MICRO INFORMATIQUE

Olivier HABERT

Notes de Cours

Université de METZ

UFR SciFa

Structuration du Cours

I. *Historique de l'informatique*

II. *Généralités sur l'informatique de nos jours*

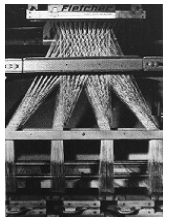
III. *Représentation de l'information*

IV. *Les composants de base de l'électronique numérique*

V. *Le Microprocesseur, concepts généraux*

VI. *Etude de cas: le 8086*

I. HISTORIQUE



Machine de JACQUARD
(Machine de FALCON
avec des cartes perforées)

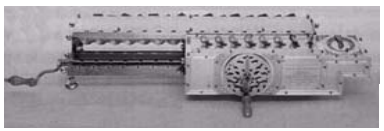
1822

Machine analytique
de BABBAGE

1801

1728

Machine de FALCON
Métier à tisser fonctionnant
à base de planchettes perforées



Machine de LEIBNIZ

1694

1641

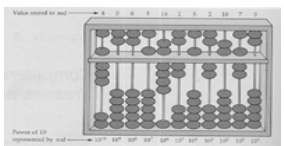
Machine de PASCAL

Machine mécanique
de SCHICKARD
(aucun spécimen)

1623

1615

Théorie des logarithmes
de NEPER



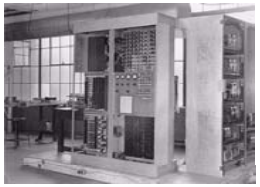
Bouliers et Abaques

10000

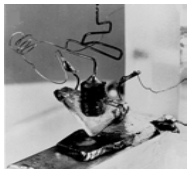
I. HISTORIQUE

1960 — Premier ordinateur 100% transistor

1957 — EDVAC, premier calculateur à programme enregistré



1947 — Premiers résultats en laboratoire sur le transistor

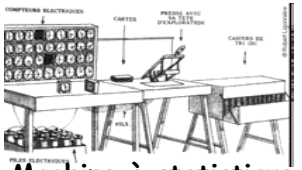


1946 — ENIAC calculateur 100% électronique de 18000 tubes un produit dure 3 millisecondes

1940/44 — Système Colussus pour le décryptage des messages codés

1935 — Calculateur Z1 binaire électromécanique (2600 relais) mutiplication dure 5 secondes

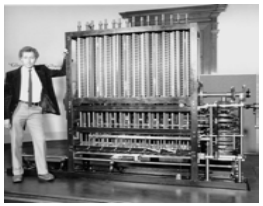
1930 — SHANNON prouve que toute opération logique et arithmétique peut s'effectuer avec des opérateurs OU,ET et NON



1900 — Machine à statistique de HOLLERITH pour le recensement des USA

1890 — Apparition des premiers calculateurs analogiques

1854 — Système logique de BOOLE Base logique de tous les ordinateurs actuels



1847 — Seconde version de la machine à différence de BABBAGE

II. GÉNÉRALITES

CLAVIER

JOYSTICK VOLANT

SOURIS

ECRAN TUBE CATHODIQUE

ECRAN LCD

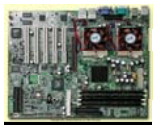
VIDEO PROJECTEUR

SCANNER

APPAREIL PHOTO NUMERIQUE

CARTES INDUSTRIELLES D'ACQUISITION

CARTE MERE



MEMOIRE

PROCESSEUR

CARTE SON

CARTE VIDEO

CARTE RESEAU

BOITIER

MODEM

ENCEINTES

CAMERA (WEB CAM)

LECTEUR/GRAVEUR DVD/CD

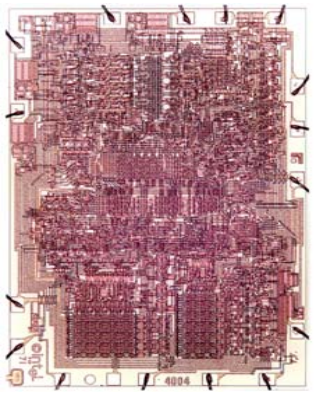
LECTEUR DE DISQUETTE

LECTEUR/ENREGISTREUR HAUTE CAPACITE

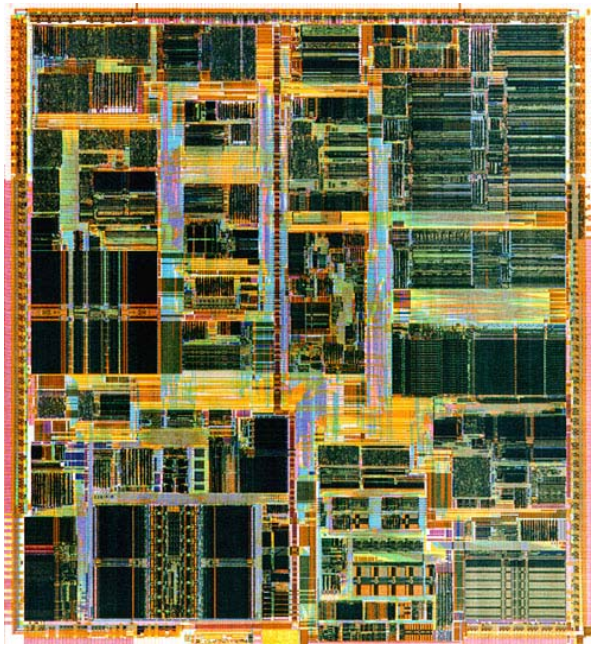
DISQUE DUR

IMPRIMANTES

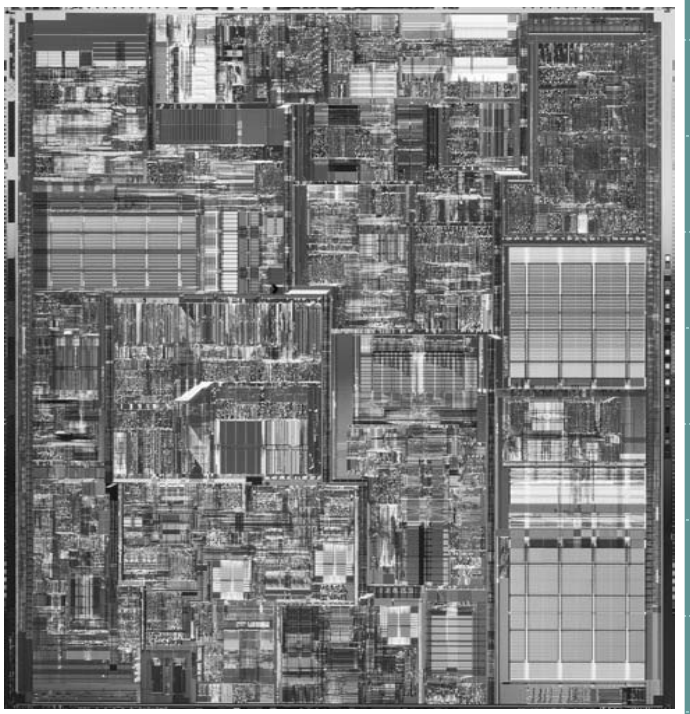
II. GÉNÉRALITES



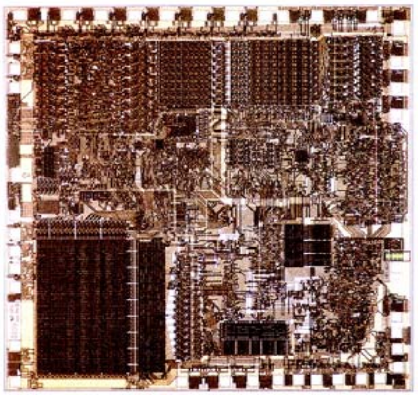
4004



Pentium II



Pentium IV



8088

Évolution des microprocesseurs INTEL de 1972 à nos jours

III. REPRÉSENTATION de L'INFORMATION

- **Deux catégories d'information**
  - **Le programme →langage machine**:
    - Code de l'instruction + opérande(s)
  - **Les données**
    - Données non numériques →caractères ASCII
    - Données numériques
      - > *Nombres entiers*
      - > *Nombres décimaux*
      - > *Nombres en notation scientifique*
      - > *…*

III. REPRÉSENTATION de L'INFORMATION

Représentation des nombres

Rappel de numération

- Dans une base B>1, les nombres 0,1,2,…,B-1 sont appelés chiffres
- Tout nombre entier >0 peut se représenter sous la forme:

$$N = a_n B^n + a_{n-1} B^{n-1} + ... + a_1 B^1 + a_0 B^0 = \sum_{i=0}^{n} a_i B^i \text{ avec } a_i \in \{0,1,...,B-1\} \text{ et } a_n \neq 0$$

> On peut écrire de manière condensée: $N = a_n a_{n-1} ... a_1 a_0$

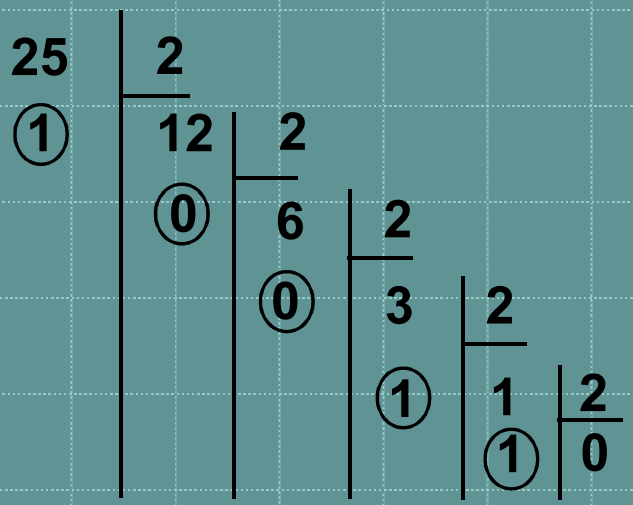
- Exemples:

$68_{(10)} = 6*10^1+8*10^0$

$1000100(2) = 1*2^6+0*2^5+0*2^4+0*2^3+1*2^2+0*2^1+0*2^0$

III. REPRÉSENTATION de L'INFORMATION

- Changement de Base
  - Binaire → Décimal *(voir exemple précédent)*
  - Décimale → Binaire
    - Division successive par 2
      - > Le nombre binaire correspond au reste en partant du dernier vers le premier



$25_{(10)} = 11001_{(2)}$

III. REPRÉSENTATION de L'INFORMATION

- **Changement de Base**
  - **Binaire → Hexadécimal**
    - Regroupement 4 par 4 de droite à gauche des bits et remplacement de chaque quartet par le chiffre hexadécimal correspondant

      > $\underline{1000}\ \underline{1111}\ \underline{0111}\ \underline{0101}_{(2)} = 8F75_{(16)}$

  - **Hexadécimal → Binaire**
    - **Transformation de chaque chiffre en son équivalent binaire sur 4 bits**

      > $4FCA_{(16)} \rightarrow 0100\ 1111\ 1100\ 1010_{(2)}$

III. REPRÉSENTATION de L'INFORMATION

Représentation des nombres entiers positifs

Représentés par leur équivalent binaire

- En base B, k chiffres permettent de représenter un nombre N tel que : $0 \leq N \leq B^k - 1$

Représentation des nombres entiers négatifs

Trois méthodes

- Signe + valeur absolue
  - > Le bit le plus à gauche représente le signe (0→nombre positif, 1→nombre négatif)
  - > Avec un nombre de k bits, on peut coder un entier positif ou négatif N tel que: $-\left(2^{k-1}-1\right) \leq N \leq +\left(2^{k-1}-1\right)$
  - > **Inconvénient**: deux codes pour représenter le 0 (00000000 et 10000000)

III. REPRÉSENTATION de L'INFORMATION

♦ Représentation des nombres entiers négatifs

- ♦ Complément logique *(complément à 1)*
  - > Pour représenter un nombre négatif, à partir du codage du nombre positif on remplace chaque bit à 0 par un 1 et vice-versa
    - » **Exemple**: $-6_{(10)} = 001_{(2)}$
    - » **Inconvénient**: il existe également deux codes pour le 0 (00000000 et 11111111)
- ♦ Complément arithmétique *(complément à 2)*
  - > C'est le complément logique +1
  - > **Exemple**: -68 sur 8 bits
    - » Signe+valeur absolue: 11000100
    - » Complément à 1: 10111011
    - » Complément à 2: 10111100
  - > Un seul code pour le 0, On peut coder $-\left(2^{k-1}\right) \leq N \leq +\left(2^{k-1}-1\right)$

III. REPRÉSENTATION de L'INFORMATION

♦ Représentation des nombres entiers négatifs

♦ Opérations arithmétiques avec les 3 codages

> Effectuons l'opération +7-6 avec les trois méthodes énoncés précédemment

Décimal	Signe+||	compl 1	Compl 2
+7	0111	0111	0111
+ -6	1110	1001	1010
1	*0*1101	*1*0000	*1*0001

- Pour le codage Signe + || → pas de correspondance évidente entre le résultat trouvé et le résultat tel qu'il devrait apparaître
- Pour le complément à 1, si on ajoute la retenue, on retrouve le résultat
- Pour le complément à 2, si on enlève la retenue, on trouve immédiatement le résultat

III. REPRÉSENTATION de L'INFORMATION

Représentation des nombres décimaux

Nombres qui comportent une partie décimale inférieure à 1

- Conversion décimale →binaire
  - > Addition des puissances de 2
  - > **Exemple:** $0.01_{(2)} = 0*2^{-1}+1*2^{-2}+1*2^{-2}$
- Conversion décimale→binaire
  - > Multiplication successive par 2 de la partie fractionnaire. On arrête lorsque on obtient une partie fractionnaire nulle ou lorsque le nombre de bits obtenus est suffisant.
    Le nombre binaire s'obtient en lisant les parties entières de la première division vers la dernière
  - > **Exemple**: 0.125*2 = **0**.250 0.250*2 = **0**.5 0.5*2 = **1**
  - > →$0.125_{(10)} = 0.001_{(2)}$

III. REPRÉSENTATION de L'INFORMATION

Représentation des nombres décimaux

Deux principes de mémorisation

- Mémorisation par virgule fixe
  - > On traite les nombres comme des entiers et on gère la virgule à part
- Mémorisation par virgule flottante
  - > Représentation des nombres sous la forme $N=M*B^E$
    - » Avec **M** la mantisse, **B** la base et **E** l'exposant
  - > Règles de normalisation:
    - » La mantisse est purement fractionnaire
    - » Elle comporte un maximum de chiffres significatifs
    - » Le signe de la mantisse se code sous la forme signe + ||
    - » L'exposant est représenté sous forme biaisé

III. REPRÉSENTATION de L'INFORMATION

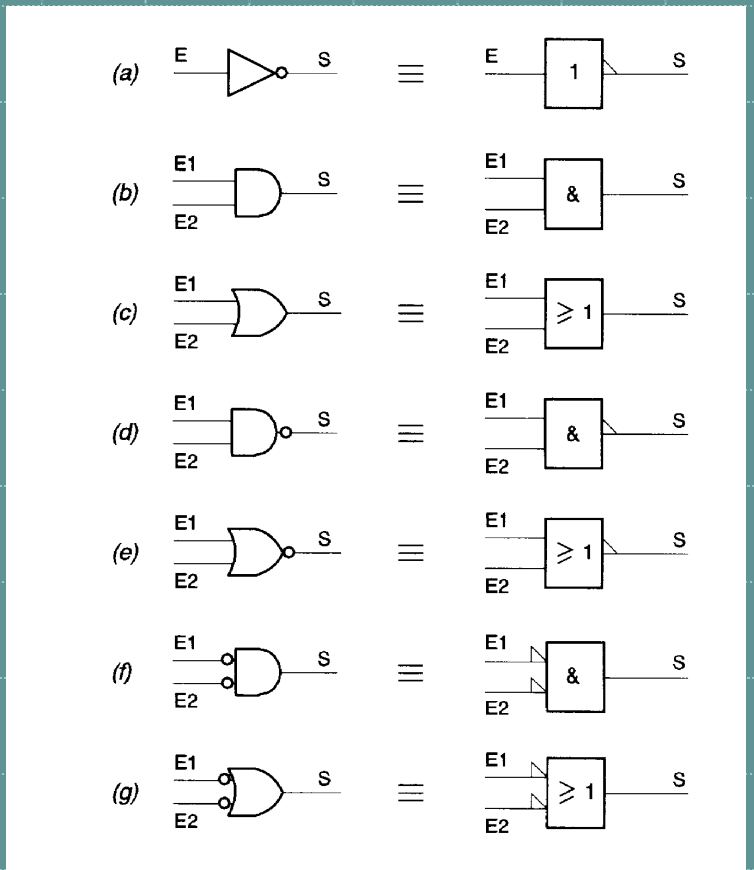
Représentation des nombres décimaux

Exemples d'opérations arithmétiques

- Multiplication de deux nombres décimaux
  - > **$0.2\ 10^{-3} * 0.3\ 10^{7}$**
    1) Addition des composants: -3+7=4
    2) Produite des mantisses 0.2*0.3 =0.06
    3) Résultat: 0.06 10^{4}
    4) Normalisation **$0.6\ 10^{3}$**
- Addition de deux nombres décimaux
  - > **$0.3\ 10^{4} + 0.998\ 10^{6}$**
    1) Dénormalisation du plus petit nombre: 0.3 10^{4} →0.003 10^{6}
    2) Addition des mantisses 0.3+0.998 = 1.001
    3) Normalisation du résultat 1.001 10^{6} →**$0.1001\ 10^{7}$**

IV. LES COMPOSANTS DE BASE DE L' ÉLECTRONIQUE NUMÉRIQUE

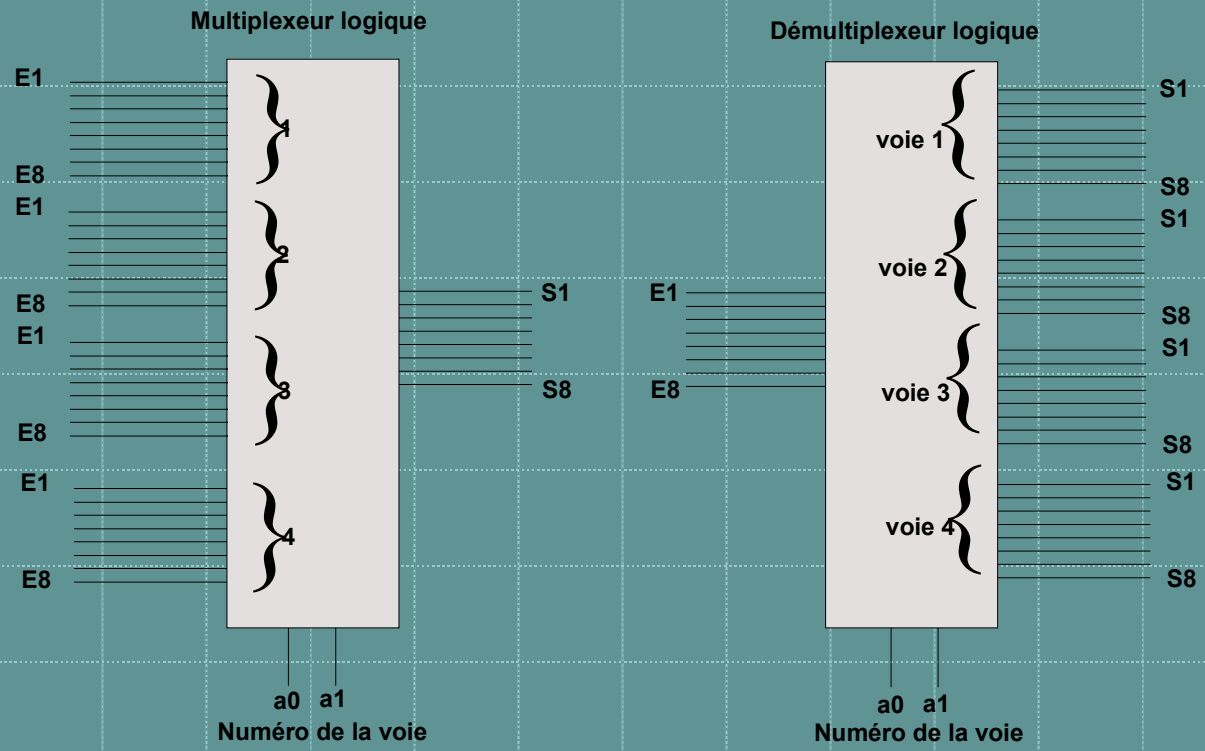
♦Circuits logiques de base



IV. LES COMPOSANTS DE BASE DE L' ÉLECTRONIQUE NUMÉRIQUE

♦Circuits combinatoires de base

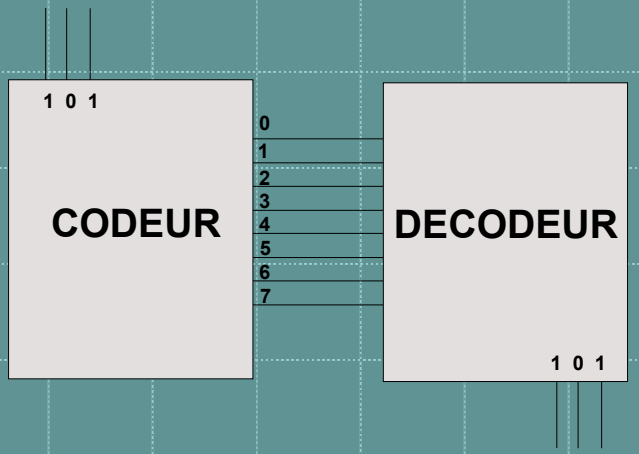
♦ Multiplexeur-Démultiplexeur



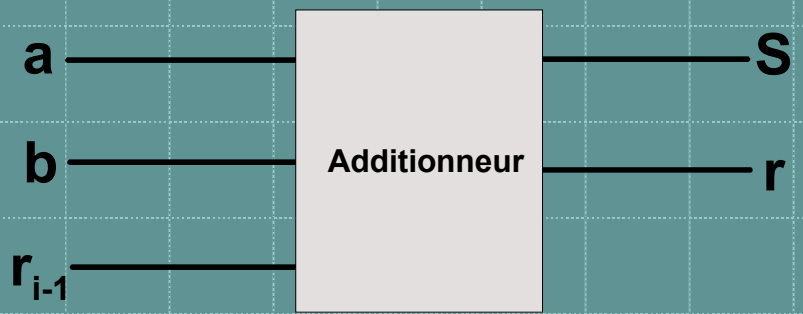
IV. LES COMPOSANTS DE BASE DE L' ÉLECTRONIQUE NUMÉRIQUE

♦Circuits combinatoires de base

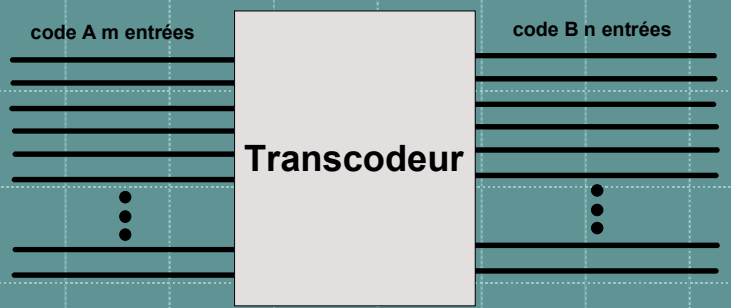
♦ Codeur-Décodeur



♦Additionneur



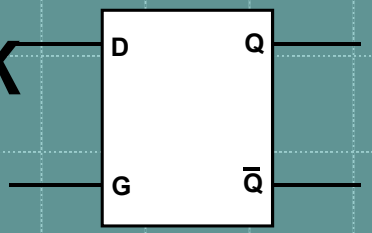
♦Transcodeur



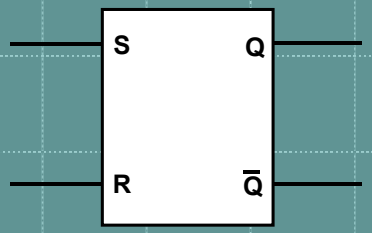
IV. LES COMPOSANTS DE BASE DE L' ÉLECTRONIQUE NUMÉRIQUE

Circuits Séquentiels de base

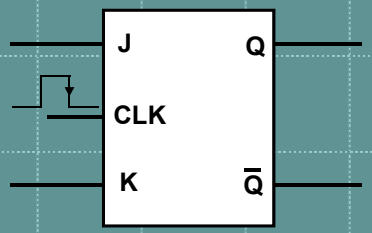
Bascule D, RS, JK



D	G	$Q_n(t)$	$\overline{Q_n(t)}$
0	1	0	1
1	1	1	0
*	0	$Q_n(t)$	$\overline{Q_n(t)}$

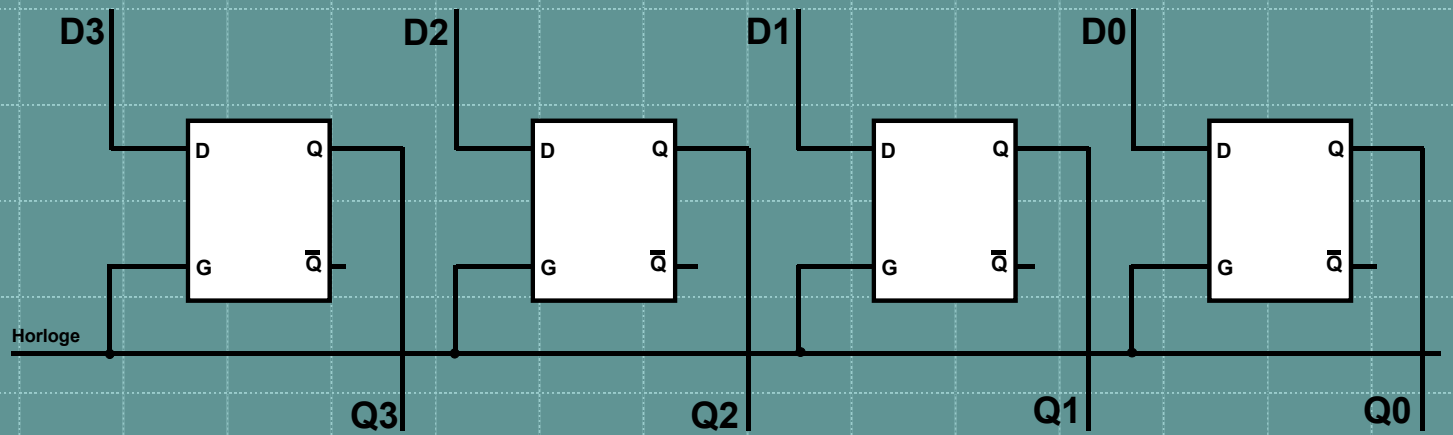


S	R	$Q_n(t)$	$\overline{Q_n(t)}$
0	1	0	1
1	0	1	0
0	0	$Q_{n-1}(t)$	$\overline{Q_{n-1}(t)}$
1	1	Indéterminé	Indéterminé



J	K	$Q_n(t)$
0	0	$Q_{n-1}(t)$
0	1	0
1	0	1
1	1	$\overline{Q_{n-1}(t)}$

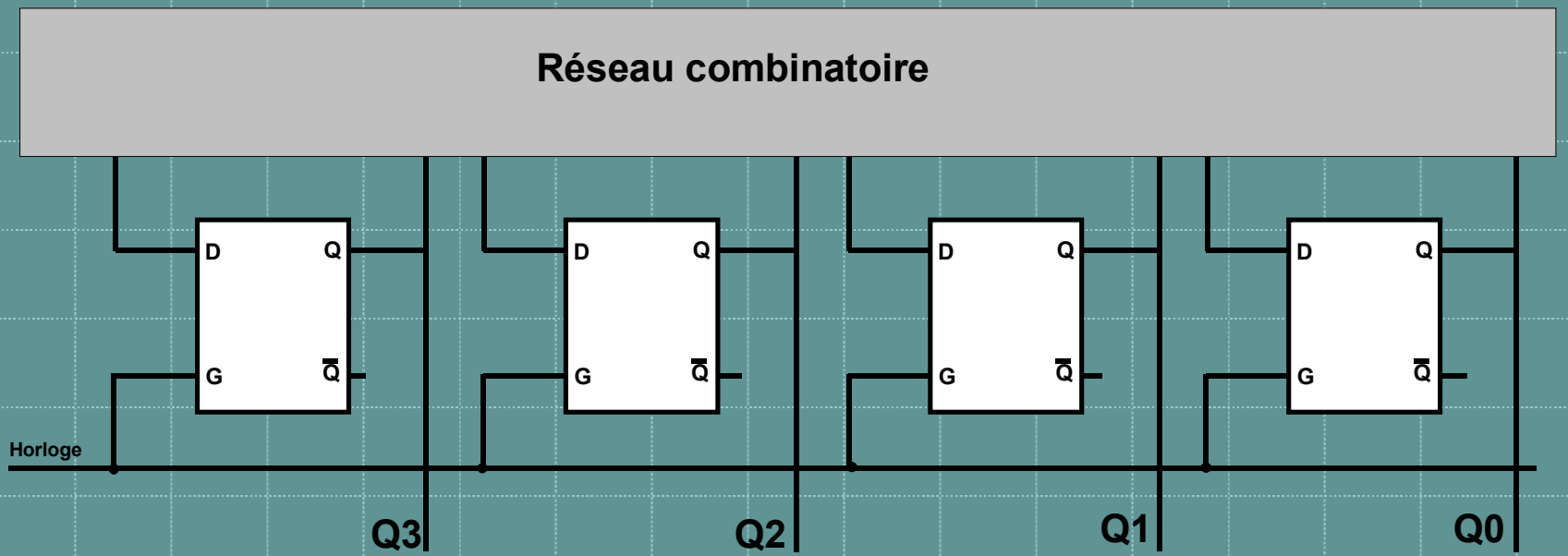
Registre 4 bits



IV. LES COMPOSANTS DE BASE DE L' ÉLECTRONIQUE NUMÉRIQUE

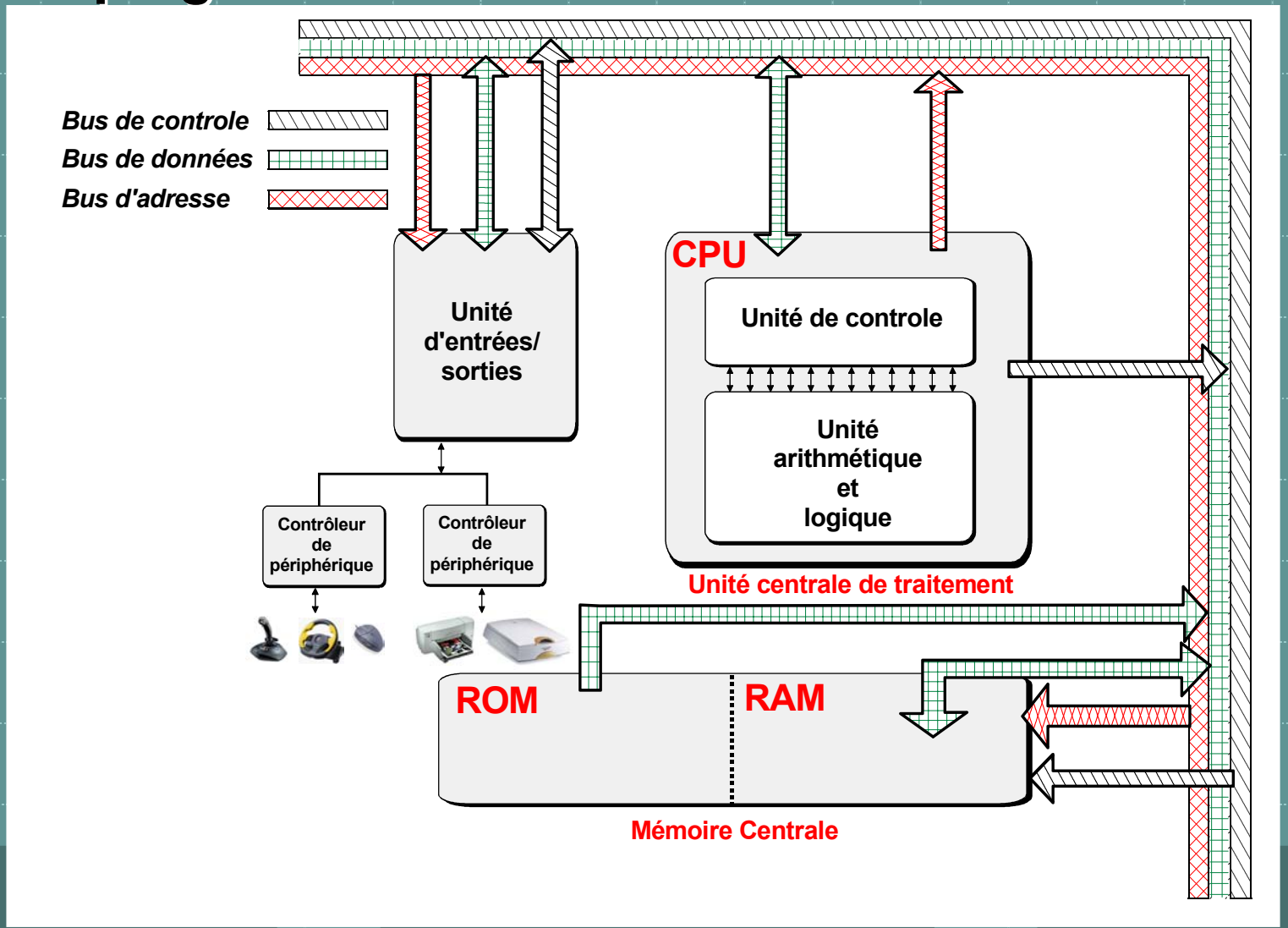
♦Circuits Séquentiels de base

♦ Schéma général d'un compteur synchrone



V. Le Microprocesseur

♦Principe général de fonctionnement



V. Le Micrprocesseur

Principe général de fonctionnement

Mémoire Centrale, Technologie

- ROM → Read Only Memory
  - > Composant permettant de stocker de façon permanente de l'information. Contient des programmes élémentaires pour initialiser le système et lancer le système d'exploitation

- RAM → Random Access Memory
  - > Mémoire à accès aléatoire très rapide, tous les mots possèdent le même temps d'accès.
    - » DRAM →Mémoire dynamique →rafraîchissement périodique des cellules
    - » SRAM →Mémoire statique →pas de rafraîchissement

V. Le Micrprocesseur

- Principe général de fonctionnement
  - Mémoire Centrale, Technologie
    - Tableau comparatif DRAM et SRAM

	Avantages	Désavantages
RAM Dynamique	- Bon marché - Haute intégration - Consomme peu d'énergie - Facile à construire	- Temps d'accès important - Besoin d'un système de rafraîchissement
RAM Statique	- Rapide - Pas de rafraîchissement	- Coûteuse - Faible intégration - Consomme plus de puissance - Génère plus de chaleur - Plus difficile à construire

V. Le Micrprocesseur

- **Le programme**
  - Programme = Données + instructions stockées dans la mémoire

Olivier Habert
mars 2001

Cours-TD d'Informatique Industrielle

Problème n°1

Soit les matrices **M1** = $\begin{pmatrix} 2 & 3 \\ 1 & 5 \end{pmatrix}$ et **M2** = $\begin{pmatrix} 9 & 1 \\ 3 & 7 \end{pmatrix}$

On se propose de calculer **M2 = M1 + M2** *(La matrice **M2** originelle est écrasée)*

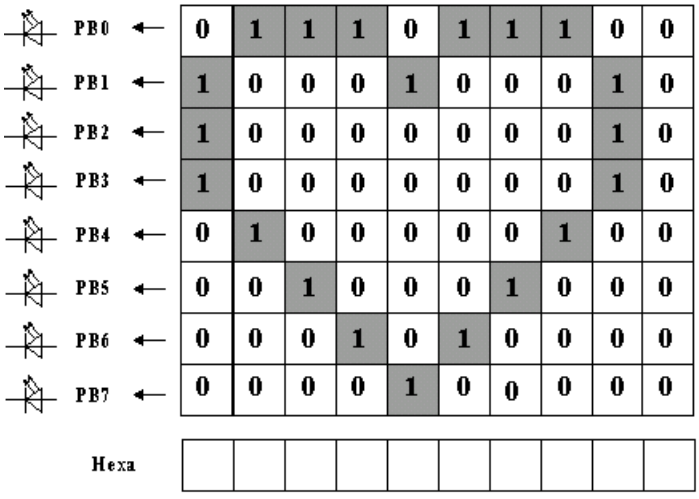
a) Dessiner l'organigramme permettant d'effectuer la somme de **M1** et **M2**. *(Prévoir un algorithme général non limité au rang de la matrice)*

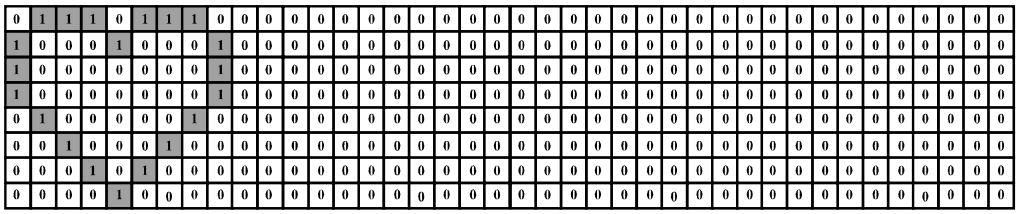
b) Ecrire le programme en assembleur 8086. **Le programme doit rester général et ne pas être figé, de part sa structure, à la somme de deux matrices de rang 2** !

Remarques et Conseils:

- Utiliser deux tableaux de la forme **MAT1 DB 2,3,1,5** et **MAT2 DB 9,1,3,7**
- Utiliser l'adressage indexé basé pour accéder aux éléments des matrices.
- Prendre le couple de registres **Bx** et **SI** pour adresser **M1** et le couple de registre **BP** et **SI** pour adresser **M2.**

Problème n°2





Constitution d'une image

On se propose de réaliser un mini afficheur lumineux. Pour cela on dispose de 8 diodes électroluminescentes de forte luminosité connectées au port **B** d'un **P.I.O**. *(voir adresse des registres du port B en ANNEXE)*.

Le problème consiste à afficher rapidement chaque colonne du motif sur les diodes disposées verticalement.

La rémanence de l'œil humain *(persistance rétinienne)* ainsi qu'un balayage recréé par le déplacement de la tête permet de percevoir fugitivement le motif.

Une image complète est constituée des 10 octets codés précédemment au bout desquels on ajoute 20 octets à la valeur 0 qui permettent de n'apercevoir qu'une seule image par balayage.

a) Remplir chaque case vide en codant la colonne correspondante en hexadécimal.

b) Dessiner l'organigramme du programme permettant d'afficher en permanence 80 images par seconde.

c) A partir de l'organigramme précédent, écrire le programme en assembleur 8086

(Voir Annexe pour les adresses des registres du port, la fréquence du microprocesseur, la durée des instructions,...)

Remarques et Conseils:

- Mettre les 30 octets dans un tableau d'octets sous la forme: **MOTIF DB 0EH, 11H,…**
- Utiliser l'adressage indexé basé avec les registres **BX** et **SI** pour accéder à chaque élément du tableau.
- On supposera le temps d'affichage de chaque image négligeable par rapport à la temporisation entre chaque image.
- Faire un sous programme de temporisation de 1/80 secondes = 12 ms. *(Voir la durée en cycle d'horloge des instructions en Annexe)*

Précisez les détails de tous les calculs !

Problème n°3

Soit le tableau suivant correspondant à des valeurs de temporisation *(nombre d'itérations-1 d'un programme de temporisation comme celui vu en T.D. avec une seule instruction NOP)*

FREQ DW 240, 271, 295, 295, 271, 295, 271, 240, 295, 240, 271, 295, 295, 271, 240, 271, 295

Le nombre 240 permet de préciser le temporisation de la note SI

Le nombre 271 ----------LA

Le nombre 295 ----------SOL

On désire jouer ces notes au moyen d'un programme en assembleur en utilisant le buzzer intégré au système.

Le Buzzer s'active en faisant varier le bit **d6** de l'octet de donnée, pour une adresse **180H**.

Exemple:

```
        MOV DX, 180H
BOU:    OUT DX, AL
        CALL TEMPO   // Tempo définissant la fréquence du son
        XOR AL, 40H  //fait varier alternativement le bit d6
        JMP BOU              //Boucle infinie
```

a) Reconnaître le début d'une berceuse bien connue.

b) En utilisant un des boutons poussoirs reliés au port **A** du **P.I.O.** pour contrôler la durée et le défilement des notes, dessiner un organigramme permettant de jouer ces notes de manière séquentielle *(Ne pas gérer le retour à la première note)*. Chaque appui sur le bouton joue la note suivante tant qu'un nouvel appui n'a pas été effectué.

c) Ecrire le programme en assembleur 8086.

Problème n°4

Soit une phrase de longueur quelconque se terminant par le caractère de code **ASCII 13** *exemple de déclaration en assembleur:*

PHRASE DB "Cet exercice est facile pour des étudiants qui sont venus en cours",13

a) Représentez l'organigramme du programme permettant de rechercher le nombre de lettre(s) **'e'** dans toute phrase respectant cette structure.

b) Ecrire le programme en assembleur 8086. Mettre le résultat dans une variable nommée ***nbe***

Problème n°5

Soit le sous programme de temporisation *(voir page suivante)* :

a) Mettre en équation en fonction de **X** et **Y** le temps perdu par ce sous programme
(on considérera le temps pris par l'instruction CALL).

b) Calculer le temps pour **X =5000**, **Y =20000**.

c) Poser **X=Y** et calculer la valeur de **X** pour effectuer une temporisation de **3 secondes**.

```
DSEG
X: DW ?
Y: DW ?
CSEG
        MOV AX, [X]
BOU1: MOV BX, [Y]
BOU2: NOP
        DEC BX
        JNZ BOU2
        DEC AX
        JNZ BOU1
        RET
```

Problème n°6

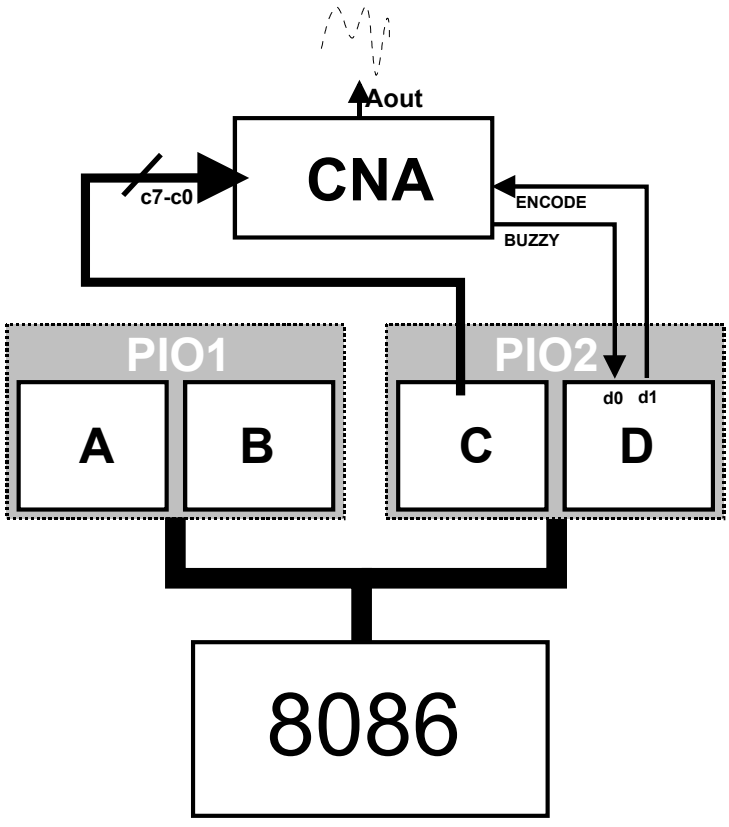
On se propose de générer des signaux analogiques à l'aide d'un système 8086 et de ses ports d'entrées/sorties. Pour cela on utilise un Convertisseur Numérique Analogique ***(CNA)***. Ce convertisseur 8 bits possède:

- 1 sortie analogique $\mathbf{A_{out}}$
- 1 entrée logique **ENCODE**
- 1 sortie logique **BUZZY**
- 8 entrées logiques $\mathbf{c_0 \ldots c_7}$

Rôle de chaque entrée/sortie:

- $\mathbf{A_{out}}$ représente le signal analogique issue de la conversion.
- Un front descendant sur l'entrée **ENCODE** lance la conversion. Ce signal est relié au bit $\mathbf{d_1}$ du port **D** du $\mathbf{PIO_2}$.

- Tant que la conversion est en cours, la sortie **BUZZY** est à un niveau logique haut. Ce signal est relié au bit $\mathbf{d_0}$ du port **D** du $\mathbf{PIO_2}$.
- Les 8 bits $\mathbf{c_0 \ldots c_7}$ du nombre à convertir proviennent du port **C** du $\mathbf{PIO_2}$. Le convertisseur est configuré pour délivrer une tension entre 0 et V_{max} volts. *(pour les programmes prendre V_{max} = 5V)*



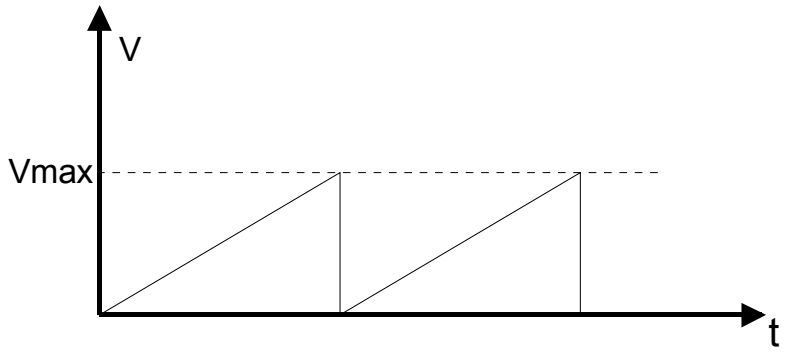
a) Configurer le port **A** en entrée, le port **B** en sortie, le port **C** en sortie et le port **D** en mode bit à bit *(voir explications pour le mode bit à bit en ANNEXES)* pour que $\mathbf{d_0}$ soit en entrée et $\mathbf{d_1}$ en sortie. Les bits non utilisés du port **D** seront programmés en sortie par défaut.

b) On désire générer des signaux de forme simple à la fréquence de 1000 Hz. Pour obtenir un signal lissé, on génère 25 échantillons par période.

- Calculez le temps séparant deux échantillons.
- A partir de ce résultat, sachant que le temps de conversion du CNA est de **5 µs** écrire un sous-programme de temporisation permettant de respecter l'échantillonnage désiré.

c) Représentez l'organigramme permettant de générer un signal en dent de scie sur la pleine échelle du CNA en respectant le cahier des charges suivant :

- L'interrupteur n° 0 du port A servira à activer le début de la génération du signal. Pour cela on détectera un front descendant sur cet interrupteur *(mise à 1 de l'interrupteur puis retour à la position repos)*.
- L'interrupteur n° 1 du port A permettra d'arrêter à tout moment le processus tant qu'il se trouve à l'état 1.
- Pendant l'arrêt du processus par l'interrupteur 1, le programme génère un son de 1000 Hz.
- Pendant la phase de génération du signal la LED n°5 du port B servira de témoin de conversion.



✍ ***Toujours s'assurer que le convertisseur n'est plus occupé avant de commencer une nouvelle conversion.***

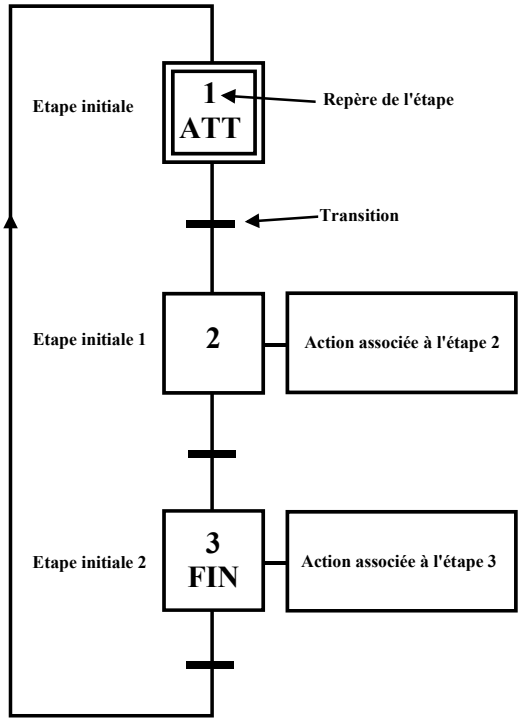
d) Ecrire le programme en assembleur 8086

Problème n°7

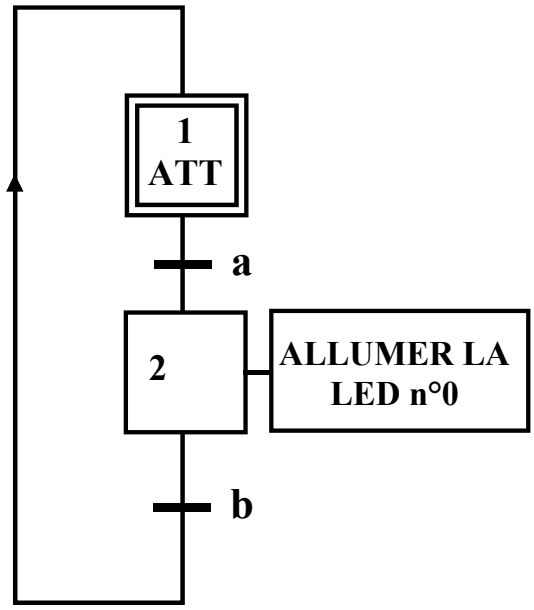
Afin de réaliser l'analyse fonctionnelle des automatismes séquentiels, le grafcet est un outil simple à utiliser et rigoureux sur le plan formel. Un grafcet se compose :

- d'étapes auxquelles sont associées des actions.
- des transitions auxquelles sont associées des réceptivités.
- des liaisons orientées reliant les étapes aux transitions.

Les actions propres à chaque étape sont actives tant que l'étape n'a pas été franchie.
Une étape est franchie quand la condition logique associée à la transition devient vraie.



a) Transformer en organigramme le grafcet élémentaire suivant, où la condition **"a"** représente l'interrupteur n°4 relié au PORT A du PIO *(**"a"** = 1, l'étape est franchie)*, la condition **b** représente l'interrupteur n°5, la LED est connectée sur la ligne 0 du port B.



b) Ecrire le programme correspondant en assembleur 8086.

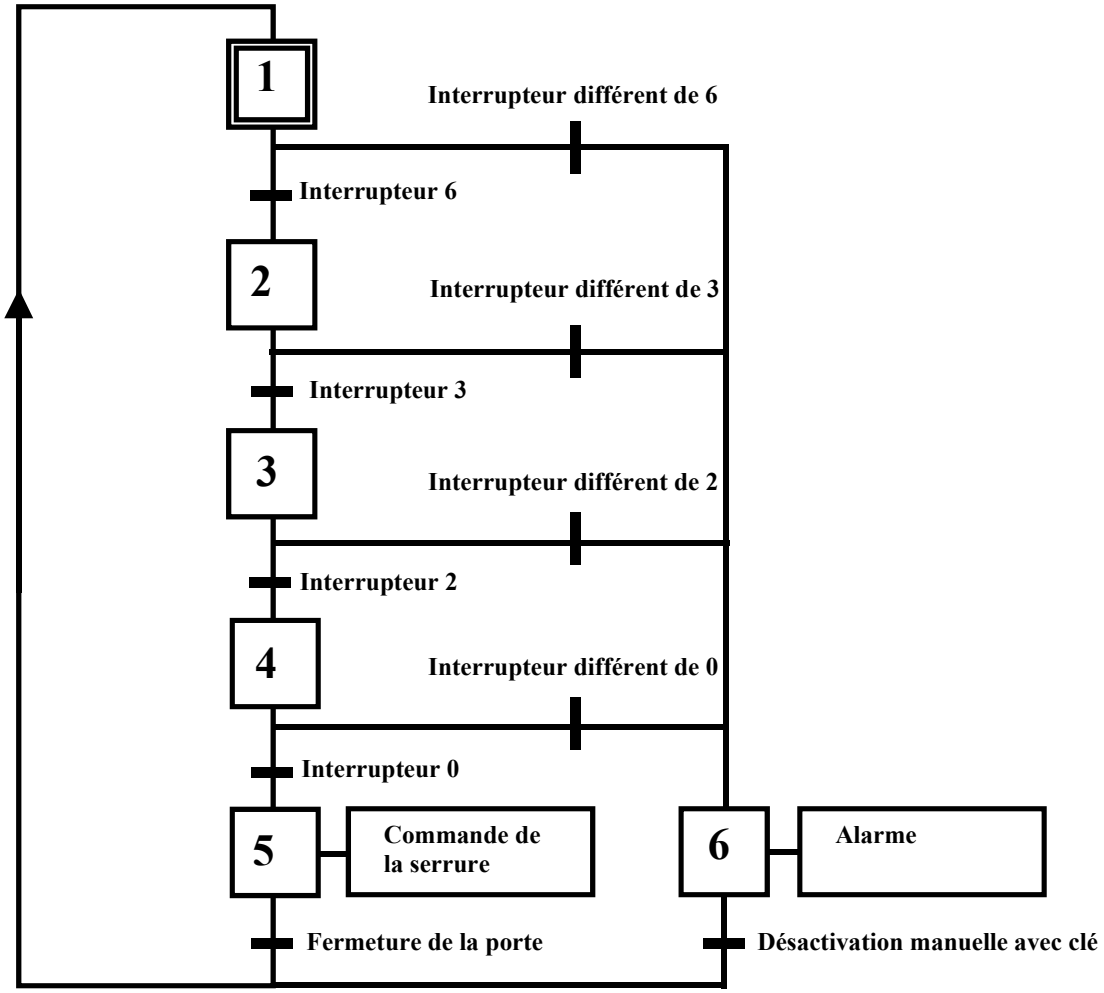
~~~~~~

On se propose maintenant de réaliser une serrure électronique. L'ouverture d'une porte est conditionnée par un code en appuyant dans un ordre précis sur les 4 touches d'un clavier.

Toute fausse manœuvre provoque une alarme sonore bloquant le dispositif qui ne peut être remis en service que par une clé de déverrouillage. Les touches du clavier sont représentées à l'aide des 4 premiers interrupteurs connectés au port A du P.I.O.. La commande de la serrure se simule à l'aide de la LED 0 du Port B. Quand la porte se referme, le système revient dans son état initial
~~~~~~

- La fermeture de la porte est simulée par l'interrupteur n°6 du Port A
- La clé de désactivation sera représentée par l'interrupteur n°7 du Port A.
- Tant que la clé n'a pas été activée, un son d'alarme de fréquence 1000 Hz retentit.
- Le code choisi est **6, 3, 2, 0,** *(ces nombres représentent le numéro des interrupteurs à enclencher successivement dans l'ordre)*

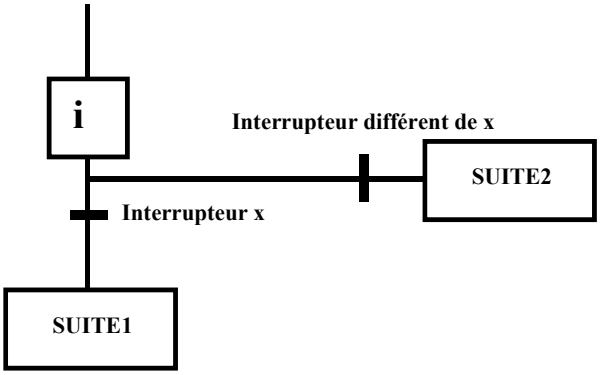
Le grafcet de la serrure est le suivant:



c) Représenter l'organigramme permettant de générer un son de 1500 Hz tant que l'interrupteur n° 7 (clé manuelle) n'est pas à 1.

d) Ecrire le programme en assembleur 8086, Détailler les calculs de la temporisation. *(Voir en annexe l'adresse du buzzer dans l'ordinateur, ainsi que le bit à alternativement mettre à 0 et 1)*

e) Transformer en organigramme la séquence suivante du grafcet



f) Transformer la totalité du grafcet de la serrure en un organigramme.

g) Ecrire le programme correspondant en assembleur 8086 en respectant les consignes suivantes:

- Mémoriser le code choisi dans un tableau *(chaque numéro correspond au numéro de l'interrupteur actionné, sachant qu'on ne peut mécaniquement activer qu'un seul interrupteur à la fois)*
- Mettre le programme de temporisation écrit en **d)** sous la forme d'un sous-programme.

ANNEXES

Registres	@ port A PIO1 (Interrupteur)	@ port B PIO1 (LED)	@ port C PIO2	@ port D PIO2
Registre de commande	06	07	10	11
Registre de données	04	05	08	09

Adresse du Buzzer: 384
Commande du buzzer: Bit 6 du bus de donnée
Port en Entrée: 79 dans le registre de commande
Port en Sortie: 15 dans le registre de commande
Fréquence du Processeur: 4.7 MHz
Durée en cycles des instructions:

JNZ →4 sans branchement et 16 avec branchement,
MOV Reg, mem→10,
CALL→19,
RET→12,
LOOP→5 sans branchement et 17 avec branchement,
MOV Reg,data →4,
NOP →3
DEC reg →3

Configuration en bit à bit d'un port:

Elle se fait en deux étapes:

- Tout d'abord, écrire le nombre **207** dans le registre de commande du port
- Ensuite, écrire sur ce même registre de commande un octet dans lequel chaque bit représente le mode de fonctionnement de la ligne correspondante selon la convention suivante:
  - Si le bit i est à 1 → la ligne i est en entrée
  - Si le bit i est à 0 → la ligne i est en sortie

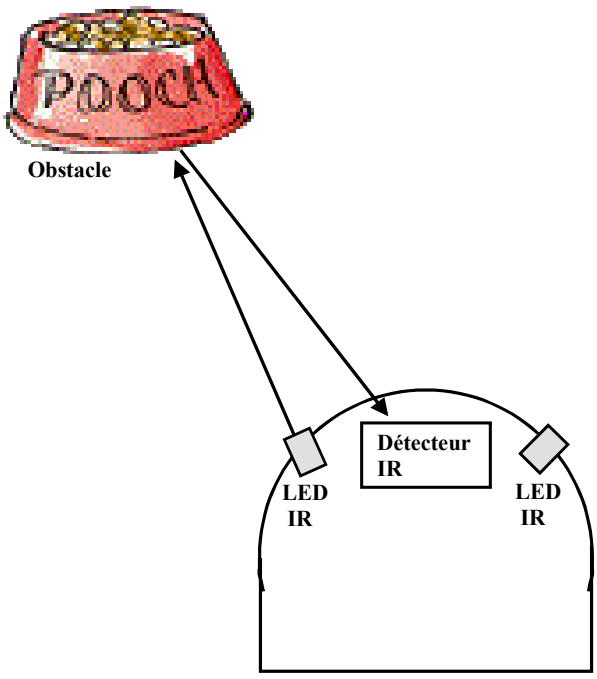
Maîtrise EEA tronc commun **Janvier 2003**
Durée: 2 h00
Documents autorisés: aucun
Enseignant: Olivier HABERT

Examen de MicroInformatique

Problème

Capteurs de proximité infrarouge pour robot

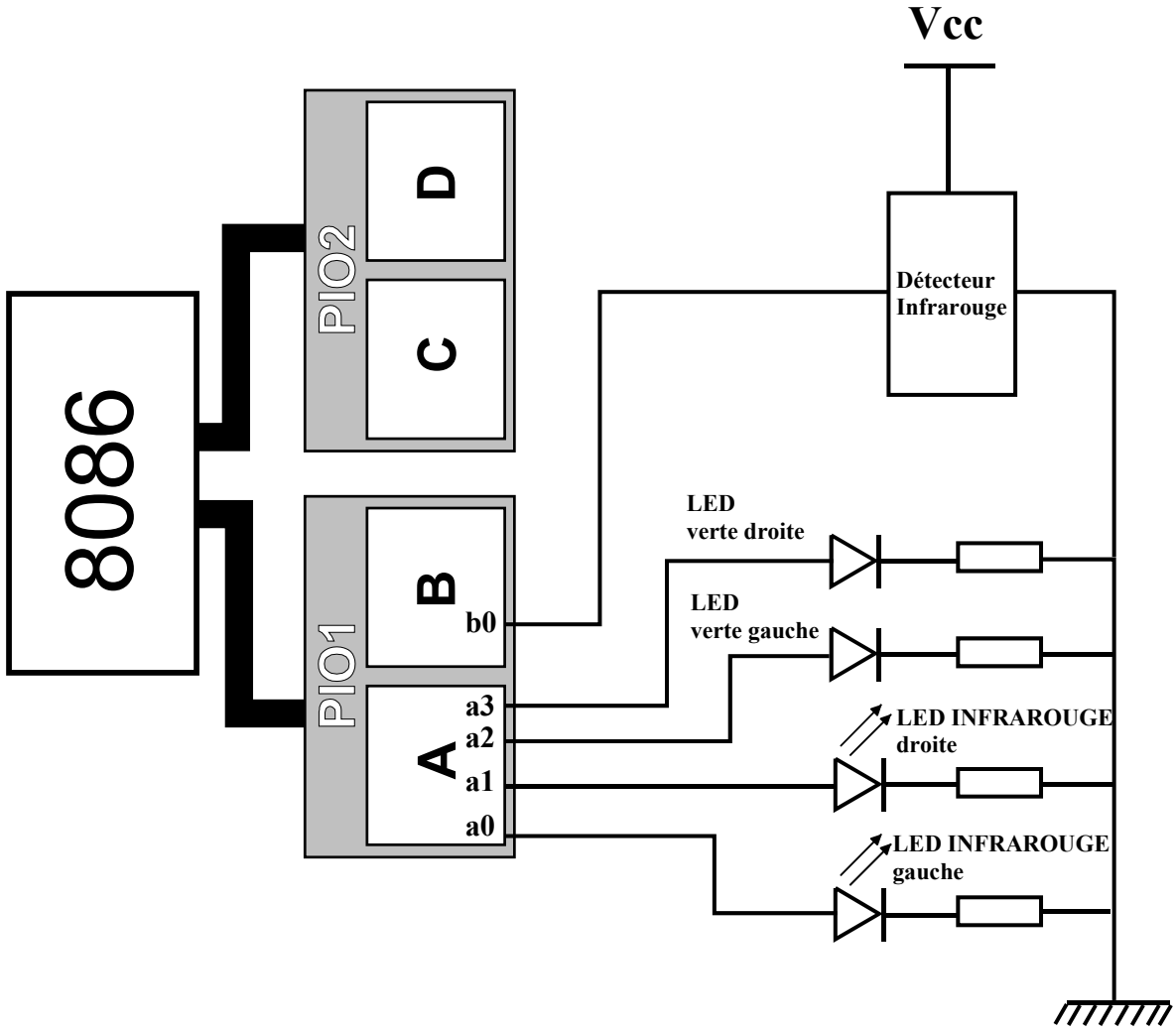
En robotique, il est courant d'utiliser la technologie infrarouge afin de détecter les obstacles proches du robot. Selon le résultat de la détection, on peut alors agir à travers un algorithme particulier sur les moteurs pour éviter *(ou atteindre)* les obstacles se trouvant sur la trajectoire du robot.
Pour cela on émet un signal par l'intermédiaire d'une LED infrarouge dans la direction que l'on désire tester. Dans le cas ou ce signal rencontre un obstacle, le signal est réfléchi et revient vers un détecteur infrarouge monté sur le robot.
Nous nous placerons dans le cas d'un robot munit de deux yeux *(deux émetteurs infrarouge)* . Un émetteur scrute à droite et l'autre scrute à gauche. Les émissions gauche et droite se feront successivement.



Nous proposons dans ce sujet d'écrire le programme:

- Qui génère le signal d'émission alternativement vers chaque direction.
- Qui teste pour chaque émission la sortie du détecteur afin de vérifier si ce dernier a reçu un signal réfléchi
- Qui allume une LED verte (une par coté) s'il y a détection d'un obstacle

Nous utiliserons un microprocesseur 8086 et ses ports d'entrées/sorties A et B pour générer les signaux d'émission et tester la sortie du détecteur infrarouge.
Les différentes lignes utilisées des ports A et B sont représentées sur le schéma suivant.

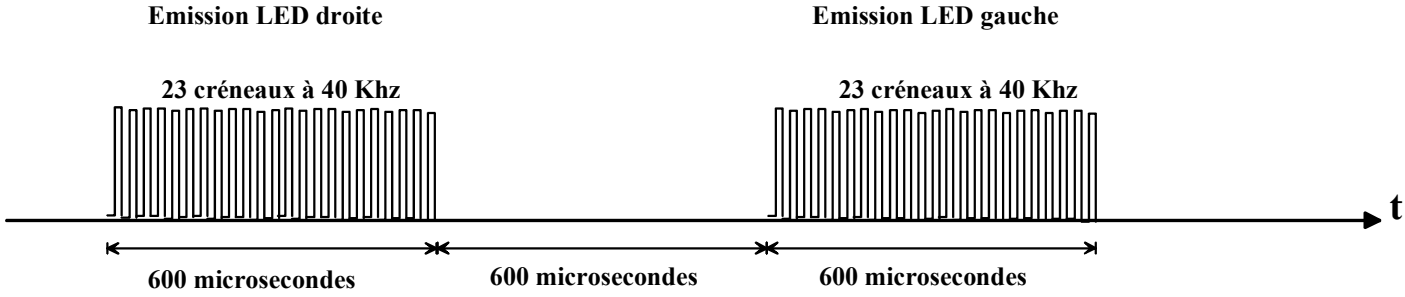


En résumé :

- Emission LED infrarouge gauche : ligne 0 (a_0) du port A
- Emission LED infrarouge droite : ligne 1 (a_1) du port A
- Allumage LED visible verte gauche : ligne 2 (a_2) du port A
- Allumage LED visible verte droite : ligne 3 (a_3) du port A
- Lecture de la sortie du détecteur infrarouge : ligne 0 (b_0) du port B

Emission d'un signal infrarouge

Afin de rendre le signal moins sensible à l'éclairage ambiant, les LED infrarouge sont alimentées par un signal rectangulaire de fréquence 38 Khz, soit une période de 26.31 **µs**.
Pour alimenter successivement les LED IR droite et gauche, chaque diode IR émet pendant 600µs avant une attente de 600**µs** entre chaque émission.
Il faut donc générer successivement environ 23 créneaux à 38 khz pour chaque LED IR.



Question 1 : *Calcul des temporisations pour générer les créneaux et pour attendre 600 **µs***

- En utilisant la structure du sous-programme de temporisation donné en annexes *(vu en COURS/TD)*, calculer la variable **x** pour que ce sous-programme génère une temporisation de (26.31/2) **µs**.

- Idem pour une temporisation de 600µs

  - Donner tous les détails des calculs !!!!
  - Expliquer pourquoi il faut diviser par deux la période pour le calcul de la temporisation des créneaux.

- Ecrire en 8086 les deux sous-programmes respectivement nommés '***TEMPO1***' et '***TEMPO2***'

Question 2 : *Initialisation des ports*

Ecrire les lignes de code 8086 qui permettent de configurer le port A en sortie et le port B en entrée. *(voir annexes pour l'adresse des ports E/S)*
Mettre ces lignes dans un sous-programme nommé '***INIT***'

Question 3 : *Sous-programmes qui génèrent 23 créneaux sur les lignes a_0 et a1 du port A*

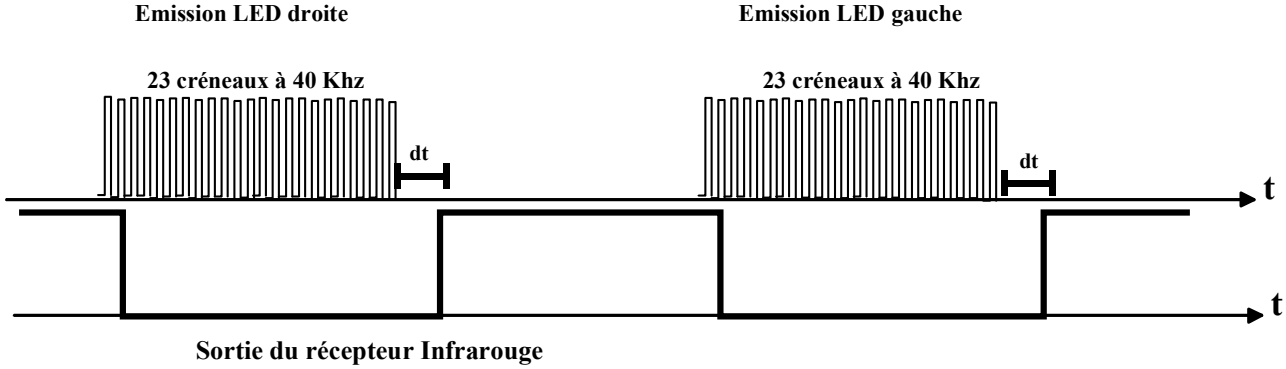
Ecrire en 8086 un sous-programme nommé '***LEDIR_G***' qui envoie 23 créneaux sur la ligne 0 du port A.

Ecrire en 8086 un sous-programme nommé ***'LEDIR_D'*** qui envoie 23 créneaux sur la ligne 1 du port A.

Réception du signal Infrarouge

Le récepteur infrarouge est un circuit spécialisé dont la sortie se trouve à un niveau logique '1' (5V) lorsqu'il ne reçoit pas d'onde réfléchie et à un niveau logique 0 lorsqu'il reçoit l'onde modulée à 38 Khz.
Entre le moment où l'on arrête l'émission et le moment ou la sortie du récepteur revient à son niveau de non-réception (5V), un délai **dt** nous laisse le temps de tester par programme l'état de la sortie du récepteur.

Le signal suivant est obtenu en sortie du récepteur infrarouge lorsque l'on place un obstacle à l'avant gauche et à l'avant droit du robot *(une feuille blanche)*.



Sortie du récepteur Infrarouge

Question 4 : *lecture de la sortie du récepteur infrarouge*

Ecrire un sous-programme nommé ***'LECT_REC'*** qui récupère l'état de la sortie du récepteur infrarouge.

Pour cela, il faut

- lire l'intégralité du port B
- faire un masque pour mettre tous les bits du registre de lecture Al à 0 sauf le bit 0 qui correspond à la sortie du récepteur
- tester si le registre Al est nul ou non
- Dans le cas où Al est nul, alors mettre 1 dans le registre Dl et 0 dans le cas contraire.

Nous avons donc un sous-programme qui lit la sortie du récepteur et qui renvoie 1 dans Dl si le récepteur IR a récupéré une onde et 0 dans le cas contraire.

Question 5 : *Programme principal*

Ecrire le programme principal en assembleur 8086 qui :

1. Emet le signal sur la LED IR gauche (23 créneaux)
2. Teste la sortie du récepteur IR
3. Allume la LED verte gauche s'il y a réception et l'éteint dans le cas contraire
4. Attend 600 **µs**
5. Emet le signal sur la LED IR droite (23 créneaux)
6. Teste la sortie du récepteur IR
7. Allume la LED verte droite s'il y a réception et l'éteint dans le cas contraire
8. Attend 600 **µs**
9. Revient en 1.

Utiliser au maximum les sous-programmes que l'on a écris précédemment *(ex : CALL LEDIR_G pour générer le signal sur la LER IR Gauche ou CALL INIT pour initialiser les ports)*

ANNEXES

Registres	@ port A PIO1 (Interrupteur)	@ port B PIO1 (LED)	@ port C PIO2	@ port D PIO2
Registre de commande	06	07	10	11
Registre de données	04	05	08	09

Adresse du Buzzer: 384
Commande du buzzer: Bit 6 du bus de donnée
Port en Entrée: 79 dans le registre de commande
Port en Sortie: 15 dans le registre de commande
Fréquence du Processeur: 4.7 MHz
Durée en cycles des instructions:

JNZ →4 sans branchement et 16 avec branchement,
MOV Reg, mem→10,
CALL→19,
RET→12,
LOOP→5 sans branchement et 17 avec branchement,
MOV Reg,data →4,
NOP →3
DEC reg →3

Structure de programme d'une temporisation:

La durée de la temporisation dépend de la donnée x !!

```
TEMPO:  MOV CX, x
TT:     NOP
        LOOP TT
        RET
```

- L'interruption **$21** permet dans une de ses variantes d'afficher un caractère à l'écran. Pour cela il faut mettre **02** dans **ah** et le code ASCII du caractère dans **dl**. Ensuite on appelle l'interruption par l'instruction **int 21**.

- L'interruption **21** permet quand **ah** = **08** d'attendre le premier caractère tapé au clavier. Le caractère se trouve alors dans **al**.

- L'interruption **$21** avec **ah** à 9 permet d'afficher des chaînes de caractères qui se terminent par le caractère 13, 10 ,'$'. *(13 → retour chariot, 10 → ligne suivante, $ → fin de chaîne de caractères).* Pour cela, il faut définir un message avec la pseudo instruction **db** en terminant ce message par 13, 10, '$'. Il faut ensuite affecter au registre **dx** l'adresse de ce message *(mov dx, mess où mess est le nom donné au message).*

Olivier HABERT
Septembre 2001

MicroInformatique
Informatique Industrielle
Assembleur 8086/8088
Seconde Session

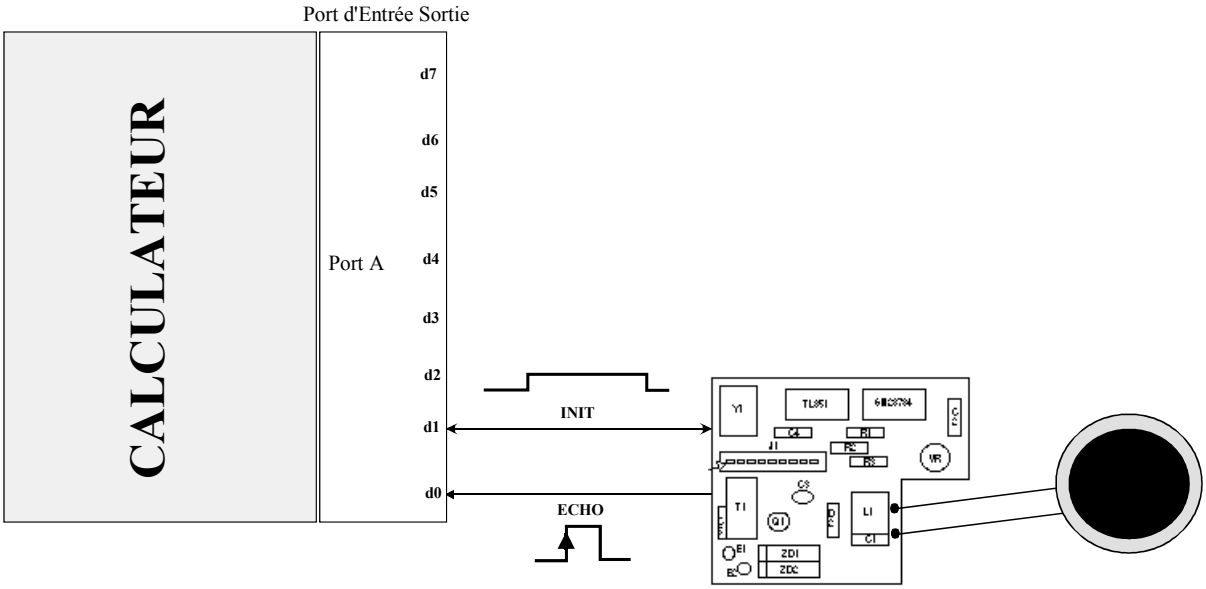
Problème

On se propose de s'interfacer avec un système de mesure de distance par capteur à Ultrasons.

1. Positionnement du problème

On dispose d'un transducteur à ultrasons et de sa carte de commande. Ces deux composants permettent de mesurer des distances allant de quelques centimètres à 10 mètres.
Pour cela la carte de commande fait transmettre une onde ultrasonore au transducteur. L'onde se propage de manière conique dans l'environnement et se réfléchit sur le premier obstacle rencontré.
Le transmetteur reçoit cette onde réfléchie et met en forme un signal d'écho.
Le temps qui sépare l'émission de l'onde et la réception de l'onde réfléchie permet de déduire la distance séparant le transducteur de l'obstacle le plus proche. *(l'onde se propage à la vitesse du son de ~ 330 m/s)*.

2. Interfaçage des différents éléments



3. Signaux de commande

La carte se commande au moyen de deux signaux: INIT et ECHO. Une mise à 1 du signal INIT déclenche la mesure *(INIT doit rester à 1 pendant la durée de la mesure)*.
Le premier front montant sur ECHO correspond à la réception de la première onde réfléchie.

4. Interface avec l'informatique

Pour envoyer des signaux logiques, nous allons utiliser un port d'entrée/sortie. Le registre de commande de ce port se trouve à l'adresse 300h. Le registre de données de ce port se trouve à l'adresse 0x301.
Il existe trois manières de configurer ce port: en entrée, en sortie en mode bit à bit *(chaque ligne du port est configurable soit en entrée, soit en sortie)*.
La configuration qui nous intéresse est le mode bit à bit.

Configuration en bit à bit d'un port:

Elle se fait en deux étapes:

- Tout d'abord, écrire le nombre **CFh** dans le registre de commande du port
- Ensuite, écrire sur ce même registre de commande un octet dans lequel chaque bit représente le mode de fonctionnement de la ligne correspondante selon la convention suivante:
  - Si le bit i est à 1 → la ligne i est en entrée
  - Si le bit i est à 0 → la ligne i est en sortie

5. Travail à effectuer

- Ecrire en assembleur le programme qui met le port d'entrée/sortie en mode bit à bit et qui configure la ligne d0 en entrée et la ligne d1 en sortie.
- Ecrire à la suite le code qui positionne l'INIT à 1
- Faire une boucle que attend le premier front montant de l'écho.
- Dans cette boucle, on incrémente à chaque fois le registre Cx que l'on aura préalablement mis à 0 avant la mesure.
- Montrer que connaissant la valeur de ce compteur lorsque l'on sort de la boucle, connaissant la fréquence du microprocesseur, connaissant la durée en cycles d'horloge des instructions, on peut déduire le temps écoulé entre le déclenchement du capteur et la réception du premier écho.
- Calculer le temps écoulé si on mesure Cx=100.

Fréquence du Processeur: 4.7 MHz
Nombre de cycles des instructions:

JNZ →4 cycles sans branchement et 16 cycles avec branchement,
MOV Reg, mem→10 cycles,
CALL→19 cycles,
RET→12 cycles,
LOOP→5 cycles sans branchement et 17 cycles avec branchement,
MOV Reg,data →4 cycles,
NOP →3 cycles
IN al, Dx→8 cycles, OUT Dx, al→8 cycles
INC reg →3 cycles
Or Reg,data, And Reg,Data →4 cycles

Licence IUP option TI **Juin 2002**

Durée: 2 h00

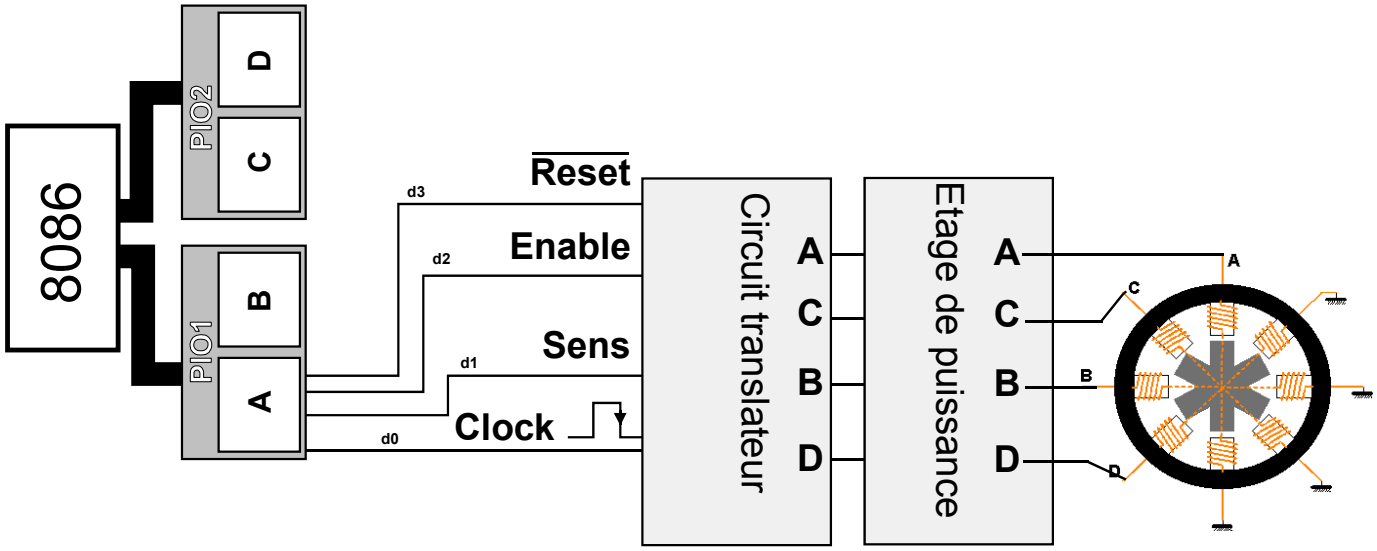
Documents autorisés: aucun

Enseignant: Olivier HABERT

Examen de MicroInformatique

Problème

On se propose de commander la rotation d'un moteur pas à pas ***(100 pas par tour)***. Pour cela on passe par un circuit spécialisé dans la commande de moteur pas à pas ***(circuit translateur)*** et d'un étage de puissance qui prend en charge l'alimentation des bobines du stator pour la génération de la rotation ***(voir dessin ci dessous)***.



Le circuit translateur dispose de 4 entrées pour la commande du moteur:

- Une entrée ***Clock*** active sur front descendant qui permet d'incrémenter d'un pas la position du rotor.
- Une entrée ***Sens*** qui permet de définir le sens de rotation ***(état haut: → sens horaire et état bas: →sens trigonométrique)***
- Une entrée ***Enable*** qui permet d'activer l'étage de puissance ***(état haut: → étage de puissance activé et état bas:→un niveau bas est envoyé à toutes bobines du moteur)***
- Une entrée ***Reset*** qui remet le moteur dans une position initiale. ***(Etat bas → pendant 200 ms remet le rotor dans une position initiale en alimentant les bobines du moteur dans un état prédéfini)***

Ces quatre entrées sont reliées au port A du PIO 1 d'un microprocesseur:

- Clock → bit d_0
- Sens → bit d_1
- Enable → bit d_2
- Reset → bit d_3

Question N°1 (3 points)

En vous aidant des renseignements donnés en annexes, écrire les lignes de programme 8086 qui configurent en sortie le port A du PIO 1.

Question N°2 (3 points)

En reprenant la structure de programme utilisée en TD ***(voir annexes)***, calculer la valeur de la donnée ***data*** pour une temporisation de 100 ms.
Expliquer en détails les calculs avec des phrases !

Question N° 3 (3 points)

Ecrire les lignes de programme 8086 qui:

- positionnent la sortie d_3 du port A à un état haut
- maintiennent **pendant 100 ms** la sortie d_3 du port A à un état bas
- remettent d_3 à un état haut

Question N° 4 (5 points)

Ecrire le programme en assembleur 8086 qui fait tourner le moteur dans le sens horaire à une vitesse de 1 tour par seconde ***(donc 100 pas par seconde).***
Dessinez l'allure du signal symétrique qu'il faut envoyer sur l'entrée **Clock** du translateur et préciser la période de ce signal. En déduire la durée de la tempo.

Calculer cette tempo comme dans la question n°2.

☝ Attention quand vous écrivez dans le registre de donnée du port A pour faire tourner le moteur, il ne faut modifier que les bits qui doivent l'être ***(le bit relié à Clock)*** et non ceux comme d_1, d_2 ou d_3 qui concernent le *sens*, le *reset* ou *l'enable*…

Question N° 5 (6 points)

Ecrire un programme en assembleur 8086 qui fait tourner le moteur dans le sens horaire à la vitesse de 1 tour/s pendant **25** tours puis ensuite à la même vitesse dans le sens trigonométrique pendant **50** tours

ANNEXES

Registres	@ port A PIO1 (Interrupteur)	@ port B PIO1 (LED)	@ port C PIO2	@ port D PIO2
Registre de commande	06	07	10	11
Registre de données	04	05	08	09

Port en Entrée: 79 dans le registre de commande
Port en Sortie: 15 dans le registre de commande
Fréquence du Processeur: 4.7 MHz
Durée en cycles des instructions:

JNZ →4 sans branchement et 16 avec branchement,
MOV Reg, mem→10,
CALL→19,
RET→12,
LOOP→5 sans branchement et 17 avec branchement,
MOV Reg,data →4,
NOP →3
DEC reg →3

Structure de programme d'une temporisation:

```
NOM SP:   MOV CX, data
TT:       NOP
          LOOP TT
          RET
```

U.F.R. SI.F.A.

Janvier 2004

Maîtrise EEA
Durée: 2h00
Documents autorisés: aucun
Enseignant: Olivier HABERT

☝ **Deux points sont réservés pour noter le soin, la lisibilité des programmes, les explications, … !**

Examen de MicroInformatique

Problème : Lecture d'un clavier par microprocesseur

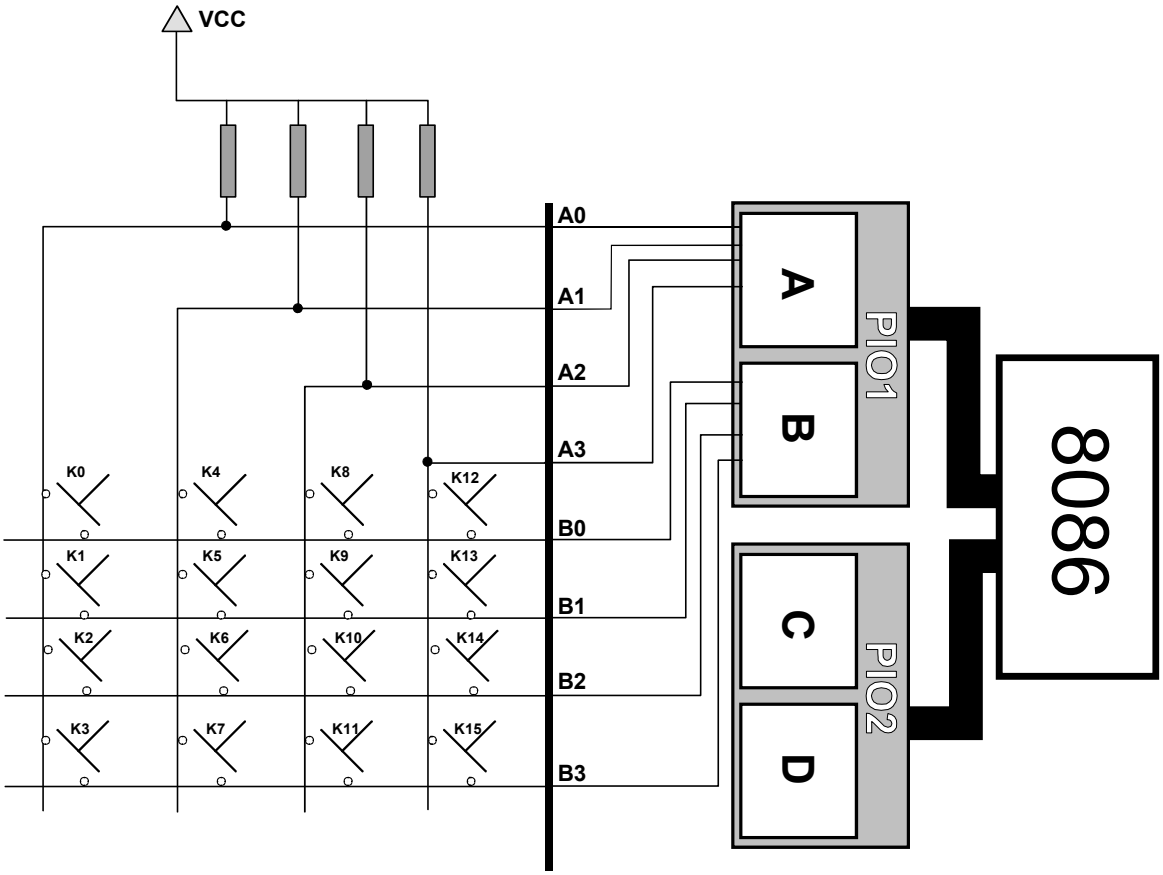
On se propose d'écrire un programme en assembleur 8086 permettant de scruter un clavier 16 touches afin de déterminer la touche dernièrement appuyée.

Soit le montage qui est représenté ci dessous :

Il est constitué d'une matrice de 4 lignes et de 4 colonnes reliées aux ports A et B d'un PIO d'un microprocesseur 8086.

16 interrupteurs relient une ligne à une colonne, appliquant le potentiel de la ligne à la colonne correspondante. Les lignes B_0, B_1, B_2, B_3 sont configurées en sortie et les colonnes A_0, A_1, A_2, A_3 sont configurées en entrée.

Il s'agit ensuite de mettre successivement chaque ligne à 0 *(les autres lignes étant à 5 Volts)* et de tester grâce aux colonnes A_0, A_1, A_2, A_3 l'état des touches de la ligne testée.



Exemple : Nous allons tester la ligne B_0. Pour cela, appliquons une tension de 0 Volts à B_0 et 5 Volts pour les lignes B_1, B_2, B_3.
Si une touche de la ligne B_0 est appuyée *(par exemple K_4)*, alors une tension de 0 Volts est appliquée sur l'entrée A_1 et un état logique 0 sera détecté par le microprocesseur. Les autres entrées A_0, A_2, A_3 seront à 5 Volts et on mesurera des états logiques 1.
Le principe de la scrutation du clavier consiste à alternativement tester chaque ligne en la mettant à un niveau bas puis à détecter si une des 4 colonnes se trouve à un niveau bas.

Question N°1 (3 points)

Ecrire les lignes de programme qui configurent le port B en sortie et le port A en entrée.
Mettre ces lignes dans un sous-programme INIT_PORT.

Question N°2 (3 points)

En utilisant la structure de temporisation vue en TD et rappelée en Annexes, calculer la valeur qu'il faut mettre dans Cx pour avoir une temporisation de 5 ms.
Ecrire le programme de temporisation en l'insérant dans un sous-programme TEMPO.

☝ Cette question ne sera validée que si toutes les explications sur la méthodologie et les calculs sont données.

Question N°3 (6 points) Test des colonnes

La logique de test des colonnes est la suivante :

- Lire le port A avec l'instruction **IN**
- Mettre à 0 les bits 4,5,6,7 du registre Al puisqu'ils ne représentent rien.
- Tester les bits 0,1,2,3 dans cet ordre en utilisant l'instruction **TEST AL, Masque** *(équivalent à un AND mais ne modifie pas le registre Al). Masque* est un octet où seul le bit testé dans Al est à 1.
- Si l'un des bits 0,1,2 ou 3 est à 0, alors la touche correspondante est appuyée :
    - Mettre dans le registre Bx le numéro de la colonne (1,2,3 ou 4) de la touche appuyée
    - Quitter le sous-programme
- Si aucun des quatre bits n'est à 0, alors aucune touche n'est appuyée :
    - Mettre 0 dans Bx et quitter le sous-programme

Représenter l'organigramme du programme

Ecrire un sous-programme TEST_COL qui teste séquentiellement chaque colonne en suivant la logique précédente

Question N°4 (6 points) Test des lignes

La logique de test des lignes est la suivante :

- Mettre la ligne testée à 0 et les autres à 1
- Appeler le sous-programme TEST_COL
- Si Bx est à 0 alors passer au test de la ligne suivante après une temporisation de 5ms

- Si on vient de tester la dernière ligne *(ligne 4)*, alors revenir à la ligne n°1.

- Si Bx est différent de 0 alors:
  - Calculer le numéro **x** de la touche qui est appuyée en utilisant la valeur de Bx et le numéro **n** de la ligne en cours de test. *(x = (Bx-1)*4 +n-1)*
  - Mettre **n** dans le registre **Dx**
  - Attendre 5 ms
  - Revenir au test de la ligne n°1

Représenter l'organigramme du programme.

Ecrire le programme complet en assembleur 8086.

ANNEXES

Registres	@ port A PIO1	@ port B PIO1	@ port C PIO2	@ port D PIO2
Registre de commande	06	07	10	11
Registre de données	04	05	08	09

Port en Entrée: 79 dans le registre de commande
Port en Sortie: 15 dans le registre de commande
Fréquence du Processeur: 4.7 MHz
Durée en cycles des instructions:

JNZ →4 sans branchement et 16 avec branchement,
MOV Reg, mem→10,
CALL→19,
RET→12,
LOOP→5 sans branchement et 17 avec branchement,
MOV Reg,data →4,
NOP →3
DEC reg →3

Structure de programme d'une temporisation:

```
NOM SP:   MOV CX, data
TT:       NOP
          LOOP TT
          RET
```

8086 Instruction Set Summary

Data Transfer Instructions

MOV	Move byte or word to register or memory
IN, OUT	Input byte or word from port, output word to port
LEA	Load effective address
LDS, LES	Load pointer using data segment, extra segment
PUSH, POP	Push word onto stack, pop word off stack
XCHG	Exchange byte or word
XLAT	Translate byte using look-up table

Logical Instructions

NOT	Logical NOT of byte or word (one's complement)
AND	Logical AND of byte or word
OR	Logical OR of byte or word
XOR	Logical exclusive-OR of byte or word
TEST	Test byte or word (AND without storing)

Shift and Rotate Instructions

SHL, SHR	Logical shift left, right byte or word? by 1 or CL
SAL, SAR	Arithmetic shift left, right byte or word? by 1 or CL
ROL, ROR	Rotate left, right byte or word? by 1 or CL
RCL, RCR	Rotate left, right through carry byte or word? by 1 or CL

Arithmetic Instructions

ADD, SUB	Add, subtract byte or word
ADC, SBB	Add, subtract byte or word and carry (borrow)
INC, DEC	Increment, decrement byte or word
NEG	Negate byte or word (two's complement)
CMP	Compare byte or word (subtract without storing)
MUL, DIV	Multiply, divide byte or word (unsigned)
IMUL, IDIV	Integer multiply, divide byte or word (signed)
CBW, CWD	Convert byte to word, word to double word (useful before multiply/divide)

Adjustments after arithmetic operations:

AAA, AAS, AAM, AAD	ASCII adjust for addition, subtraction, multiplication, division (ASCII codes 30-39)
DAA, DAS	Decimal adjust for addition, subtraction (binary coded decimal numbers)

Transfer Instructions

JMP	Unconditional jump (*short* ?127/8, *near* ?32K, *far* between segments)

Conditional jumps:

JA (JNBE)	Jump if above (not below or equal)? +127, -128 range only
JAE (JNB)	Jump if above or equal(not below)? +127, -128 range only
JB (JNAE)	Jump if below (not above or equal)? +127, -128 range only
JBE (JNA)	Jump if below or equal (not above)? +127, -128 range only
JE (JZ)	Jump if equal (zero)? +127, -128 range only
JG (JNLE)	Jump if greater (not less or equal)? +127, -128 range only
JGE (JNL)	Jump if greater or equal (not less)? +127, -128 range only
JL (JNGE)	Jump if less (not greater nor equal)? +127, -128 range only
JLE (JNG)	Jump if less or equal (not greater)? +127, -128 range only
JC, JNC	Jump if carry set, carry not set? +127, -128 range only
JO, JNO	Jump if overflow, no overflow? +127, -128 range only
JS, JNS	Jump if sign, no sign? +127, -128 range only
JNP (JPO)	Jump if no parity (parity odd)? +127, -128 range only
JP (JPE)	Jump if parity (parity even)? +127, -128 range only

Loop control:

LOOP	Loop unconditional, count in CX, short jump to target address
LOOPE (LOOPZ)	Loop if equal (zero), count in CX, short jump to target address
LOOPNE (LOOPNZ)	Loop if not equal (not zero), count in CX, short jump to target address
JCXZ	Jump if CX equals zero (used to skip code in loop)

Subroutine and Interrupt Instructions

CALL, RET	Call, return from procedure (inside or outside current segment)
INT, INTO	Software interrupt, interrupt if overflow
IRET	Return from interrupt

String Instructions

MOVS	Move byte or word string
MOVSB, MOVSW	Move byte, word string
CMPS	Compare byte or word string
SCAS	Scan byte or word string (comparing to A or AX)
LODS, STOS	Load, store byte or word string to AL or AX

Repeat instructions (placed in front of other string operations):

REP	Repeat
REPE, REPZ	Repeat while equal, zero
REPNE, REPNZ	Repeat while not equal (zero)

Processor Control Instructions

Flag manipulation:

STC, CLC, CMC	Set, clear, complement carry flag
STD, CLD	Set, clear direction flag
STI, CLI	Set, clear interrupt enable flag
LAHF, SAHF	Load AH from flags, store AH into flags
PUSHF, POPF	Push flags onto stack, pop flags off stack

Coprocessor, multiprocessor interface:

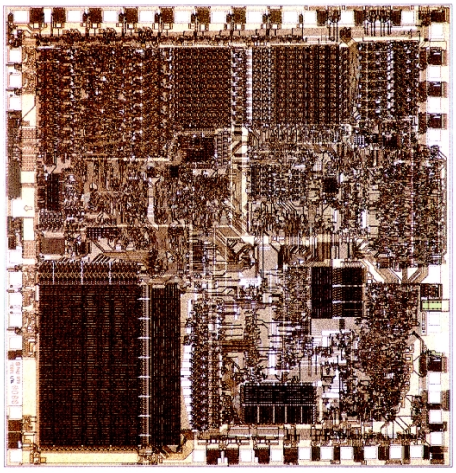
ESC	Escape to external processor interface
LOCK	Lock bus during next instruction

Inactive states:

NOP	No operation
WAIT	Wait for TEST pin activity
HLT	Halt processor

Olivier HABERT

MicroInformatique
Informatique Industrielle
Assembleur 8086/8088
TP n°1



Registres du 8086/8088

16 bits

Registres d'utilisation générale

AH	AL	Ax (pour multiplication, division, Entrées/Sorties)
BH	BL	Bx (pour pointer vers une adresse de base)
CH	CL	Cx (pour compter et décaler)
DH	DL	Dx (pour multiplication, division, Entrées/Sorties)

Registres d'Index et Pointeurs

SP	Pointeur de Pile (Stack Pointer)
BP	Pointeur de Base (Base Pointer)
SI	Index de Source (Source Index)
DI	Index de Destination(Destination Index)

Registres de Segment

CS	Segment de Code (Code Segment)
DS	Segment de Données (Data Segment)
SS	Segment de Pile (Stack Segment)
ES	Segment Extra (Extra Segment)

Autres Registres

IP	(Pointeur d'Instruction) Instruction Pointer
ST	(Registre d'Etat) Status

I. BUT DU TP

Se familiariser avec les registres du 8086 et certaines instructions de base

II. Présentation de DEBUG

DEBUG est un programme fournit avec MSDOS. Dans un premier temps, nous allons utiliser DEBUG pour consulter/modifier les registres, pour afficher/modifier le contenu de la mémoire et enfin pour éditer/exécuter des programmes élémentaires.

a) Chargement de DEBUG

- Si vous êtes déjà sous DOS, taper **DEBUG** + Retour Chariot *(CR)*
- Si vous êtes sous WINDOWS, Relancer la machine sous DOS ou ouvrir une fenêtre DOS. Taper alors **DEBUG** + Retour Chariot *(CR)*.
- Le prompt suivant apparaît: **-**

b) Consultation des registres du processeur

DEBUG permet de consulter les registres du microprocesseur. Pour cela taper la commande **R** et observer l'affichage. Noter les registres qui apparaissent et leur contenu.
CS et IP représentent l'adresse du programme dans la mémoire.
Que vaut la valeur des segments **CS** et **IP** ? Calculer l'adresse réelle *(en décimal et en hexadécimal)* que représente ce couple Segment : Déplacement.
Donner **5** autres couples Segment : Déplacement permettant d'atteindre la même adresse réelle *(justifier vos propositions)*.

Taper la commande **R reg** où reg est un registre *(Ax par exemple)*. Le contenu du registre spécifié apparaît. Editer à la suite un nombre de 16 bits en hexadécimal *(ex : ABFD)*. Relancez la commande **R reg** et vérifier que le registre contient bien le nombre.

c) Edition de lignes de programme

Remettre **0000** dans le registre **Ax**. Taper la commande **A 100** pour commencer à entrer un programme à la position **100** du segment définit dans **CS**.
Editez les lignes suivantes :

```
mov ax,a0a1 (CR)
mov bx,b0b1 (CR)
mov cx,c0c1 (CR)
mov dx,d0d1 (CR)
(CR)
```

d) Désassembler la mémoire

Vérifier que vos lignes sont bien en mémoire en tapant la commande **u 100**. DEBUG analyse le code machine *(40 octets)* qui se situe à partir de l'adresse **cs : 100** et déduit les instructions en langage source 8086.

e) Exécuter n lignes de programme

Pour exécuter **n** lignes de programme à partir de l'adresse **cs : 100**, il faut exécuter la commande **t=100 n**. Dans notre application, il y a 4 lignes donc taper **t=100 4**.
Observer l'évolution de l'exécution de votre programme et notamment l'évolution du contenu des registres ax, bx, cx et dx.
Que vaut **IP** à l'issue de l'exécution du programme ?
Conclusion sur le rôle du registre **IP**.

Remettre les registres **ax, bx, cx et dx** à **0000** et lancer votre programme en utilisant la commande **G=100 10C** *(signifie lancer le programme qui se trouve à l'adresse cs : 100 et placer un point d'arrêt à l'adresse cs:10C)*.

f) Scrutation de la mémoire

Le commande **D** permet de scruter en hexadécimal la mémoire. Exemple **D 100** permet de scruter 128 octets à partir de l'adresse **cs : 100**.
Vérifier que vos 4 lignes de programme sont toujours présentes à partir de l'adresse **cs : 100**. Lancer ensuite la commande **D 100**. En vous aidant de l'annexe A, montrer que les 12 premiers octets correspondent au code de vos lignes de programme. Conclusion concernant l'ordre de stockage des mots de 16 bits.

g) Exercices

1) Ecrire le programme suivant : *(l'instruction PUSH reg place dans la pile le registre reg. L'adresse de la pile est définie par le couple SS : SP)*

```
mov ax,35
mov bx,41
mov cx,30
mov dx,6
push ax
push bx
push cx
push dx
add ax,bx
add cx,dx
int $20
```

Exécuter le programme et vérifier son bon fonctionnement.
Retrouver les valeurs de **ax, bx, cx et dx** qui ont été sauvés dans la pile.
Conclusion concernant l'ordre de stockage dans la pile et l'ordre de stockage des parties hautes et basses des mots de 16 bits.

2) Soit le programme suivant :

```
mov al,17
mov cl,14
add cl,al
int $20
```

Transcrire le programme en hexadécimal en s'aidant de l'annexe A. Pour cela on utilisera la commande **E** qui permet d'éditer du code hexadécimal en mémoire. *(Exemple : E 100, modifier l'octet se trouvant en cs : 100 puis espace pour éditer l'octet suivant ou CR pour quitter la commande).*

3) L'interruption **21** permet dans une de ses variantes d'afficher un caractère à l'écran. Pour cela il faut mettre **02** dans **ah** et le code ASCII du caractère dans **dl**. Ensuite on appelle l'interruption par l'instruction **int $21**. Ecrire un programme qui écrit BONJOUR à l'écran. *(Le code ASCII du A est 41h)*.

4) L'interruption **21** permet quand **ah = 08** d'attendre le premier caractère tapé au clavier. Le caractère se trouve alors dans **al**. Ecrire un programme qui lit le premier caractère tapé au clavier et qui l'affiche à l'écran.

Table 2. Instruction Set Summary

Mnemonic and Description	Instruction Code			
DATA TRANSFER				
MOV = Move:	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**
Register/Memory to/from Register	1 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate to Register	1 0 1 1 w reg	data	data if w = 1	
Memory to Accumulator	1 0 1 0 0 0 0 w	addr-low	addr-high	
Accumulator to Memory	1 0 1 0 0 0 1 w	addr-low	addr-high	
Register/Memory to Segment Register	1 0 0 0 1 1 1 0	mod 0 reg r/m		
Segment Register to Register/Memory	1 0 0 0 1 1 0 0	mod 0 reg r/m		
PUSH = Push:				
Register/Memory	1 1 1 1 1 1 1 1	mod 1 1 0 r/m		
Register	0 1 0 1 0 reg			
Segment Register	0 0 0 reg 1 1 0			
POP = Pop:				
Register/Memory	1 0 0 0 1 1 1 1	mod 0 0 0 r/m		
Register	0 1 0 1 1 reg			
Segment Register	0 0 0 reg 1 1 1			
XCHG = Exchange:				
Register/Memory with Register	1 0 0 0 0 1 1 w	mod reg r/m		
Register with Accumulator	1 0 0 1 0 reg			
IN = Input from:				
Fixed Port	1 1 1 0 0 1 0 w	port		
Variable Port	1 1 1 0 1 1 0 w			
OUT = Output to:				
Fixed Port	1 1 1 0 0 1 1 w	port		
Variable Port	1 1 1 0 1 1 1 w			
XLAT = Translate Byte to AL	1 1 0 1 0 1 1 1			
LEA = Load EA to Register	1 0 0 0 1 1 0 1	mod reg r/m		
LDS = Load Pointer to DS	1 1 0 0 0 1 0 1	mod reg r/m		
LES = Load Pointer to ES	1 1 0 0 0 1 0 0	mod reg r/m		
LAHF = Load AH with Flags	1 0 0 1 1 1 1 1			
SAHF = Store AH into Flags	1 0 0 1 1 1 1 0			
PUSHF = Push Flags	1 0 0 1 1 1 0 0			
POPF = Pop Flags	1 0 0 1 1 1 0 1			

Table 2. Instruction Set Summary (Continued)

Mnemonic and Description	Instruction Code			
ARITHMETIC	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**
ADD = Add:				
Reg./Memory with Register to Either	0 0 0 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s: w = 01
Immediate to Accumulator	0 0 0 0 0 1 0 w	data	data if w = 1	
ADC = Add with Carry:				
Reg./Memory with Register to Either	0 0 0 1 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s: w = 01
Immediate to Accumulator	0 0 0 1 0 1 0 w	data	data if w = 1	
INC = Increment:				
Register/Memory	1 1 1 1 1 1 1 w	mod 0 0 0 r/m		
Register	0 1 0 0 0 reg			
AAA = ASCII Adjust for Add	0 0 1 1 0 1 1 1			
BAA = Decimal Adjust for Add	0 0 1 0 0 1 1 1			
SUB = Subtract:				
Reg./Memory and Register to Either	0 0 1 0 1 0 d w	mod reg r/m		
Immediate from Register/Memory	1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s w = 01
Immediate from Accumulator	0 0 1 0 1 1 0 w	data	data if w = 1	
SSB = Subtract with Borrow				
Reg./Memory and Register to Either	0 0 0 1 1 0 d w	mod reg r/m		
Immediate from Register/Memory	1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s w = 01
Immediate from Accumulator	0 0 0 1 1 1 w	data	data if w = 1	
DEC = Decrement:				
Register/memory	1 1 1 1 1 1 1 w	mod 0 0 1 r/m		
Register	0 1 0 0 1 reg			
NEG = Change sign	1 1 1 1 0 1 1 w	mod 0 1 1 r/m		
CMP = Compare:				
Register/Memory and Register	0 0 1 1 1 0 d w	mod reg r/m		
Immediate with Register/Memory	1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s w = 01
Immediate with Accumulator	0 0 1 1 1 1 0 w	data	data if w = 1	
AAS = ASCII Adjust for Subtract	0 0 1 1 1 1 1 1			
DAS = Decimal Adjust for Subtract	0 0 1 0 1 1 1 1			
MUL = Multiply (Unsigned)	1 1 1 1 0 1 1 w	mod 1 0 0 r/m		
IMUL = Integer Multiply (Signed)	1 1 1 1 0 1 1 w	mod 1 0 1 r/m		
AAM = ASCII Adjust for Multiply	1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0		
DIV = Divide (Unsigned)	1 1 1 1 0 1 1 w	mod 1 1 0 r/m		
IDIV = Integer Divide (Signed)	1 1 1 1 0 1 1 w	mod 1 1 1 r/m		
AAD = ASCII Adjust for Divide	1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0		
CBW = Convert Byte to Word	1 0 0 1 1 0 0 0			
CWD = Convert Word to Double Word	1 0 0 1 1 0 0 1			

Table 2. Instruction Set Summary (Continued)

Mnemonic and Description	Instruction Code			
LOGIC	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
NOT = Invert	1 1 1 1 0 1 1 w	mod 0 1 0 r/m		
SHL/SAL = Shift Logical/Arithmetic Left	1 1 0 1 0 0 v w	mod 1 0 0 r/m		
SHR = Shift Logical Right	1 1 0 1 0 0 v w	mod 1 0 1 r/m		
SAR = Shift Arithmetic Right	1 1 0 1 0 0 v w	mod 1 1 1 r/m		
ROL = Rotate Left	1 1 0 1 0 0 v w	mod 0 0 0 r/m		
ROR = Rotate Right	1 1 0 1 0 0 v w	mod 0 0 1 r/m		
RCL = Rotate Through Carry Flag Left	1 1 0 1 0 0 v w	mod 0 1 0 r/m		
RCR = Rotate Through Carry Right	1 1 0 1 0 0 v w	mod 0 1 1 r/m		
AND = **And:**				
Reg./Memory and Register to Either	0 0 1 0 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w = 1
Immediate to Accumulator	0 0 1 0 0 1 0 w	data	data if w = 1	
TEST = **And Function to Flags, No Result:**				
Register/Memory and Register	1 0 0 0 0 1 0 w	mod reg r/m		
Immediate Data and Register/Memory	1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w = 1
Immediate Data and Accumulator	1 0 1 0 1 0 0 w	data	data if w = 1	
OR = **Or:**				
Reg./Memory and Register to Either	0 0 0 0 1 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w = 1
Immediate to Accumulator	0 0 0 0 1 1 0 w	data	data if w = 1	
XOR = **Exclusive or:**				
Reg./Memory and Register to Either	0 0 1 1 0 0 d w	mod reg r/m		
Immediate to Register/Memory	1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w = 1
Immediate to Accumulator	0 0 1 1 0 1 0 w	data	data if w = 1	
STRING MANIPULATION				
REP = Repeat	1 1 1 1 0 0 1 z			
MOVS = Move Byte/Word	1 0 1 0 0 1 0 w			
CMPS = Compare Byte/Word	1 0 1 0 0 1 1 w			
SCAS = Scan Byte/Word	1 0 1 0 1 1 1 w			
LODS = Load Byte/Wd to AL/AX	1 0 1 0 1 1 0 w			
STOS = Stor Byte/Wd from AL/A	1 0 1 0 1 0 1 w			
CONTROL TRANSFER				
CALL = **Call:**				
Direct within Segment	1 1 1 0 1 0 0 0	disp-low	disp-high	
Indirect within Segment	1 1 1 1 1 1 1 1	mod 0 1 0 r/m		
Direct Intersegment	1 0 0 1 1 0 1 0	offset-low	offset-high	
		seg-low	seg-high	
Indirect Intersegment	1 1 1 1 1 1 1 1	mod 0 1 1 r/m		

Table 2. Instruction Set Summary (Continued)

Mnemonic and Description	Instruction Code		
JMP = Unconditional Jump:	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**	**7 6 5 4 3 2 1 0**
Direct within Segment	1 1 1 0 1 0 0 1	disp-low	disp-high
Direct within Segment-Short	1 1 1 0 1 0 1 1	disp	
Indirect within Segment	1 1 1 1 1 1 1 1	mod 1 0 0 r/m	
Direct Intersegment	1 1 1 0 1 0 1 0	offset-low	offset-high
		seg-low	seg-high
Indirect Intersegment	1 1 1 1 1 1 1 1	mod 1 0 1 r/m	
RET = Return from CALL:			
Within Segment	1 1 0 0 0 0 1 1		
Within Seg Adding Immed to SP	1 1 0 0 0 0 1 0	data-low	data-high
Intersegment	1 1 0 0 1 0 1 1		
Intersegment Adding Immediate to SP	1 1 0 0 1 0 1 0	data-low	data-high
JE/JZ = Jump on Equal/Zero	0 1 1 1 0 1 0 0	disp	
JL/JNGE = Jump on Less/Not Greater or Equal	0 1 1 1 1 1 0 0	disp	
JLE/JNG = Jump on Less or Equal/ Not Greater	0 1 1 1 1 1 1 0	disp	
JB/JNAE = Jump on Below/Not Above or Equal	0 1 1 1 0 0 1 0	disp	
JBE/JNA = Jump on Below or Equal/ Not Above	0 1 1 1 0 1 1 0	disp	
JP/JPE = Jump on Parity/Parity Even	0 1 1 1 1 0 1 0	disp	
JO = Jump on Overflow	0 1 1 1 0 0 0 0	disp	
JS = Jump on Sign	0 1 1 1 1 0 0 0	disp	
JNE/JNZ = Jump on Not Equal/Not Zero	0 1 1 1 0 1 0 1	disp	
JNL/JGE = Jump on Not Less/Greater or Equal	0 1 1 1 1 1 0 1	disp	
JNLE/JG = Jump on Not Less or Equal/ Greater	0 1 1 1 1 1 1 1	disp	
JNB/JAE = Jump on Not Below/Above or Equal	0 1 1 1 0 0 1 1	disp	
JNBE/JA = Jump on Not Below or Equal/Above	0 1 1 1 0 1 1 1	disp	
JNP/JPO = Jump on Not Par/Par Odd	0 1 1 1 1 0 1 1	disp	
JNO = Jump on Not Overflow	0 1 1 1 0 0 0 1	disp	
JNS = Jump on Not Sign	0 1 1 1 1 0 0 1	disp	
LOOP = Loop CX Times	1 1 1 0 0 0 1 0	disp	
LOOPZ/LOOPE = Loop While Zero/Equal	1 1 1 0 0 0 0 1	disp	
LOOPNZ/LOOPNE = Loop While Not Zero/Equal	1 1 1 0 0 0 0 0	disp	
JCXZ = Jump on CX Zero	1 1 1 0 0 0 1 1	disp	
INT = Interrupt			
Type Specified	1 1 0 0 1 1 0 1	type	
Type 3	1 1 0 0 1 1 0 0		
INTO = Interrupt on Overflow	1 1 0 0 1 1 1 0		
IRET = Interrupt Return	1 1 0 0 1 1 1 1		

Table 2. Instruction Set Summary (Continued)

Mnemonic and Description	Instruction Code	
	76543210	76543210
PROCESSOR CONTROL		
CLC = Clear Carry	11111000	
CMC = Complement Carry	11110101	
STC = Set Carry	11111001	
CLD = Clear Direction	11111100	
STD = Set Direction	11111101	
CLI = Clear Interrupt	11111010	
STI = Set Interrupt	11111011	
HLT = Halt	11110100	
WAIT = Wait	10011011	
ESC = Escape (to External Device)	11011xxx	mod x x x r/m
LOCK = Bus Lock Prefix	11110000	

NOTES:
AL = 8-bit accumulator
AX = 16-bit accumulator
CX = Count register
DS = Data segment
ES = Extra segment
Above/below refers to unsigned value
Greater = more positive;
Less = less positive (more negative) signed values
if d = 1 then "to" reg; if d = 0 then "from" reg
if w = 1 then word instruction; if w = 0 then byte instruction
if mod = 11 then r/m is treated as a REG field
if mod = 00 then DISP = 0*, disp-low and disp-high are absent
if mod = 01 then DISP = disp-low sign-extended to 16 bits, disp-high is absent
if mod = 10 then DISP = disp-high; disp-low
if r/m = 000 then EA = (BX) + (SI) + DISP
if r/m = 001 then EA = (BX) + (DI) + DISP
if r/m = 010 then EA = (BP) + (SI) + DISP
if r/m = 011 then EA = (BP) + (DI) + DISP
if r/m = 100 then EA = (SI) + DISP
if r/m = 101 then EA = (DI) + DISP
if r/m = 110 then EA = (BP) + DISP*
if r/m = 111 then EA = (BX) + DISP
DISP follows 2nd byte of instruction (before data if required)
*except if mod = 00 and r/m = 110 then EA = disp-high; disp-low.

if s w = 01 then 16 bits of immediate data form the operand
if s w = 11 then an immediate data byte is sign extended to form the 16-bit operand
if v = 0 then "count" = 1; if v = 1 then "count" in (CL)
x = don't care
z is used for string primitives for comparison with ZF FLAG

SEGMENT OVERRIDE PREFIX

0 0 1 reg 1 1 0

REG is assigned according to the following table:

16-Bit (w = 1)	8-Bit (w = 0)	Segment
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:
FLAGS = X:X:X:X:(OF):(DF):(IF):(TF):(SF):(ZF):X:(AF):X:(PF):X:(CF)

DATA SHEET REVISION REVIEW

The following list represents key differences between this and the -004 data sheet. Please review this summary carefully.

1. The Intel 8086 implementation technology (HMOS) has been changed to (HMOS-III).
2. Delete all "changes from 1985 Handbook Specification" sentences.

Les commandes de DEBUG.

Lancer debug.

Sous dos, on peut lancer Debug de plusieurs façons:

debug
Lance debug sans charger de programme; permet d'assembler une ligne pour avoir le code.
debug <programme avec extension>
Lance debug en chargeant un programme. C'est la forme la plus courante, qui permet de l'analyser. On peut lancer debug sans rien et charger un programme après coup, mais c'est beaucoup plus facile ainsi.
debug <programme avec extension> <paramètres>
Lance debug en chargeant un programme et en donnant des paramètres au programme. Si un programme se lance en faisant *gcc -g -c etape.c -o etape.o*, si on veut le débuggeur, on fait *debug gcc.exe -g -c etape.c -o etape.o* (notez l'extension qui est rajoutée).

On peut en plus rediriger l'entrée, c'est à dire donner les commandes dans un fichier plutôt que par le clavier. Mettre alors le symbole inférieur suivi du nom du fichier. debug < *<fichier>* lance debug en prenant les commandes dans un fichier. Attention ce fichier doit contenir une commande q, sinon vous ne pourrez plus sortir de debug. Exemple *debug < a.txt b.txt* ou *debug b.txt < a.txt* lance debug en chargeant la mémoire avec le programme ou le fichier *b.txt* et prend les commandes dans *a.txt*. Ceci permet d'assembler un programme, si on n'a pas compris que mon site vous en propose un mieux (et gratuit !).

On peut aussi rediriger la sortie, c'est à dire donner le résultat dans un fichier plutôt qu'à l'écran. Mettre alors le symbole supérieur suivi du nom du fichier. debug > *<fichier>* lance debug en envoyant les résultats dans un fichier. Exemple *debug > a.txt b.txt* ou *debug b.txt > a.txt* lance debug en chargeant la mémoire avec le programme ou le fichier *b.txt* et écrit les résultats dans *a.txt*. Ceci permet de désassembler dans un fichier (pour analyse). On n'a pas d'écho à l'écran, on a donc tout intérêt à appeler debug sans redirection, de tout noter, et de rediriger la sortie. On peut aussi utiliser un fichier d'entrée.

debug < a.txt > b.txt c.exe lance debug pour analyser le programme *c.exe*, en prenant les commandes dans *a.txt* et en mettant les sorties dans *b.txt*.

La commande de redirection de sortie efface le fichier si il existait. Pour écrire à la suite sans effacer le début, mettre deux symboles supérieurs au lieu d'un. Pour information, ceci n'est pas spécifique à debug, c'est du dos!

Sous windows, je conseille de faire un raccourci et de mettre derrière le nom du programme un point d'interrogation. Ainsi si vous cliquez sur le raccourci, windows demande ce qu'il doit mettre derrière debug.

Une fois lancé, debug nous donne un tiret d'invite. Les commandes ont toutes une lettre avec une suite de paramètres. Il n'y a pas de différences entre les majuscules et les minuscules. Toutes les valeurs numériques sont en hexadécimal.

Les paramètres des commandes.

<adresse>
L'adresse peut s'écrire sous un seul nombre *<offset>* ou sous un couple de nombres *:<offset>*. Il est possible de donner comme segment le mot DS ou CS. Si on ne précise pas de segment, c'est DS qui est pris par défaut, sauf pour les commandes Go, Load, Trace, Unassemble et Write (ce qui concerne le code en fait). Quelques adresses valides : 40:17
A000:0000 1000 CS:1000
<bloc>

C'est un ensemble d'adresses dans un même segment. Il y a deux formes :
<première adresse> <dernière adresse> ou *<première adresse> L <longueur en octet>*. La première adresse peut avoir un segment. La deuxième non.
<port>
C'est une adresse d'un port d'entrée sortie. Par exemple 378 pour les données de l'imprimante.
<octet>
C'est une donnée sur 1 ou 2 chiffres hexa.
<liste d'octets>
C'est une suite d'octets séparés par des espaces (ou des virgules).
<mot>
Nombre de 1 à 4 chiffres hexa.
<nom de fichier>
Donne au moins le nom et l'extension d'un fichier. On peut y mettre le chemin.
<chaîne>
Une chaîne est délimitée par des apostrophes ou des guillemets. Pour mettre un apostrophe dans une chaîne, utilisez les guillemets comme délimiteurs. Si vous voulez les deux types, doublez le délimiteur comme en C ou en Pascal.
<registre> Nom d'un registre.
<disque>
0 représente le lecteur A:, 1 le lecteur B:, 2 le disque C: et ainsi de suite.
<secteur>
Si on a *nb_tetes* têtes et *nb_secteurs* secteurs par cylindres, pour le secteur *sect* du cylindre *cyln* de la tête *tet* porte le numéro *sect+nb_secteur*(tet+nb_tetes*cyln)*. Le premier secteur porte le numéro 0.
<nombre de secteurs>
Nombre entre 1 et 80 (on ne peut pas lire ou écrire en une fois plus de 80 secteurs. Je rappelle que les secteurs sont de 512 octets.

Commande A (assemble).

Syntaxe: *A <adresse>*

debug reçoit ligne par ligne les instructions à assembler. On finit en donnant un retour chariot supplémentaire. On ne peut pas donner d'étiquettes.

MOVSB s'applique à des octets, *MOVSW* à de mots.
RET est un retour court, *RETF* un long
BYTE PTR et *WORD PTR* permettent de spécifier la taille dans certains cas : *INC BYTE PTR [100]*
DB et *DW* sont supportées.

Commande C (compare).

Syntaxe *C <bloc> <adresse>*

Compare le bloc défini par <bloc> avec le bloc qui débute à <adresse> (de la même taille). Il n'y a pas de message si ils sont identiques.

Exemple *C 100 110 200* ou *C 100 L10 200* compare les blocs (CS : 100...CS:110) avec (CS:200..CS:210).

Commande D (dump soit vider).

Syntaxe *D* ou *D <adresse>* ou *D <bloc>*

Donne la suite d'octets du bloc défini. Si on spécifie une adresse seule, le bloc fera 128 octets. D sans paramètres "dump" les 128 octets suivants.

Commande E (enter).

Syntaxe *E <adresse>* ou *E <adresse> <liste d'octets>*

Si on donne une liste d'octets, ils sont mis en mémoire à partir de <adresse>. Sinon, on peut donner les différents octets (l'ancien est affiché). Une valeur donnée remplace l'ancienne, un espace conserve l'ancienne valeur, retour chariot termine la commande et le symbole moins permet de revenir en arrière.

Commande F (fill).

Syntaxe *F <bloc> <liste d'octets>*

Remplit le bloc avec la suite d'octets. La suite est répétée pour tout remplir.

Exemple: *F 100 200 1 2 3* donne *100: 1 2 3 1 2 3 1 2 3 1 2 3*... jusqu'en 200 inclus.

Commande G (go).

Syntaxe *G [=<adresse1>] [<adresse2>...]*

Exécute le programme à partir de l'adresse 1 si elle est spécifiée (elle est facultative) ou à partir de CS : IP si elle n'est pas indiquée. Il y a des points d'arrêt aux adresses <adresses2> (on peut mettre entre = et 10 points d'arrêt qui doivent être le premier octet d'une instruction).

Exemples :
G -> lance à partir de CS:IP
G = 100 -> lance à partir de CS:100
G 100 -> lance à partir de CS:IP et s'arrête si on tombe sur l'adresse 100
G = 100 200 300 400 -> lance à partir de 100 et s'arrête si on passe par 200 300 ou 400.

Commande H (hexa).

Syntaxe : *H <mot1> <mot2>*

Affiche *<mot1>+<mot2>* et *<mot1>-<mot2>*

Commande I (input).

Syntaxe *I <port>*

Affiche le contenu de l'adresse d'entrée sortie *<port>*

Commande L (load).

Syntaxe *L* ou *L <adresse>* ou *L <adresse> <disque> <secteur> <nombre de secteurs>*

L utilisée seul charge le programme (on peut donner le nom par la commande N). Le programme est chargé à partir de <adresse> si elle est donnée (ignorée pour les .exe). par défaut, un .com est chargé en 100h. BX:CX donnent la taille du fichier chargé. Si on donne le disque, le secteur et le nombre de secteur, on charge alors des secteurs d'un disque.

Commande M (move).

Syntaxe *M <bloc> <adresse>*

Déplace le bloc à partir de l'adresse spécifiée.

Commande N (name).

Syntaxe *N <nom de fichier>*

La dernière commande entrée donne le nom du fichier qui peut être lue ou écrite par les commandes L ou W. La commande N permet aussi de donner les paramètres, mais je déconseille son usage.

Commande O (output).

Syntaxe *O <port> <octet>*

Écrit à l'adresse entrée sortie spécifiée un octet.

Commande P (proceed).

Syntaxe *P [=<adresse>] [<mot>]*

Exécute <mot> instructions (1 si mot n'est pas spécifié), en commençant à <adresse> ou en CS:IP si <adresse> n'y est pas. Cette commande ressemble à T (trace), sauf qu'elle considère INT CALL et les LOOP comme si c'était une seule instruction. Ceci permet de "passer" sur les INT 21.

Commande Q (quit).

Syntaxe *Q*

Termine la cession.

Commande R (registre).

Syntaxe *R [<registre>]*

Affiche et permet de modifier le contenu d'un registre ou affiche tous les registres si aucun nom n'est spécifié. Le registre des flags s'appelle F. Si on donne RF, l'état des flags s'affiche et on peut changer les flags individuellement : OV, DN, EI, NG, ZR, AC, PE et CY permettent de mettre à un les flags Overflow, Direction, Interruption, Signe, Zéro, Retenue auxiliaire, Parité et Retenue. NV, UP, DI, PL, NZ, NA, PO et NC permettent de les remettre à zéro.

Commande S (search).

Syntaxe *S <bloc> <liste d'octets>*

Recherche les occurrences de la liste d'octets dans le bloc. Aucun message n'est affiché si debug ne trouve rien.

Commande T (trace).

Syntaxe *T [=<adresse>] [<mot>]*

Exécute <mot> instructions (1 si mot n'est pas spécifié), en commençant à <adresse> ou en CS : IP si <adresse> n'y est pas. Cette commande ressemble à P (proceed), sauf qu'elle entre dans les INT CALL et traite les LOOP en plusieurs fois. Ceci permet de tracer l'intérieur des fonctions.

Commande U (unassemble).

Syntaxe *U ou U <adresse> ou U <bloc>*

Désassemble un bloc. Si on spécifie une adresse seule, le bloc fera 20 octets. Si on ne spécifie rien, on désassemble les 20 octets suivants.

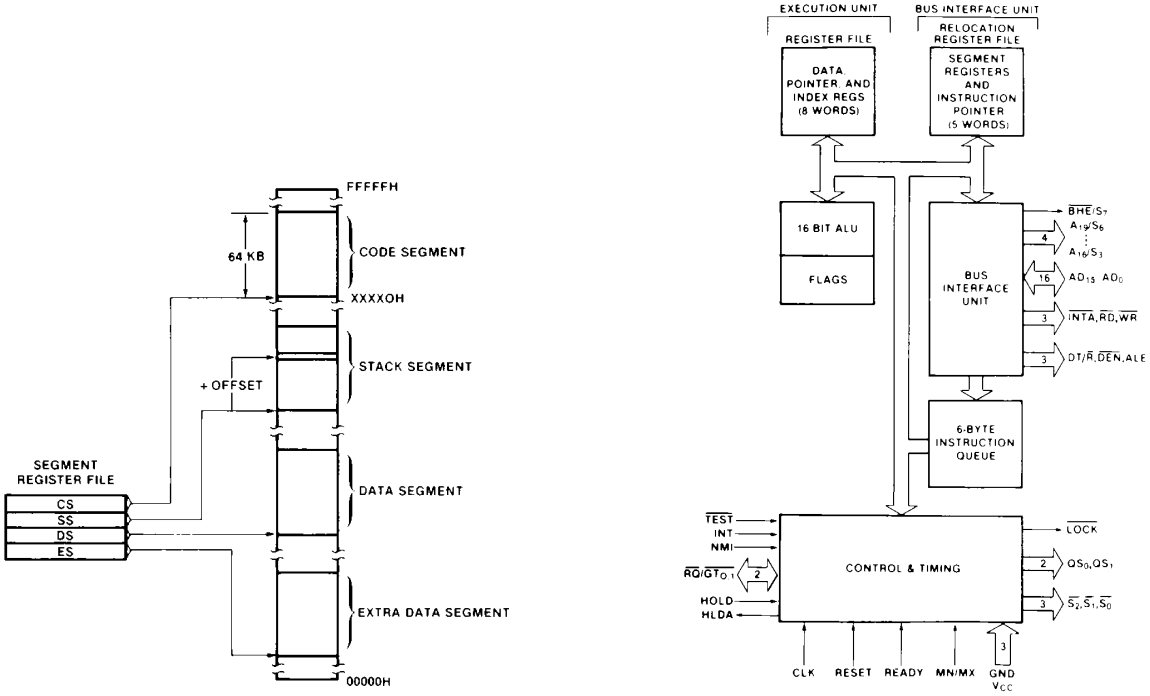
Commande W (write).

Syntaxe *W* ou *W <adresse>* ou *W <adresse> <disque> <secteur> <nombre de secteurs>*

Sauve sur disque BX:CX octets à partir de <adresse> si elle est donnée ou à partir de 100 sinon. Le nom du programme est celui qui à été chargé ou le dernier nom donné par la commande N. Si on donne le disque, le secteur et le nombre de secteur, on écrit alors des secteurs sur un disque. Je ne réponds de rien dans ce cas, c'est une commande très dangereuse sur disque dur.

Olivier HABERT
février 2001

MicroInformatique
Informatique Industrielle
Assembleur 8086/8088
TP n°2



I. BUT DU TP

Se familiariser avec assembleur freeware **NASM** associé à l'éditeur syntaxique **NASMIDE**.

II. Présentation de NASM

a) *Qu'est ce que NASM ?*

NASM (http://cryogen.com/nasm) est un assembleur 80x86 libre d'utilisation. NASMIDE (http://www.inglenook.co.uk/rob/index.html) est un environnement de développement spécialement développé pour travailler avec NASM sous DOS. *(Il existe également une version de NASMIDE sous Windows permettant de développer des programmes en assembleur 32 bits).*

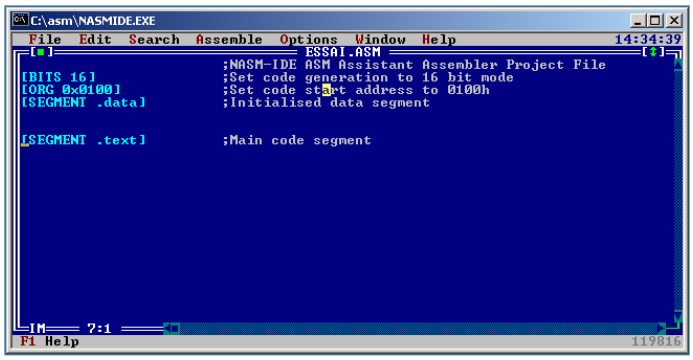
b) *Structure d'un programme assembleur*

Un programme en assembleur commence en général par la définition des données puis ensuite par le code.

Il s'agit tout d'abord de préciser à l'assembleur si on va écrire du code 16 bits ou 32 bits. Dans notre cas, nous générerons du code 16 bits en précisant la directive **[BITS 16]**. Il faut ensuite préciser l'adresse du programme dans le segment de code en utilisant la directive **[ORG 0x100**]. *(0x100 signifie que 100 est un nombre en hexadécimal)*

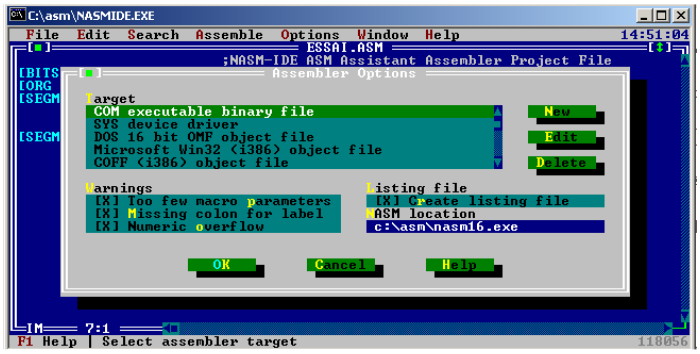
Ensuite, il faut préciser par **[SEGMENT .data]** que l'on va définir des variables et constantes dans le segment de données.

La directive **[SEGMENT .text]** permet ensuite d'introduire les instructions du programme en langage assembleur.

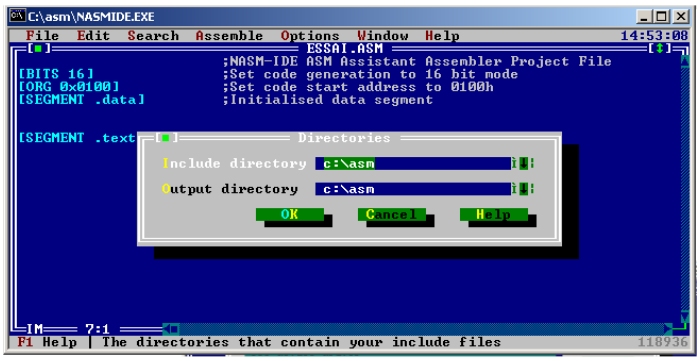


c) Edition d'un programme

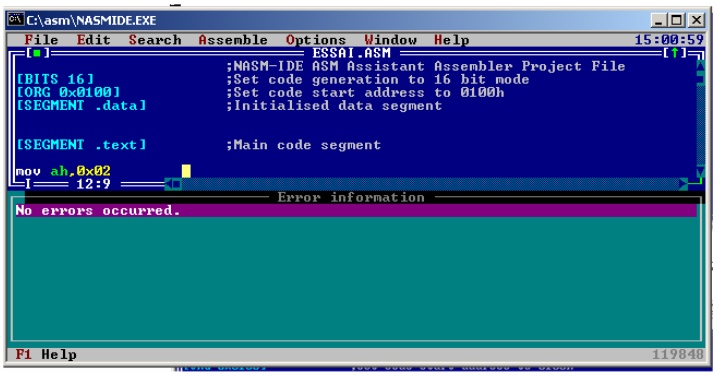
- Vérifier que les options suivantes sont configurées dans le menu ***options|assembler***



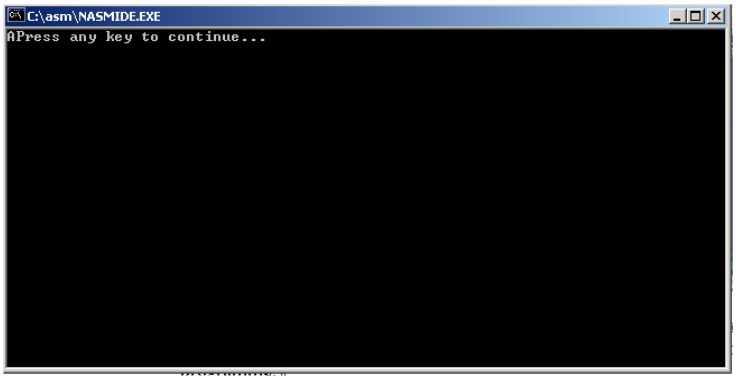
- Vérifier que les options suivantes sont configurées dans le menu ***options|directories***



- Editer le programme qui affiche un caractère **A** à l'écran *(le même celui du TP 1). (Attention, int 21 s'écrit maintenant int 0x21 ou int $21)*
- Appuyer sur la touche **F2** pour sauver le programme.
- Appuyer simultanément sur les touches **alt + F9** pour assembler le programme.
- Une fenêtre d'erreur doit normalement indiquer qu'aucune erreur n'est survenue.



- Appuyer simultanément sur les touches **Ctrl + F9** pour exécuter le programme.



En résumé, attention au format des constantes numériques :

100	→	décimal
100h	→	hexadécimal
0x100	→	hexadécimal aussi
$100	→	hexadécimal également mais le premier caractère doit être un chiffre
10000100b	→	binaire

d) Déclaration de variables dans le segment de données

Il est possible de définir des variables de type caractère, chaîne de caractères, entier, entier long, réel,.. grâce à des pseudo instruction.

Déclaration de variables entières de type octet: db

```
nb          db      $10
tab         db      1, 2 ,3 ,4 ,5 ,6 ,7 ,8 ,10
message     db      'bonjour les IUP GSI TI',13,10,'$'
```

(13 → retour chariot, 10 → ligne suivante, $ → fin de chaîne de caractères)

Déclaration de variables entières de type mot: dd

```
R       dd      1024
```

Déclaration de variables en virgule flottante: dd, dq, dt

```
A       dd      1.2
B       dd      1.e10
C       dd      1.e+10
D       dq      1.e-10
pi      dt      3.141592353589793238462
```

e) Utilisation des variables dans le programme

```
                r       db      40
Exemple:                mov al, [r]
                        mov [r], cl
```

☝ **: Il est impossible d'affecter une variable directement à une autre variable, il faut obligatoirement passer par un registre.**
Il est également impossible d'affecter une constante directement à une variable, il faut également passer par un registre.

f) Définition d'étiquettes

Contrairement à **DEBUG**, il est possible de définir des étiquettes vers lesquelles certaines instructions pourraient se brancher. Ceci évite le comptage laborieux des octets lorsqu'il s'agit de réaliser un saut dans le programme.

Une étiquette de déclare par un nom de taille inférieure à 8 caractères et se terminant par :

Exemple:

```
BOUCLE:   ADD AX,CX     ; Ax = Ax + Cx
          .
          .
          .
          JNE BOUCLE    ; jump not equal
```

III. Exercices

Exercice 1:

1) Faire l'organigramme du programme qui calcule la somme des 11 premiers entiers *(0 + 1 + 2 +...+ 10 +11)*. Ecrire ensuite le programme en assembleur 8086. On utilisera pour cela les instructions MOV, CMP, JNE, ADD, DEC ou INC… On utilisera une variable **R** pour stocker le résultat et une variable **N** pour stocker le nombre 11.

2) Afficher ensuite le résultat à l'écran en utilisant l'interruption **$21** puis déduire en fonction du caractère affiché la valeur numérique du résultat.

3) Même exercice mais en utilisant l'instruction **LOOP**

Exercice 2:

- Utilisation de l'interruption $21 en mode 9

L'interruption **$21** avec **ah** à 9 permet d'afficher des chaînes de caractères qui se terminent par le caractère 13, 10 ,'$'. *(13 → retour chariot, 10 → ligne suivante, $ → fin de chaîne de caractères)*

Pour cela, il faut définir un message avec la pseudo instruction **db** *(voir explications de db)* en terminant ce message par 13, 10, '$'.

Il faut ensuite affecter au registre **dx** l'adresse de ce message *(mov dx, mess où mess est le nom donné au message)*.

Ecrire un programme qui affiche dix fois à l'écran le message suivant avec retour à la ligne et saut de ligne :

Si je travaille bien en TP, je deviendrai un dieu de l'assembleur et le prof sera fier de moi

Exercice 3:

Concevoir une application qui réalise le produit de deux nombres **A = 100** et **B = 170**. Les nombres **A** et **B** sont codés sur 8 bits chacun.

- Quelle instruction faut il utiliser ?
- Où se trouvent les opérandes ?
- Où se trouve le résultat de l'opération.
- Afficher les octets du registre contenant le résultat de l'opération.
- Déduire le résultat numérique à partir des caractères affichés.

Mêmes questions mais avec des nombres A et B de 16 bits. Prendre **A = 1642** et **B = 10**.
Essayer avec **A=3227** et **B=5**.
Comment apparaît le résultat ?
Comment faire pour que des caractères imprimables apparaissent tout le temps ?

ASCII Table (7-bit)

Decimal	Octal	Hex	Binaire	Valeur
000	000	000	00000000	NUL (Null char.)
001	001	001	00000001	SOH (Start of Header)
002	002	002	00000010	STX (Start of Text)
003	003	003	00000011	ETX (End of Text)
004	004	004	00000100	EOT (End of Transmission)
005	005	005	00000101	ENQ (Enquiry)
006	006	006	00000110	ACK (Acknowledgment)
007	007	007	00000111	BEL (Bell)
008	010	008	00001000	BS (Backspace)
009	011	009	00001001	HT (Horizontal Tab)
010	012	00A	00001010	LF (Line Feed)
011	013	00B	00001011	VT (Vertical Tab)
012	014	00C	00001100	FF (Form Feed)
013	015	00D	00001101	CR (Carriage Return)
014	016	00E	00001110	SO (Serial In)(Shift Out)
015	017	00F	00001111	SI (Serial Out)(Shift Out)
016	020	010	00010000	DLE (Data Link Escape)
017	021	011	00010001	DC1 (XON) (Device Control 1)
018	022	012	00010010	DC2 (Device Control 2)
019	023	013	00010011	DC3 (XOFF)(Device Control 3)
020	024	014	00010100	DC4 (Device Control 4)
021	025	015	00010101	NAK (Negative Acknowledgement)
022	026	016	00010110	SYN (Synchronous Idle)
023	027	017	00010111	ETB (End of Trans. Block)
024	030	018	00011000	CAN (Cancel)
025	031	019	00011001	EM (End of Medium)
026	032	01A	00011010	SUB (Substitute)
027	033	01B	00011011	ESC (Escape)
028	034	01C	00011100	FS (File Separator)
029	035	01D	00011101	GS (Group Separator)
030	036	01E	00011110	RS (Request to Send)
031	037	01F	00011111	US (Unit Separator)
032	040	020	00100000	SP (Space)
033	041	021	00100001	!
034	042	022	00100010	"
035	043	023	00100011	#
036	044	024	00100100	$
037	045	025	00100101	%
038	046	026	00100110	&
039	047	027	00100111	'
040	050	028	00101000	(
041	051	029	00101001	)
042	052	02A	00101010	*
043	053	02B	00101011	+
044	054	02C	00101100	,
045	055	02D	00101101	-
046	056	02E	00101110	.
047	057	02F	00101111	/
048	060	030	00110000	0
049	061	031	00110001	1
050	062	032	00110010	2
051	063	033	00110011	3
052	064	034	00110100	4
053	065	035	00110101	5
054	066	036	00110110	6
055	067	037	00110111	7
056	070	038	00111000	8
057	071	039	00111001	9
058	072	03A	00111010	:
059	073	03B	00111011	;
060	074	03C	00111100	<
061	075	03D	00111101	=
062	076	03E	00111110	>
063	077	03F	00111111	?
064	100	040	01000000	@
065	101	041	01000001	A
066	102	042	01000010	B
067	103	043	01000011	C
068	104	044	01000100	D
069	105	045	01000101	E
070	106	046	01000110	F
071	107	047	01000111	G
072	110	048	01001000	H
073	111	049	01001001	I
074	112	04A	01001010	J
075	113	04B	01001011	K
076	114	04C	01001100	L
077	115	04D	01001101	M
078	116	04E	01001110	N
079	117	04F	01001111	O
080	120	050	01010000	P
081	121	051	01010001	Q
082	122	052	01010010	R
083	123	053	01010011	S
084	124	054	01010100	T
085	125	055	01010101	U
086	126	056	01010110	V
087	127	057	01011111	W
088	130	058	01011000	X
089	131	059	01011001	Y
090	132	05A	01011010	Z
091	133	05B	01011011	[
092	134	05C	01011100	\
093	135	05D	01011101	]
094	136	05E	01011110	^
095	137	05F	01011111	_
096	140	060	01100000	`
097	141	061	01100001	a
098	142	062	01100010	b
099	143	063	01100011	c
100	144	064	01100100	d
101	145	065	01100101	e
102	146	066	01100110	f
103	147	067	01100111	g
104	150	068	01101000	h
105	151	069	01101001	i
106	152	06A	01101010	j
107	153	06B	01101011	k
108	154	06C	01101100	l
109	155	06D	01101101	m
110	156	06E	01101110	n
111	157	06F	01101111	o
112	160	070	01110000	p
113	161	071	01110001	q
114	162	072	01110010	r
115	163	073	01110011	s
116	164	074	01110100	t
117	165	075	01110101	u
118	166	076	01110110	v
119	167	077	01110111	w
120	170	078	01111000	x
121	171	079	01111001	y
122	172	07A	01111010	z
123	173	07B	01111011	{
124	174	07C	01111100	|
125	175	07D	01111101	}
126	176	07E	01111110	~
127	177	07F	01111111	DEL

Olivier HABERT
février 2001

MicroInformatique
Informatique Industrielle
Assembleur 8086/8088
TP n°3

I. BUT DU TP

Maîtriser les différents types d'adressage, notamment pour accéder aux éléments de tableaux.

II. RAPPELS

Pour adresser les différents éléments d'un tableau, il faut utiliser l'adressage indexé basé en prenant le registre **BX** comme base et le registre **DI** ou **SI** comme indexe. Pour mettre la base du tableau *(déplacement du tableau dans le segment de données)* dans **BX**, il suffit d'écrire **MOV BX, nom du tableau.**

Ensuite l'adressage indexée basée s'utilise de la manière suivante :

MOV AL, [BX+SI] ; mettre le SI ème élément du tableau dans le registre AL

où **SI** représente la position de l'élément du tableau que l'on veut atteindre.

III. EXERCICES

Exercice 1:

Soit le tableau suivant:

tab db 5, 30, 20, 1,11, 8

Faire le somme des 6 éléments de ce tableau et mettre le résultat dans une variable **S**. Afficher le résultat à l'écran, en déduire la valeur numérique.

Exercice 2:

Ecrire une application qui compte le nombre de lettres **'e'** dans une phrase quelconque *(50 caractères minimum)* se terminant par le caractère **'$'**.

- Représenter l'organigramme.
- Mettre le résultat dans le registre **dl** et ajouter **32**.
- Imprimer le caractère à l'écran et vérifier si le caractère qui apparaît est cohérent.

Exercice 3:

Définir une chaîne de caractères de longueur au moins 50 caractères se terminant par '$'.

Réaliser une application *(organigramme + programme)* qui remplace toutes les lettres d'un type donné par un autre type *(par exemple tous les 'e' par des 'a')*.

On entrera les deux lettres au clavier en utilisant l'interruption 21,8.

Les saisies clavier seront précédées d'un message du type **'Remplacer :'** et **'par :'**

Exercice 4:

Tri d'un tableau

Soit le tableau **tab** suivant : **5, 8, 1, 7, 6**
Soit l'algorithme suivant :

```
i←0
j←1
n←5
tant que i < n-1
        tant que j <n
                Si tab[i]>tab[j] alors Permuter(tab,i,j)
                j = j +1
        fin tant que
        i = i+1
        j = i+1
fin tant que
```

Appliquer l'algorithme suivant au tableau **tab** en remplissant le tableau d'état suivant:

Etape	i	j	tab[0]	tab[1]	tab[2]	tab[3]	tab[4]

Soit la chaîne de caractères suivante :

message db "le but de ce probleme est de trier par ordre croissant de code ASCII cette chaîne de caractere"

- Ecrire le programme qui tri par ordre croissant de code ASCII la chaîne précédente.
- On prendra SI pour l'index i et DI pour l'index j.
- On affichera le résultat grâce à l'interruption $21, 9.

Olivier HABERT
février 2001

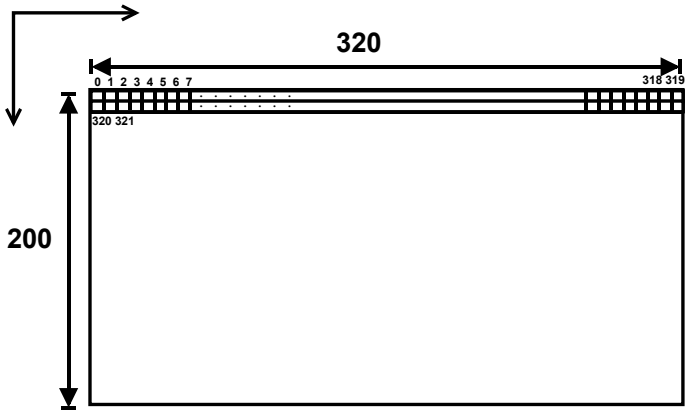
MicroInformatique
Informatique Industrielle
Assembleur 8086/8088
TP n°4

I. But du TP

Programmer l'affichage en mode Graphique **MCGA**.

II. Présentation du mode MCGA

Le mode MCGA est un mode graphique créé par IBM. Sa particularité est de posséder **256** couleurs et une définition de **320 pixels** par 200 pixels. De ce fait, chaque pixel est représenté par un octet. La mémoire écran est donc représentée par **320*200*1** octets contigus (64000).



III. Accès au mode MCGA

Pour accéder au mode **MCGA**, nous allons utiliser les interruptions logicielles du **BIOS**. L'interruption du **BIOS** est l'interruption **0x10** et la fonction qui permet de choisir son mode vidéo est la fonction **0**. Dans cette fonction il faut préciser le mode vidéo que l'on désire. Le mode correspondant au **MCGA** est le mode **0x13**.
En résumé pour activer le mode **MCGA** il faut :

- **ah← 0**
- **al ← 0x13**
- **Appel de l'interruption 0x10**

Pour se remettre en mode texte à la fin du programme, on utilisera le mode 0x03.

- **ah← 0**
- **al ← 0x03**
- **Appel de l'interruption 0x10**

IV. Accès aux pixels

Pour accéder à un pixel, il faut écrire dans la mémoire vidéo du mode **MCGA**. Cette mémoire vidéo commence à l'adresse **0xa000** et s'étend sur **64000** octets.
Pour accéder à un pixel de coordonnées **x, y,** $\mathbf{x \in [0,319]}$ **et** $\mathbf{y \in [0,199]}$ il faut donc accéder à la mémoire vidéo d'adresse **y*320+x**.
La couleur de ce pixel dépendra de l'octet que l'on écrira à l'adresse du pixel. La couleur associée à une valeur de l'octet est définie dans une palette qu'il est également possible de modifier. Mais ceci est une autre histoire …

Pour écrire à l'adresse **0xa000** + **déplacement**, il faut utiliser l'extra segment **es** associé à un déplacement représenté par **bx**. On peut éventuellement ajouter à tout cela un déplacement fixe ou variable avec les registres **si** et **di**.

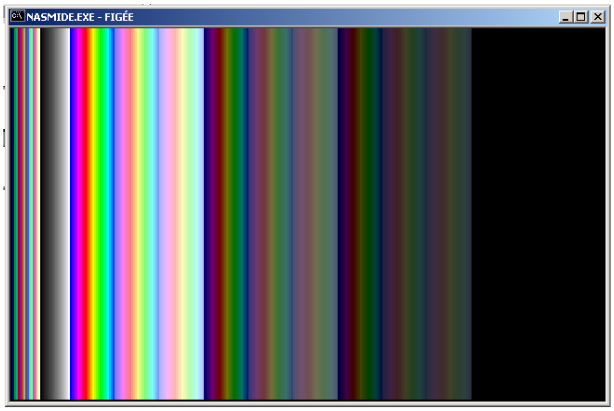
Exemple:

```
mov ax,0xa000
mov es,ax
mov dl, 0xaa
mov bx, 160
mov [es:bx], dl              ; on écrit à l'adresse 0xa000 + 160

ou
mov bx, 160
mov [es:bx + 320], dl        ; on écrit à l'adresse 0xa000 + 160 +320

ou


mov bx, 160
mov si, 320
mov [es:bx + si], dl         ; on écrit à l'adresse 0xa000 + 160 +320

ou

mov bx, 160
mov si, 320
mov [es:bx + si + 640], dl   ; on écrit à l'adresse 0xa000 + 160 +320 +640
```

V. Exercices

Exercice 1:

- Allumer le pixel de coordonnées **(200,100)** avec la couleur de code **0x32**.
- Allumer les **256** premiers pixels leur associant une couleur variant de **0** à **255**.
- Même question mais en augmentant l'épaisseur du trait *(5 lignes)*.
- Même question mais sur toute la hauteur de l'écran *(cf figure suivante)*.

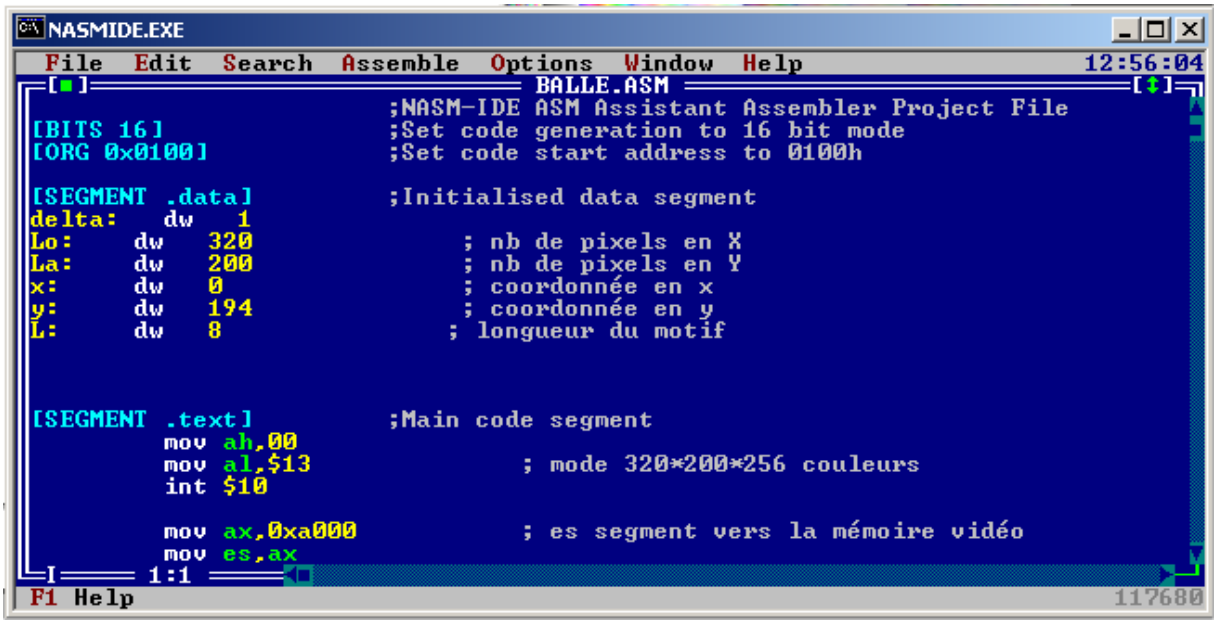


Exercice 2:

- Encadrer l'écran avec un cadre d'épaisseur **5** pixels de la couleur de votre choix
- Quadriller l'écran avec une ligne de 5 pixels d'épaisseur et un espacement de 10 pixels entre les lignes.

Exercice 3: *(pour les meilleur(e)s)*

- Ecrire un programme qui déplace un motif rectangulaire **(8*6 pixels)** en bas de la fenêtre de gauche →droite, de droite → gauche, … tant qu'aucune touche n'est appuyée.
- L'affichage et l'effacement devront se faire avec un ou exclusif pour ne pas effacer ce qui se trouve éventuellement déjà sur l'écran.
- Le test de l'appui d'une touche se fera en utilisant l'interruption **0x16** *(voir annexe)*
- Les variables à utiliser seront les suivantes :



- entre chaque affichage, il est nécessaire *(au moins sur les Pentiums)* de générer une temporisation. Pour cela on écrira un sous-programme de temporisation simple du type :

```
tempo:    mov cx,0xffff
tt:       nop
          loop tt
          ret
```

- Pour déplacer le motif vers la gauche ou vers la droite, on utilisera une variable delta qui prendra la valeur **1** ou **–1** selon le sens du déplacement. Cette variable servira d'incrément pour afficher la position suivante du motif.
- Il faudra tester si on arrive au niveau de la limite supérieure droite **(320-L)** pour donner à delta la valeur **–1** et tester la limite inférieure gauche **(0)** pour mettre delta à **+1**.

Exercice 4: *(pour les meilleur(e)s des meilleur(e)s*

- Ecrire un Casse brique en assembleur **8086** *(raquette + balle + briques)*

INT 16,1 - Get Keyboard Status

AH = 01
on return:
ZF = 0 if a key pressed (even Ctrl-Break)
AX = 0 if no scan code is available
AH = ~scan code~
AL = ASCII character or zero if special function key
- data code is not removed from buffer
- ~Ctrl-Break~ places a zero word in the keyboard buffer but does register a keypress.

Olivier HABERT
février 2001

MicroInformatique
Informatique Industrielle
Assembleur 8086/8088
TP n°5

I. But du TP

Se familiariser avec les instructions répétitives, les horloges temps réel, la génération de nombres pseudo aléatoires, …

(Ce TP a été largement inspiré de ce que l'on trouve sur le site http://www.multimania.com/abcp/ad/index.html#depart)

II. Le hasard en informatique

C'est simple, le hasard n'existe pas en informatique. Les suites de nombres "au hasard" en informatique sont presque toutes basées sur un principe : On utilise un nombre générateur, on lui fait subir une opération *(par exemple on multiplie les poids faibles par les poids forts et on inverse un bit sur deux)* qui nous donne un nouveau générateur. On extrait tout ou une partie des bits de ce générateur pour avoir le nombre au hasard.

Dans les langages évolués, il existe une fonction qui donne un nombre et une fonction qui fixe le premier générateur. Cela signifie que si vous dessinez "au hasard" en prenant deux fois le même générateur, vous allez obtenir deux fois exactement le même dessin. Pour palier à ceci, on choisit en principe pour le premier générateur l'heure du système.

Une "bonne" série aléatoire doit être telle que chacun des nombres générés doivent être équiprobables *(sur un temps de mesure suffisamment long, on doit trouver autant de 1 que de 2, que de 3...)*. D'autre part on voudrait que la suite soit aussi irrégulière que possible

Si on utilise l'horloge du PC, on va avoir des valeurs qui vont changer avec le temps, d'une façon relativement régulière, mais dans la plupart des cas cela suffit. Sur un PC, le plus simple est de récupérer le temps depuis que l'on a allumé l'ordinateur. Les premiers PC avaient une horloge à 4,77 Mhz, derrière on divisait par 4 puis par 65536 *(compteur sur 16 bits)* en on obtenait une fréquence de 4770000/(4*65536) = 18,2 Hz.

On alertait alors ainsi 18,2 fois par seconde le PC qui remettait à l'heure son compteur de temps. Lorsque l'on a changé les vitesses des horloges des PC, on a gardé cette fréquence pour l'horloge.

Ceci signifie que si vous changez la fréquence de votre ordinateur, le comptage du temps va rester le même.

III. Application à l'affichage aléatoire de points de couleurs différentes

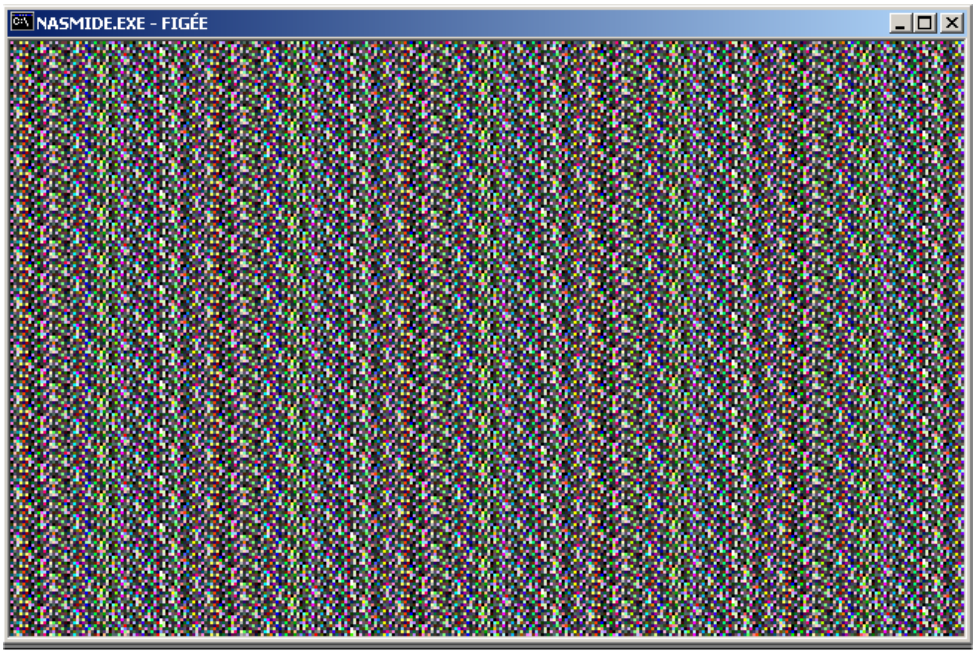
Nous proposons dans cet exercice d'afficher en permanence les points de l'écran avec une couleur pseudo aléatoire. Pour cela nous allons utiliser le compteur temps réel du PC qui se trouve à l'adresse **0040h : 006Ch**.

Ecrire un programme qui affiche allume dans le désordre les points de l'écran avec une couleur que l'on incrémentera au fur et à mesure.

Pour cela il faut suivre les étapes suivantes :

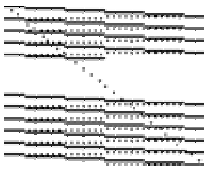
- Se mettre en mode graphique **MCGA.**
- Initialiser le segment **DS** avec **0xA000** *(passer par AX).*
- Initialiser le segment **ES** avec **0x0040** *(passer par AX).*
- **BL** contiendra la couleur du point à afficher, mettre **BL** à 0.
- Afficher le premier point en utilisant la notation **MOV [DS : SI], BL.**
- Ajouter à SI l'état du compteur temps réel **ADD SI,[ES:0x6C].**
- Insérer l'ensemble dans une boucle dont on sortira dés que l'on appui sur une touche du clavier *(interruption 16,1).*

- Revenir en mode Texte à la fin du programme *(voir TP 4)*.



IV. Synchronisation du dessin des points avec le rafraîchissement de l'affichage

De manière simpliste, l'affichage est réalisé par le balayage quasi horizontal de 3 spots *(voir figure suivante)*



Quand les spots arrivent en bas de l'écran, ils reviennent en haut et la balayage recommence. Pendant la remontée, les spots sont invisibles.

Nous allons dans cet exercice synchroniser l'affichage des points avec la remontée des spots. Pour cela nous allons détecter grâce à la lecture d'un port d'entrée sortie le moment ou les spots commencent leur remontée. Pendant cette remontée, nous allons effectuer l'affichage.

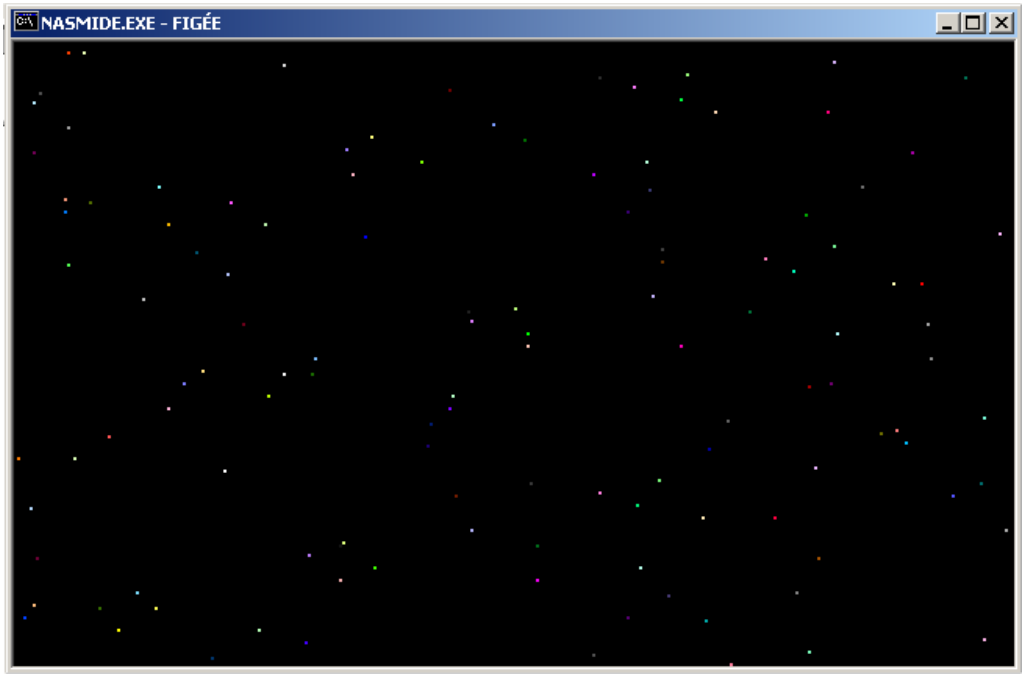
Cette information de trouve au niveaux du port E/S d'adresse 3DAh. Quand on lit ce port, le bit n°3 *(quatrième bit)* renseigne sur l'état des spots. Quand ce bit est à 0, on est entrain de dessiner *(balayage vers le bas des spots)* et quand ce bit est à 1, les spots sont entrain de remonter.

Le principe est d'attendre tant que le bit n°3 est à 1 puis d'attendre tant qu'il est à 0. Ainsi on sera certain que l'on va commencer l'affichage au début de la remontée des spots. Dans les animations graphiques, cette méthode permet d'éviter le phénomène d'instabilité au niveau de l'affichage.

Ajouter dans le programme précédent le code qui synchronise le dessin sur le rafraîchissement de l'écran.

Pour cela on suivra les étapes suivantes :

- Lire le port **3DAh** avec l'instruction IN AL, DX *(dx contient l'adresse du port)*.
- Faire un masque avec la valeur 08 *(utiliser l'instruction AND AL,08)*.
- Faire le saut tant que le résultat n'est pas nul.
- Recommencer une seconde boucle qui dure maintenant tant que le résultat est nul.



V. Utilisation des instructions de chaînes pour translater l'écran

Les instructions de chaînes:

MOVSB, MOVSW et MOVSD

Formes.

MOVSB
MOVSW
MOVSD
REP MOVSB
REP MOVSW
REP MOVSD

Commentaires.

Ces instructions permettent de déplacer un octet *(suffixe B comme Byte)*, un mot *(suffixe W comme Word)* ou un double mot à partir du processeur 386 *(suffixe D comme DWord)*.

En mode 16 bits, l'octet *(mot ou double mot)* d'origine est pris en [DS:SI], d'où le nom SI de Source Index. Le segment DS est implicite et il est possible d'en choisir un autre avec un préfixe de segment. La destination est [ES:DI], d'où le nom DI de destination index. Il n'est par contre pas possible de changer le segment ES.

En mode 32 bits, les index ESI et EDI sont utilisés en place de SI et DI.

Ces instructions permettent de faire des transferts simples ou multiples. On parle alors de chaîne de caractère, d'où le nom de l'instruction MOVE *(déplacer)* S *(string=chaîne)*.

Pour être prêt à faire le transfert suivant, les index sont incrémentés si l'indicateur de direction est à 0, et décrémentés si il est à 1. Pour mettre l'indicateur de direction à 0 on utilise l'instruction CLD *(clear direction, clear = effacer)*. Avec MOVSB, SI et DI seront incrémentés de 1. Avec MOVSW, on vient de transférer deux octets, il faut donc pointer sur le word suivant qui est deux octets plus loin. SI et DI seront donc incrémentés de 2. Avec MOVSD, les index sont incrémentés de 4. On peut aussi mettre l'indicateur de direction à 1 par l'instruction STD (set D, set se traduit par mettre), ainsi les index seront décrémentés de 1, 2 ou 4.

Si on utilise le préfixe REP devant, l'instruction est répétée CX fois en mode 16 bits *(ECX fois en mode 32 bits)*. En fait REP va décrémenter CX et faire l'instruction tant que CX n'est pas nul. Si on met 10 dans CX avant l'instruction MOVSB, on copiera 10 octets. MOVSW copiera donc 20 octets *(10 fois 2 octets)* et MOVSD copiera 40 octets. Si CX est nul, la première décrémentation mettra CX à 65535, et on fait donc 65536 copies.

Utilisation

On utilise rarement cette instruction en dehors d'une boucle. Elle est utilisée en principe pour copier une zone de mémoire dans une autre. REP MOVSD est plus rapide pour copier un nombre d'octets fixe, on la préférera si le nombre d'octet est connu ou est divisible par 4. On utilise MOVSW si le processeur peut être un 8086 ou un 80286. MOVSB ne devrait que très peu être employée.

Ces instructions sont très utiles pour déplacer des tableaux, des chaînes de caractères, des buffers, des zones d'écran...

Si les zones ne se recouvrent pas, on peut utiliser l'incrémentation ou la décrémentation des index. Bien sûr les valeurs de départ ne seront pas les mêmes dans les deux cas. Assurez vous aussi à positionner correctement l'indicateur de direction. Si les zones source et destination se recouvrent, on n'a pas le choix du sens de la copie. Pour déplacer par exemple 100h octets à partir de ES : 0000 vers ES : 0001, on est obligé de commencer par ES : 0100 et il faudra décrémenter les index. Dans le cas contraire, l'octet ES : 0000 serait mis en ES : 0001 qui serait de nouveau copié en ES : 0002 et ainsi de suite. Si c'est réellement ce que vous voulez faire, utilisez plutôt l'instruction STOSB.

Un exemple.

On vient de dessiner en mémoire une belle page que l'on veut afficher en mode 13h. La page dessinée commence en DS : 0000.

```
    xor  si,si ; DS:SI pointe sur l'image en mémoire

    xor  di,di ; ES:DI pointe sur l'écran (A000:0000)
    mov  ax,0A000h
    mov  es,ax

    cld        ; Index croissants

    mov  cx,65536/2 ; 64 ko à copier 2 octets par 2 octets

rep  movsw  ; fait réelement la copie
```

Le programme à réaliser

Dans l'étape précédente, nous avions un ciel noir dans lequel apparaissait progressivement des étoiles colorées. Nous allons monter dans une fusée et nous allons voir les étoiles défiler. Cela va donner :

Pour faire ce programme, nous allons répéter quatre choses *(une de plus que la dernière fois) :*

- Dessiner un point "au hasard"
- Attendre le retour du spot
- Décaler l'écran graphique d'une ligne vers le bas *(pour avoir l'impression de monter)*
- Tester si un caractère est arrivé

Décaler l'écran.

C'est la seule chose qui est nouvelle dans cette étape. Nous sommes en mode 13h et l'écran visible à 200 lignes. Quand on écrit un point au hasard, on l'écrit dans le segment entier. L'écran dans lequel on dessine dépasse un peu de quelques lignes. Nous allons utiliser la ligne 200, qui est invisible, car seules les lignes 0 à 199 sont affichées. Nous allons dérouler l'écran comme du papier à musique : pour faire descendre l'écran, il suffit de prendre toutes les lignes et de les redessiner une ligne plus bas. Pour avoir de plus en plus d'étoile comme dans l'étape précédente, nous allons mettre la dernière ligne à la place de la première. Pour décaler nos 200 lignes nous allons en utiliser 201 (une ligne sert de mémoire). Nous allons faire cette rotation en deux temps.

Premier temps : copiez les lignes 0 à 199 dans les lignes 1 à 200. En assembleur pour faire un tel travail, il n'y a pas vraiment le choix, il faut utiliser l'instruction MOVSW *(pour recopier les octets 2 par 2)*.

Deuxième temps : copier la ligne 200 *(l'ancienne 199)* dans la ligne 0.

Premier temps (0..199) → (1..200)

Les zones source et destination se recouvrent. Il va donc falloir réfléchir: si nous copions d'abord la ligne 0 dans la ligne 1, nous ne pourrons plus copier le ligne 1 dans la ligne 2 car la ligne 1 aura été écrasée et contiendrait la ligne 0. Il va donc falloir copier à l'envers. Mettons la ligne 199 dans la 200, la 198 dans la 199 et ainsi de suite.

Pour positionner les index, il faut mettre dans SI l'offset du dernier ensemble de 2 points de l'écran que l'on va copier *(vu que nous commençons par copier par la fin*). Chaque ligne contient 320 points, soit 320 octets. Après notre dernier ensemble de 2 points, nous allons trouver les points de la ligne 200, et donc d'offset 200*320. Le dernier couple copié est donc en 200*320-2. Nous écrirons simplement :

```
mov si,200*320-2
```

Pour le calcul de di, c'est exactement une ligne au dessous soit 320 octets plus loin.

```
mov di,200*320-2+320
```

Ne faites pas le calcul pour le programme, on sait ainsi comment cette valeur est obtenue.

L'initialisation des segments peut se faire au début du programme une fois pour toutes. Profitons de l'initialisation de l'un pour initialiser l'autre. Écrivons quelque chose du style :

```
mov ax,0A000h
```

```
mov ds,ax
mov es,ax
```

Pour déplacer 200 lignes, soit 200*320 octets, il faut initialiser CX à 200*320/2 vu que nous copions les octets 2 par 2.

Enfin pensons à mettre l'indicateur de direction à 1, pour décrémenter *(aller à l'envers)*. Quand tout cela est fait, appelons l'instruction MOVSW avec son préfixe REP pour faire tout le transfert.

Deuxième temps (200) → (0).

Ici les lignes ne se recouvrent pas, on peut donc les copier dans le sens que l'on veut. Comme nous avons copié le reste en décrémentant les indices, il y a deux solutions : soit on met l'indicateur à 1 une fois pour toutes dans les initialisations et on décrémente aussi pour cette partie, soit on positionne l'indicateur à chaque fois pour chacune des deux parties et on peut utiliser l'incrémentation. Pour une fois, c'est vous qui choisissez. On aura de toutes façon l'occasion d'utiliser cette deuxième démarche dans une étape ultérieure.

Ce que l'on voit à l'écran.

Normalement si tout se passe bien, l'écran va se remplir progressivement de points tout en descendant. Tant que les points sont peu nombreux, tout est sympathique, mais quand les points seront suffisamment denses, vous apercevrez comme des barres horizontales au milieu de l'écran. Cela est dû en partie au fait que l'écran est dessiné dans le mauvais sens *(bas vers le haut, contraire au balayage)*. Il y a des moyens de faire mieux, mais ce qui nous intéresse actuellement est la découverte des méthodes de programmation et les instructions, avec une progression pas trop rapide.

Toutes les commandes du 8086

Index :

Types d'opérande :

REG: AX, BX, CX, DX, AH, AL, BL, BH, CH, CL, DH, DL, DI, SI, BP, SP.

SREG: DS, ES, SS et uniquement comme deuxième opérande : CS.

mémoire: [BX], [BX+SI+7], variable, etc. (Voir **Accès mémoire**.)

valeur immédiate: 5, -24, 3Fh, 10001101b, etc.

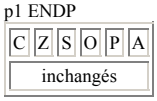
Instructions par ordre alphabétique :

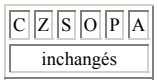
Instruction	Opérandes	Description
AAA	Pas d'opérande	Ajustement ASCII après Addition. Corrige le résultat dans AL et AH après addition lors d'une opération en valeurs BCD. Cela fonctionne selon l'algorithme suivant : Si nibble (quartet ou demi-octet) de poids faible dans AL > 9 ou AF = 1 alors : • AL = AL + 6 • AH = AH + 1 • AF = 1 • CF = 1 sinon • AF = 0 • CF = 0 dans les deux cas : effacer le nibble de poids fort dans AL. Exemple : MOV AX, 15 ; AH = 00, AL = 0Fh AAA ; AH = 01, AL = 05 RET C Z S O P A r ? ? ? ? r
AAD	Pas d'opérande	Ajustement ASCII avant Division. Prépare les deux valeurs BCD pour la division. Algorithme : • AL = (AH * 10) + AL • AH = 0 Exemple : MOV AX, 0105h ; AH = 01, AL = 05 AAD ; AH = 00, AL = 0Fh (15) RET C Z S O P A ? r r ? r ?
AAM	Pas d'opérande	Ajustement ASCII après Multiplication. Corrige le résultat de la multiplication dans les deux valeurs BCD. Algorithme : • AH = AL / 10 • AL = reste Exemple : MOV AL, 15 ; AL = 0Fh AAM ; AH = 01, AL = 05 RET C Z S O P A ? r r ? r ?
AAS	Pas d'opérande	Ajustement ASCII après Soustraction. Corrige le résultat dans AL et AH après soustraction lors d'une opération en valeurs BCD.

		Algorithme : Si nibble (quartet ou demi-octet) de poids faible dans AL > 9 ou AF = 1 alors : • AL = AL - 6 • AH = AH - 1 • AF = 1 • CF = 1 sinon • AF = 0 • CF = 0 dans les deux cas : effacer le nibble de poids fort dans AL. Exemple : MOV AX, 02FFh ; AH = 02, AL = 0FFh AAS ; AH = 01, AL = 09 RET C Z S O P A r ? ? ? ? r
ADC	REG, mémoire mémoire, REG REG, REG mémoire, immédiate REG, immédiate	Addition avec retenue. Algorithme : opérande1 = opérande1 + opérande2 + CF Exemple: STC ; place CF à 1 MOV AL, 5 ; AL = 5 ADC AL, 1 ; AL = 7 RET C Z S O P A r r r r r r

ADD	REG, mémoire
	mémoire, REG
	REG, REG
	mémoire, immédiate
	REG, immédiate
		Addition.
		
		Algorithme :
		
		opérande1 = opérande1 + opérande2
		
		Exemple:
		MOV AL, 5 ; AL = 5
		ADD AL, -3 ; AL = 2
		RET
		C Z S O P A
		r r r r r r

AND	REG, mémoire mémoire, REG REG, REG mémoire, immédiate REG, immédiate	AND (ou ET) logique entre tous les bits des deux opérandes. Le résultat est stocké dans opérande1. Table de vérité : 1 AND 1 = 1 1 AND 0 = 0 0 AND 1 = 0 0 AND 0 = 0 Exemple : MOV AL, 'a' ; AL = 01100001b AND AL, 11011111b ; AL = 01000001b ('A') RET C Z S O P 0 r r 0 r

CALL	nom de la procédure
	label
	adresse de 4 octets
		Appel de procédure, l'adresse de retour dans (IP) est placée sur la pile. L'*adresse de 4 octets* peut s'écrire sous la forme suivante : **1234h:5678h**, la première valeur est le segment et la deuxième est l'offset (c'est un appel long, CS est également placé sur la pile).
		
		Exemple :
		#make_COM#
		ORG 100h ; pour un fichier COM.
		
		CALL p1
		
		ADD AX, 1
		
		RET ; retour au système d'exploitation.
		
		p1 PROC ; déclaration de la procédure.
		MOV AX, 1234h
		RET ; retour à l'appel.
		p1 ENDP
		 

CBW	Pas d'opérande	Conversion d'octet en mot (word).
		
		Algorithme :
		
		Si le bit de poids fort (bit de signe) dans AL = 1 alors :
		
		• AH = 255 (0FFh)
		
		sinon
		
		• AH = 0
		
		Exemple:
		MOV AX, 0 ; AH = 0, AL = 0
		MOV AL, -5 ; AX = 000FBh (251)
		CBW ; AX = 0FFFBh (-5)
		RET
		 

CLC	Pas d'opérande	Mise à zéro du flag de retenue. Algorithme : CF = 0 
CLD	Pas d'opérande	Mise à zéro du flag de direction. Les registres SI et DI seront incrémentés par les instructions pour chaînes suivantes : CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithme : DF = 0 
CLI	Pas d'opérande	Mise à zéro du flag d'interruption. Désactive des interruptions matérielles. Algorithme : IF = 0

		I 0
CMC	Pas d'opérande	Complémente le flag de retenue. Inverse la valeur du flag CF. Algorithme : si CF = 1 alors CF = 0 si CF = 0 alors CF = 1 C r

CMP	REG, mémoire
	mémoire, REG
	REG, REG
	mémoire, immédiate
	REG, immédiate
		Compare deux opérandes.
		
		Algorithme :
		
		opérande1 - opérande2
		
		Le résultat n'est pas stocké, les flags (OF, SF, ZF, AF, PF, CF) sont positionnés selon le résultat.
		
		Exemple :
		MOV AL, 5
		MOV BL, 5
		CMP AL, BL ; AL = 5, ZF = 1 (égalité !)
		RET
		C Z S O P A
		r r r r r r

CMPSB	Pas d'opérande	Compare les octets de : ES:[DI] et DS:[SI]. Algorithme : • DS:[SI] - ES:[DI] • Positionne les flags : OF, SF, ZF, AF, PF, CF selon le résultat. • si DF = 0 alors o SI = SI + 1 o DI = DI + 1 sinon o SI = SI - 1 o DI = DI - 1 Exemple : Voir **cmpsb.asm** dans le dossier " Samples". C Z S O P A r r r r r r
CMPSW	Pas d'opérande	Compare les mots : ES:[DI] et DS:[SI]. Algorithme : • DS:[SI] - ES:[DI] • Positionne les flags : OF, SF, ZF, AF, PF, CF selon le résultat. • si DF = 0 alors o SI = SI + 2 o DI = DI + 2

		sinon o SI = SI - 2 o DI = DI - 2 Exemple : Voir **cmpsw.asm** dans le dossier "Samples". C Z S O P A r r r r r r

CWD	Pas d'opérande	Conversion de mot en double mot.
		
		Algorithme :
		
		si le bit de poids fort dans AX = 1 alors :
		• DX = 65535 (0FFFFh)
		
		sinon
		• DX = 0
		
		Exemple :
		MOV DX, 0 ; DX = 0
		MOV AX, 0 ; AX = 0
		MOV AX, -5 ; DX AX = 00000h:0FFFBh
		CWD ; DX AX = 0FFFFh:0FFFBh
		RET
		C Z S O P A
		inchangés

DAA	Pas d'opérande	Ajustement décimal après Addition. Corrige le résultat de l'addition des deux valeurs BCD compactées. Algorithme : si le demi-octet de poids faible dans AL > 9 ou AF = 1 alors : • AL = AL + 6 • AF = 1 si AL > 9Fh ou CF = 1 alors : • AL = AL + 60h • CF = 1 Exemple : MOV AL, 0Fh ; AL = 0Fh (15) DAA ; AL = 15h RET C Z S O P A r r r r r r
DAS	Pas d'opérande	Ajustement décimal après Soustraction. Corrige le résultat de la soustraction des deux valeurs BCD compactées. Algorithme : si le demi-octet de poids faible dans AL > 9 ou AF = 1 alors : • AL = AL - 6

		• AF = 1 si AL > 9Fh ou CF = 1 alors : • AL = AL - 60h • CF = 1 Exemple : MOV AL, 0FFh ; AL = 0FFh (-1) DAS ; AL = 99h, CF = 1 RET C Z S O P A r r r r r r
DEC	REG mémoire	Décrément. Algorithme : opérande = opérande - 1 Exemple : MOV AL, 255 ; AL = 0FFh (255 or -1) DEC AL ; AL = 0FEh (254 or -2) RET Z S O P A r r r r r CF - inchangé !

DIV	REG
	mémoire
		Division non signée.
		
		Algorithme :
		
		lorsque l'opérande est un **octet** :
		AL = AX / opérande
		AH = reste (coefficient)
		
		lorsque l'opérande est un **mot** :
		AX = (DX AX) / opérande
		DX = reste (coefficient)
		
		Exemple :
		MOV AX, 203 ; AX = 00CBh
		MOV BL, 4
		DIV BL ; AL = 50 (32h), AH = 3
		RET
		C Z S O P A
		? ? ? ? ? ?

HLT	Pas d'opérande	Suspend l'exécution et place le 8086 dans l'état HALT. Exemple : MOV AX, 5 HLT C Z S O P A inchangés
IDIV	REG mémoire	Division signée. Algorithme : lorsque l'opérande est un **octet** : AL = AX / opérande AH = reste (coefficient) lorsque l'opérande est un **mot** :

		DX = reste (coefficient)
		
		Exemple :
		MOV AX, -203 ; AX = 0FF35h
		MOV BL, 4
		IDIV BL ; AL = -50 (0CEh), AH = -3 (0FDh)
		RET
		C Z S O P A
		? ? ? ? ? ?

IMUL	REG
	mémoire
		Multiplication signée.
		
		Algorithme :
		
		orsque l'opérande est un **octet** :
		AX = AL * opérande.
		
		lorsque l'opérande est un **mot** :
		(DX AX) = AX * opérande.
		
		Exemple :
		MOV AL, -2
		MOV BL, -4
		IMUL BL ; AX = 8
		RET
		C Z S O P A
		r ? ? r ? ?
		CF=OF=0 lors des ajustements du résultat dans l'opérande par IMUL.

IN	AL, im.octet AL, DX AX, im.octet AX, DX	Transfère la donnée lue sur le port dans **AL** ou **AX**. Le second opérande est le numéro de port. Il est possible d'accéder à n'importe quel port compris entre 0 et 255. Le numéro de port se place dans le registre **DX**. Exemple : IN AX, 4 ; obtenir le statut des feux de circulation. IN AL, 7 ; obtenir le statut du moteur pas à pas. C Z S O P A inchangés
INC	REG mémoire	Incrément. Algorithme : opérande = opérande + 1 Exemple : MOV AL, 4 INC AL ; AL = 5 RET Z S O P A r r r r r CF - inchangé !

INT	octet immédiat	Appel d'une procédure d'interruption.
		Numéro d'interruption par un octet immédiat compris entre 0 et 255.
		
		Algorithme :
		
		Sauvegarde sur la pile :
		o registres de flag
		o CS
		o IP
		• IF = 0
		• Appel de la procédure d'interruption
		
		Exemple :
		MOV AH, 0Eh ; télétype.
		MOV AL, 'A'

		INT 10h ; interruption du BIOS. RET C Z S O P A I inchangés 0

INTO	Pas d'opérande	Appel d'une procédure d'interruption.
		Interruption 4 si le flag de débordement est 1.
		
		Algorithme :
		
		si OF = 1 alors INT 4
		
		Exemple :
		; -5 - 127 = -132 (n'est pas entre -128 et 127)
		; le résultat de SUB (124) est faux,
		; donc OF est placé à 1 :
		MOV AL, -5
		SUB AL, 127 ; AL = 7Ch (124)
		INTO ; erreur de procédure.
		RET

IRET	Pas d'opérande	Retour d'interruption. Algorithme : Restaure depuis la pile : o IP o CS o registres de flag C Z S O P A restaurés

JA	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est supérieur au deuxième opérande (identique à l'instruction CMP). Non signé.
		
		Algorithme :
		
		si (CF = 0) et (ZF = 0) alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 250
		CMP AL, 5
		JA label1
		PRINT 'AL n'est pas supérieur à 5'
		JMP exit
		label1:
		PRINT 'AL est supérieur à 5'
		exit:
		RET
		C Z S O P A
		inchangés

JAE	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est supérieur ou égal au deuxième opérande (identique à l'instruction CMP). Non signé.
		
		Algorithme :
		
		si CF = 0 alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 5
		CMP AL, 5
		JAE label1
		PRINT 'AL n'est pas supérieur ou égal à 5'

		JMP exit label1: PRINT 'AL est supérieur ou égal à 5' exit: RET C Z S O P A inchangés

JB	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est inférieur au deuxième opérande (identique à l'instruction CMP). Non signé.
		
		Algorithme :
		
		si CF = 1 alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 1
		CMP AL, 5
		JB label1
		PRINT 'AL n'est pas inférieur à 5'
		JMP exit
		label1:
		PRINT 'AL est inférieur à 5'
		exit:
		RET
		C Z S O P A
		inchangés

JBE	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est inférieur ou égal au deuxième opérande (identique à l'instruction CMP). Non signé.
		
		Algorithme :
		
		si CF = 1 ou ZF = 1 alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 5
		CMP AL, 5
		JBE label1
		PRINT 'AL n'est pas inférieur ou égal à 5'
		JMP exit
		label1:
		PRINT 'AL est inférieur ou égal à 5'
		exit:
		RET
		C Z S O P A
		inchangés

JC	label	Saut si la condition est réalisée.
		Saut court si le flag de retenue (carry) est à 1.
		
		Algorithme :
		
		si CF = 1 alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 255
		ADD AL, 1
		JC label1
		PRINT 'pas de retenue.'
		JMP exit
		label1:
		PRINT 'il y a retenue.'
		exit:
		RET
		C Z S O P A
		inchangés

JCXZ	label	Saut si la condition est réalisée.
		Saut court si le registre CX vaut 0.
		
		Algorithme :
		
		si CX = 0 alors saut
		Exemple :
		`include 'emu8086.inc'`
		`#make_COM#`
		`ORG 100h`
		`MOV CX, 0`
		`JCXZ label1`
		`PRINT 'CX ne vaut pas 0.'`
		`JMP exit`
		`label1:`
		`PRINT 'CX vaut zéro.'`
		`exit:`
		`RET`
		C Z S O P A
		inchangés

JE	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est égal au deuxième opérande (identique à l'instruction CMP). Signé et Non signé.
		
		Algorithme :
		
		si ZF = 1 alors saut
		Exemple :
		`include 'emu8086.inc'`
		`#make_COM#`
		`ORG 100h`
		`MOV AL, 5`
		`CMP AL, 5`
		`JE label1`
		`PRINT 'AL n'est pas égal à 5.'`
		`JMP exit`
		`label1:`
		`PRINT 'AL est égal à 5.'`
		`exit:`
		`RET`
		C Z S O P A
		inchangés

JG	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est supérieur au deuxième opérande (identique à l'instruction CMP). Signé.
		
		Algorithme :
		
		si (ZF = 0) et (SF = OF) alors saut
		Exemple :
		`include 'emu8086.inc'`
		`#make_COM#`
		`ORG 100h`
		`MOV AL, 5`
		`CMP AL, -5`
		`JG label1`
		`PRINT 'AL n'est pas supérieur à -5.'`
		`JMP exit`
		`label1:`
		`PRINT 'AL est supérieur à -5.'`
		`exit:`
		`RET`
		C Z S O P A
		inchangés

JGE	label	Saut si la condition est réalisée. Saut court si le premier opérande est supérieur ou égal au deuxième opérande (identique à l'instruction CMP). Signé. Algorithme :

		si SF = OF alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 2
		CMP AL, -5
		JGE label1
		PRINT 'AL < -5'
		JMP exit
		label1:
		PRINT 'AL >= -5'
		exit:
		RET
		C Z S O P A
		inchangés

JL	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est inférieur au deuxième opérande (identique à l'instruction CMP). Signé.
		
		Algorithme :
		
		si SF <> OF alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, -2
		CMP AL, 5
		JL label1
		PRINT 'AL >= 5.'
		JMP exit
		label1:
		PRINT 'AL < 5.'
		exit:
		RET
		C Z S O P A
		inchangés

JLE	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est inférieur ou égal au deuxième opérande (identique à l'instruction CMP). Signé.
		
		Algorithme :
		
		si SF <> OF ou ZF = 1 alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, -2
		CMP AL, 5
		JLE label1
		PRINT 'AL > 5.'
		JMP exit
		label1:
		PRINT 'AL <= 5.'
		exit:
		RET
		C Z S O P A
		inchangés

JMP	label adresse sur 4 octets	Saut inconditionnel. Transfère le contrôle à une autre partie du programme. L'adresse sur 4 octets s'écrit sous la forme suivante : **1234h:5678h**, la première valeur est la valeur de segment et la deuxième est la valeur d'offset. Algorithme : saut dans tous les cas Exemple : include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 5

		JMP label1 ; saut au-dessus de 2 lignes ! PRINT 'Pas de saut !' MOV AL, 0 label1: PRINT 'Arrivé ici !' RET C Z S O P A inchangés

JNA	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est non supérieur au deuxième opérande (identique à l'instruction CMP). Non signé.
		
		Algorithme :
		
		si CF = 1 ou ZF = 1 alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 2
		CMP AL, 5
		JNA label1
		PRINT 'AL est supérieur à 5.'
		JMP exit
		label1:
		PRINT 'AL est non supérieur à 5.'
		exit:
		RET
		C Z S O P A
		inchangés

JNAE	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est non supérieur ou non égal au deuxième opérande (identique à l'instruction CMP). Non signé.
		
		Algorithme :
		
		si CF = 1 alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 2
		CMP AL, 5
		JNAE label1
		PRINT 'AL >= 5.'
		JMP exit
		label1:
		PRINT 'AL < 5.'
		exit:
		RET
		C Z S O P A
		inchangés

JNB	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est non inférieur au deuxième opérande (identique à l'instruction CMP). Non signé.
		
		Algorithme :
		
		si CF = 0 alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 7
		CMP AL, 5
		JNB label1
		PRINT 'AL < 5.'
		JMP exit
		label1:
		PRINT 'AL >= 5.'
		exit:
		RET

		inchangés

JNBE	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est non inférieur et non égal au deuxième opérande (identique à l'instruction CMP). Non signé.
		
		Algorithme :
		
		si (CF = 0) et (ZF = 0) alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 7
		CMP AL, 5
		JNBE label1
		PRINT 'AL <= 5.'
		JMP exit
		label1:
		PRINT 'AL > 5.'
		exit:
		RET
		C Z S O P A
		inchangés

JNC	label	Saut si la condition est réalisée.
		Saut court si le flag de retenue (carry) est à 0.
		
		Algorithme :
		
		si CF = 0 alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 2
		ADD AL, 3
		JNC label1
		PRINT 'il y a retenue.'
		JMP exit
		label1:
		PRINT 'pas de retenue.'
		exit:
		RET
		C Z S O P A
		inchangés

JNE	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est non égal au deuxième opérande (identique à l'instruction CMP). Signé et non signé.
		
		Algorithme :
		
		si ZF = 0 alors saut
		Exemple :
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, 2
		CMP AL, 3
		JNE label1
		PRINT 'AL = 3.'
		JMP exit
		label1:
		PRINT 'Al <> 3.'
		exit:
		RET
		C Z S O P A
		inchangés

JNG	label	Saut si la condition est réalisée. Saut court si le premier opérande est non supérieur au deuxième opérande (identique à

		Algorithme :
		
		si (ZF = 1) et (SF <> OF) alors saut
		Exemple :
		`include 'emu8086.inc'`
		`#make_COM#`
		`ORG 100h`
		`MOV AL, 2`
		`CMP AL, 3`
		`JNG label1`
		`PRINT 'AL > 3.'`
		`JMP exit`
		`label1:`
		`PRINT 'Al <= 3.'`
		`exit:`
		`RET`
		C Z S O P A
		inchangés

JNGE	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est non supérieur et non égal au deuxième opérande (identique à l'instruction CMP). Signé.
		
		Algorithme :
		
		si SF <> OF alors saut
		Exemple :
		`include 'emu8086.inc'`
		`#make_COM#`
		`ORG 100h`
		`MOV AL, 2`
		`CMP AL, 3`
		`JNGE label1`
		`PRINT 'AL >= 3.'`
		`JMP exit`
		`label1:`
		`PRINT 'Al < 3.'`
		`exit:`
		`RET`
		C Z S O P A
		inchangés

JNL	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est non inférieur au deuxième opérande (identique à l'instruction CMP). Signé.
		
		Algorithme :
		
		si SF = OF alors saut
		Exemple :
		`include 'emu8086.inc'`
		`#make_COM#`
		`ORG 100h`
		`MOV AL, 2`
		`CMP AL, -3`
		`JNL label1`
		`PRINT 'AL < -3.'`
		`JMP exit`
		`label1:`
		`PRINT 'Al >= -3.'`
		`exit:`
		`RET`
		C Z S O P A
		inchangés

JNLE	label	Saut si la condition est réalisée.
		Saut court si le premier opérande est non inférieur et non égal au deuxième opérande (identique à l'instruction CMP). Signé.
		
		Algorithme :
		
		si (SF = OF) et (ZF = 0) alors saut
		Exemple :
		`include 'emu8086.inc'`
		`#make_COM#`

		ORG 100h
		MOV AL, 2
		CMP AL, -3
		JNLE label1
		PRINT 'AL <= -3.'
		JMP exit
		label1:
		PRINT 'Al > -3.'
		exit:
		RET
		C Z S O P A
		inchangés

JNO	label	Saut si la condition est réalisée.
		Saut court si pas de débordement.
		
		Algorithme :
		
		si OF = 0 alors saut
		Exemple :
		; -5 - 2 = -7 (se trouve entre -128 et 127)
		; le résultat de l'instruction SUB est correct,
		; donc OF = 0:
		
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AL, -5
		SUB AL, 2 ; AL = 0F9h (-7)
		JNO label1
		PRINT 'Débordement !'
		JMP exit
		label1:
		PRINT 'Pas de débordement.'
		exit:
		RET
		C Z S O P A
		inchangés

JNP	label	Saut si la condition est réalisée. Saut court si pas de parité (impaire). Seul le résultat des 8 bits de poids faible est vérifié. Ce flag est positionné par les instructions CMP, SUB, ADD, TEST, AND, OR, XOR. Algorithme : si PF = 0 alors saut Exemple : include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; positionne juste les flags. JNP label1 PRINT 'Parité paire.' JMP exit label1: PRINT 'Parité impaire.' exit: RET C Z S O P A inchangés
JNS	label	Saut si la condition est réalisée. Saut court si non signé (positif). Ce flag est positionné par les instructions CMP, SUB, ADD, TEST, AND, OR, XOR. Algorithme : si SF = 0 alors saut Exemple : include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; positionne juste les flags. JNS label1

		PRINT 'Signé.' JMP exit label1: PRINT 'Non signé.' exit: RET C Z S O P A inchangés
JNZ	label	Saut si la condition est réalisée. Saut court si non zéro (pas égal). Ce flag est positionné par les instructions CMP, SUB, ADD, TEST, AND, OR, XOR. Algorithme : si ZF = 0 alors saut Exemple : include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; positionne juste les flags. JNZ label1 PRINT 'Zéro.' JMP exit label1: PRINT 'Non zéro.' exit: RET C Z S O P A inchangés

JO	label	Saut si la condition est réalisée.
		Saut court si débordement.
		
		Algorithme :
		
		si OF = 1 alors saut
		Exemple :
		; -5 - 127 = -132 (ne se trouve pas entre -128 et 127)
		; le résultat de l'instruction SUB est faux (124),
		; donc OF = 1 :
		
		include 'emu8086.inc'
		#make_COM#
		org 100h
		MOV AL, -5
		SUB AL, 127 ; AL = 7Ch (124)
		JO label1
		PRINT 'Pas de débordement.'
		JMP exit
		label1:
		PRINT 'Débordement !'
		exit:
		RET
		C Z S O P A
		inchangés

JP	label	Saut si la condition est réalisée. Saut court si parité (paire). Seul le résultat des 8 bits de poids faible est vérifié. Ce flag est positionné par les instructions CMP, SUB, ADD, TEST, AND, OR, XOR. Algorithme : si PF = 1 alors saut Exemple : include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 00000101b ; AL = 5 OR AL, 0 ; positionne juste les flags. JP label1 PRINT 'Parité impaire.' JMP exit label1: PRINT 'Parité paire.'

		exit: RET C Z S O P A inchangés
JPE	label	Saut si la condition est réalisée. Saut court si parité (paire). Seul le résultat des 8 bits de poids faible est vérifié. Ce flag est positionné par les instructions CMP, SUB, ADD, TEST, AND, OR, XOR. Algorithme : si PF = 1 alors saut Exemple : include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 00000101b ; AL = 5 OR AL, 0 ; positionne juste les flags. JPE label1 PRINT 'Parité impaire.' JMP exit label1: PRINT 'Parité paire.' exit: RET C Z S O P A inchangés
JPO	label	Saut si la condition est réalisée. Saut court si parité (impaire). Seul le résultat des 8 bits de poids faible est vérifié. Ce flag est positionné par les instructions CMP, SUB, ADD, TEST, AND, OR, XOR. Algorithme : si PF = 0 alors saut Exemple : include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 00000111b ; AL = 7 OR AL, 0 ; positionne juste les flags. JPO label1 PRINT 'Parité paire.' JMP exit label1: PRINT 'Parité impaire.' exit: RET C Z S O P A inchangés
JS	label	Saut si la condition est réalisée. Saut court si signé (négatif). Ce flag est positionné par les instructions CMP, SUB, ADD, TEST, AND, OR, XOR. Algorithme : si SF = 1 alors saut Exemple : include 'emu8086.inc' #make_COM# ORG 100h MOV AL, 10000000b ; AL = -128 OR AL, 0 ; positionne juste les flags. JS label1 PRINT 'Non signé.' JMP exit label1: PRINT 'Signé.' exit: RET C Z S O P A inchangés

JZ	label	Saut si la condition est réalisée.
		Saut court si zéro (égal). Ce flag est positionné par les instructions
		CMP, SUB, ADD, TEST, AND, OR, XOR.
		
		Algorithme :
		
		si ZF = 1 alors saut
		Exemple :
		`include 'emu8086.inc'`
		`#make_COM#`
		`ORG 100h`
		`MOV AL, 5`
		`CMP AL, 5`
		`JZ label1`
		`PRINT 'AL n'est pas égal à 5.'`
		`JMP exit`
		`label1:`
		`PRINT 'AL est égal à 5.'`
		`exit:`
		`RET`
		C Z S O P A
		inchangés

LAHF	Pas d'opérande	Charge AH avec les 8 bits de poids faible du registre de flags. Algorithme : AH = registre de flags Bits de AH : 7 6 5 4 3 2 1 0 [SF] [ZF] [0] [AF] [0] [PF] [1] [CF] bits 1, 3, 5 sont réservés. C Z S O P A inchangés

LDS	REG, mémoire	Charge un pointeur long (double mot) à partir de la mémoire dans un registre 16 bits et dans DS.
		
		Algorithme :
		
		• REG = premier mot
		• DS = deuxième mot
		
		Exemple :
		
		`#make_COM#`
		`ORG 100h`
		
		`LDS AX, m`
		
		`RET`
		
		`m DW 1234h`
		`DW 5678h`
		
		`END`
		
		AX contient 1234h, DS contient 5678h.
		
		C Z S O P A
		inchangés

LEA	REG, mémoire	Charge une adresse effective Algorithme :

		• REG = adresse de mémoire (offset) Généralement, cette instruction est remplacée si possible par MOV lors de l'assemblage. Exemple : #make_COM# ORG 100h LEA AX, m RET m DW 1234h END AX contient : 0104h. L'instruction LEA utilise 3 octets, RET utilise 1 octet pour le retour, nous débutons à 100h, donc l'adresse de 'm' est 104h. C Z S O P A inchangés
LES	REG, mémoire	Charge un pointeur long (double mot) à partir de la mémoire dans un registre 16 bits et dans ES. Algorithme : • REG = premier mot • ES = deuxième mot Exemple : #make_COM# ORG 100h LES AX, m RET m DW 1234h DW 5678h END AX contient 1234h, ES contient 5678h. C Z S O P A inchangés

LODSB	Pas d'opérande	Charge un octet depuis DS:[SI] dans AL et mise à jour de SI.
		
		Algorithme :
		
		• AL = DS:[SI]
		• si DF = 0 alors
		o SI = SI + 1
		
		sinon
		
		o SI = SI - 1
		
		Exemple :
		#make_COM#
		ORG 100h
		
		LEA SI, a1
		MOV CX, 5
		MOV AH, 0Eh

		m: LODSB INT 10h LOOP m RET a1 DB 'H', 'e', 'l', 'l', 'o' C Z S O P A inchangés
LODSW	Pas d'opérande	Charge un mot depuis DS:[SI] dans AX et mise à jour de SI. Algorithme : • AX = DS:[SI] • si DF = 0 alors o SI = SI + 2 sinon o SI = SI - 2 Exemple : #make_COM# ORG 100h LEA SI, a1 MOV CX, 5 REP LODSW ; finalement il y aura 555h dans AX. RET a1 dw 111h, 222h, 333h, 444h, 555h C Z S O P A inchangés
LOOP	label	Décrémente CX, boucle en revenant au label tant que CX n'est pas à zéro. Algorithme : • CX = CX - 1 • si CX <> 0 alors o saut sinon o pas de saut, continue Exemple : include 'emu8086.inc' #make_COM# ORG 100h MOV CX, 5 label1: PRINTN 'boucle !' LOOP label1 RET C Z S O P A inchangés
LOOPE	label	Décrémente CX, boucle en revenant au label si CX non zéro et égal (ZF = 1). Algorithme : • CX = CX - 1 • si (CX <> 0) et (ZF = 1) alors

		saut
		else
		pas de saut, continue
		
		Exemple :
		; Boucle jusqu'à ce que le résultat dans AL soit 1
		; ou 5 fois. Le résultat sera supérieur à 255 lors
		; de la troisième boucle (100+100+100) et
		; sortira de la boucle.
		
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AX, 0
		MOV CX, 5
		label1:
		PUTC '*'
		ADD AX, 100
		CMP AH, 0
		LOOPE label1
		RET
		C Z S O P A
		inchangés

LOOPNE	label	Décrémente CX, boucle en revenant au label si CX non zéro et non égal (ZF = 0).
		
		Algorithme :
		
		• CX = CX - 1
		• si (CX <> 0) et (ZF = 0) alors
		saut
		sinon
		pas de saut, continue
		
		Exemple :
		; Boucle jusqu'à ce que '7' soit trouvé
		; ou que 5 boucles soient effectuées.
		
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV SI, 0
		MOV CX, 5
		label1:
		PUTC '*'
		MOV AL, v1[SI]
		INC SI ; octet suivant : (SI=SI+1).
		CMP AL, 7
		LOOPNE label1
		RET
		v1 db 9, 8, 7, 6, 5
		C Z S O P A
		inchangés

LOOPNZ	label	Décrémente CX, boucle en revenant au label si CX non zéro et non égal (ZF = 0). Algorithme : • CX = CX - 1 • si (CX <> 0) et (ZF = 0) alors saut sinon pas de saut, continue Exemple :

		; Boucle jusqu'à ce que '7' soit trouvé
		; ou que 5 boucles soient effectuées.
		
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV SI, 0
		MOV CX, 5
		label1:
		PUTC '*'
		MOV AL, v1[SI]
		INC SI ; octet suivant : (SI=SI+1).
		CMP AL, 7
		LOOPNZ label1
		RET
		v1 db 9, 8, 7, 6, 5
		C Z S O P A
		inchangés

LOOPZ	label	Décrémente CX, boucle en revenant au label si CX non zéro et ZF = 1.
		
		Algorithme :
		• CX = CX - 1
		• si (CX <> 0) et (ZF = 1) alors
		o saut
		sinon
		o pas de saut, continue
		
		Exemple :
		; Boucle jusqu'à ce que le résultat dans AL soit 1
		; ou 5 fois. Le résultat sera supérieur à 255 lors
		; de la troisième boucle (100+100+100) et
		; sortira de la boucle.
		
		include 'emu8086.inc'
		#make_COM#
		ORG 100h
		MOV AX, 0
		MOV CX, 5
		label1:
		PUTC '*'
		ADD AX, 100
		CMP AH, 0
		LOOPZ label1
		RET
		C Z S O P A
		inchangés

MOV	REG, mémoire
	mémoire, REG
	REG, REG
	mémoire, immédiate
	REG, immédiate
	
	SREG, mémoire
	mémoire, SREG
	REG, SREG
	SREG, REG
		Copie l'opérande2 dans l'opérande1.
		
		L'instruction MOV ne peut pas :
		• Utiliser les valeurs des registres CS et IP.
		• Copier la valeur d'un registre de segment dans un autre registre de segment (il faut d'abord copier la valeur d'un segment de registre dans un registre général et ensuite, copier dans un autre registre de segment).
		• Copier une valeur immédiate dans un registre de segment (il faut d'abord, passer par un registre général).
		
		Algorithme :
		opérande1 = opérande2
		
		Exemple :
		#make_COM#
		ORG 100h
		MOV AX, 0B800h ; AX = B800h (mémoire VGA).
		MOV DS, AX ; copie la valeur de AX dans DS.
		MOV CL, 'A' ; CL = 41h (code ASCII).
		MOV CH, 01011111b ; CL = attribut couleur.

		MOV BX, 15Eh ; BX = position d'affichage.
		MOV [BX], CX ; w.[0B800h:015Eh] = CX.
		RET ; retour au système d'exploitation.
		C Z S O P A
		inchangés

MOVSB	Pas d'opérande	Copie l'octet de DS:[SI] dans ES:[DI]. Met à jour SI et DI. Algorithme : • ES:[DI] = DS:[SI] • si DF = 0 alors o SI = SI + 1 o DI = DI + 1 sinon o SI = SI - 1 o DI = DI - 1 Exemple : #make_COM# ORG 100h LEA SI, a1 LEA DI, a2 MOV CX, 5 REP MOVSB RET a1 DB 1,2,3,4,5 a2 DB 5 DUP(0) C Z S O P A inchangés
MOVSW	Pas d'opérande	Copie le mot de DS:[SI] dans ES:[DI]. Met à jour SI et DI. Algorithme : • ES:[DI] = DS:[SI] • si DF = 0 alors o SI = SI + 2 o DI = DI + 2 sinon o SI = SI - 2 o DI = DI - 2 Exemple : #make_COM# ORG 100h LEA SI, a1 LEA DI, a2 MOV CX, 5 REP MOVSW RET a1 DW 1,2,3,4,5 a2 DW 5 DUP(0) C Z S O P A inchangés
MUL	REG mémoire	Multiplication non signée.

		Algorithme :
		
		lorsque l'opérande est un **octet** :
		AX = AL * opérande.
		
		lorsque l'opérande est un **mot** :
		(DX AX) = AX * opérande.
		
		Exemple :
		MOV AL, 200 ; AL = 0C8h
		MOV BL, 4
		MUL BL ; AX = 0320h (800)
		RET
		C Z S O P A
		r ? ? r ? ?
		CF=OF=0 si la section haute du résultat est zéro.

NEG	REG mémoire	Négation. Transforme la valeur d'un opérande en valeur négative (par son complément à deux). Algorithme : • Inverse tous les bits de l'opérande • Additionne 1 à l'opérande inversé Exemple : MOV AL, 5 ; AL = 05h NEG AL ; AL = 0FBh (-5) NEG AL ; AL = 05h (5) RET C Z S O P A r r r r r r
NOP	Pas d'opérande	Aucune opération. Algorithme : • Rien Exemple : ; rien, 3 fois : NOP NOP NOP RET C Z S O P A inchangés
NOT	REG mémory	Inverse chaque bit de l'opérande. Algorithme : • si bit est 1 remplace par 0. • si bit est 0 remplace par 1. Exemple : MOV AL, 00011011b NOT AL ; AL = 11100100b RET C Z S O P A inchangés
OR	REG, mémoire mémoire, REG REG, REG	Effectue un OR (OU) logique entre tous les bits des deux opérandes. Le résultat est stocké dans le premier opérande. Table de vérité :

	immédiate REG, immédiate	1 OR (OU) 1 = 1 1 OR (OU) 0 = 1 0 OR (OU) 1 = 1 0 OR (OU) 0 = 0 Exemple : MOV AL, 'A' ; AL = 01000001b OR AL, 00100000b ; AL = 01100001b ('a') RET C Z S O P A 0 r r 0 r ?

OUT	im.octet, AL
	im.octet, AX
	DX, AL
	DX, AX
		Place la valeur de **AL** ou **AX** sur le port.
		Le premier opérande contient le numéro de port. Pour accéder à un port dont le numéro est supérieur à 255, il faut utiliser le registre **DX**.
		
		Exemple :
		MOV AX, 0FFFh ; Allumer tous les
		OUT 4, AX ; feux de circulation.
		
		MOV AL, 100b ; Alimenter le troisième
		OUT 7, AL ; électro-aimant du moteur pas à pas.
		C Z S O P A
		inchangés

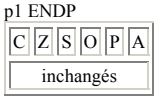
POP	REG SREG mémoire	Dépile une valeur 16 bits. Algorithme : • opérande = SS:[SP] (sommet de la pile) • SP = SP + 2 Exemple : MOV AX, 1234h PUSH AX POP DX ; DX = 1234h RET C Z S O P A inchangés
POPA	Pas d'opérande	Dépile tous les registres généraux DI, SI, BP, SP, BX, DX, CX, AX . La valeur de SP est ignorée, elle est dépilée mais pas placée dans le registre SP. Note : cette instruction fonctionne uniquement avec les CPU **80186** ou supérieurs ! Algorithme : • POP DI • POP SI • POP BP • POP xx (la valeur de SP est ignorée) • POP BX • POP DX • POP CX • POP AX C Z S O P A inchangés
POPF	Pas d'opérande	Dépile le registre flags. Algorithme : • flags = SS:[SP] (sommet de la pile)

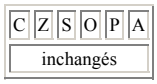
		• SP = SP + 2 C Z S O P A dépilés
PUSH	REG SREG mémoire immédiate	Empile une valeur 16 bits. Note : **EMPILER une valeur immédiate**, fonctionne uniquement avec les CPU 80186 ou supérieurs ! Algorithme : • SP = SP - 2 • SS:[SP] (sommet de la pile) = opérande Exemple : MOV AX, 1234h PUSH AX POP DX ; DX = 1234h RET C Z S O P A inchangés
PUSHA	Pas d'opérande	Empile tous les registres généraux DI, SI, BP, SP, BX, DX, CX, AX . La valeur d'origine du registre SP (avant PUSHA) est utilisée. Note : cette instruction fonctionne uniquement avec les CPU **80186** ou supérieurs ! Algorithme : • PUSH AX • PUSH CX • PUSH DX • PUSH BX • PUSH SP • PUSH BP • PUSH SI • PUSH DI C Z S O P A inchangés
PUSHF	Pas d'opérande	Empile le registre flags. Algorithme : • SP = SP - 2 • SS:[SP] (sommet de la pile) = flags C Z S O P A inchangés
RCL	mémoire, immédiate REG, immédiate mémoire, CL REG, CL	La rotation effectue un décalage de l'opérande1 à gauche de tous les bits y compris le flag de retenue. Le nombre de rotations est contenu dans l'opérande2. Si la valeur **immédiate** est supérieure à 1, l'assembleur génère plusieurs **RCL xx**, **une** instruction à la fois, car le 8086 a le code machine uniquement pour cette instruction (le même principe est utilisé pour toutes les autres instructions de rotations/décalages). Algorithme :

		placé à la position droite laissée vide par le décalage. Exemple : STC ; place la retenue (CF=1). MOV AL, 1Ch ; AL = 00011100b RCL AL, 1 ; AL = 00111001b, CF=0. RET  OF=0 si c'est le premier opérande, il conserve le signe d'origine.
RCR	mémoire, immédiate REG, immédiate mémoire, CL REG, CL	La rotation effectue un décalage de l'opérande1 à gauche de tous les bits y compris le flag de retenue. Le nombre de rotations est contenu dans l'opérande2. Algorithme : Décale tous les bits vers la droite, le bit qui déborde est placé dans CF et le bit précédent de CF est placé à la position gauche laissée vide par le décalage. Exemple : STC ; place la retenue (CF=1). MOV AL, 1Ch ; AL = 00011100b RCR AL, 1 ; AL = 10001110b, CF=0. RET  OF=0 si c'est le premier opérande, il conserve le signe d'origine.
REP	instruction chaîne	Répète les instructions MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW, le nombre de fois contenu dans CX. Algorithme : contrôle_cx : si CX <> 0 alors • exécute l'instruction chaîne suivante • CX = CX - 1 • retourne à contrôle_cx sinon • sort du cycle REP 
REPE	instruction chaîne	Répète les instructions CMPSB, CMPSW, SCASB, SCASW, tant que ZF = 1 (égalité du résultat), le nombre de fois maximum contenu dans CX. Algorithme : contrôle_cx : si CX <> 0 alors • exécute l'instruction chaîne suivante • CX = CX - 1 • si ZF = 1 alors : o retourne à contrôle_cx sinon o sort du cycle REPE sinon

		• sort du cycle REPE Exemple : voir **cmpsb.asm** dans le dossier "Samples". 
REPNE	instruction chaîne	Répète les instructions CMPSB, CMPSW, SCASB, SCASW, tant que ZF = 0 (NON égalité du résultat), le nombre de fois maximum contenu dans CX. Algorithme : contrôle_cx : si CX <> 0 alors • exécute l'instruction chaîne suivante • CX = CX - 1 • si ZF = 0 alors : o retourne à contrôle_cx sinon o sort du cycle REPNE sinon • sort du cycle REPNE 
REPNZ	instruction chaîne	Répète les instructions CMPSB, CMPSW, SCASB, SCASW tant que ZF = 0 (résultat NON zéro), le nombre de fois maximum contenu dans CX. Algorithme : contrôle_cx : si CX <> 0 alors • exécute l'instruction chaîne suivante • CX = CX - 1 • si ZF = 0 alors : o retourne à contrôle_cx sinon o sort du cycle REPNZ sinon • sort du cycle REPNZ 
REPZ	instruction chaîne	Répète les instructions CMPSB, CMPSW, SCASB, SCASW tant que ZF = 1 (résultat zéro), le nombre de fois maximum contenu dans CX.

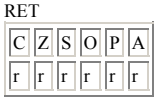
		contrôle_cx : si CX <> 0 alors • exécute l'instruction chaîne suivante • CX = CX - 1 • si ZF = 1 alors : o retourne à contrôle_cx sinon o sort du cycle REPZ sinon • sort du cycle REPZ 

RET	Pas d'opérande même immédiat	Retour de sous-programme (court).
		Algorithme :
		• Dépile :
		o IP
		• si un opérande immédiat est présent : SP = SP + opérande
		Exemple :
		#make_COM#
		ORG 100h ; pour un fichier exécutable COM.
		CALL p1
		ADD AX, 1
		RET ; retour au système d'exploitation.
		p1 PROC ; déclaration de la procédure.
		MOV AX, 1234h
		RET ; retour à l'appelant.
		p1 ENDP
		 

RETF	Pas d'opérande même immédiat	Retour de sous-programme (long). Algorithme : • Dépile : o IP o CS • si un opérande immédiat est présent : SP = SP + operand 
ROL	mémoire, immédiate REG, immédiate mémoire, CL REG, CL	La rotation effectue un décalage de l'opérande1 à gauche. Le nombre de rotations est contenu dans l'opérande2. Algorithme : Décale tous les bits vers la gauche, le bit qui déborde est placé dans CF et le bit précédent de CF est placé à la position droite laissée vide par le décalage. Exemple :

		MOV AL, 1Ch ; AL = 00011100b ROL AL, 1 ; AL = 00111000b, CF=0. RET  OF=0 si c'est le premier opérande, il conserve le signe d'origine.
ROR	mémoire, immédiate REG, immédiate mémoire, CL REG, CL	La rotation effectue un décalage de l'opérande1 à droite. Le nombre de rotations est contenu dans l'opérande2. Algorithme : Décale tous les bits vers la droite, le bit qui déborde est placé dans CF et le bit précédent de CF est placé à la position gauche laissée vide par le décalage. Exemple : MOV AL, 1Ch ; AL = 00011100b ROR AL, 1 ; AL = 00001110b, CF=0. RET  OF=0 si c'est le premier opérande, il conserve le signe d'origine.
SAHF	Pas d'opérande	Place dans le registre de flags les 8 bits de poids faible de la valeur de AH. Algorithme : AH = registre de flags Bits de AH : 7 6 5 4 3 2 1 0 [SF] [ZF] [0] [AF] [0] [PF] [1] [CF] les bits 1, 3, 5 sont réservés. 
SAL	mémoire, immédiate REG, immédiate mémoire, CL REG, CL	Effectue un décalage arithmétique de l'opérande1 à gauche. Le nombre de décalages est contenu dans l'opérande2. Algorithme : • Décale tous les bits vers la gauche, le bit qui déborde est placé dans CF. • Le bit 0 à droite est mis à 0. Exemple : MOV AL, 0E0h ; AL = 11100000b SAL AL, 1 ; AL = 11000000b, CF=1. RET  OF=0 si c'est le premier opérande, il conserve le signe d'origine.

SAR	mémoire, immédiate
	REG, immédiate
	
	mémoire, CL
	REG, CL
		Effectue un décalage arithmétique de l'opérande1 à droite. Le nombre de décalages est contenu dans l'opérande2.
		
		Algorithme :
		
		• Décale tous les bits vers la droite, le bit qui déborde est placé dans CF.
		• Le bit de signe à gauche, garde la même valeur qu'avant le décalage.
		
		Exemple :
		MOV AL, 0E0h ; AL = 11100000b
		SAR AL, 1 ; AL = 11110000b, CF=0.
		
		MOV BL, 4Ch ; BL = 01001100b
		SAR BL, 1 ; BL = 00100110b, CF=0.
		
		RET
		 
		
		OF=0 si c'est le premier opérande, il conserve le signe d'origine.

SBB	REG, mémoire mémoire, REG REG, REG mémoire, immédiate REG, immédiate	Soustraction avec retenue. Algorithme : opérande1 = opérande1 - opérande2 - CF Exemple : STC MOV AL, 5 SBB AL, 3 ; AL = 5 - 3 - 1 = 1 RET 
SCASB	Pas d'opérande	Compare les octets : AL avec ES:[DI]. Algorithme : • ES:[DI] - AL • place les flags suivant le résultat : OF, SF, ZF, AF, PF, CF • si DF = 0 alors o DI = DI + 1 sinon o DI = DI - 1 
SCASW	Pas d'opérande	Compare les octets : AX avec ES:[DI]. Algorithme : • ES:[DI] - AX • place les flags suivant le résultat : OF, SF, ZF, AF, PF, CF • si DF = 0 alors o DI = DI + 2 sinon o DI = DI - 2 
SHL	mémoire, immédiate REG, immédiate mémoire, CL REG, CL	Effectue un décalage de l'opérande1 à gauche. Le nombre de décalages est contenu dans l'opérande2. Algorithme : • Décale tous les bits vers la gauche, le bit qui déborde est placé dans CF. • Le bit 0 à droite est mis à 0. Exemple : MOV AL, 11100000b SHL AL, 1 ; AL = 11000000b, CF=1. RET  OF=0 si c'est le premier opérande, il conserve le signe d'origine.

SHR	mémoire, immédiate REG, immédiate mémoire, CL REG, CL	Effectue un décalage de l'opérande1 à droite. Le nombre de décalages est contenu dans l'opérande2. Algorithme : • Décale tous les bits vers la droite, le bit qui déborde est placé dans CF. • Le bit 0 à gauche est mis à 0. Exemple : MOV AL, 00000111b SHR AL, 1 ; AL = 00000011b, CF=1. RET  OF=0 si c'est le premier opérande, il conserve le signe d'origine.
STC	Pas d'opérande	Mise à 1 du flag de retenue. Algorithme : CF = 1 
STD	Pas d'opérande	Mise à 1 du flag de direction. SI et DI seront décrémentés par les instructions chaînes suivantes : CMPSB, CMPSW, LODSB, LODSW, MOVSB, MOVSW, STOSB, STOSW. Algorithme : DF = 1 
STI	Pas d'opérande	Mise à 1 du flag d'interruption. Désactive des interruptions matérielles. Algorithme : IF = 1 

STOSB	Pas d'opérande	Copie l'octet contenu dans AL dans ES:[DI]. Met à jour SI.
		
		Algorithme :
		
		• ES:[DI] = AL
		• si DF = 0 alors
		o DI = DI + 1
		
		sinon
		
		o DI = DI - 1
		
		Exemple :
		#make_COM#
		ORG 100h
		
		LEA DI, a1
		MOV AL, 12h
		MOV CX, 5
		
		REP STOSB

		RET a1 DB 5 dup(0) C Z S O P A inchangés

STOSW	Pas d'opérande	Copie l'octet contenu dans AX dans ES:[DI]. Met à jour SI.
		
		Algorithme :
		
		• ES:[DI] = AX
		• si DF = 0 alors
		o DI = DI + 2
		
		sinon
		
		o DI = DI - 2
		
		Exemple :
		#make_COM#
		ORG 100h
		
		LEA DI, a1
		MOV AX, 1234h
		MOV CX, 5
		
		REP STOSW
		
		RET
		
		a1 DW 5 dup(0)
		C Z S O P A
		inchangés

SUB	REG, mémoire
	mémoire, REG
	REG, REG
	mémoire, immédiate
	REG, immédiate
		Soustraction.
		
		Algorithme :
		
		opérande1 = opérande1 - opérande2
		
		Exemple :
		MOV AL, 5
		SUB AL, 1 ; AL = 4
		
		RET
		C Z S O P A
		r r r r r r

TEST	REG, mémoire mémoire, REG REG, REG mémoire, immédiate REG, immédiate	AND (ET) logique entre tous les bits des deux opérandes positionne uniquement les flags. Les flags suivants sont affectés : **ZF**, **SF**, **PF**. Le résultat n'est pas utilisé. Table de vérité : 1 AND (ET) 1 = 1 1 AND (ET) 0 = 0 0 AND (ET) 1 = 0 0 AND (ET) 0 = 0 Exemple : MOV AL, 00000101b TEST AL, 1 ; ZF = 0. TEST AL, 10b ; ZF = 1. RET C Z S O P 0 r r 0 r
XCHG	REG, mémoire mémoire, REG REG, REG	Echange les valeurs des deux opérandes.

		opérande1 < - > opérande2
		
		Exemple :
		MOV AL, 5
		MOV AH, 2
		XCHG AL, AH ; AL = 2, AH = 5
		XCHG AL, AH ; AL = 5, AH = 2
		RET
		
		C Z S O P A
		inchangés

XLATB	Pas d'opérande	Conversion d'octet depuis une table. Copie la valeur de l'octet mémoire de DS:[BX + AL non signé] dans le registre AL. Algorithme : AL = DS:[BX + AL non signé] Exemple : #make_COM# ORG 100h LEA BX, dat MOV AL, 2 XLATB ; AL = 33h RET dat DB 11h, 22h, 33h, 44h, 55h C Z S O P A inchangés
XOR	REG, mémoire mémoire, REG REG, REG mémoire, immédiate REG, immédiate	XOR logique (OU exclusif) entre tous les bits des deux opérandes. Le résultat est stocké dans le premier opérande. Table de vérité : 1 XOR (OU exclusif) 1 = 0 1 XOR (OU exclusif) 0 = 1 0 XOR (OU exclusif) 1 = 1 0 XOR (OU exclusif) 0 = 0 Exemple : MOV AL, 00000111b XOR AL, 00000010b ; AL = 00000101b RET C Z S O P A 0 r r 0 r ?