

# Systèmes Informatiques I

## Partie I: Assembleur

Jürgen Harms  
juergen.harms@cui.unige.ch  
<http://cui.unige.ch>



Systèmes Informatiques I, autres documents :

[Partie I: Assembleur \(fragmentée pour web\) \\*\)](#)

[Partie II: UNIX](#)

[Travaux pratiques](#)

\*) 1 document PDF d'environ 200 kbytes pour chaque chapitre  
pas de chaînage 'hypertexte' (PDF) entre documents

### Table des matières

- [1. Introduction](#)
- [2. L'architecture du CPU Intel 8086](#)
- [3. Principes d'un assembleur](#)
- [4. Le répertoire d'instructions](#)
- [5. Variables et expressions](#)
- [6. Aspects méta-assembleur](#)
- [7. Extensions de l'architecture de la famille 8086](#)
- [8. Principes de l'entrée / sortie](#)

[Liste des instructions du CPU Intel 8086](#)

[Littérature et références](#)

[Mots-clé](#)

'Boutons' dans  
les documents:



... début du chapitre



... fin du chapitre



... page précédente



... page en arrière



... page suivante



... page d'accueil

# Table des matières

Pour accéder à une page, sélectionner la page avec un “click” de la souris

INTRODUCTION .....	1.1	- Le rôle du bit “carry” (report arithmétique, décalage) .....	2.19
Buts (cours, document) .....	1.2	Quelques opérations typiques .....	2.20
Evolution des familles de microprocesseurs .....	1.5	Adressage Intel 8086 .....	2.21
Exemple simple .....	1.6	Principes .....	2.21
Code-source et code-objet .....	1.6	Modes d’adressage .....	2.22
Segments et adresses .....	1.7	Combinaison des modes (phase 1 et 2) .....	2.24
Parties principales d’un programme en machine .....	1.8	- Modes d’adressage (exemples Intel 8086) .....	2.25
Description de la “zone-instructions” d’un programme .....	1.9	PRINCIPES D’UN ASSEMBLEUR .....	3.1
L’ARCHITECTURE DU CPU INTEL 8086 .....	2.1	Assembleur : introduction .....	3.2
Configuration d’un système Intel 8086 .....	2.2	Caractéristiques d’un assembleur .....	3.3
Mémoire .....	2.3	Syntaxe .....	3.4
Mot-mémoire à l’adresse ‘n’, “sexe des bytes” .....	2.3	Forme générale d’une instruction MASM .....	3.4
Implantation physique .....	2.4	Signification des champs de l’instruction .....	3.5
- Plan de la mémoire (Intel 8086) .....	2.4	- Champ d’étiquette .....	3.5
Mémoire physique et mémoire logique .....	2.5	- Champ d’opération .....	3.5
Représentation externe des adresses .....	2.5	- Champ d’opérande .....	3.6
Génération une adresse-bus dans le CPU .....	2.6	- Champ de commentaire .....	3.6
La pile .....	2.7	Syntaxe : constantes .....	3.7
Séquence physique - séquence logique .....	2.9	- Format des constantes numériques .....	3.7
Instructions et données d’un programme .....	2.10	- Format des chaînes de caractères .....	3.7
Format d’une opération .....	2.11	Spécification d’adresses .....	3.8
Champs du code d’une opération .....	2.11	- Sélection du mode d’adressage .....	3.8
- Opérandes .....	2.11	Mode d’adressage .....	3.9
Propriétés des opérations du CPU Intel 8086 .....	2.12	- Calcul de l’adresse effective .....	3.9
Différences entre instructions-assembleur et instructions-machine .....	2.13	- Recherche de l’opérande .....	3.10
Registres du CPU Intel 8086 .....	2.15	Génération de code-objet .....	3.11
Registres-opérandes et d’indexage .....	2.15	Tables de code-objet .....	3.11
Recouvrement des registres à 8 et à 16 bits .....	2.16	Segment logique .....	3.11
Registres de base (“segment registers”) .....	2.17	Compteur de position .....	3.12
Compteur ordinal .....	2.17	- Rôle du compteur de position .....	3.12
Registre d’état (“flag register”) .....	2.18	- Contrôle du compteur de position .....	3.12
		- Valeur du compteur de position .....	3.12



- Compteur de position et compteur-ordinal . . . . .	3.13
- Valeur initiale du compteur-ordinal lors de l'exécution . . . . .	3.13
Contrôle des segments . . . . .	3.14
L'instruction SEGMENT . . . . .	3.14
Association du code-objet à un segment logique . . . . .	3.15
- Décomposition du code en segments logiques . . . . .	3.15
- Gestion à l'aide d'un compteur de position . . . . .	3.15
- Alternance entre segments . . . . .	3.15
- Exemple: résultat du travail de l'éditeur de liens . . . . .	3.16
Placement de segments . . . . .	3.17
Alignement de l'adresse de base d'un segment . . . . .	3.18
Gestion des registres de base . . . . .	3.19
Initialisation des registres de base . . . . .	3.19
- Réalisation (assembleur et éditeur de liens) . . . . .	3.20
Association segments logiques - registres de base . . . . .	3.21
Gestion des registres de base, exemple . . . . .	3.22
Contrôle des segments, résumé . . . . .	3.24
Exemple d'un programme . . . . .	3.25
Définition de données . . . . .	3.26
Pseudo-instructions pour la réservation et définition de données . . . . .	3.26
- La fonction DUP . . . . .	3.27
- Constantes-caractères . . . . .	3.27
- Formats possibles . . . . .	3.27
Réservation de données, exemples . . . . .	3.28
<b>LE RÉPERTOIRE D'INSTRUCTIONS . . . . .</b>	<b>4.1</b>
Opérations utilisant la pile . . . . .	4.2
Utilisation d'une pile . . . . .	4.2
Opérations utilisant la pile . . . . .	4.3
Opérations de branchement . . . . .	4.4
Forme des instructions de branchement . . . . .	4.4
Branchements sans condition . . . . .	4.4
Branchements pour itérations . . . . .	4.4
Branchements conditionnels . . . . .	4.5
Opérations arithmétiques & logiques . . . . .	4.6

Extension du bit de signe . . . . .	4.9
Opérations itératives, actions sur chaînes de caractères . . . . .	4.10
Résumé des instructions . . . . .	4.10
Contrôle des itérations . . . . .	4.11
Contrôle des bits du registre 'flags' . . . . .	4.12
Diverses opérations . . . . .	4.12
<b>VARIABLES ET EXPRESSIONS . . . . .</b>	<b>5.1</b>
Variables . . . . .	5.2
Composantes d'une variable . . . . .	5.2
Définition d'une variable . . . . .	5.3
- Visibilité d'une variable . . . . .	5.3
Types de variables . . . . .	5.4
Etiquettes ("labels") . . . . .	5.4
Variables relogeables ("variables") . . . . .	5.5
Constantes ("numbers") . . . . .	5.5
Expressions . . . . .	5.6
Fonctions . . . . .	5.7
Fonction pour l'accès au compteur de position . . . . .	5.7
Fonctions pour obtenir les attributs d'un "RECORD" . . . . .	5.7
Fonctions pour obtenir les attributs d'une variable . . . . .	5.8
Fonctions pour manipuler les attributs d'une variable . . . . .	5.9
Fonctions comme opérateurs dans une expression . . . . .	5.10
<b>ASPECTS MÉTA-ASSEMBLEUR . . . . .</b>	<b>6.1</b>
Assemblage conditionnel . . . . .	6.2
- Conditions reconnues . . . . .	6.2
- Utilisation . . . . .	6.3
Assemblage répétitif . . . . .	6.4
Substitutions de chaînes . . . . .	6.6
Macro-assemblage . . . . .	6.7
Définition d'une macro-instruction . . . . .	6.7
Appel d'une macro-instruction . . . . .	6.7
- Principes de la substitution . . . . .	6.7



Déclaration de variables locales .....	6.8
Séparateur de chaînes de caractères .....	6.8
Exemples .....	6.9
- Macro instruction pour imprimer une chaîne de caractères ....	6.9
- Macro instructions pour la gestion de files d'attente .....	6.10
Importation de code-source .....	6.13
Décomposition en modules .....	6.14
Procédures .....	6.14
Utilisation de variables globales .....	6.14
Utilisation de procédures, exemple .....	6.15
Contrôle de l'impression .....	6.16
Suppression de l'impression .....	6.16
Lignes d'en-tête .....	6.16
Mise-en-page .....	6.16
Impression de code conditionnel .....	6.17
Impression des macro-instructions .....	6.17
<b>EXTENSIONS DE L'ARCHITECTURE DE LA FAMILLE 8086 .....</b>	<b>7.1</b>
Evolution de l'architecture du CPU 8086 .....	7.2
Agrandissement de la taille d'un mot-CPU a 32 bits .....	7.3
Espace-mémoire, mode d'adressage .....	7.4
- Concept .....	7.4
- Descripteurs de segments .....	7.5
- Désignation d'un descripteur par un registre de base .....	7.6
Pagination .....	7.6
Extension du répertoire d'instructions .....	7.7
<b>ENTRÉE / SORTIE .....</b>	<b>8.1</b>
Entrée / sortie dans un système micro-informatique .....	8.2
Concept architectural .....	8.2
Fonctionnement des opérations d'entrée / sortie .....	8.3
Registres internes des équipements d'entrée / sortie .....	8.4
Déroulement d'une opération d'entrée / sortie .....	8.5
- Ecriture .....	8.5
- Lecture .....	8.6
- Esquisse d'un contrôleur DMA .....	8.7

- Fin de l'opération d'entrée / sortie physique .....	8.8
- Interruption d'entrée / sortie .....	8.9
Initialisation d'un équipement d'entrée / sortie .....	8.10
- Opération "reset" .....	8.10
- Séquence d'initialisation .....	8.10
Adressage des registres internes .....	8.11
Documentation .....	8.12
Entrée / sortie "memory-mapped" ou "non-memory-mapped" ...	8.13
ACIA (Motorola 6850) .....	8.14
Tâches de l'interface ACIA .....	8.15
Standard V.24 de l'UIT .....	8.15
- Signaux échangés entre l'interface et le modem .....	8.16
Signification des contenus des registres (ACIA) .....	8.17
- Le registre "status" .....	8.17
- Le registre "contrôle" .....	8.18
ACIA: Schéma (1) .....	8.20
Exemple de programmation .....	8.24

## LISTE DES INSTRUCTIONS DU CPU INTEL 8086

## LITTÉRATURE ET RÉFÉRENCES

# Chapitre 1 : Introduction

## Résumé :

- **Buts du cours sur l'assembleur**
- **Familles de microprocesseurs**
- **Illustration simple d'un programme assembleur**



# Buts (cours, document)

## ☛ Buts du cours

### ➔ Maîtrise de la programmation d'une machine en langage de base

- base pour des activités professionnelles toujours d'actualité (bien que taux d'utilisation relativement faible):
  - programmation en langage de base,
  - compréhension des principes de bas niveau,
- compréhension du fonctionnement du matériel:
  - contrôle directe du CPU,
  - mécanisme de base des opérations d'entrée/sortie,
- aspects "culturels" (développement historique).

### ➔ Familiarisation avec un système simple du genre MS-DOS:

- système = support de programmes écrit en assembleur,
- système = support pour les actions fondamentales de l'utilisateur.



## ☛ Mise-en-garde

### Importance des travaux pratiques:

- aspect “artisanal”: la maîtrise d’un assembleur s’apprend en faisant des exercices,
- mélange de difficultés conceptuelles et pratiques - concentration sur les concepts en automatisant les activités de “programmation”.

### Choix problématique du CPU Intel 8086:

- l’architecture de l’Intel 8086 est quelque peu “baroque”,
- confrontation “choc” avec la complexité du matériel,
- passablement de concepts se retrouvent dans les architectures plus modernes, méthodes d’adressage, concept de segment, etc.

## ☛ Le présent document propose:

- une documentation de base en support aux leçons (éviter distraction “prendre des notes”),
- une documentation technique supplémentaire “à prendre à la maison”.








### Ce document n’est pas:

- ✗ le remplacement d’un livre,
- ✗ un “polycopié” traditionnel.



## ☛ Ce document est disponible sur le serveur Web du CUI

### Navigation par lien 'hypertext'

- références par mots clé (accès par  ou la marque 'liste de références')
- liens dans la table des matières (accès par  ou la marque 'table des matières'),
- parcours séquentiel de pages sans limite de chapitre:
  -  = avancer une page,  = reculer une page;
  -  = avancer au début du chapitre,  = reculer à la fin du chapitre;
  -  = recharger à la dernière page visitée.

### Version pour utilisation locale

<http://cui.unige.ch/tios/cours/asm/slides.pdf>

- un seul fichier d'environ 2 Mbytes
- liens 'hypertext' sans contraintes

### Version pour accès par liaison à faible débit

<http://cui.unige.ch/tios/cours/asm/home.pdf>

- un fichier par chapitre (chacun environ 200 Kbytes)
- liens 'hypertext' restreints (pas de parcours séquentiel d'un chapitre à l'autre, changement de chapitre = recherche d'un nouveau fichier par le réseau).





# Evolution des familles de microprocesseurs

type		vitesse (mbps)	mot (# bits)	date d'annonce	commentaire
<b>Intel</b>	<b>4004</b>	?	4	11. 1971	
	<b>8008</b>	?	8	4. 1972	
	<b>8080</b>	2	8.5	4. 1974	
	<b>8085</b>	2	8.5	3. 1976	CPU "single chip" 8080
	<b>8086</b>	8	16	6 1978	
	<b>8088</b>	8	16	6. 1979	8086, bus de 8 bits
	<b>80186</b>	10	16		8086 + DMA + périphériques
	<b>80286</b>	10	16	2. 1982	MMU ("memory management unit"), support système
...					
	<b>Pentium</b>	>>100	64	dès 1993	
<b>ZILOG</b>	<b>Z 80</b>	2.5	8		8080 + extensions
	<b>Z 8000</b>	10	16		8086 + extensions
<b>Motorola</b>	<b>680x</b>	2	8.5		
	<b>68000</b>	10	16		
	<b>68010</b>	10	16		MMU
	<b>6802x</b>	25	32		MMU
	<b>6803x</b>	> 25	32		MMU
<b>RISC</b>		au début: alternative aux architectures traditionnelles; aujourd'hui: structure de base en support de toute implantation de processeur.			



# Exemple simple

## Code-source et code-objet

Remplir une zone de mémoire de nombres entiers allant de 0 à la val. de la variable `l_tab - 1`

1	<i>Adr.</i>	<i>Code généré</i>	<i>Instruction</i>	<i>Commentaire</i>
2	0000		<code>m_seg SEGMENT</code>	; début du segment logique "m_seg"
3			<code>ASSUME CS:m_seg,</code>	; reg. de base <-> segment logique
4			<code>DS:m_seg</code>	
5	= 004D		<code>l_tab EQU 77</code>	; définir constante: longueur de la table
6	0000	4D [	<code>table DB l_tab DUP (?)</code>	; rés.de la mémoire pour la table
7		??]		
8	004D	B8 ---- R	<code>debut: MOV AX,m_seg</code>	; première instruction à exécuter
9	0050	8E D8	<code>MOV DS,AX</code>	; initialiser DS = base du segment
10	0052	BB 00 4D	<code>MOV BX,l_tab</code>	; initialiser BX = pointeur dans la table
11	0055	4B	<code>rempl: DEC BX</code>	; mise-à-jour pointeur (vers le début)
12	0056	88 9F 0000 R	<code>MOV table[BX],BL</code>	; stocker (BL = byte inférieur de BX)
13	005A	83 FB 00	<code>CMP BX,0</code>	; 0 = début de la table
14	005D	75 F6	<code>JNZ rempl</code>	; continuer sauf si au début
15	005F	B4 4C	<code>MOV AH,4CH</code>	; prép. la requête (4CH = fin exécution)
16	0061	CD 21	<code>INT 21H</code>	; faire la requête à MS/DOS
17	0063		<code>m_seg ENDS</code>	; fin du segment
18			<code>END debut</code>	; fin, définir l'adresse du début

*Note : Distinguer entre "interruption DOS" et "interruption CPU" (= mécanisme matériel)!*

 [Liste des instructions](#)

## Segments et adresses

Microsoft MACRO Assembler Version 3.00

Segments and Groups:

Name	Size	Align	Combine	Class
M_SEG	0063	PARA	NONE	

Symbols:

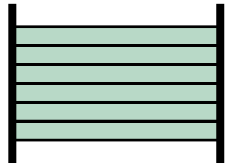
Name	Type	Value	Attr
REMP	L NEAR	0055	M_SEG
DEBUT	L NEAR	004D	M_SEG
TABLE	L BYTE	0000	M_SEG
L_TAB	Number	004D	

49708 Bytes free

Warning	Severe
Errors	Errors
0	0

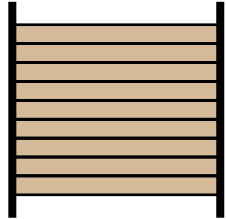


## Parties principales d'un programme en machine



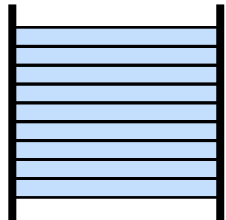
### ***zone-pile:***

... gestion dynamique des données temporaires.



### ***zone-données:***

... positions-mémoire pour la représentation et la gestion des valeurs des variables du programme.



### ***zone-instructions:***

... séquence de positions-mémoire contenant la représentation binaire des instructions à exécuter l'une après l'autre (= le code exécutable du programme).

**Note :** *Le CPU cherche les instructions du programme à exécuter, l'une après l'autre, à des positions consécutives de la mémoire; les positions-mémoire servant au stockage des variables doivent donc être réservées "ailleurs" - dans une autre zone.*

**En règle générale, ces zones ("segments") sont stockées dans différentes parties de la mémoire, disjointes les unes des autres.**

**Afin que le programme puisse accéder aux positions de la mémoire utilisées pour la gestion des valeurs des variables, il en doit connaître les adresses ("collaboration" assembleur - éditeur de liens!)**



## Description de la “zone-instructions” d’un programme

- ➔ Le système d’exploitation connaît l’adresse du début du programme - de la première instruction à exécuter (“adresse de transfert”, définie dans l’instruction “end”).

Cette adresse ne doit pas être nécessairement au début de la zone-mémoire allouée au programme.

- ➔ En générale, les premières instructions \*) du programme servent à définir l’état de différents registres de contrôle du CPU.

- ➔ Cette partie “d’initialisation de l’environnement” est suivie des instructions servant à accomplir la tâche proprement dit du programme.

Normalement, le début de cette partie sert à initialiser l’état des variables (tableaux, structures de données) utilisées par le programme.

- ➔ La dernière instruction \*) du programme sert à rendre le contrôle au système d’exploitation.

\*) ... *“premier” et “dernier” dans l’ordre de l’exécution des instructions (= de la “séquence logique”) - qui peut être différent de l’ordre physique du stockage des instructions (= de leur “séquence physique”)*.



## Chapitre 2 :

# L'architecture du CPU INTEL 8086

### Résumé :

- **Architecture: mémoire et CPU**
- **Organisation de la mémoire**
- **Registres du CPU**
- **Instructions**
- **Adressage**

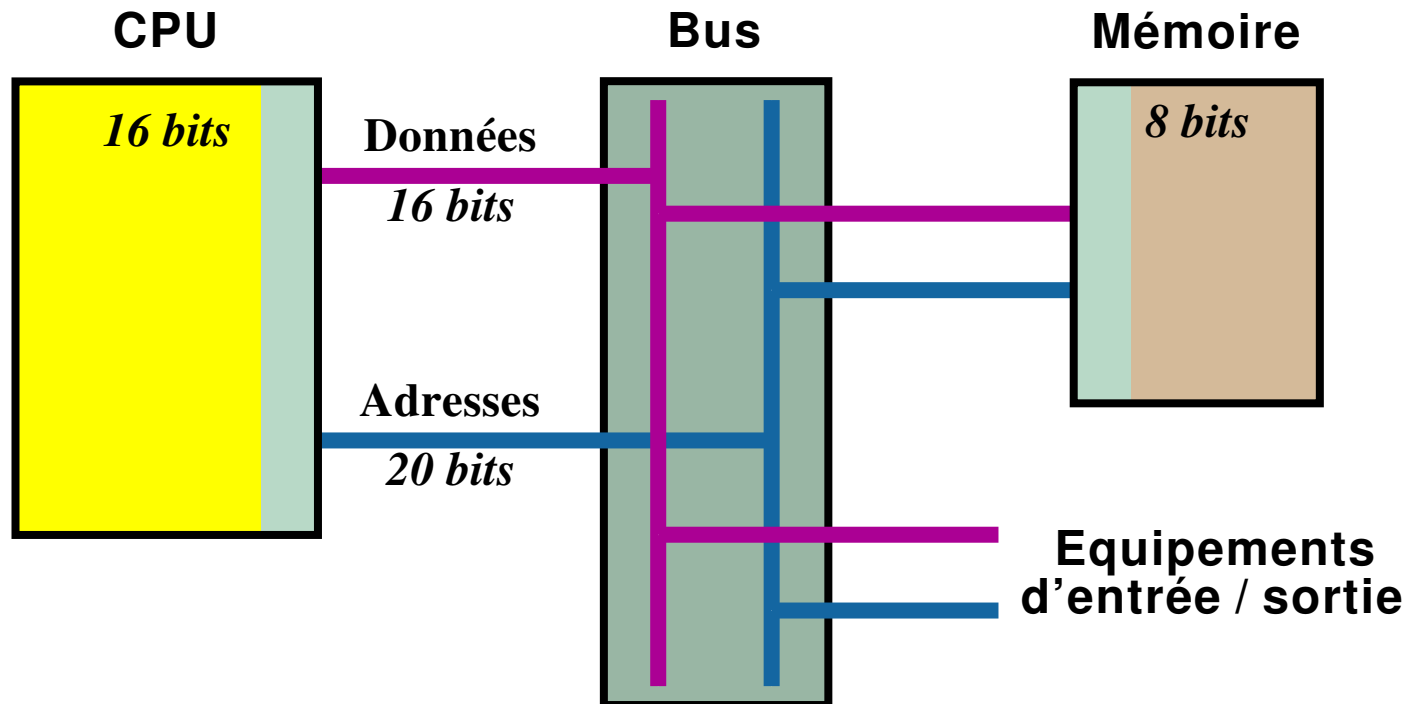


# Configuration d'un système Intel 8086

*Tout système micro-informatique est organisé autour d'un bus*

Architecture de la famille Intel 80xx:

- CPU à 16 bits
- mémoire organisée en bytes (8 bits)
- espace-mémoire avec des adresses représentées sur 20 bits



*Note : Evolution ultérieure de la taille des mots: 16 bits  $\Rightarrow$  32 (80386)  $\Rightarrow$  64 (Pentium) - et de la taille de l'espace d'adresses: 1M  $\Rightarrow$  64 M (80386)  $\Rightarrow$  4 G (80486, Pentium)*



# Mémoire

**Mémoire = espace contigu d'adresses**

→ **adresses** (= numérotation des positions de stockage)  
en général bornées entre 0 et une limite supérieure

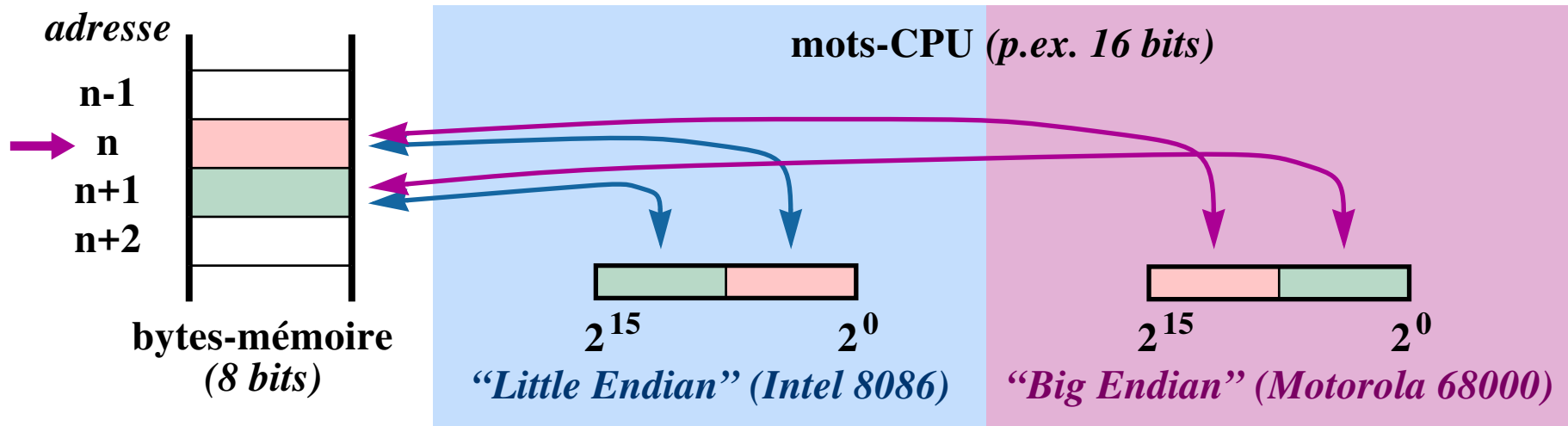
→ **limite supérieure** de la mémoire (la plus grande adresse)

normalement imposée par la plus grande valeur qui peut être représentée en machine:

- soit dans un registre,
- soit dans une instruction.

## Mot-mémoire à l'adresse 'n', "sexe des bytes"

La séquence des bytes stockés pour représenter un mot en mémoire diffère de machine à machine

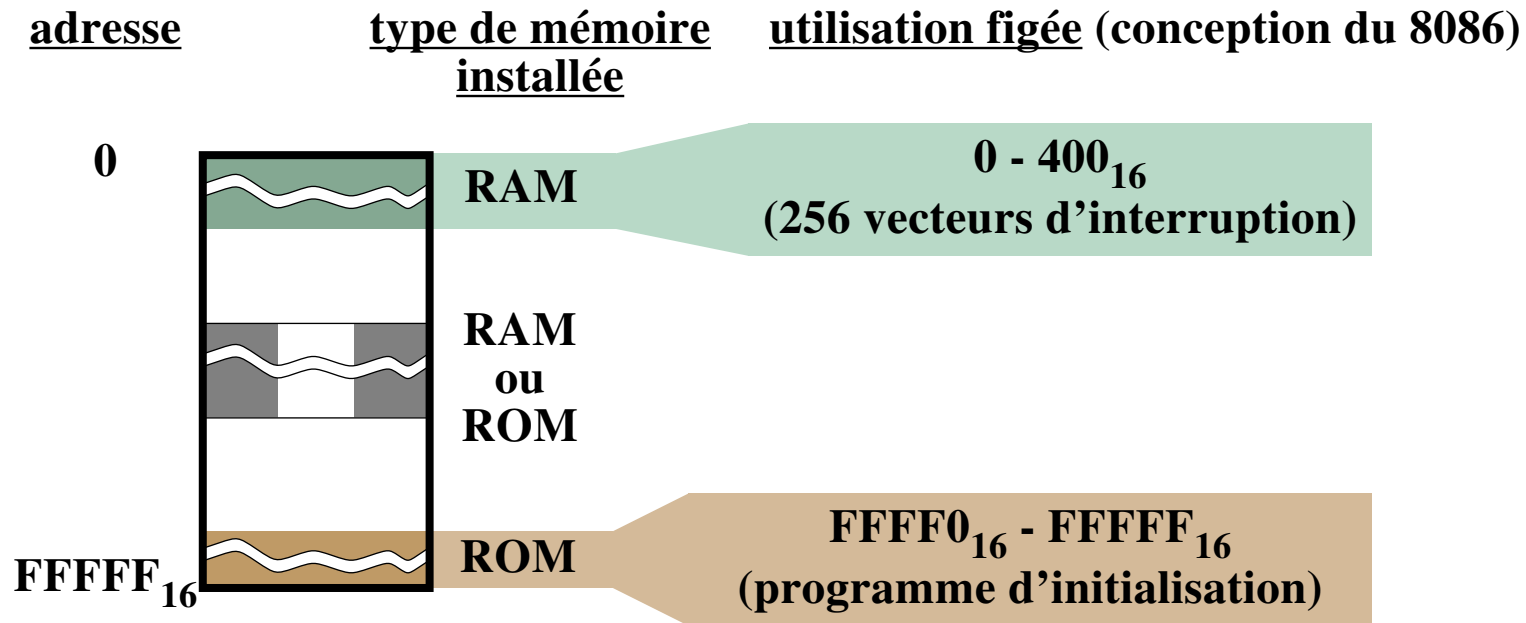




## Implantation physique

**Implantation physique de la mémoire =**  
*sous-ensemble de l'espace d'adresses,*  
*effectivement équipé de mémoire (vive ou morte), organisé en zones contiguës*

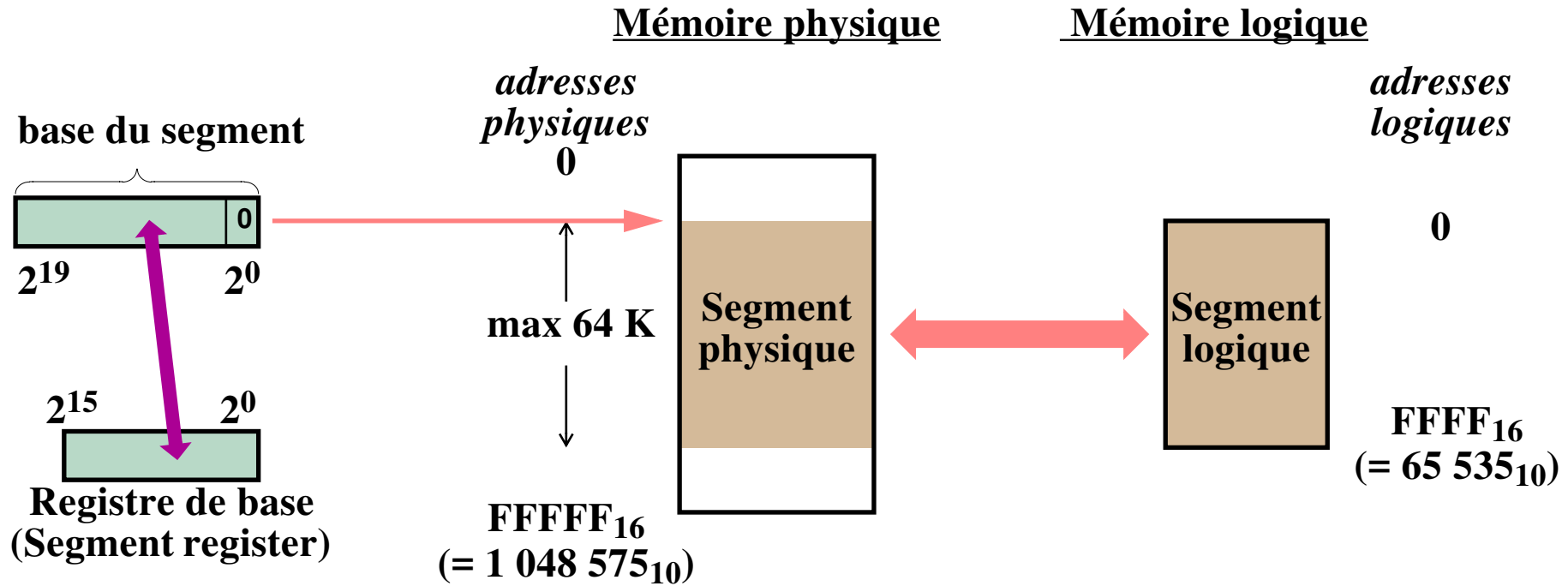
### Plan de la mémoire (Intel 8086)



**Segment de mémoire =**  
*concept pour désigner une partie contiguë de la mémoire*



# Mémoire physique et mémoire logique



$$\begin{array}{ccc}
 \text{adresse physique} & & \text{adresse logique} & & \text{base} \\
 \hline
 a_p & = & a_l & + & (\text{registre de base} * 16_{10})
 \end{array}$$

## Représentation externe des adresses

convention de Intel

exemple:  $1C395 =$  base : adresse-logique

$1000 : C395$  ou

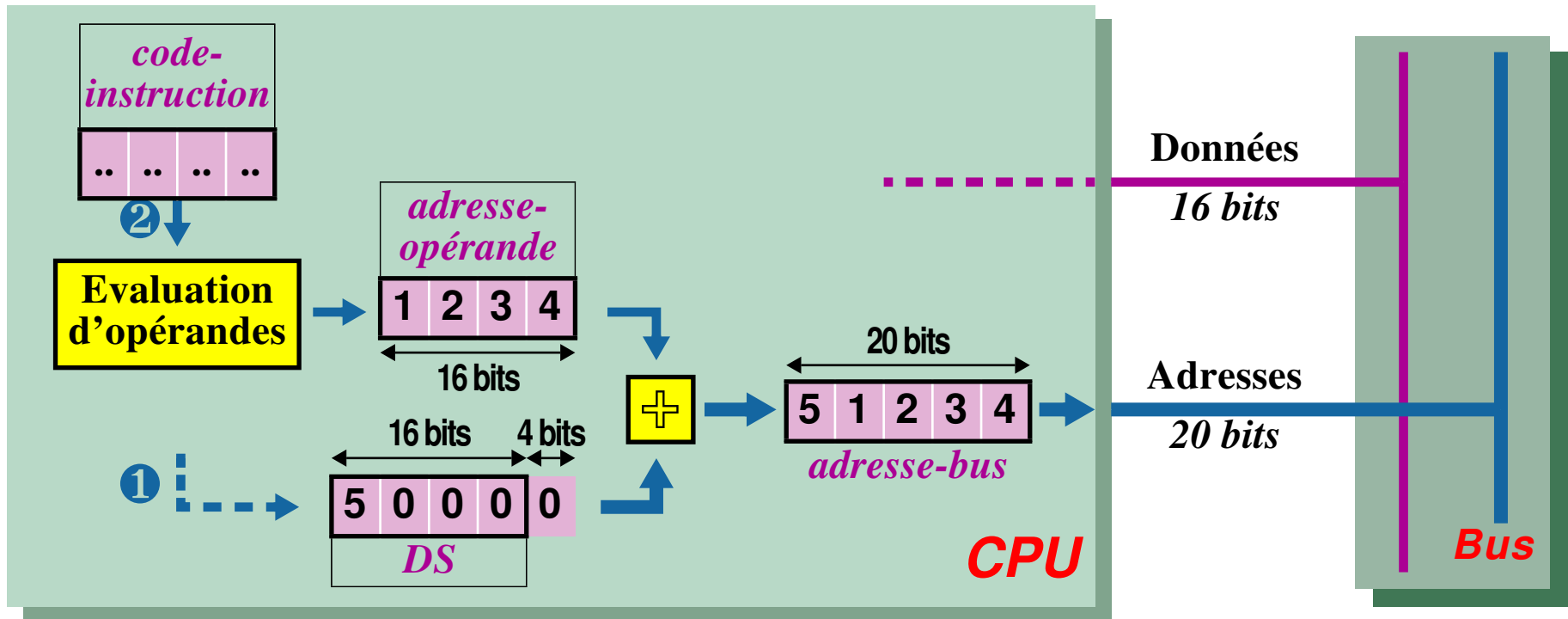
$1B29 : 1105$



# Génération une adresse-bus dans le CPU

## Modèle fonctionnel

illustration:    MOV    AX, 5000H  
① MOV    DS, AX            ; 5000H ... définir la base du segment DS = 50000H  
          ...  
② MOV    AL, dummy        ; 'dummy' représente une donnée dont la place est  
                                 ; réservée à l'adresse 1234H du 'data segment'



= registre

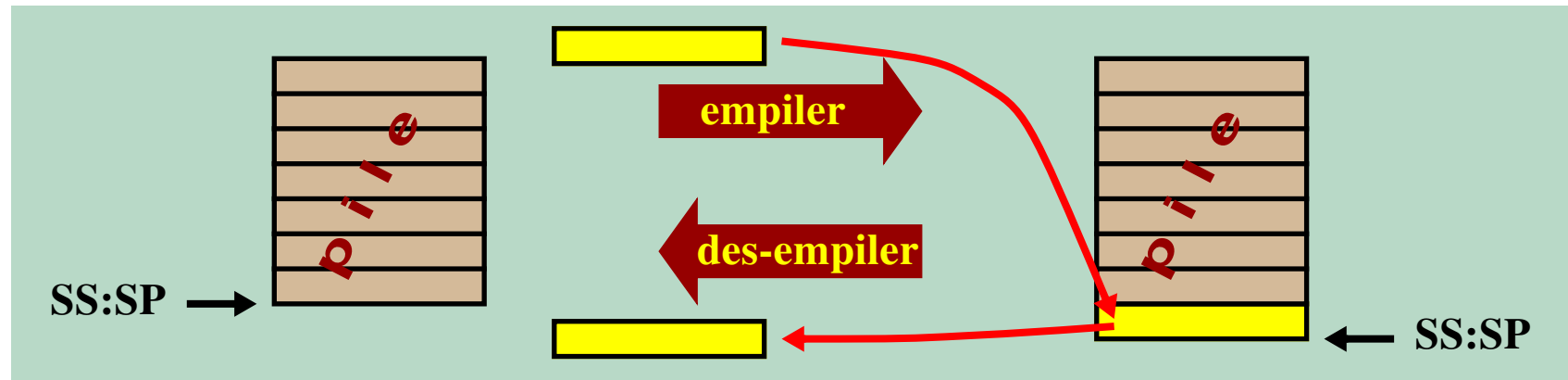


= élément actif

## La pile

L'architecture du 8086 (comme la plupart des autres machines) supporte un mécanisme particulier d'accès à la mémoire: **la pile** ("stack")

- zone de mémoire à part, réservée au stockage temporaire de données;
  - implantation 8086: la valeur SS:SP désigne l'adresse courante sur la pile (= **le pointeur-pile** - "stack pointer");
- le CPU supporte des instructions conçues pour exploiter le mécanisme de la pile;
- concept:
  - zone de mémoire gérée dynamiquement;
  - accès limité à une extrémité de la pile (= le 'haut de la pile');
  - actions de base: **empiler, des-empiler** (i.e. ajout/retrait d'une valeur de l'adresse SS:SP avec mise-à-jour automatique du contenu de SP).



*Note : La pile du 8086 "grandit vers le bas" - le haut de la pile correspond à son adresse la plus petite (autre machines: l'inverse)*

**Segment physique =**  
*zone contiguë de la mémoire physique, déterminée par:*

- des **contraintes d'adressage** (p.ex. accessible sous contrôle d'un certain registre de base),
- des **caractéristiques de la mémoire** physique installée (p.ex. RAM/ROM, vitesse d'accès),
- la **fragmentation de la mémoire** physique.

**Segment logique =**  
*partie du code-objet d'un programme*

- destiné pour le chargement dans un segment physique,
- définie comme une partie de la structure modulaire du code-objet.

*Note : La notion de "mémoire virtuelle" se réfère à une technique particulière pour implanter la mémoire logique dans le cadre d'un système d'exploitation.*

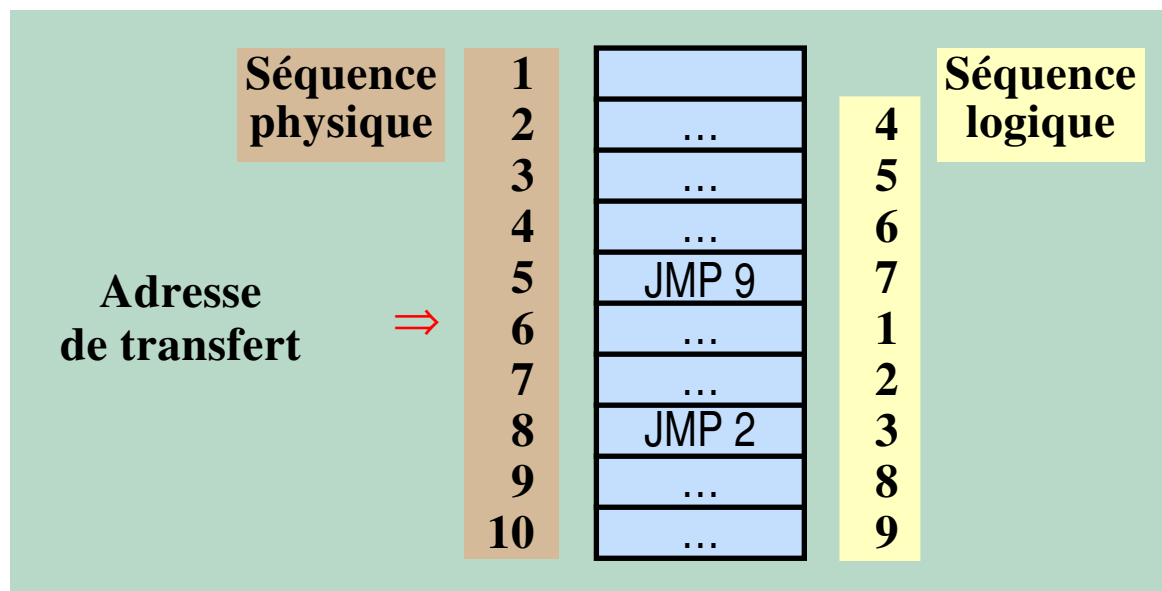


## Séquence physique - séquence logique

La numérotation des positions de stockage (= succession des adresses) implique la **séquence physique des positions de la mémoire**

→ La séquence physique détermine l'ordre de l'accès aux positions de la mémoire lors d'un accès séquentiel, p.ex. par une opération d'entrée / sortie.

### Instructions d'un programme stocké en mémoire



→ **séquence physique des instructions**

suite des positions de la mémoire utilisées pour le stockage des instructions.

→ **séquence logique des instructions**

séquence déterminée par l'exécution des instructions lorsque le CPU les cherche et les interprète.

## Instructions et données d'un programme

...
ADD ...
MOV ...
1234
MOV ...
...

- En exécutant un programme, le CPU cherchera les instructions dans la mémoire, une instruction après l'autre.
- Cette recherche sera faite "bêtement" en suivant la séquence logique.
- Chaque résultat de cette recherche est interprété comme une instruction.

**Donc, attention!**

Lorsque - dans l'exemple - la recherche de la prochaine instruction fournit la valeur '1234', le CPU la comprendra comme le code d'une instruction à exécuter, peut importe "le sens" sous-entendu par le programmeur lors de la création du programme.



**Tri entre instructions et données d'un programme!**

Ce tri doit être fait par le programmeur (assembleur) ou le compilateur (langage élevé): éviter un mélange entre code et données (= représentation de variables).

Normalement, cela aboutit dans le regroupement d'instructions et de données dans des segments (logiques) distincts:

- segment de code,
- segment de données



# Format d'une opération

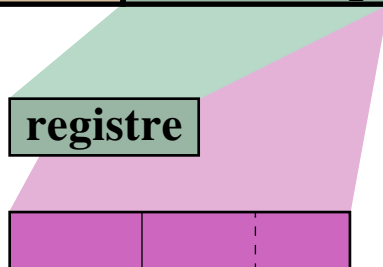
## Champs du code d'une opération

*opération* *opérande1* (*opérande2*) ...

### Opérandes

interprétées en fonction du type de l'opération et de la représentation de l'opérande

- soit comme le contenu d'un registre,
- soit comme une adresse de la mémoire.



opérande =

→ registre ... valeur = contenu du registre

ou

→ adresse ... valeur déterminée par les sous-champs

Les sous-champs d'une opérande interprétée comme une adresse déterminent :

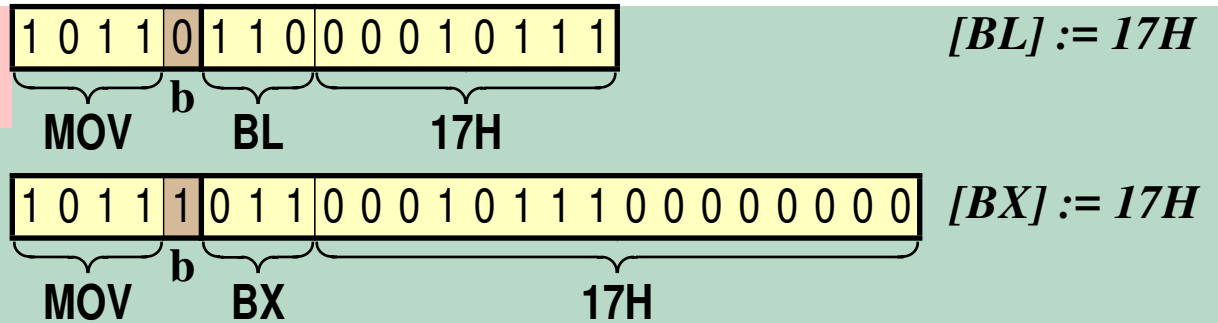
- le **“mode d'adressage”** (= “fonction”, la méthode pour calculer l'adresse et pour en obtenir la valeur de l'opérande) (= “*algorithme d'adressage*”),
- une **valeur constante** à utiliser dans ce calcul ; dans le cas le plus simple (le mode d'adressage “absolu”) cette valeur est directement l'adresse de l'opérande,
- le cas échéant, un ou plusieurs **registres d'indexage**.





# Propriétés des opérations du CPU Intel 8086

1. Opération = chaînes de bytes, longueur variable (1-8 bytes)



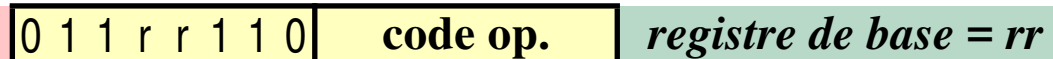
2. Opérations arithmétiques en complément à deux

3. Inversion apparente des bytes dans les opérandes-mot

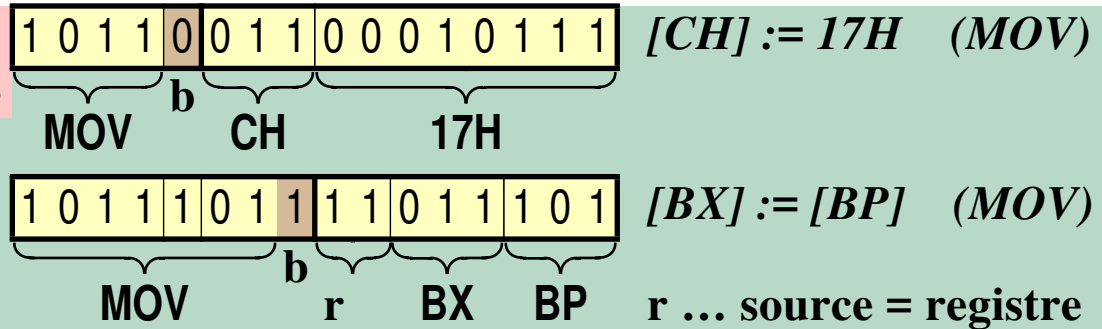
4. Opérandes implicites



5. Instructions avec préfixe



6. Un seul mnémonique  $\Leftrightarrow$  plusieurs instructions



7. Trois opérandes ... deux champs

e.g.  $ADD\ AX, BX$  ... dest. = source1 + source2 (= dest.)

8. Jeu d'instructions relativement riche, complexe, non-orthogonal

[Liste des instructions](#)

# Différences entre instructions-assembleur et instructions-machine

Dans certains cas il existe une différence entre

- la **machine réelle**: comportement déterminé par le CPU
- la **machine abstraite** (“*machine assembleur*”): comportement déterminé par la définition des instructions figurant dans un programme

## 1. Adressage absolu (instructions de branchement Intel 8086) :

machine abstraite ... adresses absolues

*conversion automatique par l'assembleur*



machine réelle (CPU) ... adresses relatives.

## 2. Sélection automatique de l'opération (certains instructions Intel 8086) :

machine abstraite ... une seule instruction

sélection automatique par l'assembleur



machine réelle ... ensemble d'opérations du CPU.



Par exemple (Intel 8086) :

Machine réelle	Machine abstraite
CALL ⇒	CALL intrasegment et CALL intersegment
MOV ⇒	MOV immédiat et MOV direct etc.

*Note : La sélection automatique de l'opération n'est possible que si l'assembleur reconnaît les types qui déterminent le choix, p. ex*

- *étiquette "NEAR" ou "FAR"*
- *valeur absolue ou relogeable*
- *un assembleur ne connaissant pas les types "entier" et "réel" ne peut pas choisir entre les additions correspondantes, p. ex. ADD et FADD.*



# Registres du CPU Intel 8086

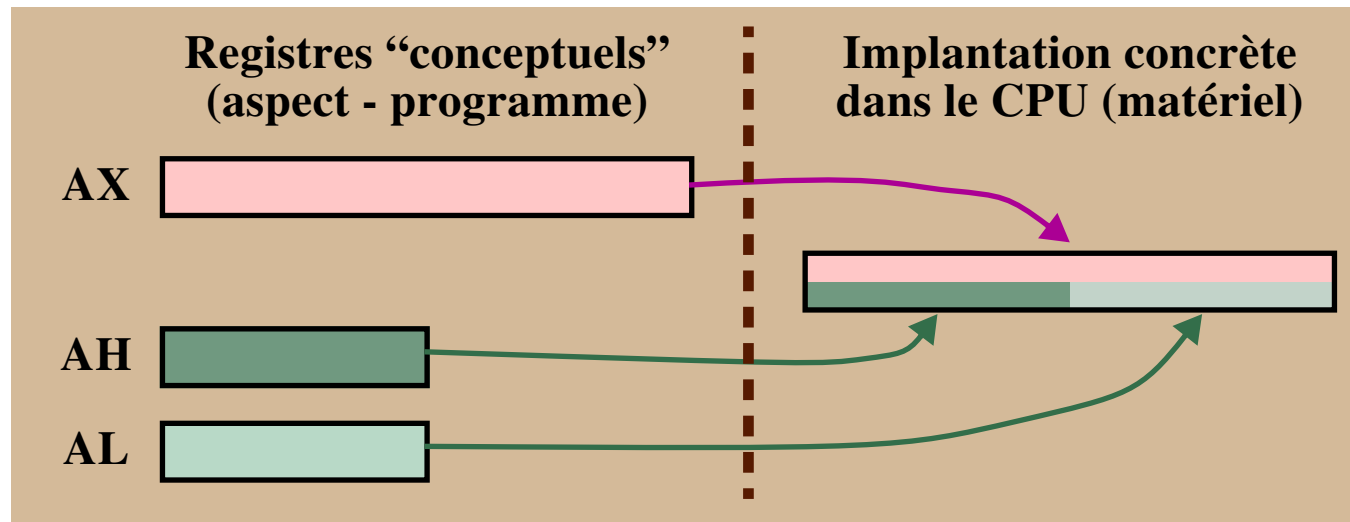
## Registres-opérandes et d'indexage

reg. 16 bits		reg. 8 bits		utilisation implicite
mném.	code	mném.	code	
AX	000			accumulateur (opérations arithmétiques), opérande E/S
		AL	000	comme AX (mais opérande-byte)
		AH	001	aucune
CX	001			compteur pour itérations (boucles, opération sur chaînes)
		CL	010	compteur pour décalage
		CH	011	aucune
DX	010			porte E/S, extens. de AX → 32 bit (DX = msb) (MUL, DIV)
		DL	100	aucune
		DH	101	aucune
BX	011			indexage <sup>1)</sup> , XLAT
		BL	110	aucune
		BH	111	aucune
SP	100			indexage pile (= pointeur pile)
BP	101			indexage <sup>1)</sup> , accès données par registre segment = SS (registre)
SI	110			indexage-source [DS] <sup>2)</sup> (opérations sur chaînes)
DI	111			indexage-destination [ES] <sup>2)</sup> (opérations sur chaînes)



## Recouvrement des registres à 8 et à 16 bits

L'implantation effective d'un  
**registre-byte**  
se recouvre avec la partie supérieure ou inférieure du  
**registre-mot correspondant**



**Conséquences sur l'extension du bit de signe !**

**Par exemple :**

**MOV AX, -3 ; AX devient 0FFFD**

**MOV AL, -3 ; AX devient 0xxFD (0xx = valeur antérieure)**

*Note : Ce concept a probablement son origine dans la volonté d'Intel de garantir un important degré de portabilité de programmes du 8080 vers le 8086*



## Registres de base (“segment registers”)

nom	mném.	code	utilisation implicite (= <i>choix automatique d'un registre de base par le CPU</i> )
Extra Segment	ES	00	opérations sur chaînes (chaîne-destination)
Code Segment	CS	01	code à exécuter
Stack Segment	SS	10	opérations utilisant la pile (y compris les appels de procédures et les interruptions) accès aux opérandes en mémoire si indexage par BP
Data Segment	DS	11	accès aux opérandes en mémoire sauf si indexage par BP opérations sur chaînes (chaîne-source)

### Compteur ordinal

(= “instruction pointer”, “IP”)

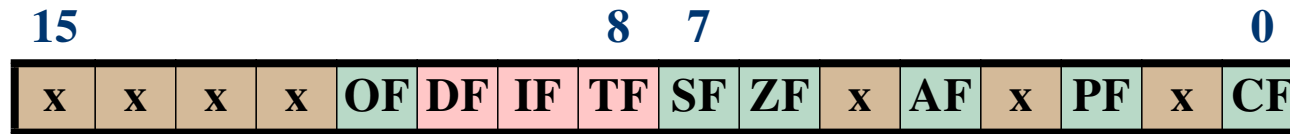
- Seulement indirectement accessible au programme (**JMP**, **CALL**, etc.)
- Autres machines : registre(s) d'état +compteur ordinal (= “Processor StatusWord”, “PSW”)



## Registre d'état ("flag register")

Le registre d'état contient une liste de bits représentant l'état du CPU

- *bits de contrôle* (état de fonctionnement du CPU, déterminé par des instructions de contrôle, respectivement le système d'interruption),
- *bits arithmétiques* (la valeur décrit le résultat de l'exécution d'une opération).



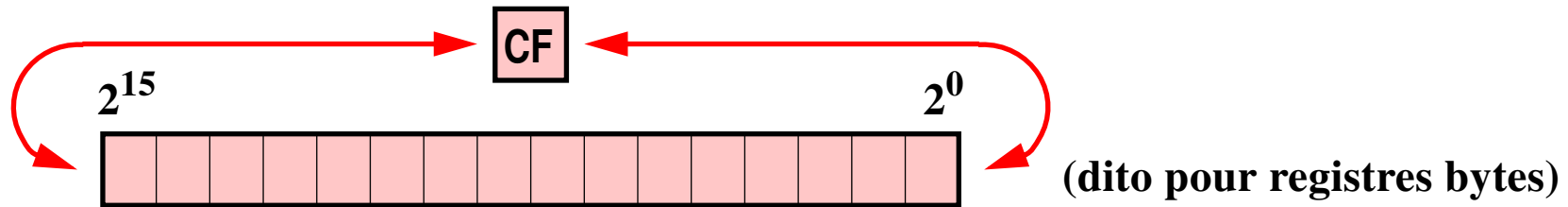
Bit (nom)	Signification: 1 si ...
<b>OF</b> ("overflow")	report MSB ↔ bit devant MSB
<b>DF</b> ("direction")	marche-arrière (op. sur chaînes)
<b>IF</b> ("interrupt")	les interruption sont acceptées
<b>TF</b> ("trap")	interruption après chaque opération
<b>SF</b> ("sign")	résultat négatif (MSB = 1)
<b>ZF</b> ("zero")	résultat = 0
<b>AF</b> ("arithmetic")	report bit #4 ↔ bit #3
<b>PF</b> ("parity")	la parité du résultat est paire
<b>CF</b> ("carry")	report du / vers le MSB

*Note :*      MSB = "most significant bit" (bit de poids le plus fort)  
                   LSB = "least significant bit" (bit de poids le plus faible)

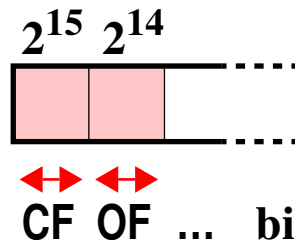


## Le rôle du bit “carry” (report arithmétique, décalage)

### Opérations de rotation et le bit “CF”



### Opérations arithmétiques et les bits “CF”, “OF”



bits modifiés en cas de report du / vers le bits du plus grand poids:

- registres à 16 bits: report des bits  $2^{15}$  ou  $2^{14}$
- registres à 8 bits: report des bits  $2^7$  et  $2^6$





# Quelques opérations typiques

Mném.	Opérandes	Sémantique
<b>SUB</b>	dest 1)      source 1)	dest := dest - source
<b>AND</b>	dest 1)      source 1)	<i>dest := dest <math>\wedge</math> source</i>
<b>SHL</b>	dest 1)      count 1)	<i>dest := dest * 2<sup>count</sup></i>
<b>CMP</b>	dest 1)      source 1)	dest - source 4)
<b>TEST</b>	dest 1)      source 1)	dest $\wedge$ source 4)
<b>JMP</b>	valeur	IP := IP + valeur
<b>JE</b>	valeur	si ZF = 1 ... IP := IP + valeur
<b>MOV</b>	dest 2)      source 2)	dest := source
<b>PUSH</b>	source 3)	empiler source
<b>CALL</b>	valeur	(pas 1) empiler IP, (pas 2) IP := IP + valeur
<b>CLI</b>		désactiver le système d'interruption (IF := 0)
<b>OUT</b>	valeur	envoyer AL vers l'équipement avec l'adresse "valeur"

**Notes:**  $\wedge$ ... "et" logique

1)... registre 8-bits, registre 16-bits, position-mémoire, valeur immédiate ("source" seulement)

2)... comme 1); en plus: registre de base

3)... registre 16-bits, registre de base, position-mémoire

4)... ne modifie pas le registre-résultat (seulement les bits arithmétiques du registre d'état sont touchés)

 [Liste des instructions](#)



# Adressage Intel 8086

## Principes

### → Adressage implicite

par exemple : opération sur pile (PUSH, POP)

### → Adressage explicite

information fournie dans le champ d'opérande, évaluée en deux phases :

#### 1. **Calcul de l'adresse effective** ( $adr_{eff}$ ) pour un des modes :

- adressage absolu
- adressage indexé
- adressage relatif

#### 2. **Recherche de l'opérande** en utilisant $adr_{eff}$ pour un des modes :

- adressage immédiat
- adressage direct
- adressage indirect

*Note : La sélection d'un mode d'adressage fait partie de l'information figurant dans le champ d'opérande.*

*L'Intel 8086 ne supporte pas tous ces modes et seulement certaines combinaisons entre les modes de la phase 1 et de la phase 2 (voir "Modes d'adressage").*



## Modes d'adressage

### 1. Modes pour le calcul de l'adresse effective

- **adressage absolu**  $\text{adr}_{\text{eff}} = \text{const.}^{1)}$
- **adressage indexé**  $\text{adr}_{\text{eff}} = \text{const.}^{2)} + r_i [ + r_k ]^{3)}$

*Note : Le double indexage est une option particulière du CPU Intel 8086.*

*Dans certaines conditions, l'adressage indexé permet une incrémentation ou décrémentation automatique avant ou après le calcul de l'adresse effective (Intel 8086 : opérations itératives tel que MOV B etc.).*

- **adressage relatif**  $\text{adr}_{\text{eff}} = \text{const.}^{2)} + \text{IP}$   
 → *Le CPU Intel 8086 ne permet l'adressage relatif que pour les instructions de branchement (JMP, CALL, JE etc.); voir remarque sous "Syntaxe : adressage"*

- 1) const. ... contenu d'un sous-champ particulier, interprété comme une valeur sans signe
- 2) const. ... contenu d'un sous-champ particulier, interprété comme une valeur positive ou négative (complément à 2)
- 3) [ ... ] ... champ optionnel
- $r_i, r_k$  ... contenu d'un registre d'indexage
- IP ... contenu du compteur ordinal



## 2. Modes pour la recherche de l'opérande

- **adressage immédiat** opérande =  $\text{adr}_{\text{eff}}$   
→ *Le CPU Intel 8086 ne supporte l'adressage immédiat qu'en combinaison avec le mode absolu*
- **adressage direct** opérande =  $[ \text{adr}_{\text{eff}} ]^{4)}$
- **adressage indirect** opérande =  $[ [ \text{adr}_{\text{eff}} ] ]^{4)}$   
→ *L'Intel 8086 ne supporte pas l'adressage indirect. Certains CPU permettent une multiple indirection.*

4) [ adresse ] ... obtenir le contenu de la position de la mémoire "adresse"



## Combinaison des modes (phase 1 et 2)

En principe, un CPU devrait permettre de combiner chaque mode traité en phase 1 (“calcul de l’adresse effective”) avec chaque mode traité en phase 2 (“recherche de l’opérande”).

L’architecture du CPU Intel 8086 cependant ne supporte pas toutes les combinaisons théoriquement possibles

### Modes d’adressage possibles (Intel 8086)

mode	immédiat	direct	indirect
absolu	✓	✓	✗
indexé	✗	✓	✗
relatif	✗	✓	✗

✓ ... *combinaison possible*  
 ✗ ... *combinaison défendue*



# Modes d'adressage (exemples Intel 8086)

**Mode immédiat** assembleur: *MOV BX,17H*

opérande 1 = mot registre BX

opérande 2 = 17H (implicitement: opérande2 = constante)

opération =  $MOV_{imm.}$

**Mode direct, absolu** assembleur: *MOV BX, adresse*

opérandes = mot opérande 1 = BX (implicitement: opérande1 = registre)

opération =  $MOV_{registre, direct}$

opérande 2 = adresse absolue

**Mode direct, indexé** assembleur: *MOV BX,4[SI]*

opérandes = mot opérande 1 = BX (implicitement: opérande1 = registre)

opération =  $MOV_{registre, direct}$

opérande 2 = adresse indexée (SI), constante-byte

valeur immédiate = 4



## Chapitre 3 :

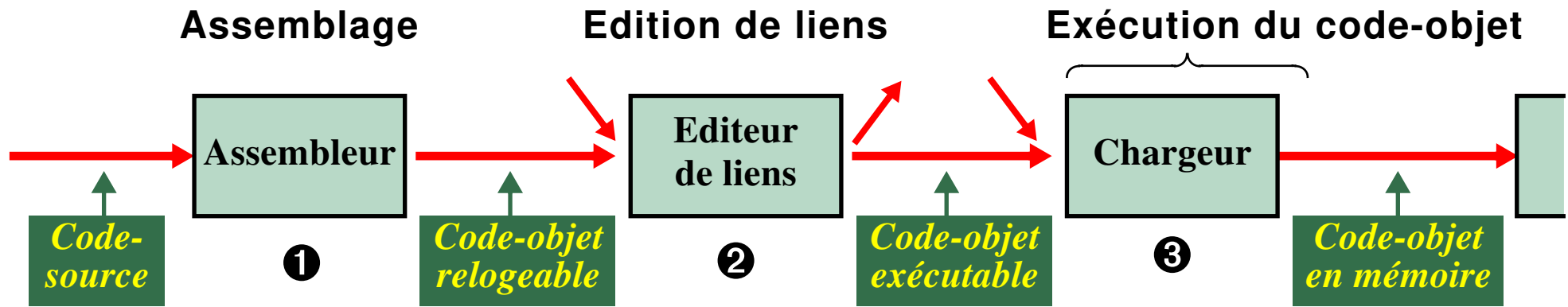
# Principes d'un assembleur

### Résumé :

- **Concepts généraux**
- **Instructions**
  - Représentation, syntaxe des instructions
  - Adressage
- **Génération de code**
  - Compteurs de positions
  - Segments et gestion des registres de base
- **Instructions**



# Assembleur : introduction



## Code-source:

**Description symbolique d'un programme**

- des instructions-machine,
- des directives pour l'assembleur,
- (des directives pour l'éditeur de liens).

## Code-objet relogeable:

**Description binaire d'un programme**

- des instructions-machine,
- de l'information pour l'éditeur de liens (relocation, chargement, exécution).

## Code-objet absolu:

**Représentation binaire d'un programme**

- des instructions-machine,
- de l'information pour le chargement et l'exécution du programme-objet.





## Caractéristiques d'un assembleur

- Représentation à l'aide de ***mnémoniques***.
- Représentation de ***valeurs constantes***:
  - constantes numériques,
  - chaînes de caractères.
- Gestion de l'***espace-mémoire, symboles***.
- ***Méta assemblage***:
  - “fonctions” de l'assembleur,
  - assemblage conditionnel,
  - assemblage répétitif,
  - macro-assemblage.
- Assemblage en plusieurs ***“passes”*** (en général 2 passes)

*Notes:* ➤ pour un type de CPU ... plusieurs assembleurs,  
... syntaxes différentes,  
... mnémoniques différents !

*par exemple :* MASM (Microsoft),  
Turbo Assembleur (Borland)

➤ *l'assembleur accomplit son travail en lisant les instructions du programme-source l'une après l'autre; il s'arrête lorsqu'il trouve l'instruction “END”.*



# Syntaxe

## Forme générale d'une instruction MASM

***[[ étiquette [ : ]♦] opération [♦ opérande ] [ ; commentaire ] ¶***

### Syntaxe:

- représentation d'une instruction par un enregistrement,
- décomposition d'une instruction en champs, séparés de caractères-séparateurs,
- si nécessaire, décomposition d'un champ en sous-champs.

#### *Notation utilisée :*

♦ ... *espace ou tabulation ("whitespace") :*

*des espaces ou tabulations (supplémentaires) peuvent figurer au début et à la fin des champs et sous-champs*

[ ] ... *optionnel*

¶ ... *fin de ligne*

par exemple :

**CALC :    MOV    CX , 7   ; définir le compteur**



## Signification des champs de l'instruction

### Champ d'étiquette

**définition d'un symbole** dont le nom est la chaîne de caractères du champ d'étiquette :

- max 31 caractères {A..Z} {a..z} {0..9} {.?@\$},
- {0..9} défendu comme 1er caractère,
- “.” uniquement permis comme 1er caractère,
- pas de distinction entre minuscules et majuscules,
- terminaison par “:” ... le symbole aura le type “near” (valeur sans partie “segment”, voir aussi “Variables, symboles”);

la valeur attribuée au symbole dépend du champ d'opération.

### Champ d'opération

**opération à effectuer**, indiquée à l'aide d'un “mnémonique” ; deux significations possibles :

- **opération** ...  
élément du répertoire d'instructions du CPU dont le code-objet est à générer,
- **pseudo-opération** (“pseudo-instruction”) ...  
directive au programme “assembleur” (ne produit pas forcément du code-objet).

☛ Si le premier champ d'une instruction est une (pseudo-)opération, ce champ sera considéré comme un champ d'opération - sinon comme un champ d'étiquette.



## Champ d'opérande

### **argument(s) pour la (pseudo-)opération :**

1. le champ d'opérande se décompose en sous-champs séparés de virgules (suivies comme option d'un ou plusieurs caractères "white space") :

**sous-champ-1 [ , [♦] sous-champ-2 [ , [♦] ... ] ]**

*rappel: ♦ ... espace ou tabulation ("whitespace")*

2. le sous-champs décrivant l'opérande-destination précède celui décrivant l'opérande-source, donc :

**opération destination, source**

*Note : cette ordre est inverse pour les assembleurs de la plupart des autres machines, par exemple celui du Motorola 68000*

3. la signification spécifique du champ d'opérande et de ses sous-champs dépend de l'opération et, par conséquent, est déterminée par le champ d'opération.

## Champ de commentaire

### **texte explicatif :**

champ sans signification syntaxique et sémantique pour l'assembleur; chaîne de caractères terminée par la fin de l'instruction (= fin de ligne)

**; commentaire**



## Syntaxe : constantes

### Format des constantes numériques

*chiffre [ chiffre ... ] [ selecteur ]*

type	selecteur	chiffre	exemple
binaire	B	{0,1}	00011101B
octal	O ou Q	{0..7}	035O
hexadécimal	H	{0..9}{A..F}{a..f} <sup>1)</sup>	1dH
décimal	D ou rien <sup>2)</sup>	{0..9}	29

1) contrainte: premier chiffre = obligatoirement {0..9}

2) base utilisée par défaut pour la conversion (initialisée au début de l'assemblage à 10), peut être changée par la directive

**.RADIX n (2 ≤ n ≤ 16)**

### Format des chaînes de caractères

(constantes alpha-numériques)

*séparateur [ caractère [ caractère... ] séparateur*

séparateur ... { ' " }

caractère ... tout caractère du jeu ASCII sauf séparateur

par exemple :

"abc' \1X" ou ' abc"e'



## Spécification d'adresses

Décomposition de l'opérande désignant une adresse en sous-champs; chaque adresse a la forme:

**[ *segreg*: ][ *valeur* [ [ *indreg* [ *+indreg* ][ *+valeur* ] ] ] ]**

**[ ... ] = [ et ]** indiquent sous-champs optionnel

**[ ... ] = [ et ]** sont des caractères significatifs

### Sélection du mode d'adressage

Le choix des modes d'adressage disponibles est déterminé par l'architecture du CPU. La syntaxe de l'assembleur permet de sélectionner pour chaque opérande un mode spécifique.

Le mode d'adressage est déterminé par :

- la ***syntaxe particulière*** d'une opérande  
(*par exemple adressage indexé*),
- la ***présence*** ou absence de ***certaines sous-champs***  
(*par exemple un champ avec un registre d'indexage*),
- les ***caractéristiques*** associées par l'assembleur à ces ***sous-champs***  
(*par exemple le type du champ "valeur"*).



## Mode d'adressage

### Calcul de l'adresse effective

- ☛ l'assembleur reconnaît le mode "**adressage indexé**" par la présence d'au moins un sous-champ spécifiant un registre d'indexage (syntaxe = [*registre-d'indexage* ])

par exemple:     4 [SI]  
                   [SI+4]  
                   SYMBOL [SI+BP]  
                   ES : [DI+BX+3]

- ☛ sinon le mode "**adressage absolu**" est pris (le programmeur ne peut pas choisir le mode relatif) ;

par exemple:     SYMBOL  
                   ES : SYMBOL  
                   ' '  
                   0fff

- ☛ l'assembleur simule la disponibilité d'instructions de branchement permettant l'adressage absolue (JMP, CALL, JE ...), bien que les instructions correspondant de l'Intel 8086 ne permettent que **l'adressage relatif** :

l'assembleur effectue automatiquement une conversion en adressage relatif

par exemple:     JMP            LABEL  
                   CALL         EXIT



## Recherche de l'opérande

- ☛ l'assembleur reconnaît le mode **“adressage immédiat”** par l'absence de tous sous-champ spécifiant un registre d'indexage (terme en notion [*registre*]), mais seulement si le champ **“valeur”** représente une constante (par opposition à une adresse relogeable) ;

par exemple:     '   '  
                  0ffff

- ☛ sinon le mode **“adressage direct”** est choisi ;

par exemple:     *symbole\_relogeable*  
                  4 [SI]

- ☛ en cherchant une valeur dans la mémoire, l'Intel 8086 utilise des **règles par défaut pour sélectionner un registre de base** (voir “Registres de base”) ;

si un sous-champ spécifiant un registre de base (nom de registre-segment suivi de “:”) est rencontré, l'assembleur composera un code-préfixe (“Segment Prefix Code”) qui, lors de l'exécution, forcera le CPU d'utiliser ce registre de base.

par exemple:     ES : SYMBOL

*Note : Contraintes imposés par l'architecture du 8086 quant à la disponibilité et au choix des combinaisons de modes! voir aussi: adressage Intel 8086, combinaison des modes).*

- ➔ *Un non-respect de ces contraintes est traitée comme une erreur de programmation, l'assembleur fournira un message d'erreur correspondant.*
- ➔ *par exemple:     ES:7*





# Génération de code-objet

La génération de code exécutable par un assembleur ou par un compilateur aboutit, en général, dans  
*la création de plusieurs zones disjointes de code-objet.*

## Tables de code-objet

- L'assemblage - l'action de générer du code-objet - peut être considérée comme le remplissage d'une ou de plusieurs **"tables de code-objet"**; une telle table est gérée pour chaque segment.
- A chacune des tables de code-objet il faut associer un **"compteur de position"**, qui représente la position (= l'adresse relative dans le segment) où le prochain byte généré sera placé.

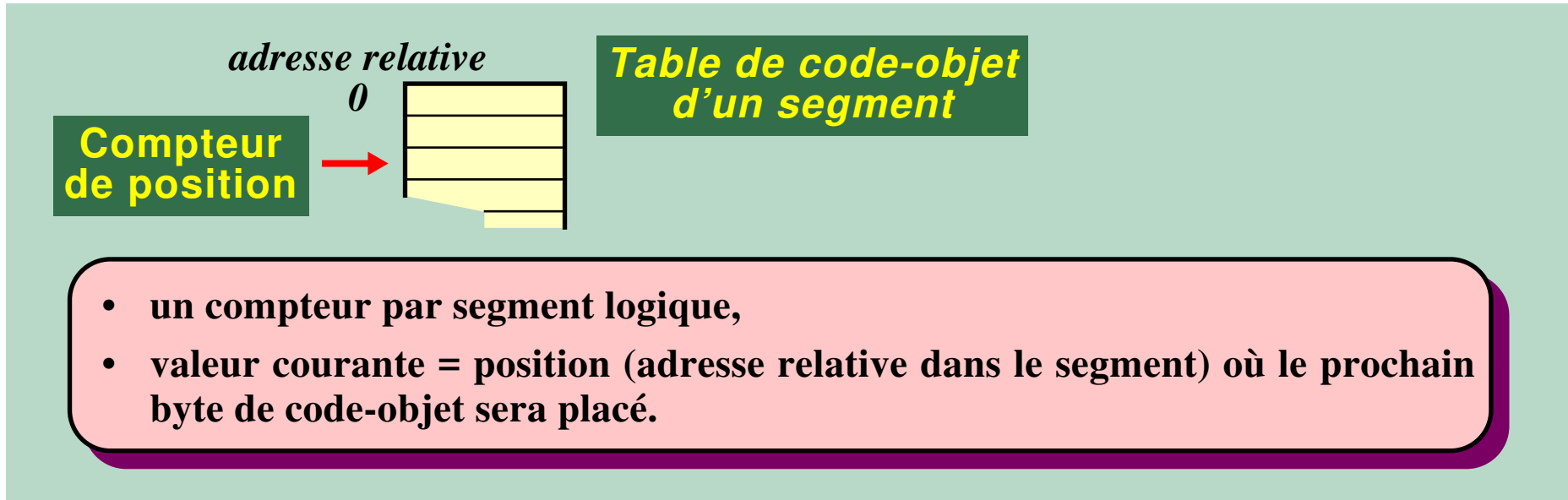
## Segment logique

- En créant du code-objet, l'assembleur regroupe le code généré (instructions, données) en différentes zones contiguës, indépendantes l'une de l'autre - des **"segments logiques"**.
- Le contenu de chaque segment logique représente un **espace d'adresses indépendant**.
- L'emplacement des segments logiques dans la mémoire et la position d'un segment logique par rapport à l'autre ne seront déterminés que lors de l'édition de liens.



## Compteur de position

### Rôle du compteur de position



### Contrôle du compteur de position

**ORG**     *valeur*

Mettre la valeur du compteur de position du segment courant à “valeur”

**EVEN**

Assurer que la valeur du compteur de position du segment courant soit paire; si nécessaire, générer un byte de code-objet (“no-op”)

### Valeur du compteur de position

La fonction “\$” fournit à tout instant la valeur courante du compteur de position du segment courant.

## Compteur de position et compteur-ordinal

Il y a une relation étroite entre le compteur de position du segment de code (tel qu'il est géré au moment de l'assemblage) et le compteur-ordinal du CPU (l'adresse de l'instruction que le CPU cherche au moment de l'exécution):

→ *le compteur de position correspond à la valeur du compteur-ordinal au moment où l'instruction sera exécutée.*

## Valeur initiale du compteur-ordinal lors de l'exécution

END    *valeur*

- Utiliser l'adresse "valeur" pour déterminer *l'adresse de transfert*.
- Considérer l'instruction comme la dernière du programme à traduire.

L'adresse de transfert sera mémorisée dans le code-objet du programme généré. Cette valeur sera utilisée par le système d'exploitation pour initialiser le compteur-ordinal (= déterminer la première instruction du programme à exécuter).

L'adresse de transfert spécifie l'adresse de la première instruction exécutée suite au chargement du programme



# Contrôle des segments

## L'instruction SEGMENT

La pseudo-instruction **SEGMENT** permet de contrôler la relation entre code et segments logiques. Elle sert à :

1. contrôler le **placement du code-objet** dans des segments spécifiques;
2. **associer les symboles** représentant des adresses à un segment en considérant leur valeur comme un déplacement par rapport au début du segment;
3. spécifier des **directives pour l'éditeur de liens** (nom du segment, champs d'opérande de l'instruction **SEGMENT** déterminant le traitement du segment par l'éditeur de liens); ces informations sont passées telles quelles.

Utilisation de l'instruction **SEGMENT**:

<i>nom</i>	<b>SEGMENT</b>	<i>opérande (s)</i>
	⋮	
<i>nom</i>	<b>ENDS</b>	



# Association du code-objet à un segment logique

## Décomposition du code en segments logiques

Le code-objet correspondant au bloc d'instructions figurant entre les pseudo-instructions **SEGMENT** et **ENDS** sera placé dans un segment dont le nom figure dans le champ d'étiquette.

## Gestion à l'aide d'un compteur de position

Un "compteur de position" est associé à chaque segment.

Ce compteur de position vaut initialement (= lors de la première sélection d'un segment) 0.

Si le segment est dé-sélectionné pour sélectionner un autre segment, la valeur du compteur est sauvée jusqu'au moment où le segment est de nouveau sélectionné.

## Alternance entre segments

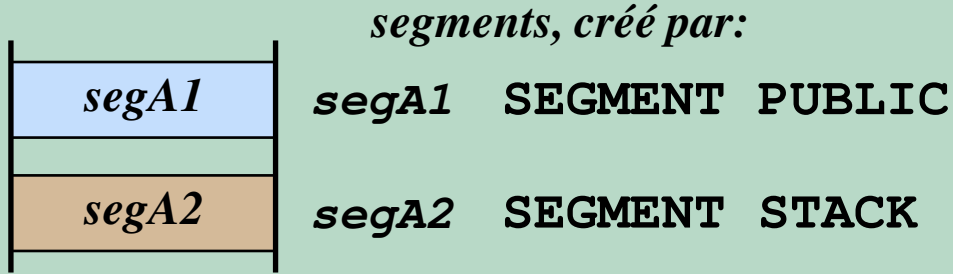
Un programme peut alterner entre différents segments pour y générer de code:

- l'instruction **SEGMENT** permet de re-ouvrir un segment déjà existant (donc, **SEGMENT** soit crée un nouveau segment, soit ouvre un segment en vue d'y ajouter de code supplémentaire);
- suite à une telle re-ouverture, le compteur de positions pointera à la fin des données déjà existant dans le segment;
- ne pas oublier l'instruction **ENDS** avant une telle opération, elle permet de (temporairement) clore l'ancien segment!

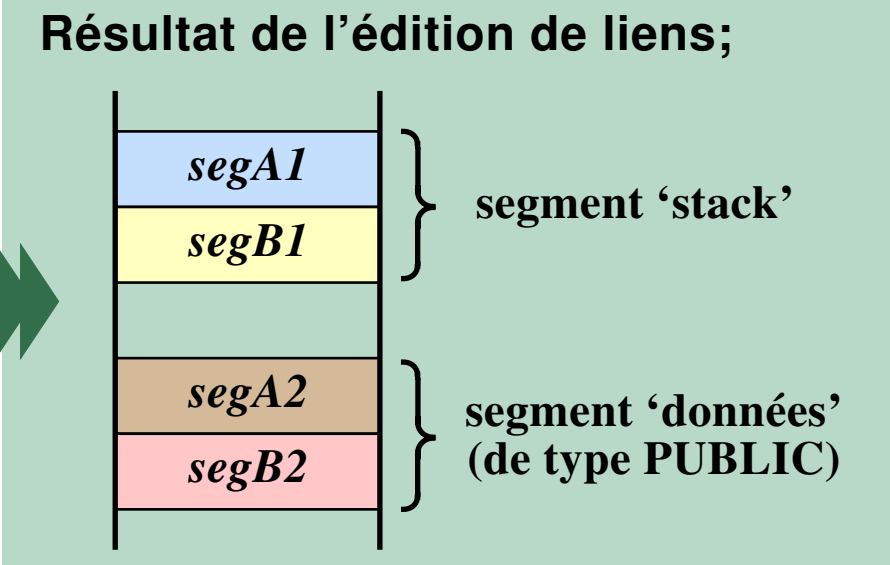
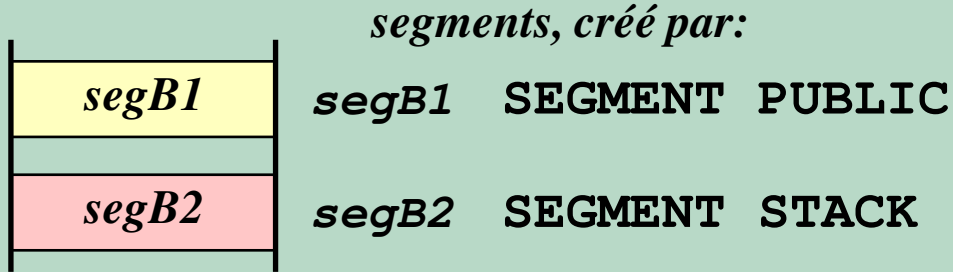


# Exemple: résultat du travail de l'éditeur de liens

## code-objet du module A;



## code-objet du module B;



## Placement de segments

Les opérandes de l'instruction **SEGMENT** déterminent la manière dont l'éditeur de liens traitera le segment :

```

nom          SEGMENT  [♦ COMMON
                        [♦ PUBLIC
                        [♦ STACK
                        [♦ MEMORY
                        AT ♦ adr
                        [♦ BYTE
                        [♦ WORD
                        [♦ PARA
                        [♦ PAGE
                        ][♦ classe]]
```

*Note : rappel: ♦ ... espace ou tabulation ("whitespace")*

**COMMON** tous les segments avec l'étiquette "*classe*" seront placés à la même adresse de base (= ils se recouvriront) ; des zones du type "**COMMON**" avec différents noms ("*classe*") seront placés l'un derrière l'autre ;

**PUBLIC** tous les segments avec ce qualificatif seront regroupés dans un seul segment-résultat, l'un derrière l'autre ;

**STACK** un seul segment avec ce qualificatif est accepté, il est destiné à la gestion de la pile ;

**MEMORY** le premier segment portant ce qualificatif sera placé à une position de mémoire en dessus de tout autre segment; s'il y a d'avantage de segments de ce genre, ils seront traités comme les segments du type "**COMMON**" ;

**AT ♦ adr** les étiquettes définies dans un tel segment sont définies comme étant relatives à la valeur ( "*adr*" / 16 ) \* 16.



## Alignement de l'adresse de base d'un segment

Le segment sera placé à une adresse alignant le positionnement du segment de manière spécifique:

mot-clé	alignement sur	adresse modulo
<b>BYTE</b>	... frontière de bytes	1
<b>WORD</b>	... frontière de mots	2
<b>PARA</b>	... frontière de paragraphes	16
<b>PAGE</b>	... frontière de pages	256

*Note : Les opérations nécessaires pour le regroupement et le placement des segments seront entreprises par l'éditeur de liens - l'assembleur ne fait que placer les directives correspondantes dans le code-objet.*

*Le mode "BYTE" est utilisé par défaut, les segments seront donc placés directement l'un derrière l'autre.*

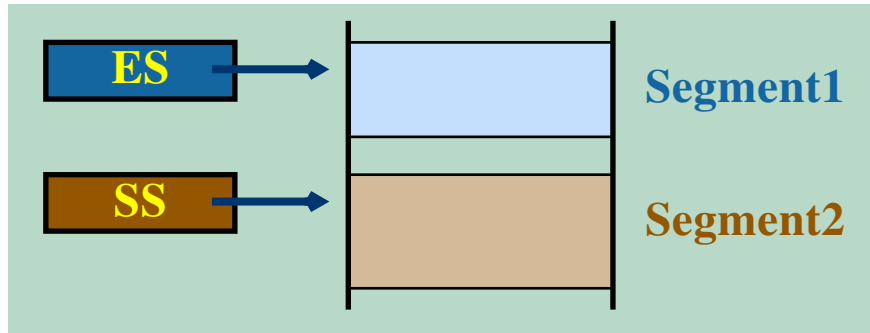
*En mode "WORD", par exemple, un byte supplémentaire sera inséré au besoin, ce qui assure que les adresses paires d'un segment correspondent à des adresses paires de la mémoire physique; ceci permet d'écrire du code "rapide", exploitant le fait que le 8086 utilise un seul cycle du bus pour chercher une opérande-mot si elle est stockée à une adresse paire (sinon, le mot est découpé en 2 bytes transmis de manière séquentielle).*





# Gestion des registres de base

## Initialisation des registres de base



Au début de l'exécution, le programme doit initialiser les registres de base : leur contenu doit correspondre à l'emplacement des segments physiques, contenant le code des segment logiques correspondants :

```
1  Déclarations:  Segment1  SEGMENT
                        ...
2                        Segment1  ENDS
3                        Segment2  SEGMENT          STACK
                        ...
4                        Segment2  ENDS
                        ...
5  Initialisation:  MOV          AX, Segment1
6                        MOV          ES, AX
7                        MOV          AX, Segment2
8                        MOV          SS, AX
                        ...
```



## Réalisation (assembleur et éditeur de liens)

- l'assembleur crée un symbole désignant le segment logique; ce symbole peut être utilisé comme une variable externe (un symbole défini dans un autre module);
- l'éditeur de liens définit la valeur du symbole comme l'origine du segment (= la valeur de l'adresse de base / 16).

*Note : Le contenu des registres de segment CS et SS (ainsi que les registres IP et SP) sont définis par le système d'exploitation avant le début de l'exécution du programme (segment contenant le code, petite pile mise à disposition par le système).*

*Néanmoins il est préférable de redéfinir la pile (SS et SP) pour disposer d'une zone de mémoire suffisante lors de l'exécution du programme.*



## Association segments logiques - registres de base (= segments physiques)

```
ASSUME  CS      CS  
        ES      ES : nom1[, ES : nom2 ... ]  
        SS      SS  
        DS      DS
```

La pseudo-opération **ASSUME** indique à l'assembleur quel registre de segment il doit utiliser pour accéder aux opérandes :

- comme résultat, l'assembleur associe le(s) registre(s) de base (CS, ES, SS, DS) au nom des segments logiques "nom<sub>k</sub>" ;
- par la suite, l'assembleur ajoutera automatiquement des préfixes de segment où cela est nécessaire.

*Note : L'assembleur n'ajoutera un préfixe qu'aux instructions où le choix par défaut du registre de base (voir "Registres") ne fournira pas le bon résultat.*

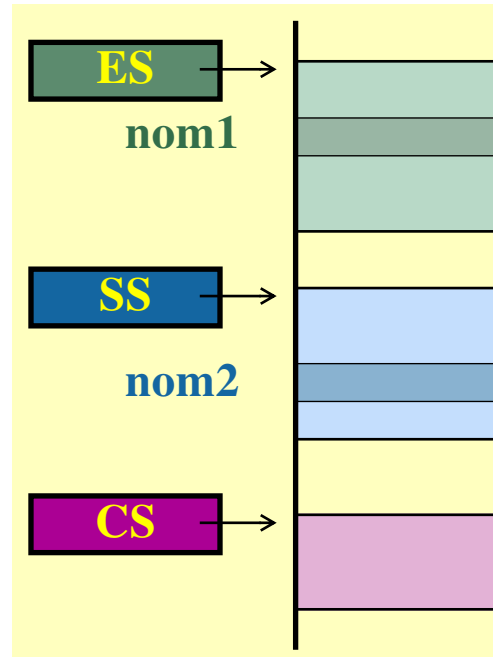


## Gestion des registres de base, exemple

```

1  seg1  SEGMENT
      . . .
2  nom1  DW
      . . .
3  seg1  ENDS
4  seg2  SEGMENT
      . . .
5  nom2  DW
      . . .
6  seg2  ENDS
7  seg3  SEGMENT
      . . .
8
      ASSUME
      . . .
9      MOV    AX, seg1
10     MOV    ES, AX
11     MOV    AX, seg2
12     MOV    SS, AX
      . . .
13     MOV    AX, nom1
14     MOV    BX, nom2
    
```

STACK



Code-source assembleur :



Code correspondant à générer :

```

MOV    AX, ES : nom1
MOV    BX, SS : nom2
    
```

*ES* : *seg1*, *SS* : *seg2*, *CS* : *seg3*...



## Au moment de l'assemblage

### 1. Définition du symbole 'nom1'

→ 'nom1' est défini comme une adresse dans le segment 'seg1' ( 1 et 2 )

### 2. Définition du symbole 'nom2'

→ 'nom2' est défini comme une adresse dans le segment 'seg2' ( 4 et 5 )

### 3. Directive **ASSUME** = "promesse" du programmeur ( 8 ) : lors de l'exécution,

→ ES va contenir la base du segment 'seg1'

→ SS va contenir la base du segment 'seg2'

### 4. Traduction de l'instruction 13 `MOV AX, nom1` : l'assembleur va

→ se souvenir que le symbole 'nom1' avait été défini dans le segment 'seg1' ( 2 )

→ savoir que, lors de l'exécution, ES contiendra l'adresse du segment 'seg1' ( 8 )

→ donc, va interpréter l'instruction comme `MOV AX, ES : nom1`

### 5. Traduction de l'instruction 14 `MOV BX, nom2` par

→ dito ( 4 et 8 ), traduction comme `MOV BX, SS : nom2`

donc

## Au moment de l'exécution

1. L'exécution de 9 et 10 chargera, "comme promis", dans ES l'adresse du segment 'seg1'

2. L'exécution de 11 et 12 chargera, "comme promis", dans SS l'adresse du segment 'seg2'

3. Les instructions 13 et 14 utiliseront donc les bon registres de base (ES, resp. SS) et ces registres contiendront les adresses de base de ces segments.

## Contrôle des segments, résumé

Mesures pour le contrôle de la séparation du code en segments, respectivement de l'association de variables à un segment spécifique:

- Association du code objet à un segment logique (pseudo-instructions **SEGMENT**, **ENDS**) :
  - définition de blocs de code-source et, par conséquent, du code-objet correspondant;
  - création des symboles représentant des adresses dans le code-objet.
- Association entre segments logiques et registres de base (pseudo-instruction **ASSUME**).
- Initialisation des registres de base aux valeurs correspondant aux adresses des segments physiques (opérations d'initialisation à incorporer dans le programme).
- Utilisation du registre de base correct par le CPU (géré par le système d'exploitation).



## Exemple d'un programme

1	INCLUDE	MACRO.LIB	;	ajouter les instructions contenu	
			;	dans le fichier MACRO.LIB	
2	pile	SEGMENT	STACK	;	réservation d'une pile
3		DW	100 DUP (?)	;	mots sans initialisation
4	haut	LABEL	WORD	;	adresse-mot suivant la pile
5	pile	ENDS			
6	data	SEGMENT	PUBLIC	;	ici seront placées les données du programme
		...			
7	data	ENDS			
8	code	SEGMENT	PUBLIC		
9		ASSUME	CS:code, SS:pile, DS:data	;	
10	start	LABEL	FAR		
11		MOV	AX, data	;	lier DS au segment data
12		MOV	DS, AX	;	(pas de MOV imm. DS !)
13		MOV	AX, pile	;	ditto pour SS
14		MOV	SS, AX		
15		MOV	SP, haut	;	haut de la pile (vide !)
		...			
16		EXIT			
17	code	ENDS			
18		END	start	;	fin du code-source, adresse de la première instruction

**MOV AH, 4CH** (de "MACRO.LIB")  
**INT 21H**



# Définition de données

## Pseudo-instructions pour la réservation et définition de données

Un programme-assembleur ne permet pas seulement de définir du code-objet exécutable, mais également de **réserver et initialiser des positions de mémoire** représentant les données sur lesquelles agit le programme:

- réserver une zone de code-objet pour le placement de données constantes ou variables,
- créer du code-objet correspondant aux constantes fournies dans le champ d'opérande (optionnel),
- définir une variable (champ d'étiquette !) portant le type déterminé par l'instruction (BYTE, WORD, DWORD, QWORD, TBYTE) (optionnel).

```

DB          valeur          value
DW          ?
DD          fact DUP (valeur) [, fact DUP (valeur) [, ...]]
DQ          fact DUP (?)
DT          fact DUP (?)
  
```

Les sous-champs du champ d'opérande de ces pseudo-opérations déterminent le contenu du code réservé ou généré. Ce code se compose d'éléments dont la taille est implicitement déterminée par la pseudo-instruction (1, 2, 4, 8, 10 bytes).





## La fonction DUP

La fonction DUP permet de créer des copies multiples d'une constante, d'une liste de constantes ou d'un champ vide.

Si la représentation interne d'un sous-champ dépasse cette taille, un message d'erreur sera donné. Exception : une chaîne de caractères dans une instruction DB est considérée comme une liste de sous-champs d'un caractère.

## Constantes-caractères

La longueur des chaînes de caractères dans tous les instructions sauf DB est limitée à 2 caractères; le code correspondant est ajusté à droite, en ajoutant des 0-bits à gauche si nécessaire.

## Formats possibles

Comme indiqué, 4 formats sont possibles pour chaque sous-champ :

1. *valeur* ... réserver et générer **un seul élément** de code-objet, le contenu correspondant à "**valeur**";
2. *?* ... réserver **un seul élément** de code-objet sans générer de code (contenu **non-défini**);
3. *DUP (valeur)* ... réserver et générer **de multiples éléments** (nombre d'éléments = "fact") de code-objet, chacun avec un contenu correspondant à "**valeur**";
4. *DUP (?)* ... réserver **de multiples éléments** de code-objet **sans générer de code**



## Réservation de données, exemples

1	DB	16	
2	DB	' a '	
3	DB	' ab '	
4	DB	' a ' , ' b ' , 10	
5	DB	10 DUP ( ' ' )	
6	DB	10 DUP ( ? )	
7	DW	' xy '	
8	DB	257	; faux
9	DT	' abc '	; faux





## Chapitre 4 :

# Le répertoire d'instructions

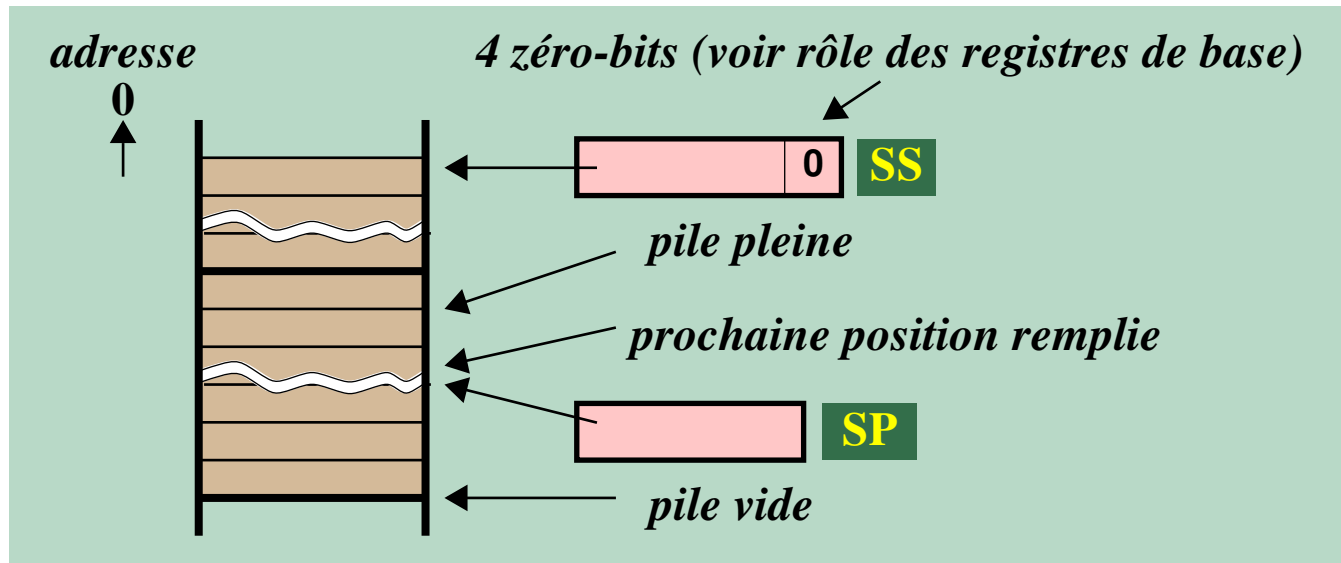
### Résumé :

#### ➤ Opération de l'Intel 8086

- Opérations utilisant la pile
- Branchements
- Opérations arithmétiques et logiques
- Opérations itératives, actions sur chaînes de caractères

# Opérations utilisant la pile

## Utilisation d'une pile



**Ajouter une valeur sur la pile :**

1.  $SP := SP - 2$
2. déposer la valeur à l'adresse déterminée par SP

**Retirer une valeur de la pile :**

1. Retirer la valeur de l'adresse déterminée par SP
2.  $SP := SP + 2$

*Note : la pile "grandit" vers le bas de la mémoire (Intel 8086!)  
choix du constructeur!*

## Opérations utilisant la pile

<b>PUSH</b> <i>opérande-mot</i>	<ol style="list-style-type: none"> <li>1. <math>SP := SP - 2</math></li> <li>2. <math>cont(SS:SP) := \text{"op.-mot"}^1)</math></li> </ol>	1) <i>cont(pos) = contenu de la position-mémoire "pos"</i>
<b>POP</b> <i>opérande-mot</i>	<ol style="list-style-type: none"> <li>1. <math>\text{"op.-mot"} := cont(SS:SP)^1)</math></li> <li>2. <math>SP := SP + 2</math></li> </ol>	
<b>CALL</b> <i>opérande-near</i>	<ol style="list-style-type: none"> <li>1. PUSH IP</li> <li>2. <math>IP := \text{"op.-near"}</math></li> </ol>	
<b>CALL</b> <i>opérande-far</i>	<ol style="list-style-type: none"> <li>1. PUSH CS</li> <li>2. PUSH IP</li> <li>3. <math>CS:IP := \text{"op.-far"}</math></li> </ol>	
<b>INT</b> <i>valeur</i>	<ol style="list-style-type: none"> <li>1. PUSH flags<sup>2)</sup></li> <li>2. PUSH CS</li> <li>3. PUSH IP</li> <li>4. <math>IF := 0 \quad TF := 0</math></li> <li>5. <math>CS:IP := cont(\text{"valeur"} * 4)^1) 3)</math></li> </ol>	2) <i>flags = contenu du registre d'état ("flag register")</i> 3) <i>adresse absolue (valeur * 4)</i>
<b>RET</b>	<ol style="list-style-type: none"> <li>1. POP IP</li> <li>2. POP CS (seulement si "FAR")</li> </ol>	
<b>IRET</b>	<ol style="list-style-type: none"> <li>1. POP IP</li> <li>2. POP CS</li> <li>3. POP flags<sup>2)</sup></li> </ol>	



# Opérations de branchement

## Forme des instructions de branchement

*instruction*                      *opérande*

*Note* : “opérande” doit être du type “NEAR”, sauf pour les instructions **JMP** et **CALL** où il peut aussi être “FAR”

L’assembleur transforme la valeur de l’opérande du code-source (= adresse immédiate) en une valeur de déplacement (= adresse relative, seule permise par les instructions-machine).

### Branchements sans condition

<b>JMP</b>	branchement simple
<b>CALL</b>	avant le branchement, sauver l’adresse de retour sur la pile

### Branchements pour itérations

Instruction	Actions (phases)	
	1. mise-à-jour de CX	2. branchement si :
<b>LOOP</b>	CX := CX-1	CX≠0
<b>LOOPZ , LOOPE</b>	CX := CX-1	(CX≠0)^(ZF=1)
<b>LOOPNZ , LOOPNE</b>	CX := CX-1	(CX≠0)^(ZF=0)
<b>JCXZ</b>	pas d’action	CX=0

 [Liste des instructions](#)



## Branchements conditionnels

ZF	SF	CF	OF	PF	opération	condition de branchement (à la suite de l'exécution de l'instruction "CMP A,B")
1					JE, JZ	A=B
0					JNE, JNZ	A≠B
	1				JS	<div style="display: flex; align-items: center;"> <div style="margin-right: 10px;">} comparaison de valeurs sans signe</div> <div style="border: 1px solid black; padding: 5px;"> <math>\vee</math> = "ou" inclusif  <math>\nabla</math> = "ou" exclusif                 </div> </div>
	0				JNS	
		1			JB, JNAE	
		0			JNB, JAE	
					JBE, JNA	
					JNBE, JA	
					JL, JNGE	
					JNL, JGE	
					JLE, JNG	
					JNLE, JG	
			1		JO	dépassement arithmétique
			0		JNO	pas de dépassement arithmétique
				1	JP, JPE	parité paire ("even")
				0	JNP, JPO	parité impaire ("odd")

*Notes :* **Caractères mnémoniques :** Equal, Zero, Below, Above, Less-than, Greater-than, Not

[Liste des instructions](#)

# Opérations arithmétiques & logiques

opération	destination						source						flags modifiés
	reg8	reg16	seg	mém.	imm.	flag	reg8	reg16	seg	mém.	imm.	flag	
MOV	✓	✓		✓			✓	✓		✓	✓		aucun
LEA 1)	✓	✓	✓					✓			✓		
XCHG	✓	✓					✓	✓			✓		aucun
PUSH			<i>pile</i>					✓	✓				aucun
PUSHF			<i>pile</i>									✓	aucun
POP		✓	2)								<i>pile</i>		aucun
POPF						✓					<i>pile</i>		tous
LAHF	AH											<i>inf</i>	aucun
SAHF						<i>inf</i>	AH						ACOPSZ

Notes : ✓ ... peut figurer dans ce champ à condition si

- source et opérande sont de la même taille
- il a au plus une seule opérande-mémoire

*inf* ... 8 bits inférieurs

*pile* ... adressage implicite par une action sur la pile

1) évaluation de l'adresse-mémoire, résultat (= adresse effective) → registre

2) sauf CS

 [Liste des instructions](#)





## Opérations arithmétiques & logiques (suite)

opération	destination						source						flags modifiés
	reg8	reg16	seg	mém.	imm.	flag	reg8	reg16	seg	mém.	imm.	flag	
ADD 1)	✓	✓		✓			✓	✓		✓	✓		ACOPSZ
INC, DEC	✓	✓		✓					= dest				AOPSZ
NEG 2)	✓	✓		✓					= dest				ACOPSZ
NOT 3)	✓	✓		✓					= dest				aucun
SHL, SAL, SHR, SAR	✓	✓		✓					= dest				ACOPSZ
ROL, ROR, RCL, RCR	✓	✓		✓					= dest				CO
MUL, IMUL	DX:AX							✓		✓		4)	} ACOPSZ
	AX						✓		✓			4)	
DIV, IDIV	AX					5)		✓		✓			} ACOPSZ
	AL					5)	✓		✓				

**Notes :** ✓... peut figurer dans ce champ à condition si

- source et opérande sont de la même taille
- il a au plus une seule opérande-mémoire

1) *dito* : ADC, CMP, SUB, SBB, TEST, AND, OR, XOR

2) *complément à deux de tous les bits*

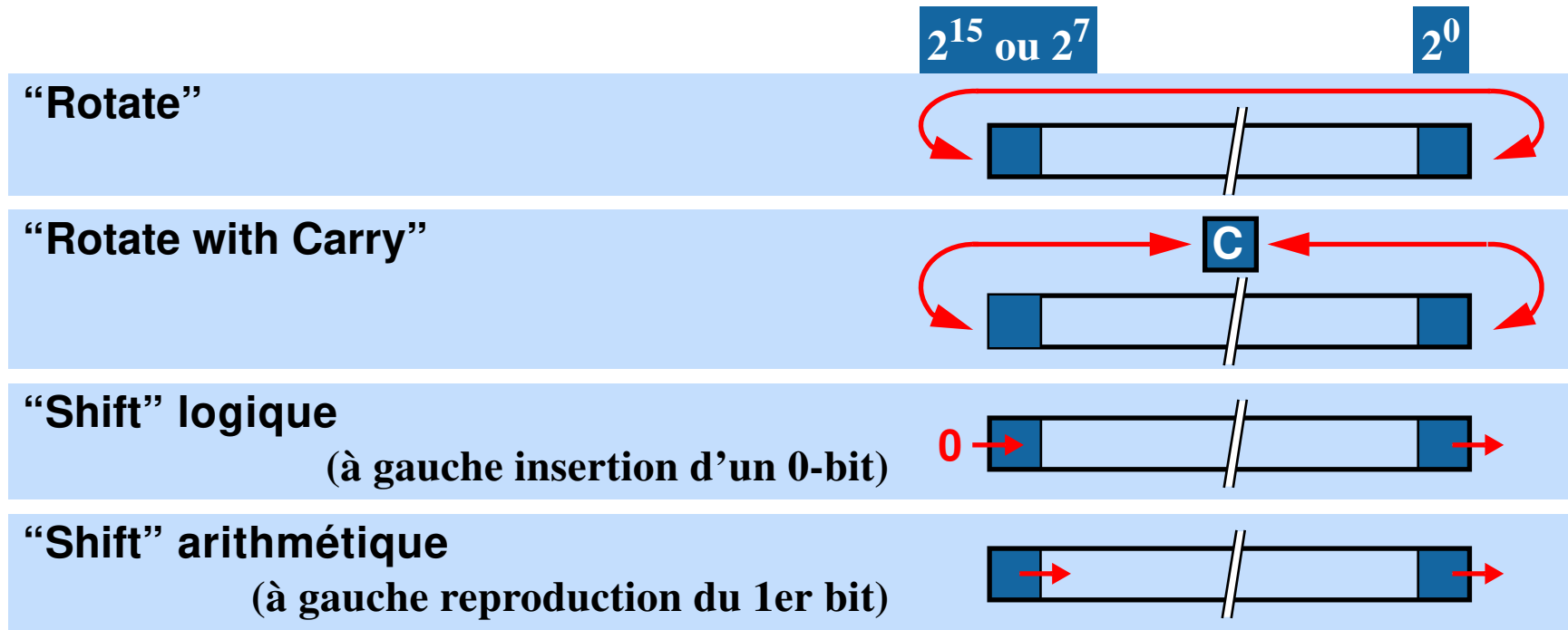
3) *complément à un de tous les bits*

4) *l'autre opérande-source = AX, resp. AL*

5) *dest. = résultat; reste = DX, resp. DL, diviseur = source; dividende = AX:DX, resp. AX*

 [Liste des instructions](#)





# Extension du bit de signe

Certaines opérations doivent ajouter des bits significatifs en début d'une valeur  
p.ex. conversion byte → mot, décalages

Illustration à l'aide d'un exemple non-Intel (l'opération "MOV AX, AL" n'es pas permise):

**MOV AX, AL !! Opération défendue !!**

Deux implantations possibles de l'extension du contenu de AL en valeur de 16 bits (choix du constructeur) :

1. Extension logique : adjonction de huit 0-bits en position de fort poids
2. Extension du bit de signe : reproduction du 1er bit

**CBW** extension du bit de signe de AL vers AH  
**CWD** extension du bit de signe de AX vers DX

Les CPUs à partir du 80386 ont des instructions qui permettent d'effectuer une telle extension:

**MOVZX AX, AL** adjonction de huit 0-bits  
**MOVSX AX, AL** reproduction du 1er bit

*Note : un problème semblable se présente si une valeur immédiate représentée sur 8 bits doit déterminer une opérande de 16 bits. Normalement ceci est traité par l'assembleur sans que le programmeur ne s'en aperçoive.*



# Opérations itératives, actions sur chaînes de caractères

## Résumé des instructions

opération	destination						source						flags modifiés
	reg 8	reg 16	seg	mém	imm.	flag	reg 8	reg 16	seg	mém	imm.	flag	
MOVSW, MOVSB				[ES:DI]							[DS:SI]		aucun
CMP SW, CMP SB				[ES:DI]							[DS:SI]		ACOPSZ
SCAW				[ES:DI]					AX				ACOPSZ
SCAB				[ES:DI]			AL						ACOPSZ
LODW		AX									[DS:SI]		aucun
LODB	AL										[DS:SI]		aucun
STOW				[ES:DI]					AX				aucun
STOB				[ES:DI]			AL						aucun

*Note : [ES:DI] et [DS:SI] ... contenu de la position-mémoire*

*Les registres SI et DI sont modifiés après l'exécution de ces instructions, en fonction du contenu du bit DF du registre d'état (0 ... incrémenter, 1 ... décrémenter; l'incrément ou décrément est de 1 pour les opérandes-byte, de 2 pour les opérandes-mot).*

 [Liste des instructions](#)

## Contrôle des itérations

Les instructions de ce groupe peuvent être exécutées de manière itérative en les faisant précéder d'un code de répétition (**REP**, **REPE**, **REPZ**, **REPNE**, **REPNZ**).

Ce préfixe provoque l'exécution répétitive (**CX** fois) de l'opération et détermine une condition d'arrêt supplémentaire.

Avant chaque itération (et avant le teste de la condition de sortie), **CX** se décrémente de 1.

Préfixe	Répétition si <b>CX</b> ≠0 ou si :
<b>REP</b>	pas d'autre condition
<b>REPE</b> , <b>REPZ</b>	<b>ZF</b> =1
<b>REPNE</b> , <b>REPZ</b>	<b>ZF</b> ≠0

**Exemple :**

```

MOV      CX, 17
REP      ; Recopier la valeur de AL
STOB    ; dans 17 bytes consécutifs
  
```

# Contrôle des bits du registre 'flags'

Opération	Action
CLC , CMC , STC	modification <sup>1)</sup> du bit 'C' ("carry")
CLD , STD	modification <sup>1)</sup> du bit 'D' (direction opérations-chaînes)
CLI , STI	modification <sup>1)</sup> du bit 'I' ("interrupt enable")

1) CLx ... "clear", CMx ... "complement", STx ... "set"

## Diverses opérations

Opération	Action
AAM , AAS , DAA DAS	Adaptation de la représentation d'opérandes ASCII et BCD avant/ après une opération arithmétique (e.g. addition de 2 car. ASCII).
CBW	Extension du bit de signe du registre AL vers le registre AX
CWD	Extension du bit de signe du registre AX vers le registre DX
ESC	Mettre une opérande destinée au co-processeur sur le bus.
HLT	Arrêter le CPU avant l'exécution de la prochaine instruction; reprise p.ex. par une interruption.
WAIT	Suspendre le CPU, attendre un niveau 1 au "pin" TEST du CPU.
LOCK	Instruction préfixe pour bloquer le bus pour la totalité des cycles nécessaires pour exécuter une opération.

 [Liste des instructions](#)



## Chapitre 5 :

# Variables et expressions

### Résumé :

- L'utilisation de variables dans un assembleur
- Les expressions
- L'utilisation de 'fonctions' pour l'accès aux conditions gérées par l'assembleur



# Variables

**Variable = représentation d'un objet gérée par l'assembleur**

En général, cet objet est la représentation d'une valeur

→ assembleur pour le CPU 8086: objets typés (= exception par rapport aux autres assembleurs)

Justifications principales pour le typage:

- associer un registre de base spécifique aux symboles désignant une adresse-mémoire,
- distinguer entre contenant-mot et contenant byte,
- distinguer entre cibles pour branchement avec ou sans modification du registre CS.

## Composantes d'une variable

Une variable est définie par:

**1. Le nom** de la variable :

**un symbole unique** qui désigne la variable (= chaîne de caractères, introduite par le programme-source, obéissant à certaines règles syntaxiques) (voir "champ d'étiquette")

**2. Les attributs** de la variable :

**des valeurs et caractéristiques** (types) associées à la variable ; ceux-ci sont déterminées par l'assembleur

- en fonction de l'opération par laquelle la variable est définie,
- de l'état de l'assemblage au moment où la variable est définie.





## Définition d'une variable

Une variable est (re-)définie quand son nom figure dans le champ d'étiquette d'une instruction (dont les autres champs peuvent être vides).

*Note : Problème des “références en avant” ... assemblage “en deux passes”*

### Visibilité d'une variable

La définition de la variable se solde par la création d'une entrée dans la table des symboles, table gérée par l'assembleur et qui disparaît à la fin de l'assemblage.

Visibilité = contexte de la validité de l'association nom  $\Leftrightarrow$  valeur; correspond à la “durée de vie” de la variable - la période pendant laquelle cette association reste définie.

Cas normal:

#### variable locale

visibilité restreinte au code-source définissant la variable: la variable n'est définie que dans un **contexte local**.

Cas particulier:

#### variable globale

visibilité au delà du code dans lequel est défini la variable: la variable est définie dans un **contexte global**.

→ l'assembleur enregistre la valeur finale des variables globales dans un tableau auxiliaire du code-objet.

exemples:

- les variables déclarées “PUBLIC”
- code créé en vue d'une mise-au-point à l'aide d'un “debugger symbolique”



# Types de variables

Distinction entre 3 types de variables :

- *étiquettes*
- *variables relogeables*
- *constantes*

## Étiquettes (“labels”)

Étiquettes ... opérandes pour les opérations de branchement

### Attributs

segment	...	nom du segment où l'étiquette est définie
offset	...	adresse relative dans le segment (= sa “valeur”)
type	...	étiquette-cible pour les branchements
	<b>NEAR</b>	... à l'intérieur d'un segment,
	<b>FAR</b>	... dans un autre segment (syntaxe = <i>segment : offset</i> )

### Définition

dans une instruction  
de la forme :

<i>étiquette</i> [ $\surd$ <i>instruction</i> ]	...	type implicite = <b>NEAR</b>
<i>étiquette:</i> [ $\surd$ <i>instruction</i> ]	...	type implicite = <b>FAR</b>
<i>étiquette</i> $\surd$ <b>NEAR</b> $\surd$ <i>LABEL</i> $\surd$ <b>FAR</b>	...	choix explicite du type

Ce type de variable ne peut être définie qu'à l'intérieur de segments liés à CS (voir “**ASSUME**”).



## Variables relogeables (“variables”)

Variables ... opérandes pour les opérations se référant à la mémoire (accès aux données)

### Attributs

segment ... voir “étiquettes”  
 offset ... voir “étiquettes”  
 type ... dépend de la pseudo-opération utilisée pour définir la variable :  
 BYTE, WORD, DWORD,  
 QWORD, TBYTE, STRUCT

### Définition

dans une instruction de la forme :

*variable*  $\surd$  DB, DW, DD, DQ, DT, [  $\surd$ opérande ]  
 RECORD, STRUCT

## Constantes (“numbers”)

Constantes ... représentation d’une valeur numérique

### Attributs

offset ... valeur numérique attribuée à la variable

### Définition

dans une instruction de la forme :

*variable*  $\surd$  =  $\surd$  *valeur* ... re-définition permise  
*variable*  $\surd$  EQU  $\surd$  *valeur.* ... re-définition défendue



# Expressions

L'assembleur permet d'utiliser des expressions partout où une valeur numérique est attendue; le résultat de l'évaluation sera substitué à l'expression.

L'expression sera évaluée en respectant l'ordre de précedence des opérateurs, à précedence égale, de gauche à droite



opérateur	précedence	exemples, commentaires	
		expression	résultat
* / MOD	5	14 MOD 3	2
SHL SHR	25	101B SHL 3	010100B (décalage)
+ -	6	2 + 3	5
EQ NE LT	7	3 EQ 3	0FFFFH
LE GT GE	7	3 NE 3	0
NOT	8	NOT 0	0FFFFH
AND	9	12H AND 3	2
OR	10	11H OR 3	13H
XOR	10	11H XOR 3	1H
HIGH LOW	4	HIGH 1234H	12H (high, low byte)
:	1	<i>symbole</i> :	définir "symbole" (type = NEAR)
( )	1		

# Fonctions

Les “fonctions” permettent d’avoir accès à certaines conditions gérées par l’assembleur. Lorsque le nom d’une fonction figure dans une instruction, l’assembleur substitue le résultat de l’exécution de la fonction au nom de la fonction.

*Note : L’assembleur 8086 offre un nombre important de fonctions (accès et gestion des attributs associés aux symboles!). Cet aspect complexe ne se retrouve pas dans les assembleurs d’autres CPU.*

*Ce cours se concentre sur la fonction ‘\$’ (pour laquelle un correspondant existe dans la plupart des assembleurs. Les autres fonctions ne sont décrites qu’à titre de documentation.*

## Fonction pour l’accès au compteur de position

fonction	valeur
\$	La fonction “\$” fournit une valeur qui, à tout instant, est égal au compteur de positions actuel avec les attributs correspondant.

## Fonctions pour obtenir les attributs d’un “RECORD”

fonction	valeur
<b>MASK</b>	Masque pour isoler un champ d’un record par un “et” logique.
<b>WIDTH</b>	Nombre de bits dans un record.



## Fonctions pour obtenir les attributs d'une variable

Obtenir la valeur d'un attribut par l'utilisation d'une fonction :

fonction	valeur
<b>SEG</b>	Base du segment dont le nom = symbole ou base du segment où le symbole est défini.
<b>OFFSET</b>	Adresse relative ou valeur attribuée au symbole.
<b>TYPE</b>	<ul style="list-style-type: none"><li>• code représentant le type : nombre de bytes (variable d'adresse),</li><li>• -2 pour <b>FAR</b>, -1 pour <b>NEAR</b> (étiquette),</li><li>• 0 (variable absolue).</li></ul>
<b>LENGTH</b>	Nombre d'éléments associés à l'opérande, p.ex.  <b>XX</b> <b>DW</b> <b>100 DUP (1)</b> <b>LENGTH</b> <b>XX</b> ; donne 100
<b>SIZE</b>	Nombre de bytes réservés (ne peut être utilisé qu'avec les opérandes du genre <b>DUP</b> ); résultat = " <b>LENGTH</b> " * " <b>TYPE</b> " (200 dans l'exemple précédant).



## Fonctions pour manipuler les attributs d'une variable

Déterminer la valeur d'un attribut par l'utilisation d'une fonction :

fonction	valeur
<b>SHORT</b>	Utiliser une étiquette du type "FAR" en faisant apparaître son type comme "NEAR".
<b>PTR</b>	Utiliser la valeur d'une expression (argument suivant PTR) en lui attribuant un type explicitement spécifié par l'argument suivant PTR <sup>1)</sup> , qui est en règle général différent du type implicite de l'expression. Par exemple <pre>xyz    DB    4       MOV  AX, WORD PTR xyz</pre>
<b>THIS</b>	Définir le symbole figurant dans le champ d'étiquette comme une variable d'un type spécifié (argument suivant THIS); le reste des attributs est déterminé par le compteur de position et le segment courant. Par exemple <pre>xyz    DB    4 abc    EQU  THIS WORD</pre>

<sup>1)</sup> un des types

BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, FAR



## Fonctions comme opérateurs dans une expression

opérateur	précé- dence	exemples, commentaires	
		expression	résultat
<b>SHORT</b>	<b>11</b>	<b>JMP</b>	<b>SHORT</b> <i>farsymbol</i>
<b>SEG</b>	<b>3</b>	<b>SEG</b>	<i>seg</i>
		<b>SEG</b>	<i>var</i>
<b>LENGTH</b>	<b>1</b>	<b>LENTH</b>	<i>dup</i>
<b>SIZE</b>	<b>1</b>	<b>SIZE</b>	<i>dup</i>
<b>TYPE</b>	<b>3</b>	<b>TYPE</b>	<i>symb</i>
<b>MASK</b>	<b>1</b>	<b>MASK</b>	<i>champ</i>
<b>WIDTH</b>	<b>1</b>	<b>WIDTH</b>	<i>champ</i>
<b>OFFSET</b>	<b>3</b>	<b>OFFSET</b>	<i>symb</i>
<b>PTR</b>	<b>3</b>	<b>WORD</b>	<b>PTR</b> <i>var</i>
<b>THIS</b>	<b>3</b>	<b>THIS</b>	<b>BYTE</b>





## Chapitre 6 :

# Aspects méta-assembleur

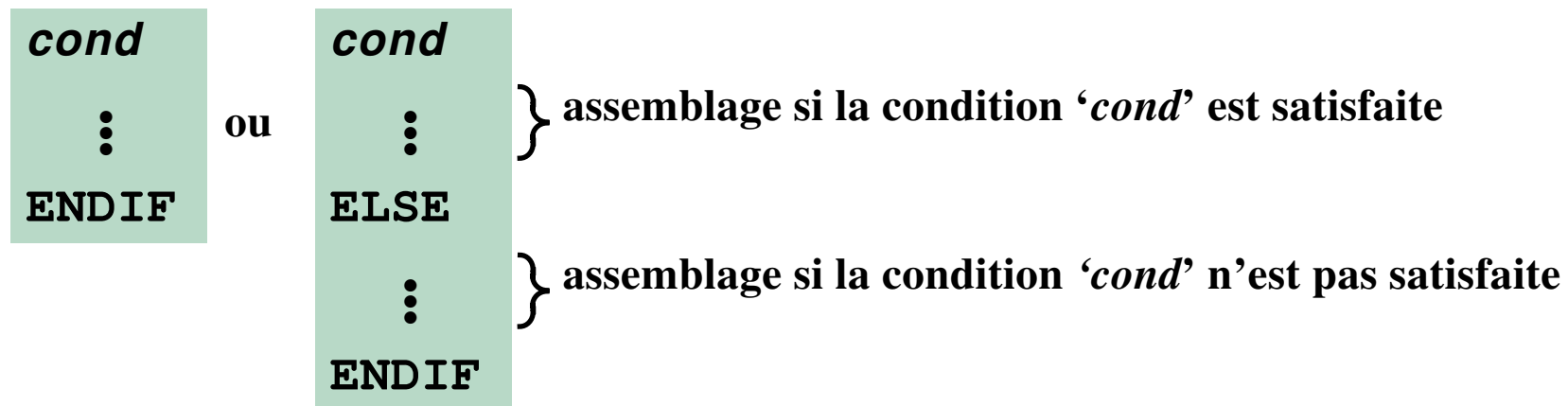
### Résumé :

- **Assemblage conditionnel et répétitif**
- **Macro-assemblage**
- **Procédures et décomposition du programme en modules**
- **Contrôle de l'impression**



# Assemblage conditionnel

Suppression conditionnelle de l'assemblage d'un bloc de lignes de code-source.



## Conditions reconnues

condition	opérande(s)	assemblage si :
IF	<i>expression</i>	"expression" ≠ 0
IFE	<i>expression</i>	"expression" = 0
IF1		en 1er passage de l'assembleur
IF2		en 2me passage de l'assembleur
IFDEF	<i>variable</i>	"variable" est définie
IFNDEF	<i>variable</i>	"variable" n'est pas définie
IFB	<arg>	"arg" est blanc
IFNB	<arg>	"arg" n'est pas blanc
IFIDN	<arg1>,<arg2>	"arg1" = "arg2" <i>comparaison de</i>
IFNIDN	<arg1>,<arg2>	"arg1" ≠ "arg2" <i>chaînes de caractères</i>

## Utilisation

- **Assemblage optionnel**

Conditions figées par déclaration (EQU), p. ex. code pour différentes configurations, options de mise-au-point, etc.

- **Conditions dynamiques**

Conditions variant en fonction de l'état de l'assemblage, utilisation surtout dans des macro-instructions

### Exemple 1

```

MACH    EQU        780           ; type de machine (750 / 780)
        IFE        MACH-750
        ...
        ENDIF
        IFE        MACH-780
        ...
        ENDIF

```

; code spécifique VAX 11/750

; code spécifique VAX 11/780

### Exemple 2

```

IF      (OFFSET $) MOD 256
        DB      256-((OFFSET $) MOD 256) DUP (?)
ENDIF

```

; le prochain byte généré sera placé à une  
; adresse relative qui est un multiple de 256



# Assemblage répétitif

Assemblage répétitif d'un bloc de lignes de code-source (= dédoublement automatique du bloc)

Implantation dans l'assembleur par une opération de substitution de chaînes :

1. "re-écriture" du code-source (n fois, en effectuant les modifications éventuellement nécessaires)
2. assemblage du code-source ainsi obtenu

```
REPT    count
  ⋮
ENDM
```

les instructions figurant entre les deux pseudo-instructions sont traitées "count" fois

```
IRP    symb,<arg1 , ... , argn>
  ⋮
ENDM
```





comme REPT, mais le nombre de répétitions = nombre d'arguments; "symb" prend successivement les valeurs des arguments

```
IRPC   symb,<chaîne> ou symb,chaîne
  ⋮
ENDM
```

comme REPT, mais le nombre de répétitions = nombre de caractères dans "chaîne"; toute occurrence de "symb" est successivement remplacée par les caractères de la chaîne



## Exemples

<b>x</b>	=	0		DB	0
	REPT	3		DB	1
	DB	<b>x</b>		DB	2
<b>x</b>	=	<b>x+1</b>		ENDM	
	IRPC	<b>xx</b> , 012		DB	0
	DB	<b>xx</b>		DB	1
	ENDM			DB	2
	IRPC	<b>xx</b> , 012		DB	0
	DB	<b>xx</b>		DB	1
	ENDM			DB	2
	IRPC	<b>xx</b> , 012		DB	0
	DB	<b>3&amp;xx</b>		DB	1
	ENDM			DB	2



# Substitutions de chaînes

## Traitement interne par l'assembleur

### Modèle "substitution de chaînes" en support de la compréhension des opérations

- de l'assemblage conditionnel
- de l'assemblage répétitif
- du macro-assemblage

#### 1. Manipulation du code source

#### 2. Assemblage conditionnel : suppression conditionnelle de code-source

- Assemblage répétitif :
  - a. lecture répétitive du code-source,
  - b. opération de substitution de chaînes de caractères
- Macro-assemblage :
  - a. remplacement de l'appel par le corps
  - b. opération de substitution de chaînes de caractères

#### 3. Assemblage du code ainsi obtenu

*Note : Ce modèle est utile pour l'aspect conceptuel; la réalisation concrète et effective dans un assembleur est, en général, différent.*



# Macro-assemblage

## Définition d'une macro-instruction

Définition du nom de la macro-instruction, association du corps au nom :

```

macname   MACRO   [parf1 [, parf2 ... ] ]   ← entête
           ⋮
           ENDM
  
```

← corps

## Appel d'une macro-instruction

- Substitution
- de l'instruction d'appel par le corps de la macro-instruction
  - des **arguments actuels** aux **paramètres formels** :

```

macname   [ arg1 [, arg2 ... ] ]
  
```

## Principes de la substitution

- substitution = traitement de chaînes;
- chaque occurrence d'un paramètre formel ( $parf_i$ ) est remplacée par l'argument correspondant ( $arg_i$ );
- le reste des opérations de l'assembleur n'interviennent qu'après cette substitution;
- des symboles rencontrés dans les champs d'étiquettes des instruction générées au cours de la substitution donnent lieu à la définition de variables qui sont globalement visibles.



## Déclaration de variables locales

```
LOCAL [ parf1 [,parf2 ... ] ]
```

La validité de la définition de variables peut être restreinte à la zone correspondant au code généré par l'appel de la macro-instruction (fautes d'assemblage suite à la définition répétée du même symbole si une macro-instruction est appelée plusieurs fois !).

Ces variables doivent être déclarées comme paramètres formels et figurer comme argument d'une pseudo-instruction **LOCAL**.

L'assembleur remplacera alors chacun de ces symboles par un symbole unique (..0001, ..0002, etc.) lors de chaque appel.

*Note : Cette utilisation du terme "variable local" est quelque peu différente de celle introduite dans le chapitre sur les "variables"! Notion commune: restriction de la visibilité d'une variable à un contexte local. Cette définition de la notion de "variable local" est compatible avec l'utilisation dans les langages élevés à structure de bloc.*

## Séparateur de chaînes de caractères

**& ...** séparateur transparent pour le reste de l'assemblage, permettant de séparer l'occurrence d'un paramètre formel des caractères avoisinants.

Par exemple :

abc&xx&efg

(xx = paramètre formel, 12 dans l'exemple)

⇒ abc12efg





## Exemples

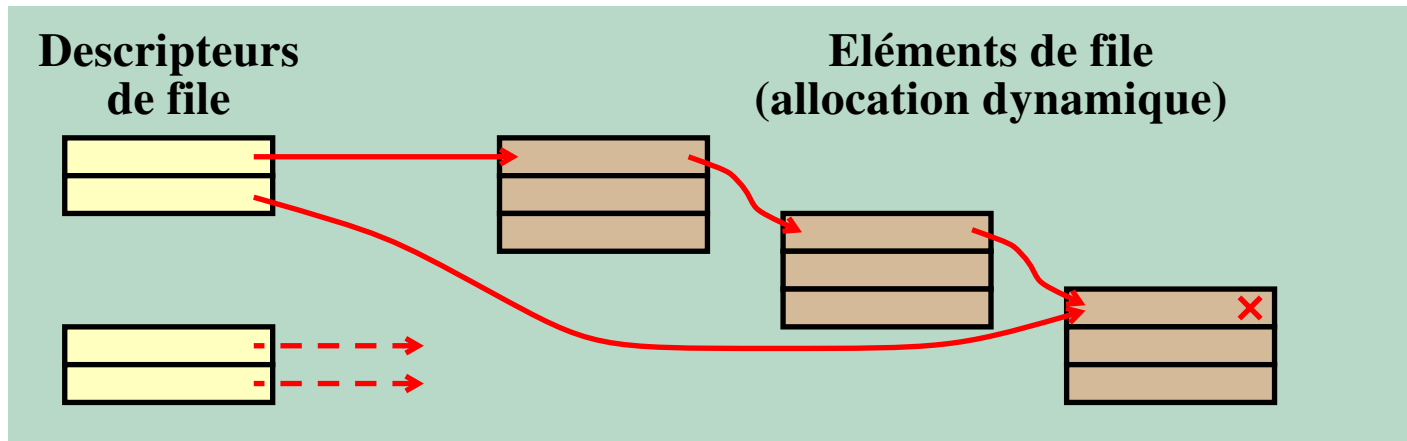
### Macro instruction pour imprimer une chaîne de caractères

Impression d'une chaîne fournie comme argument (sauvetage de tous les registres utilisés sur la pile). Pour imprimer la chaîne *abcd* :

1	<i>Appel (illustration):</i>	PRSTR	'abcd'	
2	<i>Définition:</i>	PRSTR	MACRO	chaine
3			LOCAL	temp
4		tdata	SEGMENT	
5		temp	DB	chaine ; chaîne en zone données
6			DB	'\$' ; terminaison de la chaîne
7		tdata	ENDS	
8			PUSH	AX ; sauver les
9			PUSH	DX ; registres
10			PUSH	DS ; utilises
11			MOV	AX, tdata
12			MOV	DS, AX ; segment avec chaîne
13			MOV	DX, OFFSET temp ; déplacement au début
14			MOV	AH, 9h ; no. fonction "impression"
15			INT	21h ; appel fonction du DOS
16			POP	DS ; rétablir les
17			POP	DX ; registres
18			POP	AX ; utilises
19			ENDM	

## Macro instructions pour la gestion de files d'attente

Définir une structure de données en support d'un traitement de files: usage de macro-instructions en support de structures complexes et d'opération pour leur gestion.



1	<code>VOID</code>	<code>EQU</code>	<code>0ffffh</code>	<code>; élément vide</code>
2	<code>Q_ITEM</code>	<code>STRUC</code>		<code>; descripteur d'élément de file</code>
3	<code>  i_next</code>	<code>  DW</code>	<code>  ?</code>	<code>  ; chaînage prochain élément</code>
4	<code>  i_data1</code>	<code>  DW</code>	<code>  ?</code>	<code>  ; données (bidon)</code>
5	<code>  i_data2</code>	<code>  DW</code>	<code>  ?</code>	<code>  ; données (bidon)</code>
6	<code>Q_ITEM</code>	<code>ENDS</code>		
7	<code>Q_DESCR</code>	<code>STRUC</code>		<code>; descripteur de file</code>
8	<code>  q_head</code>	<code>  DW</code>	<code>  ?</code>	<code>  ; premier élément (VOID si vide)</code>
9	<code>  q_tail</code>	<code>  DW</code>	<code>  ?</code>	<code>  ; dernier élément (VOID si vide)</code>
10	<code>  q_stamp</code>	<code>  DW</code>	<code>  ?</code>	<code>  ; heure dernière action</code>
11	<code>Q_DESCR</code>	<code>ENDS</code>		

12	<b>Q_new</b>	<b>MACRO</b>	<b>\$q</b>	<b>; définir une file de nom = arg1</b>
13	tdata	SEGMENT		
14	\$q	Q_DESCR	<?>	<b>; créer le descripteur de file</b>
15	tdata	ENDS		
16		MOV	\$q.q_head, VOID	<b>; la file est vide</b>
17		MOV	\$q.q_tail, VOID	
18		MOV	AX, system_time	<b>; horloge système</b>
19		MOV	\$q.q_stamp, AX	<b>; temps dernière action</b>
20		ENDM		
21	<b>Q_app</b>	<b>MACRO</b>	<b>\$q</b>	<b>; ajouter un él. (BX) à la fin de arg1</b>
22		LOCAL	Q_app1, Q_app2	
23		MOV	OFFSET i_next [BX], VOID	<b>; pas de successeur</b>
24		MOV	DI, \$q.q_tail	<b>; DI = ancien dernier élément</b>
25		CMP	DI, VOID	
26		JE	Q_app1	
27		MOV	OFFSET i_next [DI], BX	<b>; non-vide: élément suit élément</b>
28		JMP	Q_app2	
29	Q_app1:	MOV	\$q.q_head, BX	<b>; vide: nouveau 1er élément</b>
30	Q_app2:	MOV	\$q.q_tail, BX	<b>; nouveau dernier élément</b>
31		MOV	AX, system_time	<b>; horloge système</b>
32		MOV	\$q.q_stamp, AX	<b>; temps dernière action</b>
33		ENDM		

```

34 ; Initialisation
    ...
35     Q_new     high_pr     ; file pour priorité élevée
36     Q_new     low_pr      ; file pour priorité basse
    ...
37 ; Produire un élément
    ...
38     CALL     new_element  ; créer un nouvel élément (base = BX)
39     MOV     i_data1[BX], ... ; définir
40     MOV     i_data2[BX], ... ; son contenu
41     Q_app    high_pr      ; insérer l'élément en fin de la file
    ...

```

Non-inclu dans l'exemple:

- la procédure 'new\_element'
  - elle alloue la mémoire (gestion dynamique) pour un nouvel élément
  - au retour, le registre BX désigne l'adresse de cet élément
- la gestion de la position 'system\_time' - on admet que cette position contient à tout instant la valeur de l'horloge du système
- les instructions ASSUME et SEGMENT pour la procédure principale, ainsi que la préparation correspondante des registres de base



# Importation de code-source

**INCLUDE**

*name*

Insérer le contenu du fichier “name” en lieu et place de l’instruction **INCLUDE**



# Décomposition en modules

```
symb PROC NEAR  
FAR
```

## Procédures

Définir “*symb*” comme une étiquette, par défaut du type “NEAR”. Les instructions RET à l’intérieur du bloc seront du type intra-segment si le type est “NEAR”, sinon du type inter-segment.

```
symb ENDP
```

Fin du bloc défini par “PROC”.

## Utilisation de variables globales

```
PUBLIC symb1 [ ,symb2 ... ]
```

Les symboles indiqués sont accessibles à l’éditeur de liens, donc à d’autres (sous-) programmes

```
EXTRN symb1 : type1 [ ,symb2 : type2 ... ]
```

Les symboles indiqués ne sont pas définis dans le programme actuellement assemblé et devront être fixés par l’éditeur de liens.

“*type<sub>i</sub>*” ... BYTE, WORD, DWORD,  
NEAR, FAR, ABS



## Utilisation de procédures, exemple

<i>nrp</i>	PROC	NEAR	le symbole “nrp” obtient le type “NEAR”
	...		
	RET		utilisation de l’instruction RET intra-segment (PROC précédant était “NEAR”)
	...		
	RET		
<i>nrp</i>	ENDP		
<i>frp</i>	PROC	FAR	le symbole “frp” obtient le type “FAR”
	...		
	RET		utilisation de l’instruction RET-inter segment (PROC précédant était “FAR”)
	...		
	RET		
<i>frp</i>	ENDP		
	...		
	CALL	<i>nrp</i>	utilisation de l’instruction CALL intra-segment (“nrp” est du type “NEAR”)
	...		
	CALL	<i>frp</i>	utilisation de l’instruction CALL-inter segment (“frp” est du type “FAR”)
	...		

*Note : Le type de l’instruction CALL utilisé ne dépend que du type du symbole fourni en argument :*

- *si le type est “FAR”, un appel inter-segment est composé, même si la procédure appelée se trouve dans le même segment et qu’un appel inter-segment ne serait pas nécessaire;*
- *l’appel d’un symbole du type “NEAR”, mais déclaré dans un autre segment, correspond à une erreur de programmation*



# Contrôle de l'impression

## Suppression de l'impression

**.LIST**  
:  
**.XLIST**

Permettre (**.LIST**) / suspendre (**.XLIST**)  
l'impression des instructions assemblées

## Lignes d'en-tête

**TITLE**  
**SUBTTL** *text*

Déterminer le contenu de la 1ère (**TITLE**) et  
2ème (**SUBTTL**) ligne imprimées sur chaque  
page

## Mise-en-page

**PAGE** *ydim,xdim*  
+

Déterminer les dimensions d'une page (nombre  
de lignes et de colonnes), le numéro de la section  
("+"), ou provoquer un saut de page (rien)

*Note : L'effet de toutes les pseudo-instructions contrôlant l'état du mode d'impression persiste*





## Impression de code conditionnel

**. SFCOND**  
**. LFCOND**  
**. TFCOND**

Supprimer (.SFCOND) ou provoquer (.LFCOND) l'impression des zones de code conditionnel. .TFCOND rétablit le mode pris par défaut.

## Impression des macro-instructions

Contrôle de l'impression du code-source résultant de la substitution de l'appel d'une macro-instruction :

**. XALL**

Supprimer (.SFCOND) ou provoquer (.LFCOND) l'impression des zones de code conditionnel. .TFCOND rétablit le mode pris par défaut.

**. SALL**

Supprimer l'impression de toute instruction résultant de l'appel d'une macro-instruction.

**. LALL**

Imprimer toutes les instructions résultant de l'appel d'une macro-instruction, à l'exception de celles commençant par “;;” (deux point-virgules).



## Chapitre 7 :

# Extensions de l'architecture de la famille 8086

### Résumé :

- **Evolution, aspects communs entre processeurs de la famille 8086**
- **Agrandissement de la taille d'un mot CPU**
- **Agrandissement de l'espace-mémoire**
  - **Descripteurs de segment, "real mode" et "protected mode"**
  - **Pagination**
- **Extension du répertoire d'instructions**



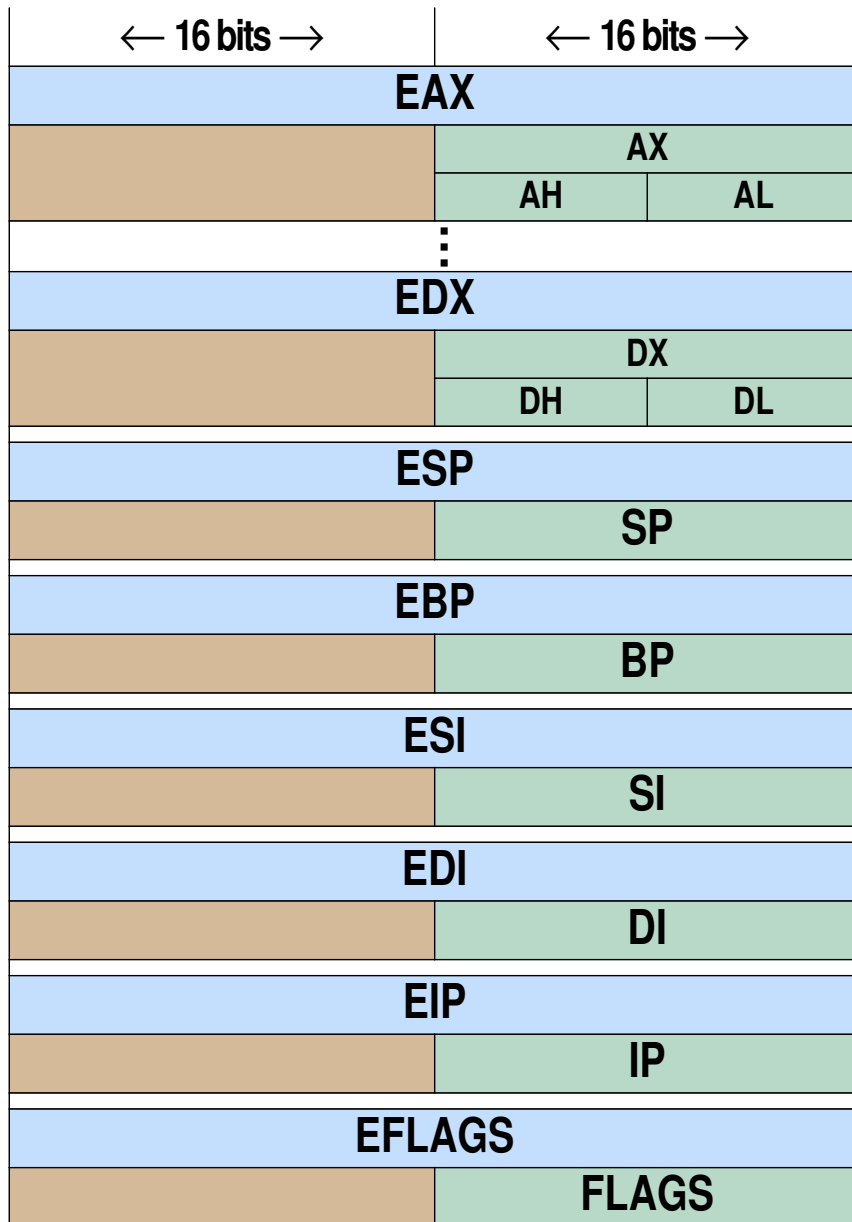
## Evolution de l'architecture du CPU 8086

	<b>mot-CPU (bits)</b>	<b>vitesse horloge</b>	<b>bus- données</b>	<b>mémoire cache</b>	<b>espace mémoire</b>	<b>bus- adresses</b>	<b>éléments nouveaux</b>
	<b>(bits)</b>	<b>(MHz)</b>	<b>(bits)</b>	<b>(bytes)</b>	<b>(bytes)</b>	<b>(bits)</b>	
<b>8086</b>	16		16	-	1 M	20	
<b>80186</b>	16		16	-	1 M	20	
<b>80286</b>	16		16	-	16 M	24	mode protégé (OS ↔ utilisateur!) instructions pour permettre l'accès à un espace de mémoire agrandi
<b>80386</b>	32		32	-	4 G	32	MMU ("memory management unit")
<b>80486</b>	32	50 →	32	8 K	4 G	32	co-processeur réel, instructions avec double vitesse
<b>Pentium (80586)</b>	32	60 →	64	16 K	4 G	32	co-processeur réel, deux processeurs (entiers) travaillant en parallèle

- Notes:**
- *Le Pentium fût initialement conçu sous le nom de 80586, nom abandonné par la suite pour des raisons de protection de marque (pas possible de protéger un nom 'numérique')*
  - *Pour chaque type de CPU, des variantes existent, p.ex. avec de largeur de bus différentes (tel que le 8088 avec un bus données de 8 bits) ou différents taux de l'horloge CPU.*



## Agrandissement de la taille d'un mot-CPU a 32 bits



→ implantation physique des registres

- 80386 et suite... registres de 32 bits,
- avant 80386 ... style 8086 (16 bits);

→ 80386 et suite... 2 modes de fonctionnement du CPU: choix entre

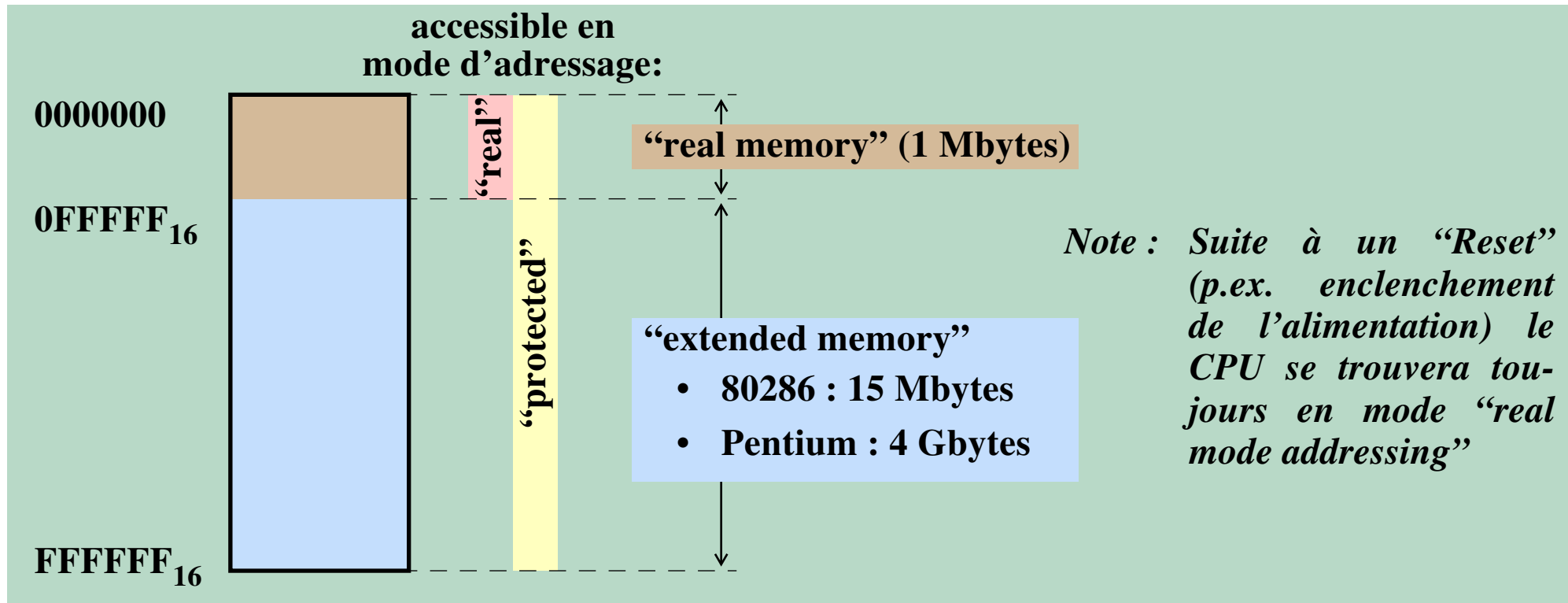
- “mode 16 bits”
- “mode 32 bits”;

→ en “mode 32 bits”: si un registre de 8 ou 16 bits est modifié, seulement les 8 ou 16 bits correspondant du registre physique de 32 bits seront modifiés;



## Espace-mémoire, mode d'adressage

### Concept



### Modifications du mécanisme d'adressage de segments physiques:

- **adresses de base >  $2^{20}$**  : contenu des registres de base = *descripteur* (au lieu d'une adresse);
- **taille des segments > 64 Kbytes** : adresses effectives sur 32 bits, extension de la portée de l'adresse-limite par le “granularity bit” du descripteur.
- **contrôle de l'accès** à la mémoire: les descripteurs contiennent une *adresse-limite* et des bits de contrôle (*protection*);

## Descripteurs de segments

byte #	Signification du contenu du descripteur					
0	adresse-limite (bits #15 - #0)					
2	adresse de base (bits #15 - #0)					
4	adresse de base (bits #23 - #16)			contrôle de l'accès		
6	adresse-limite (bits #19 - #16)	AV	O	D	G	adresse de base (bits #31 - #24)

*Note : Les bytes 6 et 7 dans un descripteur du 80286 sont toujours 0 (n'ont pas de signification).*

**adresse-limite ...** valeur (80286: 20 bits, 80386 et au delà: 32 bits) pour déterminer la dernière adresse valable du segment;

**adresse de base ...** valeur (80286: 16 bits, 80386 et au delà: 20 bits) déterminant l'adresse de base du segment; valeur \* 4096 ( $1000_{16}$ ) si le bit 'G' vaut 1

**contrôle de l'accès ...** liste de bits: contrôle du mode de sélection et protection du segment

**AV ...** 1 ... le segment est disponible ("available"), sinon 0;

**O ...**

**D ...** 1 ... instructions sur opérandes à 32 bits, 0 ... instructions compatible 8086;

**G ...** "granularity bit", voir adresse de base.

## Désignation d'un descripteur par un registre de base

Concept: registre de base devient une indirection sur une structure déterminant l'adresse

Registre de base: 

Sélecteur	TI	PR
-----------	----	----

Sélecteur ... par l'intermédiaire d'un **numéro de descripteur** désigner la représentation du descripteur en mémoire (table de descripteurs locale ou globale)

TI ... 0 : table globale, 1 : table locale

PR ... niveau de privilège (mécanisme de contrôle de l'accès)

### Pagination

Un mécanisme de pagination existe pour le CPU 80386 et ses successeurs; ce mécanisme s'applique au résultat de l'évaluation du registre de segment, respectivement du descripteur de segment.

Concepts:

- L'exécution d'un programme (y compris l'application du mécanisme de segmentation - style 8086, ou utilisation de descripteurs) produit des "**linear address**" (= adresses virtuelles).
- Avant d'accéder à la mémoire physique, une telle adresse est convertie en "**physical address**" en utilisant un dispositif de pagination (technique de pagination habituelle).
- Le CPU contient un ensemble de registres de contrôle ("**control registers**" CR0 ... CR3, un registre CR4 n'existe que dans le Pentium); ces registres déterminent:
  - si ou sinon le mécanisme de pagination sera utilisé,
  - les conditions particulières et les paramètres à appliquer.



# Extension du répertoire d'instructions

## → Maintien de la **compatibilité avec le 8086**

- recouvrement entre registres de différentes tailles;
- deux modes pour l'exécution d'opérations
  - “16-bit mode”: correspond au 8086,
  - “32-bit mode”: support de l'accès aux registres de 32 bits et du calcul d'adresses (virtuelles) de 32 bits;

le mode est déterminé par le descripteur de segment (i.e. son bit ‘D’) d'où provient l'instruction.

## → Enrichissement des **modes d'adressage**

- Possibilité d'utiliser tous les registres de 32 bits (EAX ... ESI) pour l'indexation.
- Nouveau mode: “Scaled Index Addressing” (double indexage, 2me registre fois 2, 4, 8)

## → **Instructions** pour l'exploitation des nouvelles **caractéristiques du CPU**, p.ex.

INVD (“invalid data cache”): marquer le contenu de la mémoire-cache comme non-valide

LGDT, LLDT (“load global/local descriptor table”): définition de la table de descripteurs.

## → **Instructions** supplémentaires pour augmenter le **confort de programmation**, p.ex.

LDS, LSS etc.: faciliter la définition du contenu des registres segment

BTC (“bit test and complement”) et opérations similaires: manipulation de bits.

## → **2 registres de base supplémentaires** (FS, GS)





## Chapitre 8 : Entrée / sortie

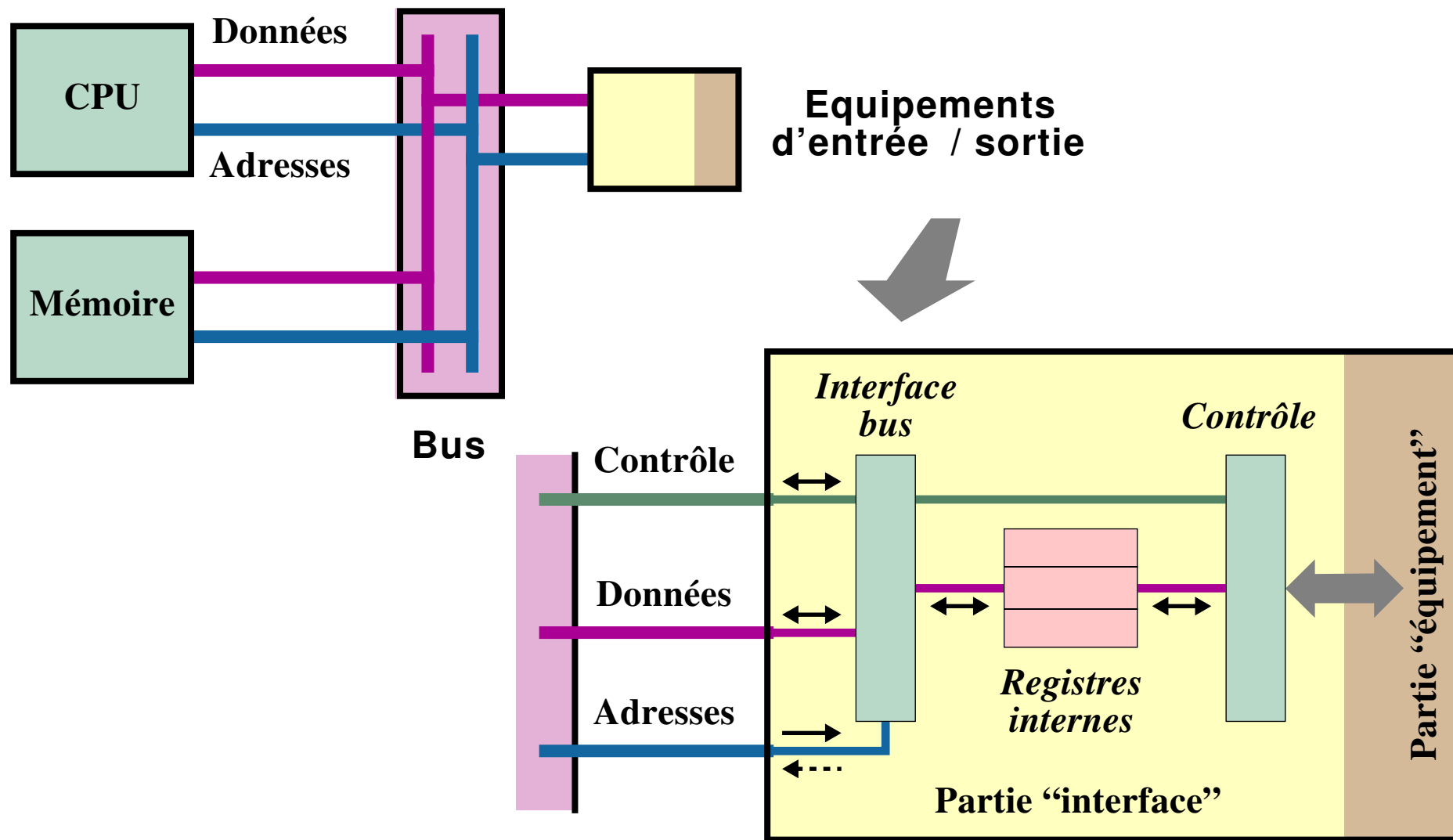
### Résumé :

- **Equipements d'entrée / sortie**
  - Architecture
  - Principe de fonctionnement
  - Adressage des registres internes
- **Programmation des opérations d'entrée / sortie**
  - Synchronisation de la fin des opérations
  - Initialisation des équipements d'entre / sortie
- **Exemple: l'interface ACIA (Motorola 6850)**



# Entrée / sortie dans un système micro-informatique

## Concept architectural



## Fonctionnement des opérations d'entrée / sortie

### ☛ *Echange de données entre*

- l'équipement central (CPU / mémoire) et
- un équipement périphérique

### ☛ Synchronisation entre activités se déroulant à différentes vitesses et rythmes



*registres internes des équipements*  
(dépôt intermédiaire des données)



### *Déroulement de l'entrée / sortie en 2 phases :*

#### ➔ Transfert interne

échange de données entre l'équipement central (e.g. un registre du CPU) et un registre interne de l'équipement périphérique,

#### ➔ Transfert physique

échange de données entre un registre interne de l'équipement périphérique et un support de données extérieur.



## Registres internes des équipements d'entrée / sortie

### ☛ Registres pour le transfert des données

- **Registre de réception** (accès-bus = lecture) :  
Données reçues par l'équipement - à transférer par la suite au CPU ou à la mémoire.
- **Registre de transmission** (accès-bus = écriture) :  
Données à transmettre par l'équipement - elles avaient été au préalable transférées du CPU ou de la mémoire.

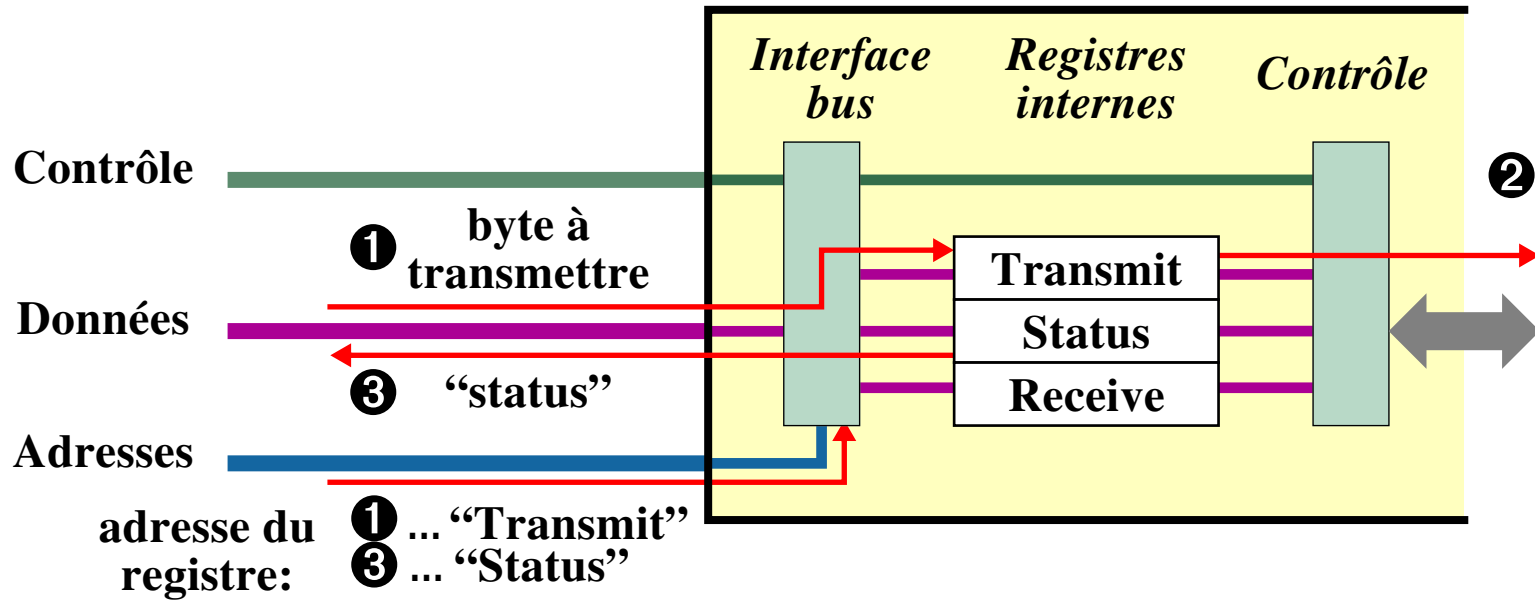
### ☛ Registres pour le contrôle des opérations d'entrées / sorties

- **Registre(s) de status** (accès-bus = lecture) :  
Information binaire représentant l'état de fonctionnement actuel de l'équipement, prêt pour la consultation par le CPU.
- **Registre(s) de contrôle** (accès-bus = écriture) :  
Liste de bits pour contrôler des fonctions particulières de l'équipement; initialisée par le CPU pour contrôler le déroulement des opérations d'entrées / sorties.
- **Registres numériques auxiliaires** (sens de l'accès selon fonction du registre) :  
Valeurs numériques liées au déroulement de l'opération d'entrée / sortie (p. ex. compteur de bytes / mots, adresse physique du bloc lu/écrit, etc.).



# Déroulement d'une opération d'entrée / sortie

## Ecriture

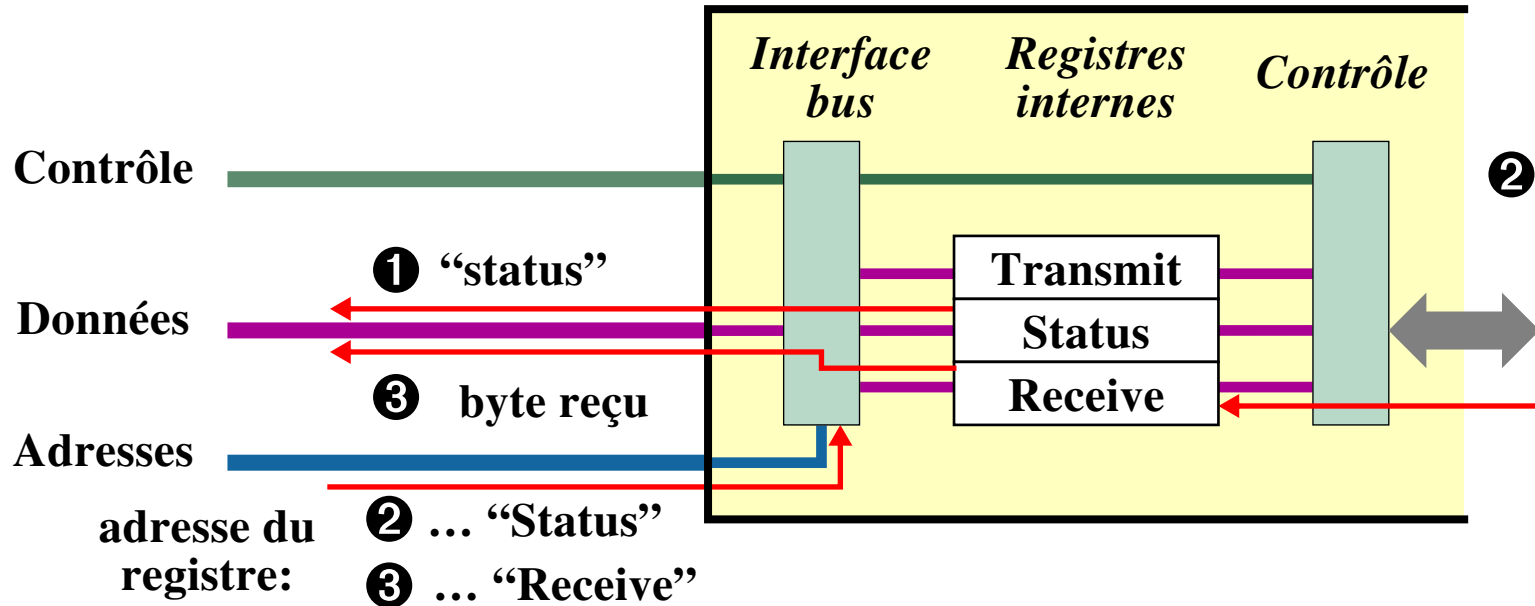


- ①** ... Transfert des données à transmettre du CPU (ou de la mémoire) au registre “Transmit”
- ②** ... Déclenchement de l’opération physique d’écriture
- ③** ... Vérification si l’opération d’écriture est effectivement terminé

### Deux méthodes pour l’action **①** :

- **Déclenchement automatique** : Le dépôt d’une nouvelle valeur dans le registre “Transmit” déclenche automatiquement sa transmission physique par l’équipement.
- **Déclenchement explicite** : Le CPU doit exécuter une instruction d’entrée / sortie, p.ex. envoyer une valeur particulière à un registre de contrôle.

## Lecture



- ① ... Déclenchement de l'opération physique de lecture
- ② ... Vérification si l'opération de lecture est effectivement terminée
- ③ ... Transfert des données reçues du registre "Receive" au CPU (ou à la mémoire)

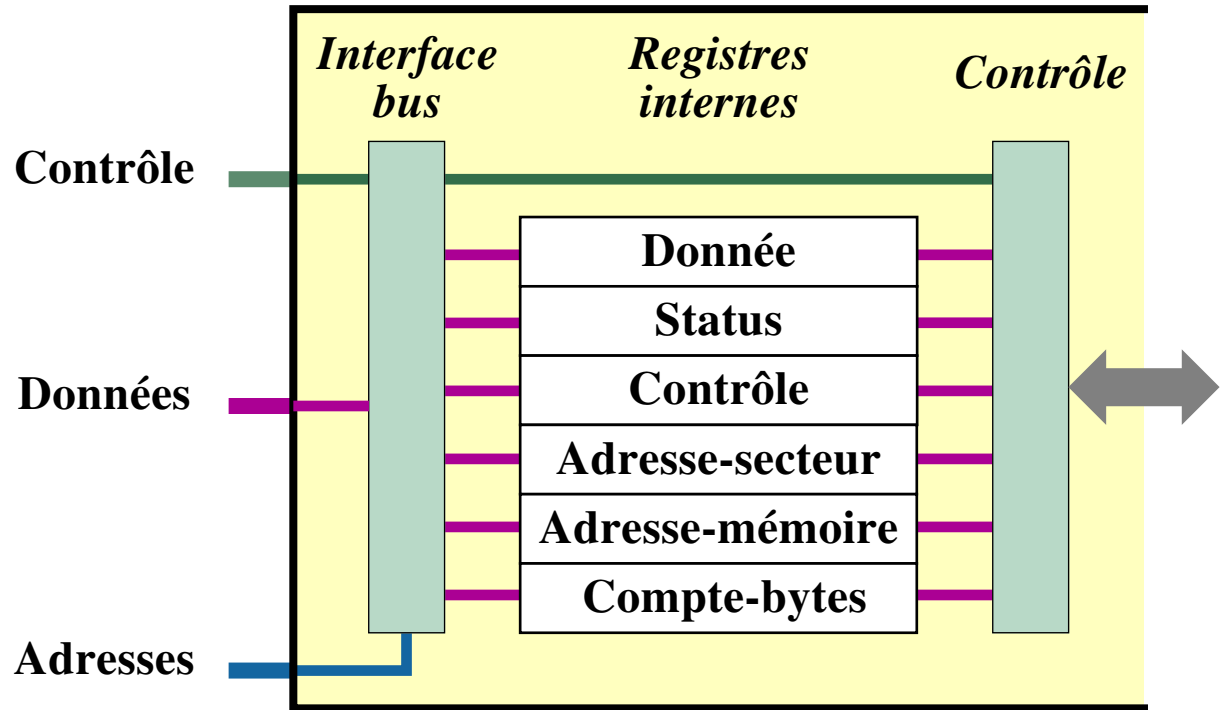
### Trois méthodes pour l'action ② :

- **Déclenchement automatique** : le prélèvement d'une valeur du registre "Receive" déclenche automatiquement la prochaine lecture physique.
- **Déclenchement explicite** : le CPU doit exécuter une instruction d'entrée / sortie, p. ex. envoyer une valeur particulière à un registre de contrôle.
- **Lecture permanente** : Stocker toute donnée reçue par l'équipement immédiatement dans le registre "Receive".

## Esquisse d'un contrôleur DMA

(**'Direct Memory Access'**):

illustration du principe de fonctionnement (exemple: contrôleur de disque)



1. Le CPU définit: Adresse-secteur, Adresse-mémoire, Compte-bytes, Contrôle
2. L'interface commence l'opération: positionnement de la tête ("Adresse-secteur")
3. L'interface exécute un processus itératif jusqu'à ce que "Compte-bytes" arrive à 0
  - lecture/écriture d'un byte: registre "Donnée" (séquence avec DMA inversée si écriture)
  - exécution d'un cycle-bus DMA: transfert registre "Donnée"  $\Leftrightarrow$  "Adresse-mémoire"
  - mise-à-jour des registres: "Adresse-mémoire" + 1, "Nombre-bytes" - 1
4. Fin itération
  - modification du registre "Status" pour indiquer 'opération complète'
  - génération d'une interruption (conditionnel, selon liste de bits dans "Contrôle")

## Fin de l'opération d'entrée / sortie physique

1. *Observation du registre "status"* : Le CPU consulte (périodiquement) le contenu du registre "status" pour connaître l'état d'avancement de l'opération en cours.

= "**Attente active**"

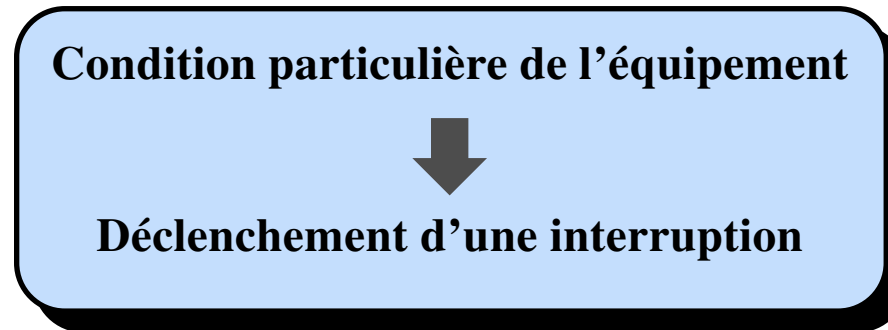
2. *Interruption* : L'équipement déclenche une interruption et, par conséquent, l'exécution d'une routine de traitement d'interruption par le CPU.





## Interruption d'entrée / sortie

### Déroulement d'une interruption entrée / sortie



### Conditions déclenchant une interruption

La sélection des conditions “actives” est déterminée par le contenu d'un registre de contrôle.

### Exemples :

- nouvelle donnée reçue,
- transmission d'une donnée accomplie,
- faute détecté par l'équipement, etc.

**Pour une opération sans interruption : ne sélectionner aucune de ces conditions, p. ex. dans des application très simples.**



# Initialisation d'un équipement d'entrée / sortie

## Opération "reset"

= remise dans un état de repos bien défini :

- après l'enclenchement de l'alimentation électrique,
- suite à la détection d'erreurs,
- lors de modifications du mode d'opération.

Deux types de "reset" :

### 1. "*Hard Reset*" :

Déclenché par un signal externe (p. ex. ligne du bus), souvent commun à tous les équipements d'un système (= "Reset général").

### 2. "*Soft Reset*" : (= "Reset programmé")

Déclenché par le dépôt d'une valeur particulière dans un registre de contrôle de l'équipement.

## Séquence d'initialisation

- Définit le mode d'opération d'un équipement par une suite d'opérations d'entrées / sorties, déterminant le contenu de chaque registre de contrôle spécifique au mode.
- Généralement nécessaire après une opération "Reset".



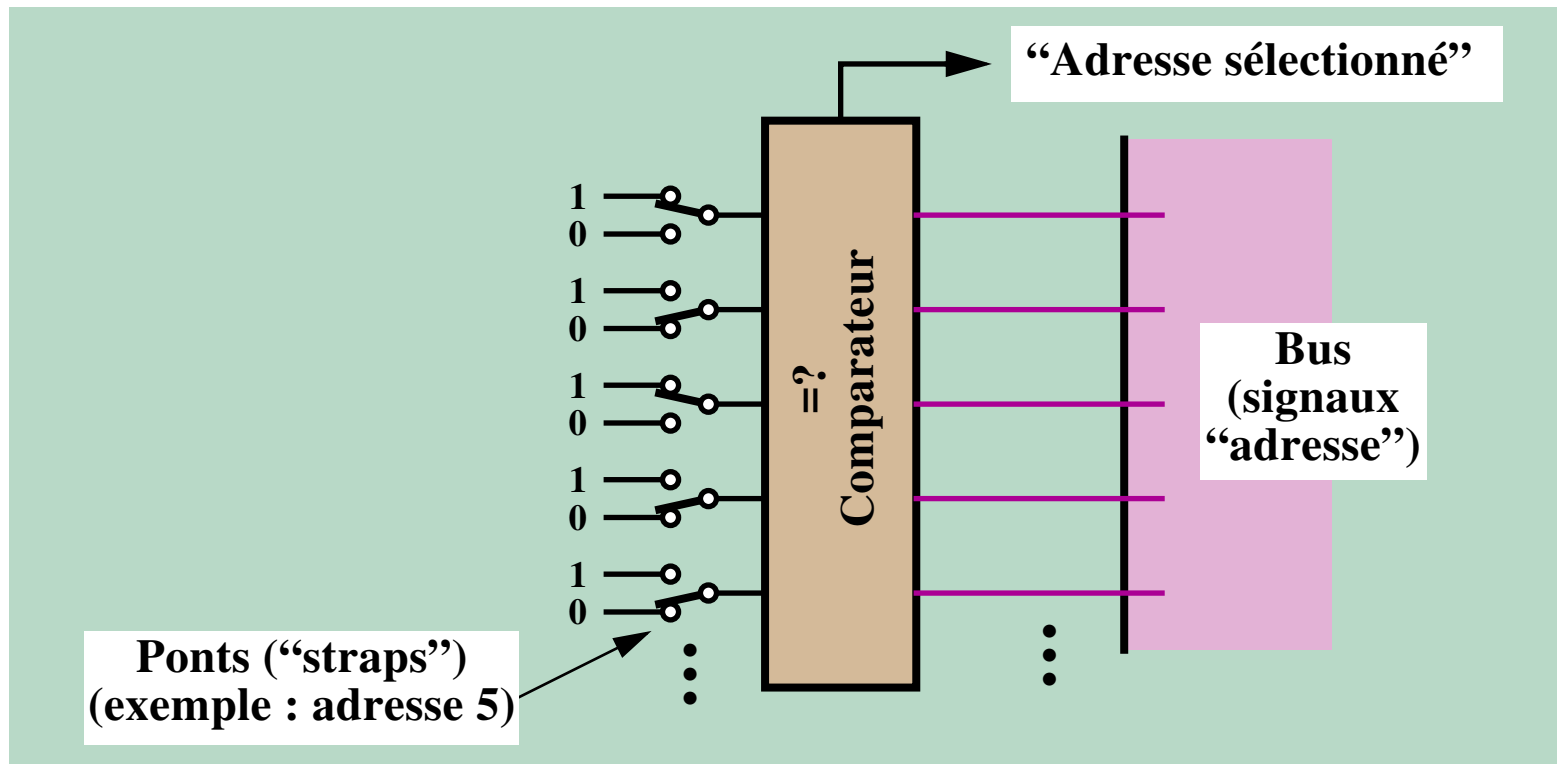
## Adressage des registres internes

### Adresses-bus individuelles

Adresse individuelle pour chaque registre  $\Rightarrow$  permet de choisir un registre spécifique.

Génération d'un signal "adresse sélectionnée" lorsque cette adresse apparaît sur le bus.

Choix d'une adresse, détection de la sélection de l'adresse:



## Regroupement des adresses des registres d'un équipement :

- Normalement une seule adresse de base peut être configurée pour un équipement.
- Adresses des registres de l'équipement ... suite contiguë commençant à cette adresse.

## Recouvrement d'adresses :

Deux registres dont un est accessible seulement en lecture et l'autre seulement en écriture peuvent porter la même adresse.

Cette technique est souvent utilisée pour des paires de registres dont les fonctions sont complémentaires (p. ex. registres de transmission / de réception, registres "Status" / "Contrôle").

## Documentation

Les documents des fournisseurs des interfaces d'équipement d'entrées / sorties doivent fournir toutes les informations sur l'utilisation et l'adressage des registres nécessaires pour la programmation de l'interface.

L'expérience montre cependant que ces renseignements sont souvent défailants et que l'expérience pratique et la communication entre utilisateurs jouent un rôle important ("news", etc.)



## Entrée / sortie “memory-mapped” ou “non-memory-mapped”

### ➤ Entrée / sortie “memory-mapped” :

Existence d'un seul et unique espace d'adresses pour la mémoire et pour les registres;

- ➔ pas de distinction entre une adresse-mémoire et celle d'un registre d'entrée / sortie.
- ➔ Pas de nécessité de disposer d'instructions spécifiques pour les opérations d'entrée / sortie, accès aux registres internes des équipements d'entrées / sorties par les mêmes instructions que celles utilisées pour l'accès à la mémoire.

### ➤ Entrée / sortie “non-memory-mapped” :

Existence d'espaces d'adresses différents et distincts pour la mémoire et pour les registres;

- ➔ notion de “porte d'entrée / sortie” : adresse d'un équipement = adresse du registre correspondant (ou première adresse d'une suite de registres).
- ➔ Nécessité de disposer d'instructions spécifiques pour les opérations d'entrées / sorties (p.ex. IN, OUT, RBYTE, RWORD, WBYTE, etc.).

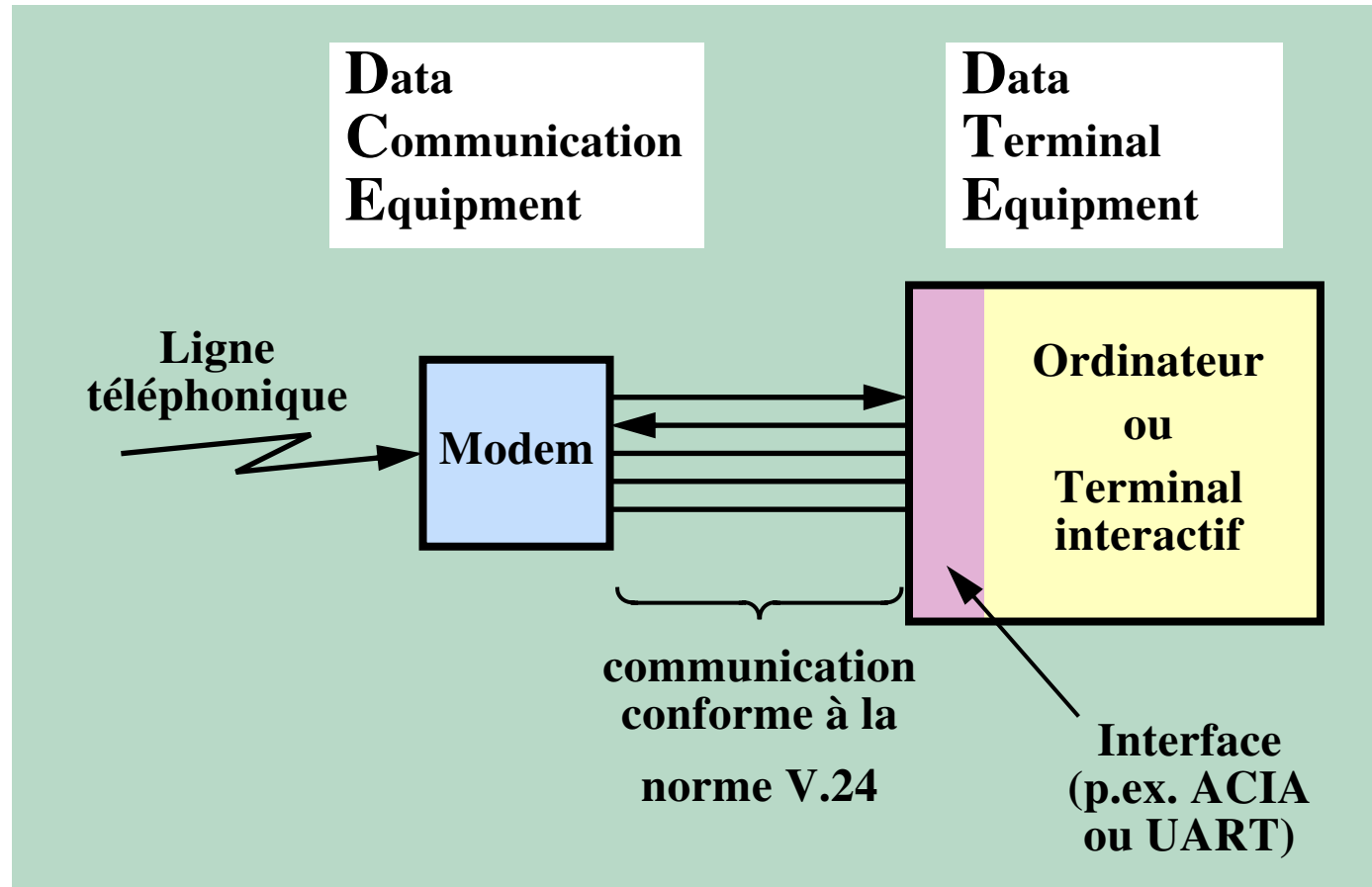


# ACIA (Motorola 6850)

(Asynchronous Communication Interface Adaptor)

Interface pour la communication sérielle de données selon le standard V.24.

Correspondant chez Intel : UART (= Universal Asynchronous Receiver-Transmitter)



## Tâches de l'interface ACIA

### ➤ Mise-en-série / re-assemblage des données

Conversion entre représentation interne (byte) et représentation externe (chaîne de bits).

### ➤ Gestion des signaux pour le contrôle des opérations d'entrée/sortie

Contrôle du sens de la communication, synchronisation CPU - évènements externes.

### ➤ Détection d'erreurs de transmission.

Signalisation des erreurs au CPU, actions pour la remise en situation normale.

### Standard V.24 de l'UIT

- Utilisation de “modems” pour la communication entre équipements informatiques à l'aide de lignes analogiques; V.24 : communication entre l'ordinateur et le modem.
- Représentation des données échangées entre modem et ordinateur : binaire, sérielle.
- Multiplexage du sens de la communication entre les modems à l'aide de signaux de contrôle échangés entre l'ordinateur et le modem.

*Note : UIT = Union Internationale des Télécommunications) (son organe de standardisation fut anciennement connu sous le nom CCITT = Comité Consultatif International Télégraphique et Téléphonique)'*

*Norme américaine correspondante = RS 232C*



## Signaux échangés entre l'interface et le modem

### → Transfert sériel de données :

**“Receive Data”**

**“Transmit Data”**

Séquences de bits = représentation sérielle des données reçues et transmises.

### → Echantillonnage des données :

**“Receive Clock”**

**“Transmit Clock”**

Impulsions d'horloge pour l'échantillonnage lors de la mise-en-série et la re-constitution des données

### → Multiplexage du sens de la communication :

**“Request-to-Send”**

Indication de l'interface au modem local que le modem du partenaire peut commencer une transmission.

**“Clear-to-Send”**

Indication du modem à l'interface qu'il peut commencer une transmission.

### → Ligne en état de fonctionnement :

**“Carrier Detect”**

Indication du modem à l'interface qu'il reçoit le signal “porteuse”.





## Signification des contenus des registres (ACIA)

### Le registre "status"

bit #								signification	
7	6	5	4	3	2	1	0		
0	.	.	.	.	.	.	.	<b>Requête d'interruption</b>	<b>absent</b>
1	.	.	.	.	.	.	.		<b>présent</b>
.	0	.	.	.	.	.	.	<b>Erreur parité</b>	<b>non</b>
.	1	.	.	.	.	.	.		<b>oui</b>
.	.	0	.	.	.	.	.	<b>Erreur "receiver overrun"</b>	<b>non</b>
.	.	1	.	.	.	.	.		<b>oui</b>
.	.	.	0	.	.	.	.	<b>Erreur "framing"</b>	<b>non</b>
.	.	.	1	.	.	.	.		<b>oui</b>
.	.	.	.	0	.	.	.	<b>"Clear-to-Send"</b>	<b>état de repos</b>
.	.	.	.	1	.	.	.		<b>actif</b>
.	.	.	.	.	0	.	.	<b>"Carrier Detect"</b>	<b>état de repos</b>
.	.	.	.	.	1	.	.		<b>actif</b>
.	.	.	.	.	.	0	.	<b>Registre "Transmit Data"</b>	<b>plein</b>
.	.	.	.	.	.	1	.		<b>vide</b>
.	.	.	.	.	.	.	0	<b>Registre "Receive Data"</b>	<b>plein</b>
.	.	.	.	.	.	.	1		<b>vide</b>



# Le registre "contrôle"

bit #								signification		
7	6	5	4	3	2	1	0			
1	.	.	.	.	.	.	.	Interruption réception	possible	
0	.	.	.	.	.	.	.		bloquée	
.	0	0	.	.	.	.	.	} Interruptions de transmission bloquée		
.	1	0	.	.	.	.	.			
.	0	1	.	.	.	.	.		Interruption de transmission possible	
.	1	0	.	.	.	.	.		Transmettre un 'break'	
.	.	.	1	1	1	.	.	# de data bits	parité	# de stop bits
.	.	.	1	1	0	.	.	8	impaire	1
.	.	.	1	0	1	.	.	8	paire	1
.	.	.	1	0	0	.	.	8	aucune	1
.	.	.	0	1	1	.	.	8	aucune	2
.	.	.	0	1	0	.	.	7	impaire	1
.	.	.	0	0	1	.	.	7	paire	1
.	.	.	0	0	0	.	.	7	impaire	2
.	.	.	0	0	0	.	.	7	paire	2
.	.	.	.	.	.	0	0	horloge	} tel que fournie	
.	.	.	.	.	.	0	1		/16	
.	.	.	.	.	.	1	0		/64	
.	.	.	.	.	.	1	1		"reset" général	



### **Note : La fonction 'break':**

*Un 'break' est un signal transmis sur une ligne de communication, transmis comme un caractère; mais*

- *conceptuellement, le 'break' est transmis sur une voie de transmission indépendante de tout autre communication,*
- *le 'break' est donc transmis "en parallèle" à l'échange des données "normales".*

*La fonction d'un 'break' peut être utilisée pour signaler une condition particulière, par exemple pour interrompre la transmission d'un flot de données par une sorte "d'interruption".*

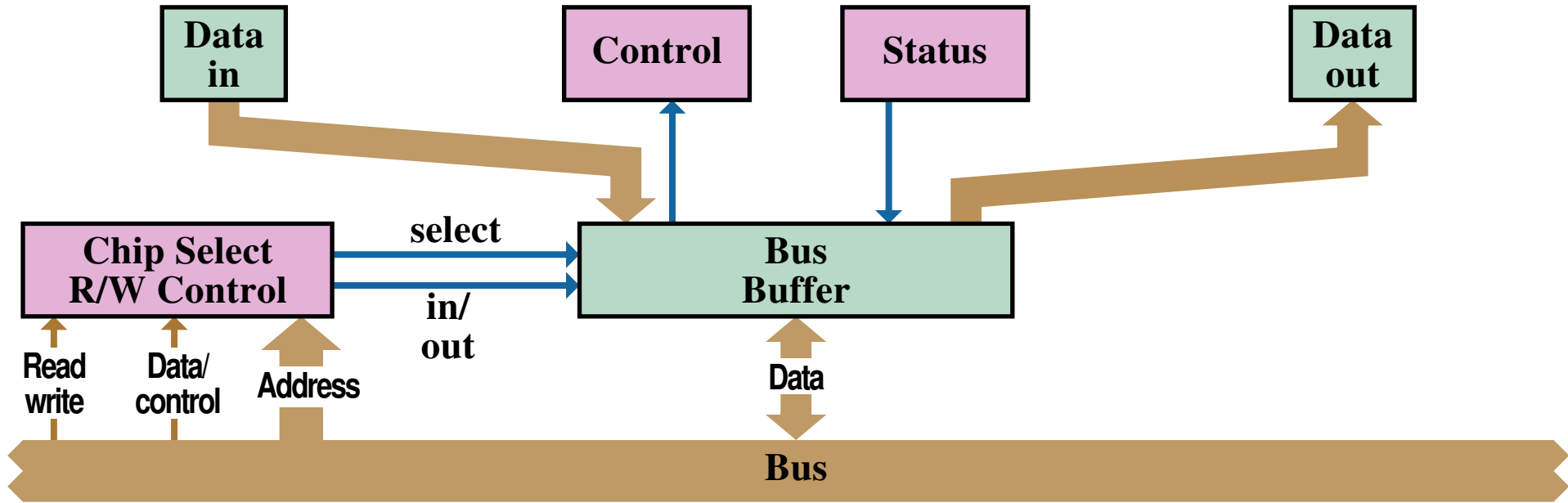
*L'implantation de la fonction 'break' se fait*

- *en mettant la ligne de communication dans un état particulier, différent de celui observé pendant la transmission de données "normales" (p.ex. une condition d'erreur de transmission toute particulière),*
- *la définition de cet état doit être convenu entre les deux partenaire d'une communication.*



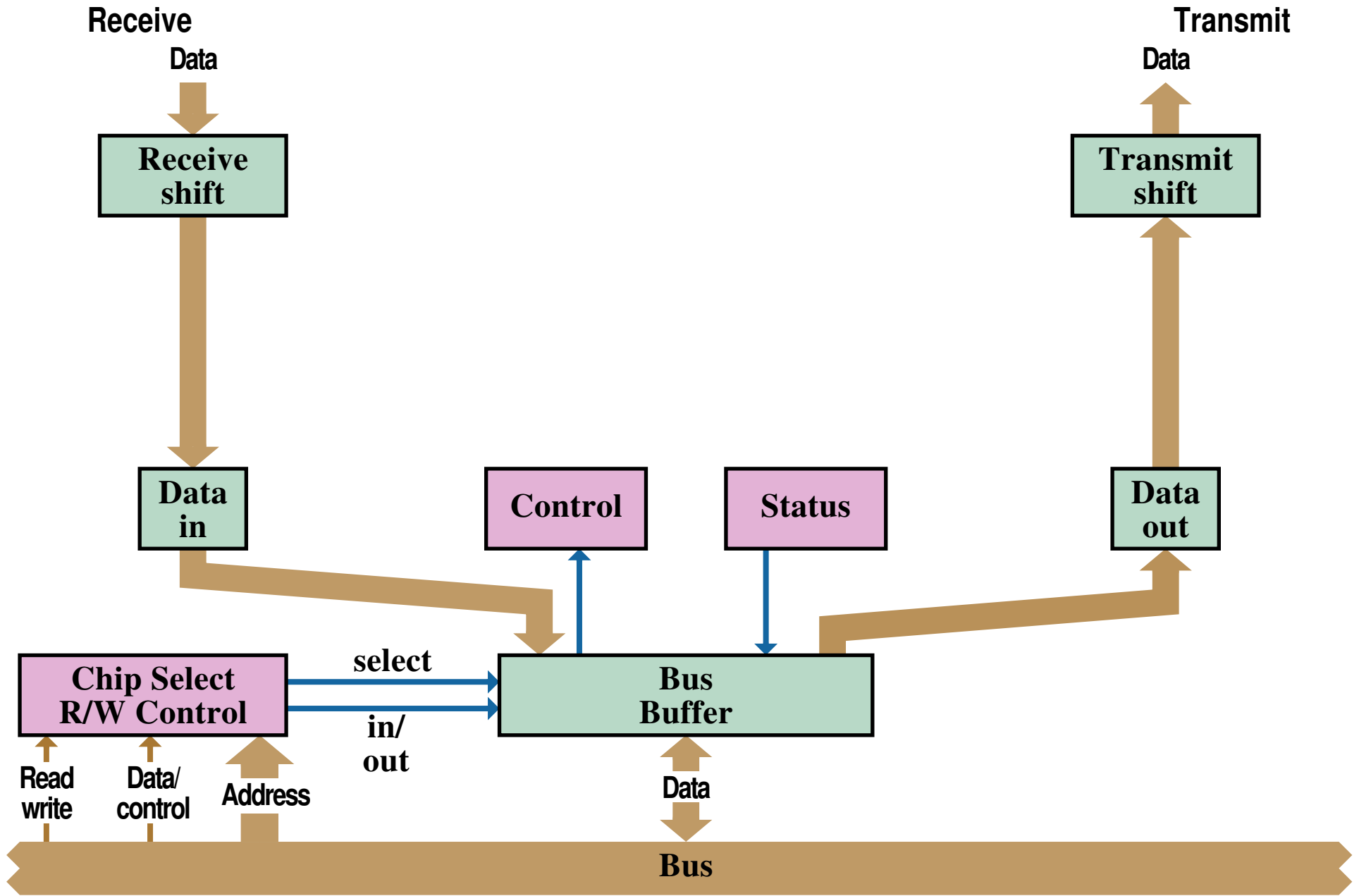
Receive

Transmit

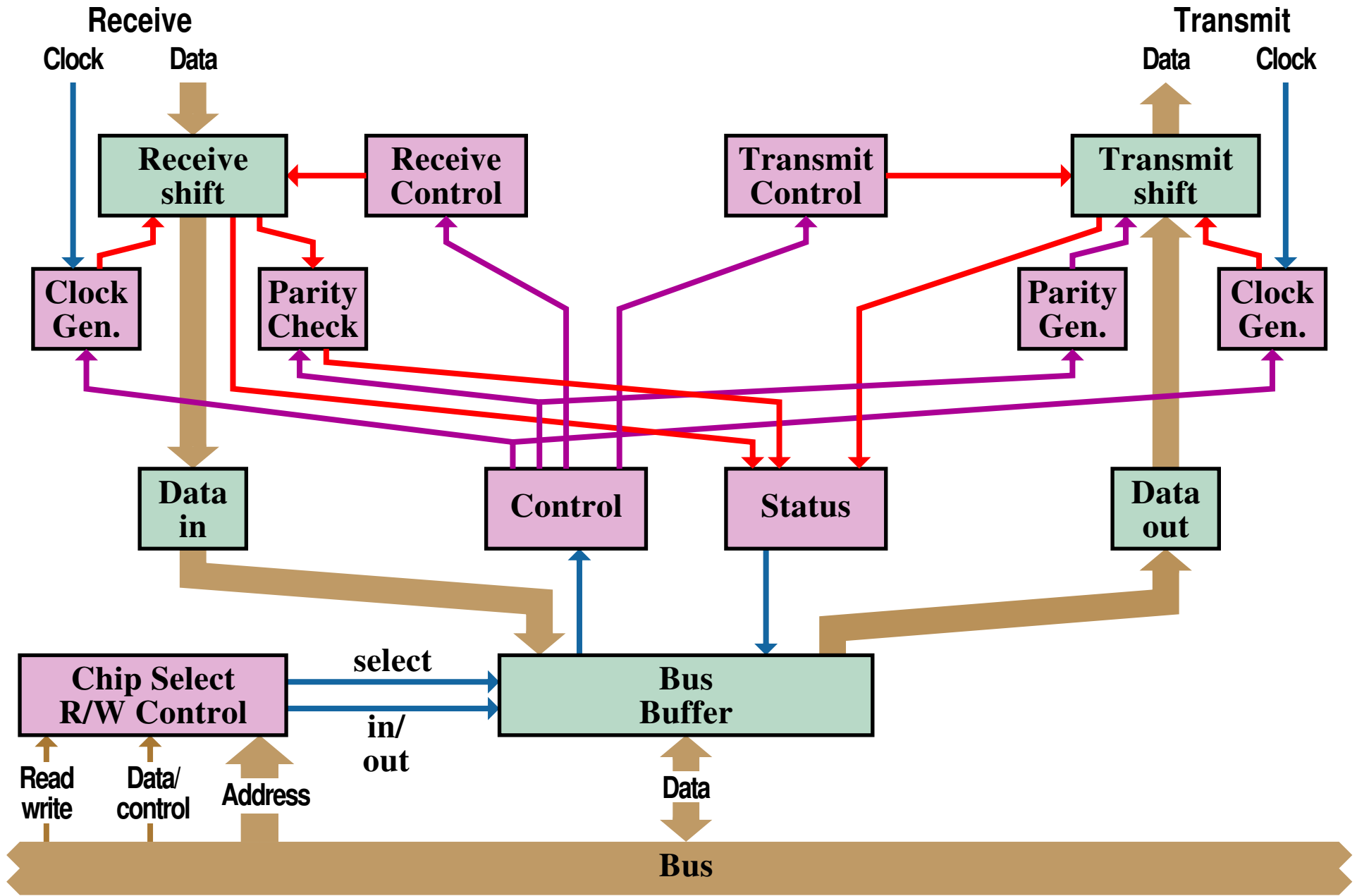


ACIA: Schéma (1)

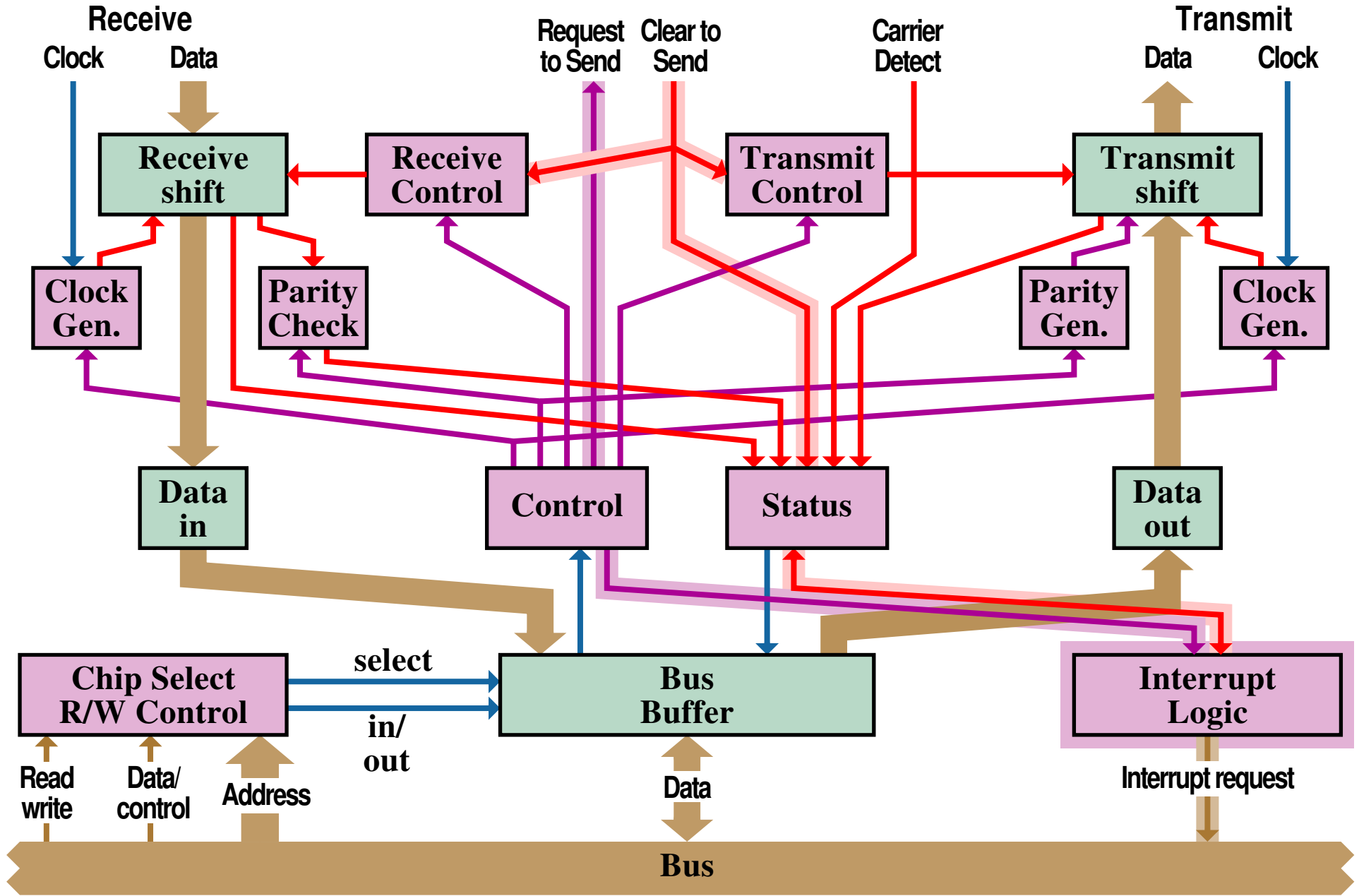




ACIA: Schéma (2)



ACIA: Schéma (3)



ACIA: Schéma (4)

## Exemple de programmation

; Définir les adresses des registres  
; (doit correspondre à la configuration câblée)

<b>1</b>	<b>ACIADR</b>	<b>EQU</b>	<b>\$810</b>	<b>; reg. transmit et receive</b>
<b>2</b>	<b>ACIASR</b>	<b>EQU</b>	<b>\$811</b>	<b>; reg. status</b>
<b>3</b>	<b>ACIACR</b>	<b>EQU</b>	<b>\$811</b>	<b>; reg. contrôle</b>

; Initialisation

<b>4</b>	<b>MOVE</b>	<b>#3, ACIACR</b>	<b>; Reset</b>
<b>5</b>	<b>MOVE</b>	<b>#4+2, ACIACR</b>	<b>; 7 bits, impair, 2 stop-bits, horloge / 64</b>

; Réception d'un byte

<b>6</b>	<b>ATTRCV</b>	<b>BTST</b>	<b>#0, ACIASR</b>	<b>; ZF=1 si "receiver plein"</b>
<b>7</b>	<b>BNE</b>	<b>ATTRCV</b>		<b>; attendre si vide</b>
<b>8</b>	<b>MOVE</b>	<b>ACIADR, dest</b>		<b>; chercher le byte reçu</b>

; Transmission d'un byte

<b>9</b>		<b>MOVE</b>	<b>srce, ACIADR</b>	<b>; transmettre le byte</b>
<b>10</b>	<b>ATTTRM</b>	<b>BTST</b>	<b>#1, ACIASR</b>	<b>; ZF=1 si "transm. plein"</b>
<b>11</b>		<b>BEQ</b>	<b>ATTTRM</b>	<b>; attendre si plein</b>





## Notes sur l'assembleur 68000

<b>BTST</b>	<i>num, oper</i>	<b>Zero-flag := 1</b> si le bit “ <i>num</i> ” de l’opérande “ <i>oper</i> ” vaut <b>0</b> , sinon <b>Zero-flag := 0</b>
<b>BEQ</b>	<i>adresse</i>	<b>branchement si le Zero-flag == 1</b>
<b>BNE</b>	<i>adresse</i>	<b>branchement si le Zero-flag == 0</b>
<b>MOVE</b>	<i>srce,dest</i>	<b>Transfert d’une opérande de “<i>srce</i>” à “<i>dest</i>” (# ... valeur immédiate)</b>



# Liste des instructions du CPU Intel 8086

Nom (mnémonique)		Action	Nom (mnémonique)		Action
<b>AAA</b>	ASCII Adjust for Addition	opération complexe, voir littérature	<b>JA</b>	Jump if Above =JNB	branchement cond. ( $op2 > op1$ )
<b>AAD</b>	ASCII Adjust for Division	opération complexe, voir littérature	<b>JAE</b>	Jump if Above or Eq. =JNB	branchement cond. ( $op2 \geq op1$ )
<b>AAM</b>	ASCII Adjust for Multiplic.	opération complexe, voir littérature	<b>JB</b>	Jump if Below =JNAE	branchement cond. ( $op2 < op1$ )
<b>AAS</b>	ASCII Adjust for Subtr..	opération complexe, voir littérature	<b>JBE</b>	Jump if Below or Eq. =JNA	branchement cond. ( $op2 \leq op1$ )
<b>ADC</b>	Add with Carry	$op1 + op2 (+1) \rightarrow op1 (+1 \text{ si } CF = 1)$	<b>JCXZ</b>	Jump if CX = 0	branchement si CX = 0
<b>ADD</b>	Add	$op1 + op2 \rightarrow op1$	<b>JE</b>	Jump if Equal =JZ	branchement cond. ( $op2 = op1$ )
<b>AND</b>		$op1 \text{ AND } op2 \rightarrow op1$	<b>JG</b>	Jump if Greater =JNL	branchement cond. ( $op2 > op1$ )
<b>CALL</b>		appel de procédure	<b>JGE</b>	Jump if Greater or Eq. =JNL	branchement cond. ( $op2 \geq op1$ )
<b>CBW</b>	Convert Byte to Word	AL $\rightarrow$ AX (extension bit de signe)	<b>JL</b>	Jump if Less =JNGE	branchement cond. ( $op2 < op1$ )
<b>CLC</b>	Clear Carry flag	0 $\rightarrow$ CF ('Carry' bit, 'flag-register')	<b>JLE</b>	Jump if Less or Equal =JNG	branchement cond. ( $op2 \leq op1$ )
<b>CLD</b>	Clear Direction flag	0 $\rightarrow$ DF ('Direction' bit, 'flag-reg.')	<b>JMP</b>	Jump	branchement inconditionnel
<b>CLI</b>	Clear Interrupt flag	0 $\rightarrow$ IF ('Interrupt' bit, 'flag-register')	<b>JNA</b>	Jump if Not Above =JBE	branchement cond. ( $op2 \leq op1$ )
<b>CMC</b>	Complement Carry flag	NOT CF $\rightarrow$ CF ('Carry' bit, 'flag-reg.')	<b>JNAE</b>	Jump if Not Above or Eq. =JB	branchement cond. ( $op2 < op1$ )
<b>CMP</b>	Compare	$op1 - op2$	<b>JNB</b>	Jump if Not Below =JAE	branchement cond. ( $op2 \geq op1$ )
<b>CMPB</b>	Compare Byte	zone-mémoire – zone-mémoire (byte)	<b>JNBE</b>	Jump if Not Below or Eq. =JA	branchement cond. ( $op2 > op1$ )
<b>CMPW</b>	Compare Word	zone-mémoire – zone-mémoire (mot)	<b>JNE</b>	Jump if Not Equal =JNZ	branchement cond. ( $op2 \neq op1$ )
<b>CWD</b>	Convert Word to Double	AX $\rightarrow$ DX:AX (extension bit de signe)	<b>JNG</b>	Jump if Not Greater =JLE	branchement cond. ( $op2 \leq op1$ )
<b>DAA</b>	Decimal Adjust for Addition	opération complexe, voir littérature	<b>JNGE</b>	Jump if Not Greater or Eq. =JL	branchement cond. ( $op2 < op1$ )
<b>DAS</b>	Decimal Adjust for Subtr.	opération complexe, voir littérature	<b>JNL</b>	Jump if Not Less =JGE	branchement cond. ( $op2 \geq op1$ )
<b>DEC</b>	Decrement	$op1 - 1 \rightarrow op1$	<b>JNLE</b>	Jump if Not Less or Eq. =JG	branchement cond. ( $op2 > op1$ )
<b>DIV</b>	Divide	DX:AX / $op1 \rightarrow$ AX, reste $\rightarrow$ DX AX / $op1 \rightarrow$ AL, reste $\rightarrow$ AH	<b>JNO</b>	Jump if Not Overflow	branchement cond. (si pas 'overflow')
<b>ESC</b>	Escape	$op1 \rightarrow$ bus (pas d'autre action CPU)	<b>JNP</b>	Jump if Not Parity =JPO	branchement cond. (si parité impaire)
<b>HLT</b>	Halt	arrêter le CPU	<b>JNS</b>	Jump if Not Sign	branchement cond. (si valeur positive)
<b>IDIV</b>	Integer Divide	DX:AX / $op1 \rightarrow$ AX, reste $\rightarrow$ DX AX / $op1 \rightarrow$ AL, reste $\rightarrow$ AH (op. abs.)	<b>JNZ</b>	Jump if Not Zero =JNE	branchement cond. (si résultat $\neq$ 0)
<b>IMUL</b>	Integer Multiply	$op1 * AX \rightarrow$ DX:AX $op1 * AL \rightarrow$ AX (op. abs.)	<b>JO</b>	Jump if Overflow	branchement cond. (si 'overflow')
<b>IN</b>	Input	entrée/sortie $\rightarrow$ AL	<b>JP</b>	Jump if Parity =JPE	branchement cond. (si parité paire)
<b>INC</b>	Increment	$op1 + 1 \rightarrow op1$	<b>JPE</b>	Jump if Parity Even =JP	branchement cond. (si parité paire)
<b>INT</b>	Interrupt	interruption par vecteur numéro $op1$	<b>JPO</b>	Jump if Parity Odd =JNP	branchement cond. (si parité impaire)
<b>INTO</b>	Interrupt Overflow	interr. par vecteur numéro 4 (si OF = 1)	<b>JS</b>	Jump if Sign	branchement cond. (si valeur négative)
<b>INW</b>	Input Word	entrée/sortie $\rightarrow$ AX	<b>JZ</b>	Jump if Zero =JE	branchement cond. (si résultat = 0)
<b>IRET</b>	Interrupt Return	retour du traitement d'interruption	<b>LAHF</b>	Load AH with Flags	bits arithmétiques du 'flag-reg.' $\rightarrow$ AH
			<b>LDS</b>	Load pointer to DS	adresse de $op2 \rightarrow$ DS:op1
			<b>LEA</b>	Load Effective Addr.	adresse de $op2 \rightarrow op1$
			<b>LES</b>	Load pointer to ES	adresse de $op2 \rightarrow$ ES:op1

opérandes absolues

opérandes absolues



Nom (mnémonique)	Action
<b>LOCK</b>	réserve du bus pour > 1 cycle
<b>LODB</b> Load Byte	zone-mémoire → AL
<b>LODW</b> Load Word	zone-mémoire → AX
<b>LOOP</b>	branchement si CX = 0
<b>LOOPE</b> Loop while Equal =LOOPZ	branchement si CX = 0 et ZF = 1
<b>LOOPNE</b> Loop while Not Eq. =LOOPNZ	branchement si CX = 0 et ZF = 0
<b>LOOPNZ</b> Loop while Not Zero =LOOPNE	branchement si CX = 0 et ZF = 0
<b>LOOPZ</b> Loop while Zero =LOOPE	branchement si CX = 0 et ZF = 1
<b>MOV</b> Move	op2 → op1
<b>MOVB</b> Move Byte	zone-mémoire → zone-mémoire
<b>MOVW</b> Move Word	zone-mémoire → zone-mémoire
<b>MUL</b> Multiply	op1 * AX → DX:AX op1 * AL → AX
<b>NEG</b> Negate	0 - op1 → op1
<b>NOT</b>	NOT op1 → op1
<b>OR</b>	op1 OR op2 → op1
<b>OUT</b> Output	AL → entrée/sortie
<b>OUTW</b> Output Word	AX → entrée/sortie
<b>POP</b>	des-empiler → op1
<b>POPF</b> Pop Flags	des-empiler → 'flag-register'
<b>PUSH</b>	op1 → empiler
<b>PUSHF</b> Push Flags	'flag-register' → empiler
<b>RCL</b> Rotate Left with Carry	décalage circulaire gauche par CF
<b>RCR</b> Rotate Right with Carry	décalage circulaire droite par CF
<b>REP</b> Repeat	pré-fixe: répétition sur zone-mémoire
<b>RET</b> Return	retour d'une procédure
<b>ROL</b> Rotate Left	décalage circulaire gauche
<b>ROR</b> Rotate Right	décalage circulaire droite
<b>SAHF</b> Store AH to Flags	AH → bits arithm. du 'flag-register'
<b>SAL</b> Shift Arithm. Left =SHL	décalage à gauche (0-remplissage)
<b>SAR</b> Shift Arithmetic Right	décalage à droite, extension du signe
<b>SBB</b> Subtract with Borrow	op1 - op2 (-1) → op1 (-1 si CF = 1)
<b>SCAB</b> Scan Byte	AL - zone-mémoire
<b>SCAW</b> Scan Word	AX - zone-mémoire
<b>SHL</b> Shift Logical Left =SAL	décalage à gauche (0-remplissage)
<b>SHR</b> Shift Logical Right	décalage à droite (0-remplissage)

1ère action:  
CX - 1 → CX

Nom (mnémonique)	Action
<b>STC</b> Set Carry flag	1 → CF ('Carry' bit, 'flag-register')
<b>STD</b> Set Direction flag	1 → DF ('Direction' bit 'flag-reg.')
<b>STI</b> Set Interrupt flag	1 → IF ('Interrupt' bit, 'flag-register')
<b>STOB</b> Store Byte	AL → zone-mémoire
<b>STOW</b> Store Word	AX → zone-mémoire
<b>SUB</b> Subtract	op1 - op2 → op1
<b>TEST</b>	op1 AND op2
<b>WAIT</b>	le CPU entre dans une boucle d'attente
<b>XCHG</b> Exchange	op1 ↔ op2
<b>XLAT</b> Translate	conversion de code par table de corresp.
<b>XOR</b> Exclusive Or	op1 XOR op2
op1, op2 ...	opérandes de l'instruction (op1 = 1er, op2 = 2ème champ,)
opérandes absolues ...	comparaison de valeurs en représentation binaire-entier sans complémentation (i.e. valeurs sans signe)
zone-mémoire ...	opérande pour une opération itérative sur chaînes de bytes ou de mots, adressée par DS:SI (source) et ES:DI (dest.)
AND ...	'et' logique, bit par bit
NOT ...	inversion logique, bit par bit
OR ...	'ou' logique inclusif, bit par bit
XOR ...	'ou' logique exclusif, bit par bit



## Littérature et références

La bible PC, Programmation système

Micro Application.

ref 1144, ISBN 2-7429-0144-2

CUI: D.4.m TIS

The undocumented PC

Franck Van GILLUWE.

Addison-Wesley

ISBN 0-201-62277-7

The undocumented DOS

Andrew Shulman

Addison-Wesley

CUI: D.4m UND; Phys: 197 NOR

La Programmation en Assembleur sur PC et PS

Perter Norton; John Socha

PSI

ISBN 2.86595.584.X

Introduction à l'infographie

Foley, Van Dam, Feiner, Huges, Philips

Addison Wesley

ISBN 2 87308 058

Programming the 80286, 80386, 80486 and Pentium-Base Personal Computer

Barry B. Brey

Prentice Hall

ISBN 0-02-314263-4

Ralph Brown's Interrupt List

<http://www.ctyme.com/intr/INT.HTM>

Intel Secrets

<http://www.x86.org/>

The Art of Assembly Language Programming

<http://sil0.csci.unt.edu/home/brackeen/vga/> ??

C restreint à la consultation

CUI, Phys. ... bibliothèques à Genève

P disponible en prêt

CUI  
Phys.

C P

C P

C



# Mots-clé

Références imprimées en *caractères italiques* : commande MASM. Références imprimées en **caractères gras** : référence principale.  
Pour accéder à la page d'une référence, sélectionner la page par un "click" de la souris.

<i>.LALL</i> .....	6.17	- de transfert.....	1.6, 1.9, <b>3.13</b> , 3.25	<i>CBW</i> .....	4.9, 4.12
<i>.LFCOND</i> .....	6.17	- effective.....	2.21, 2.22, 2.24, <b>3.9</b> , 4.6, 7.4	CCITT.....	8.15
<i>.LIST</i> .....	6.16	- espace d'adresses.....	2.3, 2.4, 3.11	CF (registre d'état) ....	<b>2.18</b> , 2.19, 4.5
<i>.RADIX</i> .....	3.7	- logique.....	2.5	champ	
<i>.SALL</i> .....	6.17	- physique.....	2.5	- d'étiquette .....	<b>3.5</b> , 3.15, 3.26, 5.2, 5.3, 6.7
<i>.SFCOND</i> .....	6.17	- registre e/s .....	8.11, 8.12, 8.13	- d'opérande .....	<b>3.6</b>
<i>.TFCOND</i> .....	6.17	- relogeable .....	3.10	- d'opération.....	2.11, <b>3.5</b>
<i>.XALL</i> .....	6.17	- virtuelle .....	7.6, 7.7	<i>CLC</i> .....	4.12
<i>.XLIST</i> .....	6.16	AF (registre d'état)....	2.18	<i>CLD</i> .....	4.12
AAM.....	4.12	AND .....	2.20, <b>4.7</b> , 5.6	<i>CLI</i> .....	2.20, 4.12
AAS .....	4.12	ASCII.....	3.7, 4.12	<i>CMC</i> .....	4.12
ABS .....	6.14	assemblage.....	<b>3.2</b> , 3.3, 3.7, 3.11, 3.13, 3.23, 5.2, 5.3, 6.3	<i>CMP</i> .....	1.6, 2.20, 4.5, <b>4.7</b>
accumulateur .....	2.15	- conditionnel.....	3.3, <b>6.2</b> , 6.6, 6.17	<i>CMPSB</i> .....	4.10
ACIA.....	8.14, 8.15, 8.17, 8.20– 8.24	- en deux passes.....	3.3, 5.3	<i>CMPSW</i> .....	4.10
ADC.....	4.7	- macro.....	3.3, 6.6, <b>6.7</b>	Code Segment, voir CS	
ADD.....	2.10, 2.12, 2.14, 4.7	- répétitif .....	3.3, <b>6.4</b> , 6.6	code-objet .....	2.8, 3.2, 3.5, <b>3.11</b> , 3.12, 3.14, 3.15, 3.18, 3.27
adressage .....	<b>2.21</b>	ASSUME.....	1.6, <b>3.21</b> , 3.22, 3.23, 3.24, 3.25, 5.4	code-source.....	<b>3.2</b> , 3.2, 3.24, 6.13
- absolu .....	2.11, 2.21, <b>2.22</b> , 2.24, 2.25, 3.9	AT.....	3.17	COMMON .....	3.17
- direct .....	2.21, <b>2.23</b> , 3.10	attente active .....	8.8	complément	
- immédiat.....	2.21, <b>2.23</b> , 3.10, 4.4	attributs d'une variable, voir variable		- à deux .....	2.12, 2.22, 4.7, 5.10
- indexé .....	2.15, 2.21, 2.24, 2.25, 3.8, 3.9	BCD .....	4.12	- à un.....	4.7
- indirect .....	2.21, <b>2.23</b>	branchement.....	2.13, 2.22, <b>4.4</b> , 4.5, 5.4, 8.25	compteur	
- mode d'adressage .....	2.11, 2.25, 3.8, <b>3.8</b> , 3.9, 7.4, 7.7	BTC .....	7.7	- de position .....	3.11, 3.12, <b>3.12</b> , 3.13, 3.15, 5.7, 5.9
- relatif.....	2.21, <b>2.22</b> , 2.24, 3.9, 4.4	bus .....	<b>2.2</b> , <b>2.6</b> , 3.18, 4.12, 7.2, 8.2, 8.4, 8.5, 8.6, 8.7, 8.10, 8.11	- itérations et décalage ...	2.15
adresse .....	2.3, 2.6, 2.9, 2.11, 2.23, 3.8, 3.12, 3.13, 3.18, 3.23, 5.4, 5.8, 5.10, 8.4, 8.11, 8.13	BYTE .....	3.17, <b>3.18</b> , 3.26, <b>5.5</b> , 5.9, 5.10, 6.14	- ordinal, voir IP	
- de base (segment) .....	2.5, 3.17, 3.18, 3.20, 3.23, 3.24, 7.4	CALL .....	2.14, 2.17, 2.20, 2.22, 3.9, 4.3, 4.4, <b>4.4</b> , 6.15	constante .....	5.5
		Carry-flag, see CF		- alpha-numérique .....	3.7
				- binaire .....	3.7
				- décimal.....	3.7
				- hexadécimal .....	3.7
				- numérique .....	3.3, <b>3.7</b>
				- octal .....	3.7
				control register, voir registre de contrôle	



CPU ..... 1.8, 1.9, **2.2**, 2.6, 2.7, 2.10, 2.13, 2.18, 3.5, 7.2, 7.3, 7.4, 7.6, 8.2, 8.3, 8.4, 8.5, 8.6, 8.8

CS (registre) ..... **2.17**, 3.20, 3.21, 3.22, 4.3, 4.6, 5.4

CWD ..... 4.9, 4.12

DAA ..... 4.12

DAS ..... 4.12

Data Segment, voir DS

DB ..... 1.6, **3.26**, **3.27**, 3.28, 5.5, 5.9, 6.3, 6.5, 6.9

DCE ..... 8.14

DD ..... 3.26, 5.5

DEC ..... 1.6, 4.7

descripteur de segment . 7.4, 7.5, 7.6

DF (registre d'état) .... **2.18**, 4.10

Direct Memory Access, voir DMA

DIV ..... 2.15, **4.7**

DMA ..... 1.5, 8.7

DQ ..... 3.26, 5.5

DS (registre) ..... 2.15, **2.17**

DT ..... **3.26**, 3.28, 5.5

DTE ..... 8.14

DUP ..... 1.6, 3.25, 3.26, **3.27**, 3.28, 5.8, 6.3

DW ..... 3.22, 3.25, **3.26**, 3.28, 5.5, 5.8

DWORD ..... 3.26, **5.5**, 5.9, 6.14

éditeur de liens ..... **3.2**, 3.11, 3.14, 3.17, 3.18, 3.20

ELSE ..... 6.2

END ..... 1.6, 3.3, **3.13**, 3.14, 3.19

ENDIF ..... 6.2, 6.3

ENDM ..... 6.4, 6.5, 6.7, 6.9, 6.11

ENDP ..... **6.14**, 6.15

ENDS ..... 1.6, **3.15**, 3.22, 3.24, 3.25, 6.9, 6.11

entrée / sortie ..... 2.2, **8.2**

- écriture ..... 8.5

- initialisation ..... 8.10

- lecture ..... 8.6

- memory-mapped ..... 8.13

- non-memory-mapped .. 8.13

- porte ..... 2.12, 2.15, 8.13

EQ ..... 5.6

EQU ..... 1.6, **5.5**, 5.9, 6.3

ES (registre) ..... 2.15, **2.17**, 3.9, 3.10, 3.19, 3.21, 3.22

ESC ..... 4.12

étiquette ..... 2.14, 3.4, 3.17, 5.4, **5.4**, 5.5, 5.8, 5.9, 6.14

EVEN ..... 3.12

expression ..... **5.6**, 5.9, 5.10, 6.2

extended memory ..... 7.4

extension du bit de signe 2.16, **4.9**, 4.12

EXTERN ..... 6.14

Extra Segment, voir ES

FADD ..... 2.14

FAR ..... 2.14, 3.25, 4.3, 4.4, 5.4, **5.4**, 5.8, 5.9, 6.14, 6.15

Flag Register, voir registre d'état

fonction ..... 3.3, 3.12, 3.27, **5.7**, 5.7, 5.8, 5.9, 5.10

GE ..... 5.6

granularity bit ..... 7.4, 7.5

GT ..... 5.6

HIGH ..... 5.6

HLT ..... 4.12

IDIV ..... 4.7

IF ..... **6.2**, 6.3

IF (registre d'état) .... **2.18**, 2.20, 4.3

IFI ..... 6.2

IF2 ..... **6.2**

IFB ..... 6.2

IFDEF ..... 6.2

IFE ..... **6.2**, 6.3

IFIDN ..... 6.2

IFNB ..... 6.2

IFNDEF ..... 6.2

IFNIDN ..... 6.2

impression ..... **6.16**, 6.17

IMUL ..... 4.7

IN ..... 8.13

INC ..... 4.7

INCLUDE ..... 3.25, **6.13**

instruction pointer, voir IP

INT ..... 1.6, 3.25, 4.3, 6.9

Intel ..... 1.5, 2.2

interruption ..... 8.9

- CPU ..... 1.6, 2.4, 2.17, 2.18, 2.20, 4.12, 8.8, 8.9, 8.17, 8.18

- DOS ..... 1.6

INVD ..... 7.7

IP (compteur-ordinal) .. **2.17**, 2.20, 2.22, 3.13, 3.20, 4.3, 7.3

IRET ..... 4.3

IRP ..... **6.4**

IRPC ..... **6.4**, 6.5

JA ..... 4.5

JAE ..... 4.5

JB ..... 4.5

JBE ..... 4.5

JBNE ..... 4.5

JCXZ ..... 4.4

JE ..... 2.20, 2.22, 3.9, 4.5

JG ..... 4.5

JGE ..... 4.5

JL ..... 4.5

JLE ..... 4.5



*JMP* ..... 2.9, 2.17, 2.20, 2.22,  
3.9, 4.4, **4.4**, 5.10

*JNA* ..... 4.5

*JNAE* ..... 4.5

*JNB* ..... 4.5

*JNE* ..... 4.5

*JNG* ..... 4.5

*JNGE* ..... 4.5

*JNL* ..... 4.5

*JNLE* ..... 4.5

*JNO* ..... 4.5

*JNP* ..... 4.5

*JNS* ..... 4.5

*JNZ* ..... 1.6, 4.5

*JO* ..... 4.5

*JP* ..... 4.5

*JPE* ..... 4.5

*JPO* ..... 4.5

*JS* ..... 4.5

*JZ* ..... 4.5

*LABEL* ..... 3.25

*LAHF* ..... 4.6

*LDS* ..... 7.7

*LE* ..... 5.6

*LEA* ..... 4.6

*LENGTH* ..... **5.8**, 5.10

*LGDT* ..... 7.7

*LLDT* ..... 7.7

*LOCAL* ..... **6.8**, 6.9

*LOCK* ..... 4.12

*LODB* ..... 4.10

*LODW* ..... 4.10

*LOOP* ..... 4.4

*LOOPE* ..... 4.4

*LOOPNE* ..... 4.4

*LOOPNZ* ..... 4.4

*LOOPZ* ..... 4.4

*LOW* ..... 5.6

*LSB* ..... 2.18

*LSS* ..... 7.7

*LT* ..... 5.6

*MACRO* ..... **6.7**, 6.9, 6.11

macro-instruction ..... **6.7**, 6.17

*MASK* ..... **5.7**, **5.8**, **5.9**, 5.10

mémoire ..... 2.2, 2.3, **2.5**, 8.2

- logique ..... 2.5

- physique ..... 2.4, 2.5

mémoire virtuelle ..... 2.8

*MEMORY* ..... 3.17

MMU ..... 7.2

mnémonique ..... 3.3

*MOD* ..... **5.6**

mode

- 16-bit mode ..... 7.7

- 32-bit mode ..... 7.7

modem ..... 8.14, 8.15, 8.16

Motorola ..... 1.5

*MOV* ..... 1.6, 2.10, 2.14, 2.16,  
2.20, 3.19, 3.25, **4.6**,  
4.9, 5.9, 6.9

*MOVSB* ..... 4.10

*MOVSW* ..... 4.10

*MOVSX* ..... 4.9

*MOVZX* ..... 4.9

*MSB* ..... 2.18

*MUL* ..... 2.15, **4.7**

*NE* ..... 5.6

*NEAR* ..... 2.14, 4.4, **5.4**, 5.6, 5.8,  
5.9, 6.14, 6.15

*NEG* ..... 4.7

*NOT* ..... **4.7**, 5.6

OF (registre d'état) ..... **2.18**, 4.5

*OFFSET* ..... **5.8**, 5.10

opérande ..... 2.23

opération ..... 2.9, **2.12**, 2.18, 2.19,  
2.20, 3.4, 3.5, 3.6, 5.2

opération sur chaînes .. 2.15, **4.10**

*OR* ..... **4.7**, 5.6

*ORG* ..... 3.12

*OUT* ..... 2.20, 8.13

*PAGE* ..... 3.17, **3.18**

pagination ..... 7.6

*PARA* ..... 3.17, **3.18**

paramètre formel ..... 6.7

parité ..... 2.18, 4.5, 8.17, 8.18

PF (registre d'état) .... 2.18, 4.5

pile ..... **2.7**, 2.17, 2.21, 3.17,  
3.20, 3.25, **4.2**

pointeur-pile, voir SP

*POP* ..... 2.21, 4.3, 6.9

*POPF* ..... 4.6

précédence ..... **5.6**

préfixe

- répétition ..... 4.11

- segment ..... **3.10**

- verrouillage bus ..... 4.12

*PROC* ..... **6.14**, 6.15

procédure ..... 2.17, 6.14, 6.15

Processor Status Word . 2.17

protected mode addressing 7.4

pseudo-opération ..... 3.5, 3.26, 5.5, 6.4, 6.8,  
6.16

*PTR* ..... **5.9**, 5.10

*PUBLIC* ..... **3.17**, 5.3, **6.14**

*PUSH* ..... 2.20, 2.21, 4.3, **4.6**, 6.9

*PUSHF* ..... 4.6

*QWORD* ..... 3.26, **5.5**, 5.9

RAM ..... 2.4

*RBYTE* ..... 8.13

*RCL* ..... 4.7

*RCR* ..... 4.7



real memory. . . . . 7.4  
 real mode addressing. . . 7.4  
*RECORD*. . . . . 5.5  
 record. . . . . 5.7  
 références en avant . . . . 5.3  
 registre  
 - d'entrée / sortie . . . . . 8.11  
 - d'état . . . . . 2.17, **2.18**, 4.3  
 - d'indexage. . . . . 2.11, **2.22**, 3.9, 3.10  
 - d'opérandes. . . . . 2.15  
 - de base . . . . . 2.5, **2.17**, 2.20, 3.10,  
 3.19, 3.24, 7.4, 7.6,  
 7.7  
 - de byte. . . . . 2.16  
 - de contrôle . . . . . 8.9  
 - de réception. . . . . 8.4  
 - de segment, voir registre de base  
*REP* . . . . . 4.11  
*REPE* . . . . . 4.11  
*REPNE*. . . . . 4.11  
*REPNZ*. . . . . 4.11  
*REPT*. . . . . **6.4**, 6.5  
*REPZ*. . . . . 4.11  
 Reset . . . . . 7.4, 8.10  
*RET* . . . . . **4.3**, 6.14, 6.15  
 RISC . . . . . 1.5  
*ROL* . . . . . 4.7  
 ROM . . . . . 2.4  
*ROR*. . . . . 4.7  
 RS 232C . . . . . 8.15  
*RWORD*. . . . . 8.13  
  
*SAHF*. . . . . 4.6  
*SAL* . . . . . 4.7  
*SAR* . . . . . 4.7  
*SBB* . . . . . 4.7  
*SCAB*. . . . . 4.10  
*SCAW*. . . . . 4.10  
*SEG* . . . . . **5.8**, 5.10

*SEGMENT* . . . . . 1.6, **3.14**, 3.15, **3.17**,  
 3.19, 3.22, 3.24, 3.25,  
 6.9, 6.11  
 segment  
 - logique. . . . . 2.5, **3.11**, 3.12, 3.15,  
 3.18, 3.19, 3.20, 3.21,  
 3.24  
 - physique. . . . . 2.4, **2.5**, 2.8, 3.19,  
 3.21, 3.24  
 Segment Prefix Code, voir préfixe  
 Segment Register, voir registre de base  
 séparateur de chaîne. . . 3.7, 6.8  
 sexe des bytes. . . . . 2.3  
 SF (registre d'état) . . . . **2.18**, 4.5  
*SHL*. . . . . 2.20, **4.7**, 5.6  
*SHORT*. . . . . 5.10  
*SHR* . . . . . 4.7, 5.6  
*SIZE* . . . . . **5.8**, 5.10  
 SP (registre) . . . . . 2.7, 2.15, 3.20  
 SS (registre) . . . . . 2.15, **2.17**, 3.19, 3.20,  
 3.21, 3.22, 4.3  
*STACK* . . . . . 3.17  
 Stack Segment, voir SS  
 status. . . . . 8.8  
*STC*. . . . . 4.12  
*STD*. . . . . 4.12  
*STI* . . . . . 4.12  
*STOB* . . . . . 4.10  
*STOW*. . . . . 4.10  
*STRUCT*. . . . . 5.5  
*SUB* . . . . . 2.20, **4.7**  
*SUBTTL* . . . . . 6.16  
 symbole . . . . . **3.5**, 3.14, 3.24, 5.2, 5.6,  
 5.8, 5.9, 6.7, 6.8, 6.14,  
 6.15  
 table de code-objet. . . . 3.11  
*TBYTE* . . . . . 3.26, **5.5**, 5.9  
*TEST*. . . . . 2.20, **4.7**

TF (registre d'état) . . . . **2.18**, 4.3  
*THIS* . . . . . **5.9**, 5.10  
*TITLE* . . . . . 6.16  
 transfert sériel . . . . . 8.16  
*TYPE*. . . . . **5.8**, 5.10  
  
 UART . . . . . 8.14  
 V.24 . . . . . 8.14, **8.15**  
 variable . . . . . 5.2  
 - attribut . . . . . **5.2**, 5.5, 5.7, 5.8, 5.9  
 - globale . . . . . 5.3, 6.14  
 - locale . . . . . 5.3, 6.8  
 - relogeable. . . . . 3.10, 5.4, 5.5  
 - table des variables . . . . 5.3  
 vecteur d'interruption . . 2.4  
  
*WAIT*. . . . . 4.12  
*WBYTE* . . . . . 8.13  
 white space . . . . . **3.4**, 3.6  
*WIDTH* . . . . . **5.7**, 5.10  
*WORD*. . . . . 3.17, **3.18**, 3.26, **5.5**,  
 5.9, 5.10, 6.14  
  
*XCHG* . . . . . 4.6  
*XLAT*. . . . . 2.15  
*XOR* . . . . . **4.7**, 5.6  
  
 ZF (registre d'état) . . . . **2.18**, 2.20, 4.5, 4.11  
 ZILOG . . . . . 1.5

