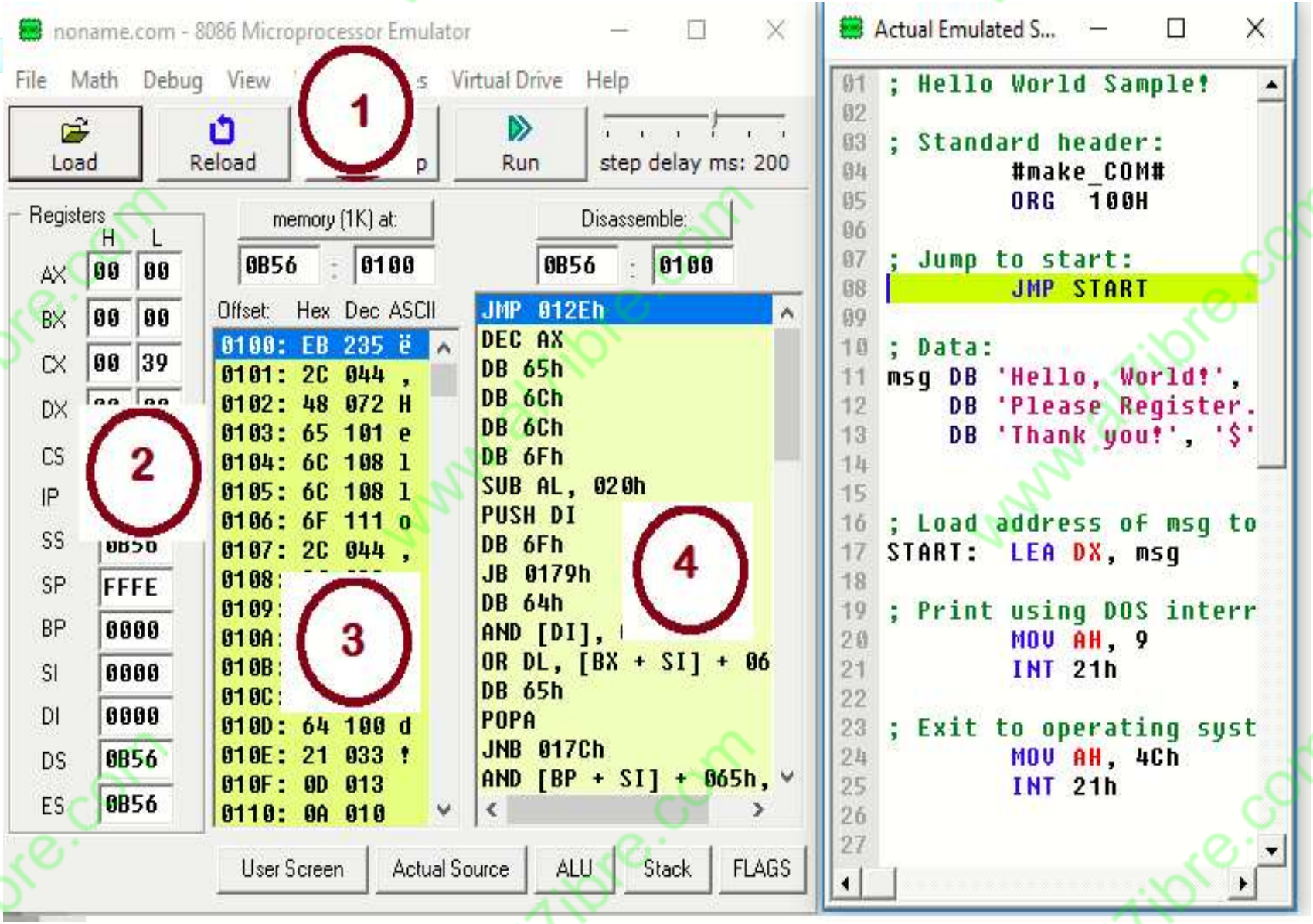
Assembleur

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**.

Assembleur


noname.com - 8086 Microprocessor Emulator
File Math Debug View Virtual Drive Help
Load
Reload
Run
step delay ms: 200
Registers
H L
AX 00 00
BX 00 00
CX 00 39
DX
CS
IP
SS
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0B56
ES 0B56
memory (1K) at:
0B56 : 0100
Disassemble:
0B56 : 0100
Offset: Hex Dec ASCII
0100: EB 235 ë
0101: 2C 044 ,
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
010D: 64 100 d
010E: 21 033 !
010F: 0D 013
0110: 0A 010
JMP 012Eh
DEC AX
DB 65h
DB 6Ch
DB 6Ch
DB 6Fh
SUB AL, 020h
PUSH DI
DB 6Fh
JB 0179h
DB 64h
OR DL, [BX + SI] + 06
DB 65h
POPA
JNB 017Ch
AND [BP + SI] + 065h,
1
2
3
4
User Screen
Actual Source
ALU
Stack
FLAGS
Actual Emulated S...
; Hello World Sample!
; Standard header:
#make_COM#
ORG 100H
; Jump to start:
JMP START
; Data:
msg DB 'Hello, World!',
DB 'Please Register.
DB 'Thank you!', '$'
; Load address of msg to
START: LEA DX, msg
; Print using DOS interr
MOV AH, 9
INT 21h
; Exit to operating syst
MOV AH, 4Ch
INT 21h

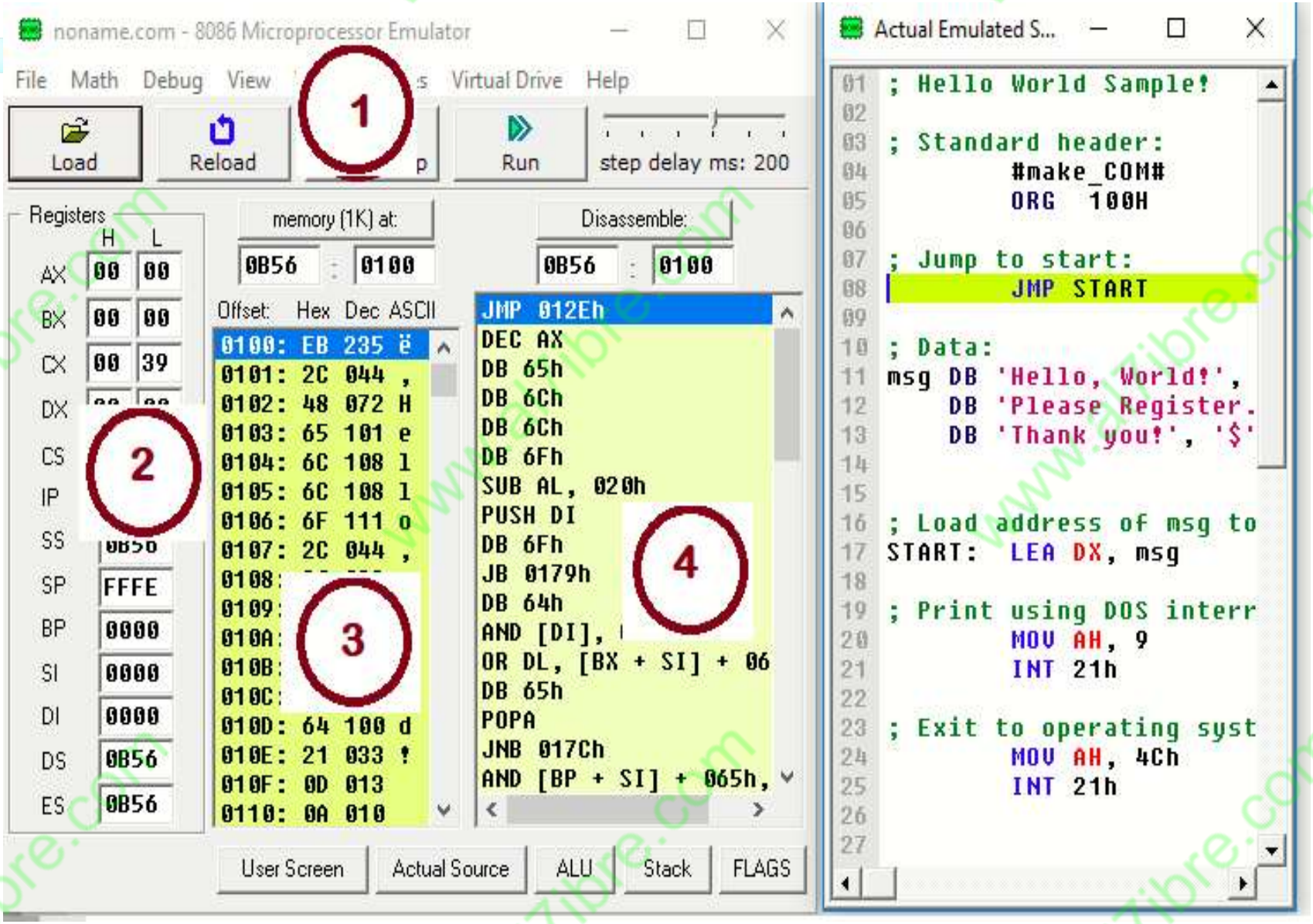
Assembleur

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**.

Assembleur


noname.com - 8086 Microprocessor Emulator
File Math Debug View Virtual Drive Help
Load
Reload
Run
step delay ms: 200
Registers
H L
AX 00 00
BX 00 00
CX 00 39
DX
CS
IP
SS
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0B56
ES 0B56
memory (1K) at:
0B56 : 0100
Disassemble:
0B56 : 0100
Offset: Hex Dec ASCII
0100: EB 235 ë
0101: 2C 044 ,
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
010D: 64 100 d
010E: 21 033 !
010F: 0D 013
0110: 0A 010
JMP 012Eh
DEC AX
DB 65h
DB 6Ch
DB 6Ch
DB 6Fh
SUB AL, 020h
PUSH DI
DB 6Fh
JB 0179h
DB 64h
OR DL, [BX + SI] + 06
DB 65h
POPA
JNB 017Ch
AND [BP + SI] + 065h,
1
2
3
4
User Screen
Actual Source
ALU
Stack
FLAGS
Actual Emulated S...
; Hello World Sample!
; Standard header:
#make_COM#
ORG 100H
; Jump to start:
JMP START
; Data:
msg DB 'Hello, World!',
DB 'Please Register.
DB 'Thank you!', '$'
; Load address of msg to
START: LEA DX, msg
; Print using DOS interr
MOV AH, 9
INT 21h
; Exit to operating syst
MOV AH, 4Ch
INT 21h

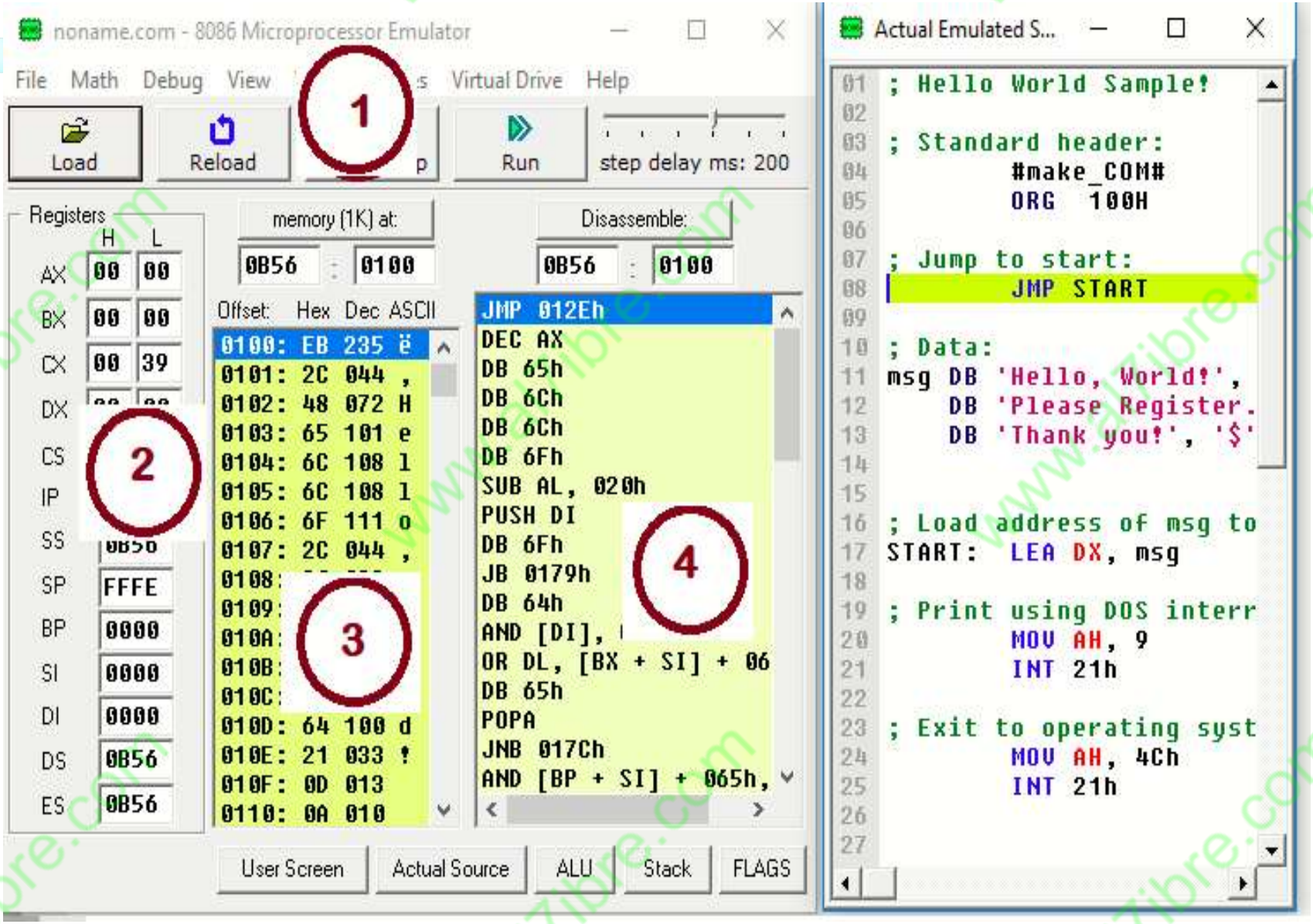
Assembleur

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**.

Assembleur


noname.com - 8086 Microprocessor Emulator
File Math Debug View Virtual Drive Help
Load
Reload
Run
step delay ms: 200
Registers
H L
AX 00 00
BX 00 00
CX 00 39
DX
CS
IP
SS
SP FFFE
BP 0000
SI 0000
DI 0000
DS 0B56
ES 0B56
memory (1K) at:
0B56 : 0100
Disassemble:
0B56 : 0100
Offset: Hex Dec ASCII
0100: EB 235 ë
0101: 2C 044 ,
0102: 48 072 H
0103: 65 101 e
0104: 6C 108 l
0105: 6C 108 l
0106: 6F 111 o
0107: 2C 044 ,
010D: 64 100 d
010E: 21 033 !
010F: 0D 013
0110: 0A 010
JMP 012Eh
DEC AX
DB 65h
DB 6Ch
DB 6Ch
DB 6Fh
SUB AL, 020h
PUSH DI
DB 6Fh
JB 0179h
DB 64h
OR DL, [BX + SI] + 06
DB 65h
POPA
JNB 017Ch
AND [BP + SI] + 065h,
1
2
3
4
User Screen
Actual Source
ALU
Stack
FLAGS
Actual Emulated S...
; Hello World Sample!
; Standard header:
#make_COM#
ORG 100H
; Jump to start:
JMP START
; Data:
msg DB 'Hello, World!',
DB 'Please Register.
DB 'Thank you!', '$'
; Load address of msg to
START: LEA DX, msg
; Print using DOS interr
MOV AH, 9
INT 21h
; Exit to operating syst
MOV AH, 4Ch
INT 21h

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**

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

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**

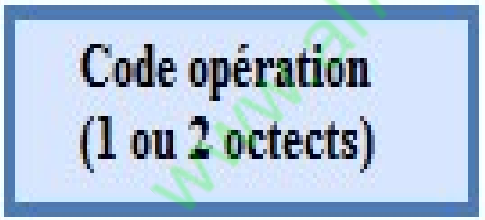
Jeu d'instructions

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

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 243

ADD AX, 0B24h : additionner le registre AX avec le nombre hexadécimal 243

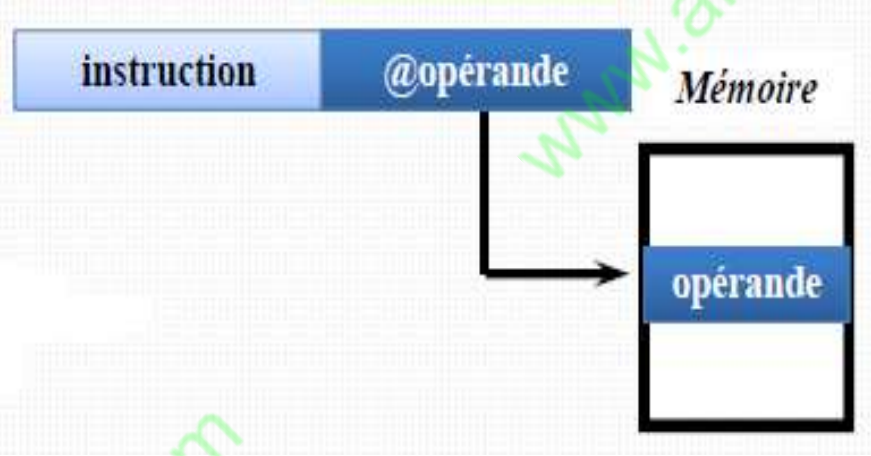
Jeu d'instructions

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 :

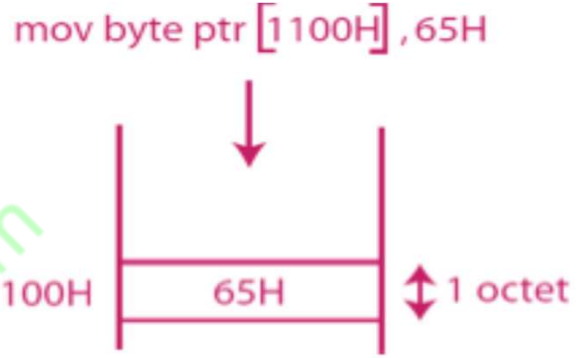
MOV AX, [2413h] : charger

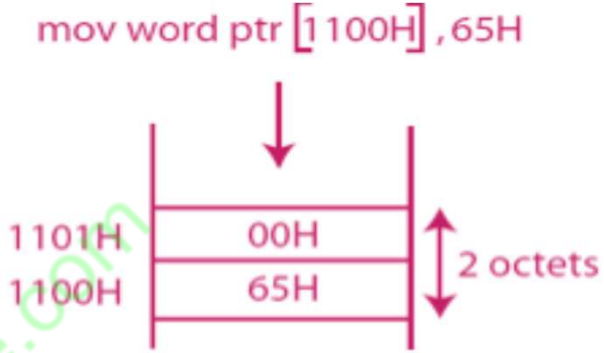
MOV [2413h] , 0B24h : Copie le nombre 0B24h dans l'@:2413

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.





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]

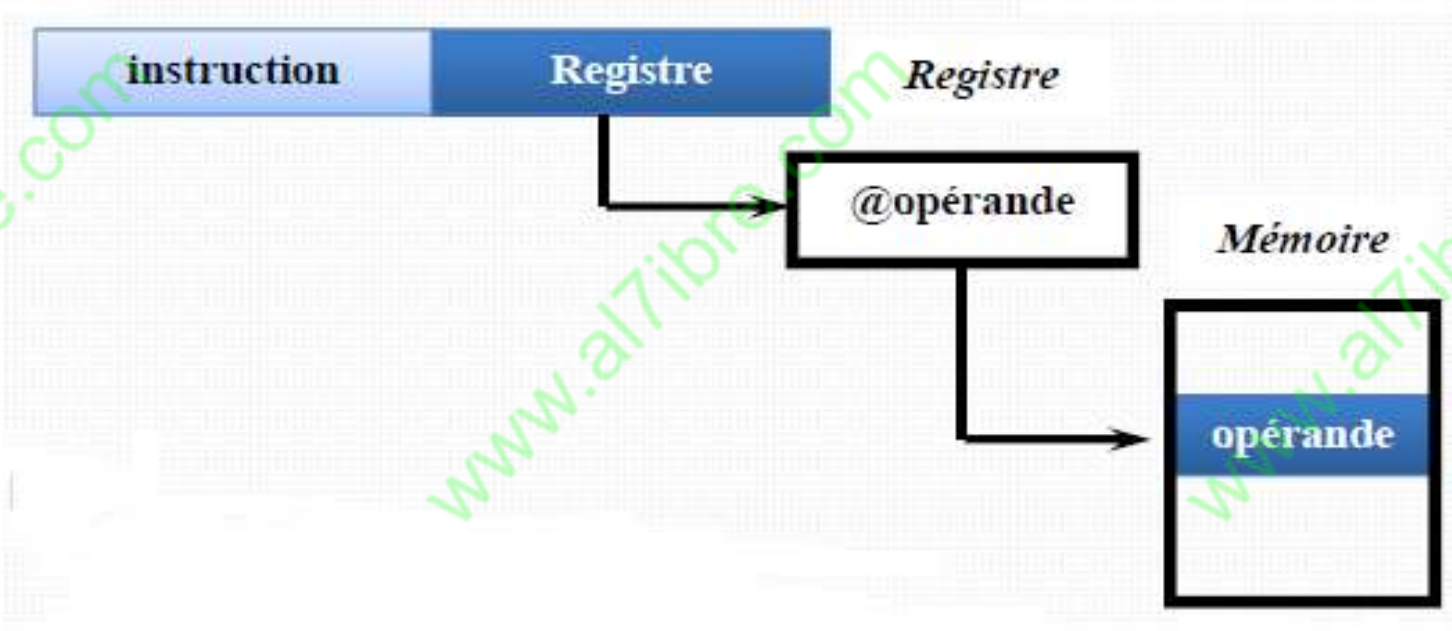
  MNQ [Rseg : Roff] , R

  MNQ table[Rseg : Roff],R

- Si **Rseg n'est pas spécifié**, le segment **par défaut** sera utilisé.

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**.

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

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

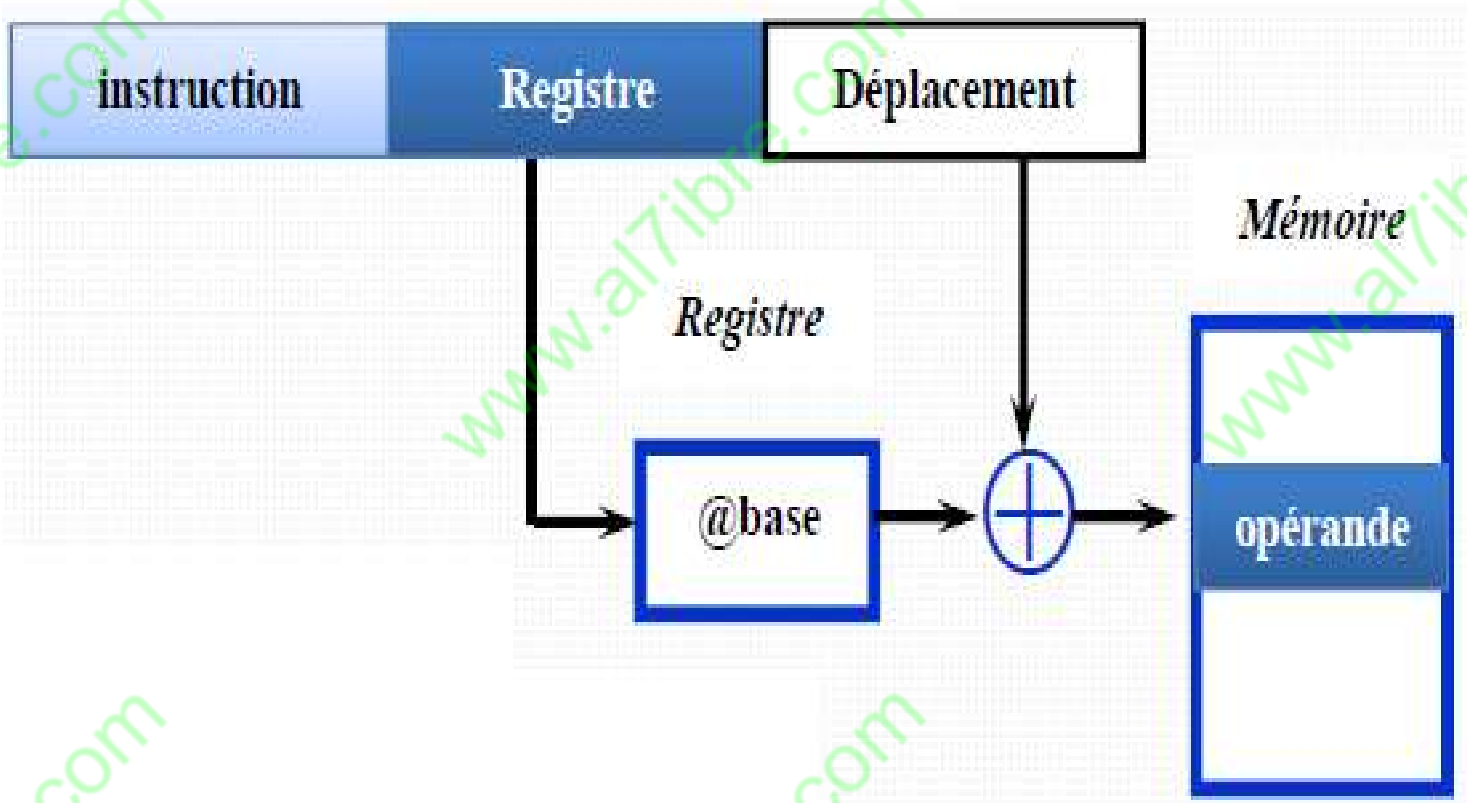
  MNQ Table [Rseg : Rb+dep]

- **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**;

Jeu d'instructions

Adressage basé :



Jeu d'instructions

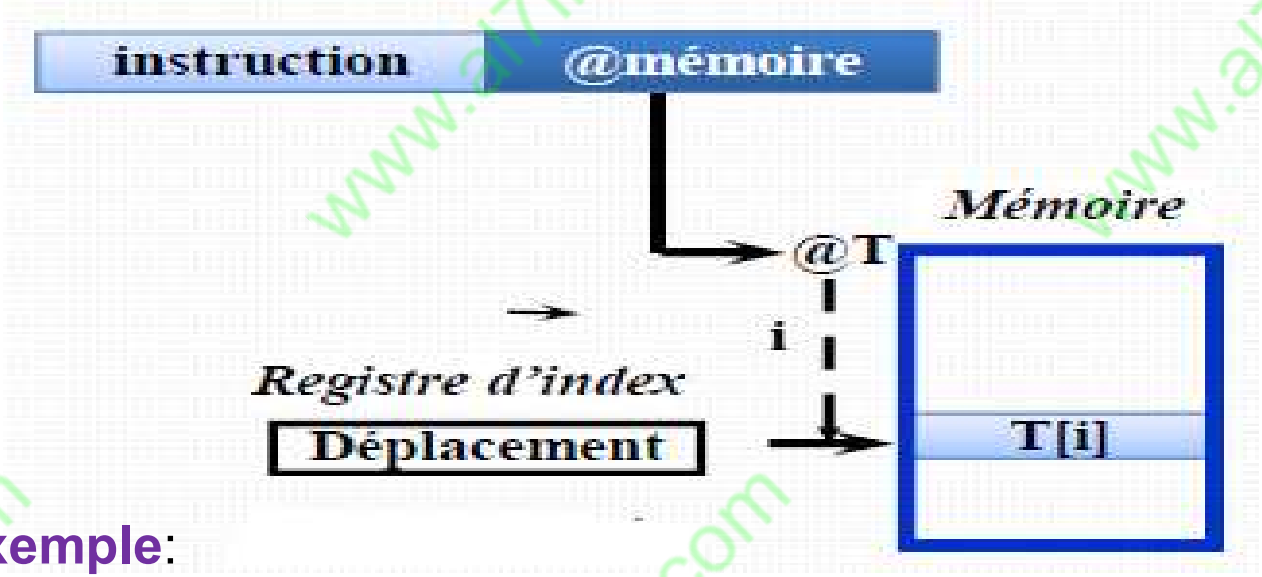
Exemples d'adressage basé :

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

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]**

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

Jeu d'instructions

Type d'instruction

- Les **instructions machines** permettent d'effectuer des **opérations élémentaires** ou plus **complexes**, et peuvent être classées **4 groupes** :
  - **Instructions de transfert de données;**
  - **Instructions arithmétiques;**
  - **Instructions logiques;**
  - **Instructions de branchement.**

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 »**

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**

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**

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**

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

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**.

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]**

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**

Jeu d'instructions

Les instructions arithmétiques

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

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

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

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).

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 AX , 39h puis **sub AX , 18h** ➔ AX= 21h, Z=S=C=0

mov AX , 26h puis **sub AX , 59h** ➔ AX= cdh, Z=0, C=S=A=1.

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

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**

Jeu d'instructions

Multiplication : MUL/IMUL op8/op16

- **Exemple:**
  - **mov al, 4** puis **mov ah, 25** puis **imul ah** : AH=2*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.

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

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**.

Jeu d'instructions

Division : DIV/IDIV op8/op16

- **Exemple:**
  - **mov AX , 37** puis **mov dx, 5** puis **div dx :** AX = 7, DX=2.
  - **mov AX , 37** puis **mov dx, 5** puis **div dl :** AH = 7, AL=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= FFFbh , DX=197. Si l'on remplace IDIV par DIV, le résultat n'a pas de sens.

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 [ ]**

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.

Jeu d'instructions

Autres instructions arithmétiques :

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

  OpDst = OpDst + OpSrc + C

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

  OpDst = OpDst – (OpSrc + C)

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

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

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

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.

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
```

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

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**.

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
```

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 regisre, registre/variable

XOR registre/variable, constante

Jeu d'instructions

Le OU exclusif : XOR op1,op2

- **Exemple :**

mov al, 36h
and al, 5ch

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

$$\begin{array}{rll} & 36h & 0011\ 0110 \\ \oplus & \underline{5Ch} & \underline{0101\ 1100} \\ & 6Ah & 0110\ 1010 \end{array}$$

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

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**.

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

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.

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 2k** et un **décalage de k positions vers la droite** correspond à une **division par 2k** .

Jeu d'instructions

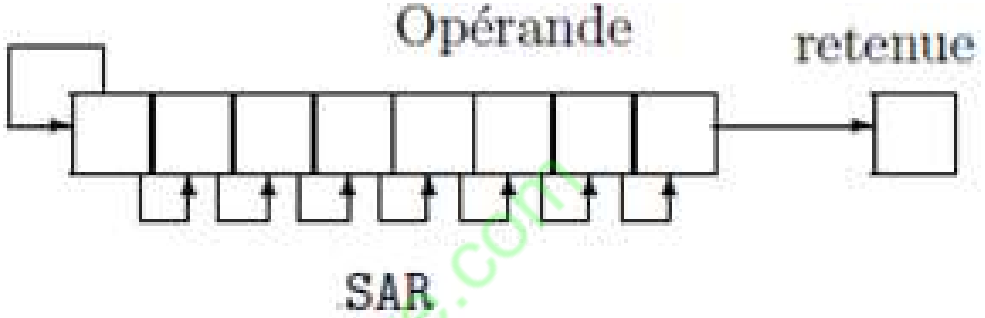
Décalage arithmétique vers la droite :

L'instruction SAR :

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.



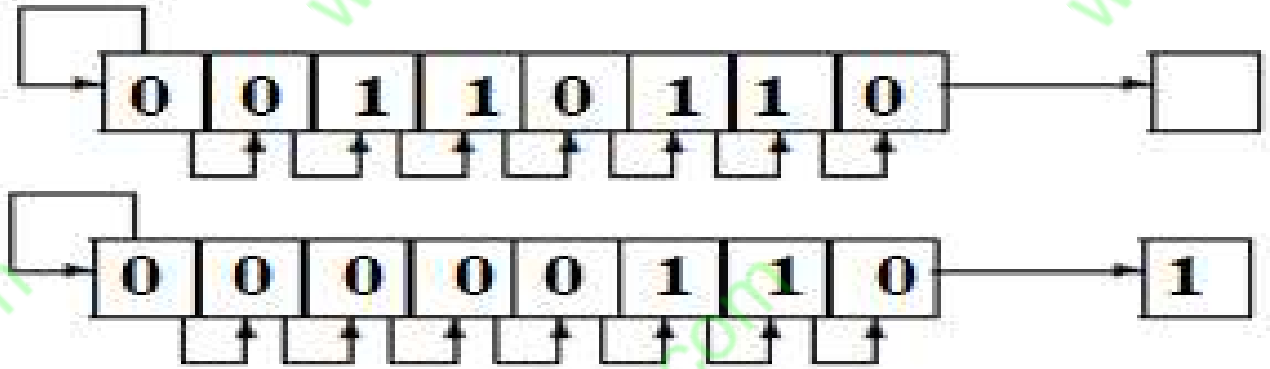
Jeu d'instructions

- **Décalage arithmétique vers la droite :**

Exemple SAR :

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

- Le registre AL contient ensuite la valeur 05h. Le bit (C) est mis à 1.



Jeu d'instructions

Décalage logique vers la droit :

L'instruction SHR :

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

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

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.

Jeu d'instructions

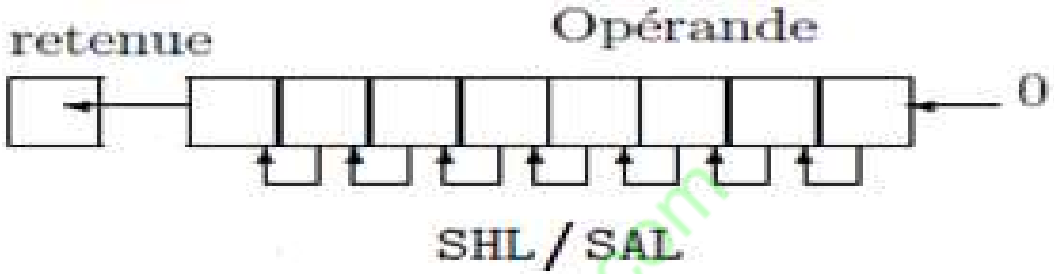
Décalage arithmétique ou logique vers la gauche:

Les instructions SAL et SHL :

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



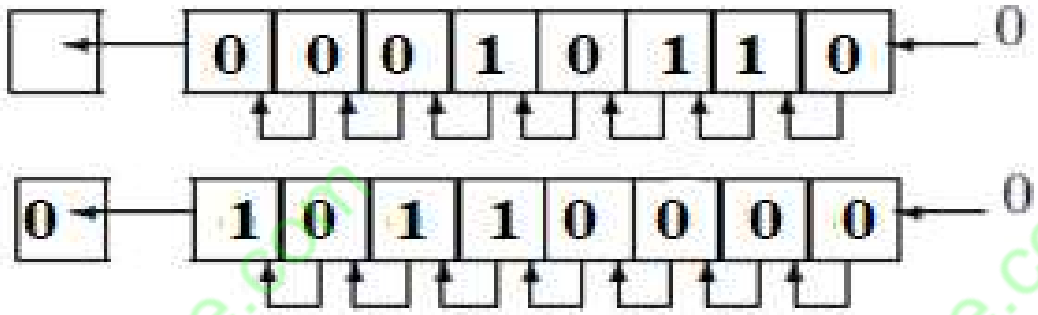
Jeu d'instructions

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.



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.

Jeu d'instructions

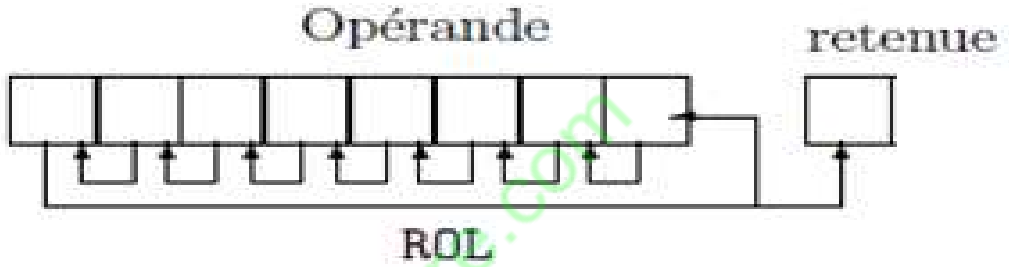
Rotation à gauche

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**.



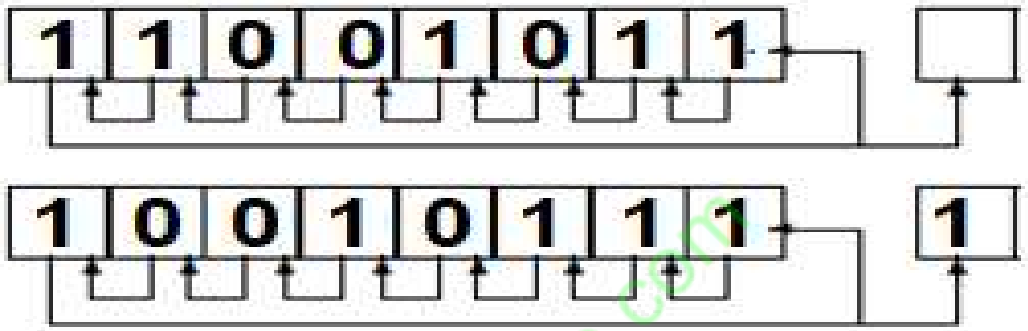
Jeu d'instructions

Rotation à gauche

Exemple de l'instruction ROL (ROtate Left) :

```
mov al,11001011B
rol al,1
```

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



Jeu d'instructions

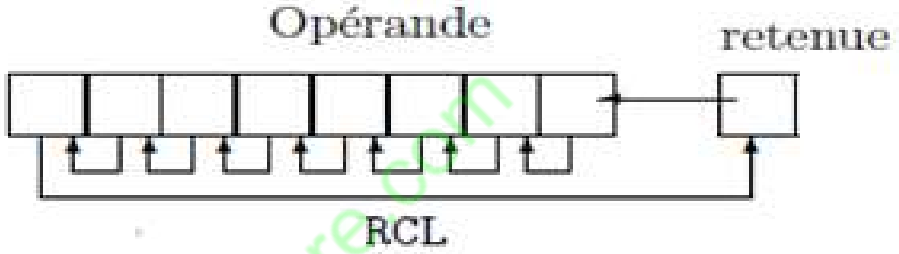
Rotation à gauche avec retenue

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**.



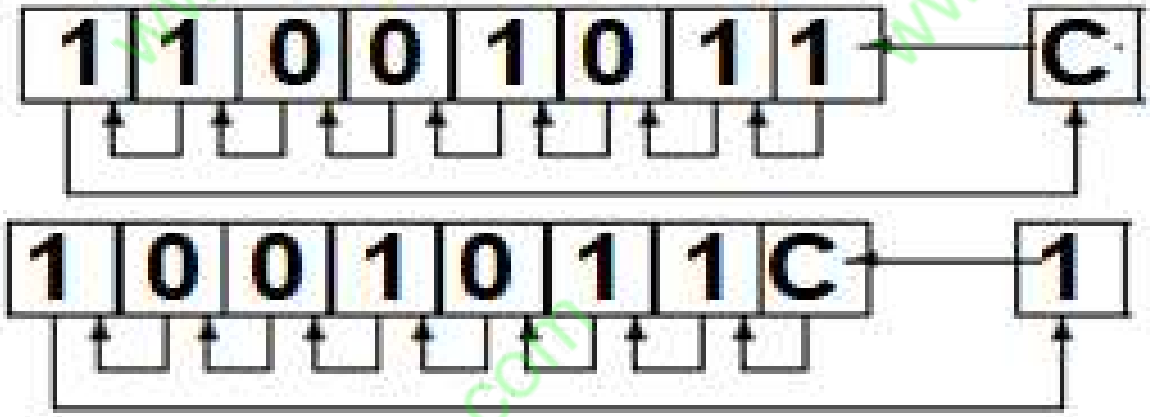
Jeu d'instructions

Rotation à gauche avec retenue

L'instruction RCL (Rotate Left through Carry) :

```
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**.



Jeu d'instructions

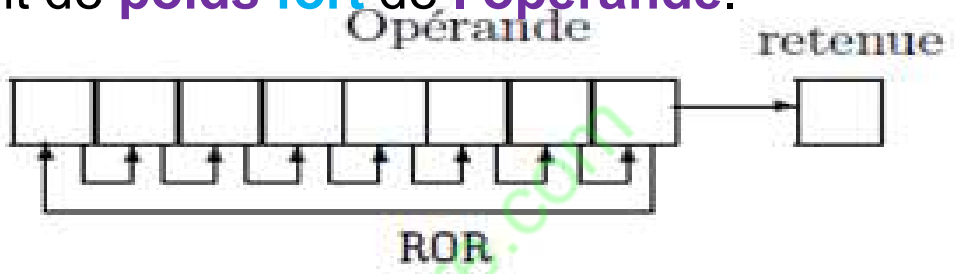
Rotation à droite

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**.



Jeu d'instructions

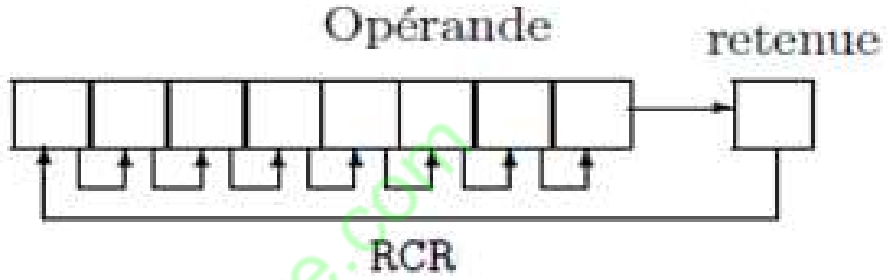
Rotation à droite avec retenue

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**.



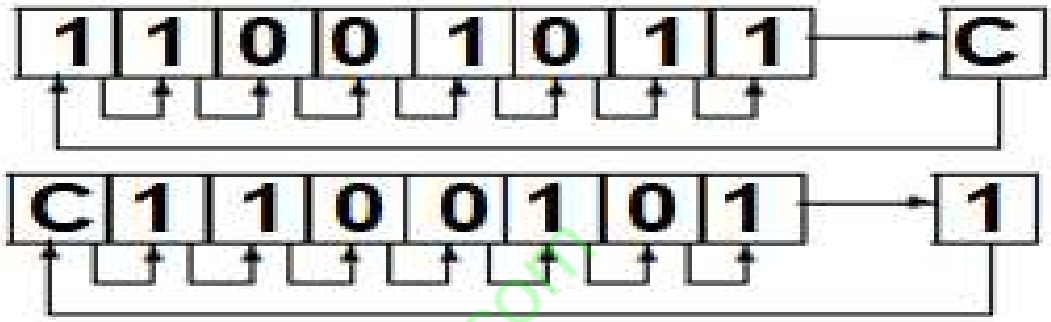
Jeu d'instructions

Rotation a droite avec retenue

Exemple de l'instruction RCR (Rotate Right through Carry) :

```
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***



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***

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

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.

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

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***).

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**.

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

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**.

Jeu d'instructions

Exemple instruction de sauts conditionnels : :

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

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

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

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.

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.

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

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 **si la condition d'arrêt n'est pas vérifiée**
- Une étiquette **FTQ** indiquant la **fin de la boucle**

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

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**

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'**

Entrée/Sortie

Interruption

- **BIOS ≈ librairie de fonctions** : Chaque fonction effectue une tache bien précise.
- **Code du BIOS** en ROM est **non modifiable** solution 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)**
- **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**

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
```

Entrée/Sortie

Fonction d'écriture à l'écran

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

```
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

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 ”2”** : **Saisie sans écho**

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

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

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**

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 information 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**