

Module M25:
**Architecture des Ordinateurs et le Langage
Assembleur 8086**

FILIERE SMI – SEMESTRE 4

Prof. ABDELTIF EL BYED

2020-2021

Objectifs de ce module

- **Apprendre** l'architecture générale d'un ordinateur, l'organisation interne d'un 8086, le CPU qui servait de base au premier PC d'IBM.
- **Comprendre** l'utilité des différents registres du CPU.
- **Découvrir** les bases du langage Assembleur du 8086 au travers de l'émulateur EMU 8086.

Table des matières

Objectifs de ce module	2
1 Architecture des ordinateurs	10
1.1 Ordinateur et logiciel.....	10
1.2 Représentation de l'information	10
1.3 Constituants élémentaires	10
1.4 Performance.....	11
1.5 Évolutions architecturales	11
1.6 Barrières à l'évolution	12
1.6.1 Barrière de la chaleur	12
1.6.2 Barrière de la complexité	12
1.7 Structure d'un ordinateur.....	13
1.7.1 Unité Centrale de Traitement (CPU).....	14
1.7.2 Chemin de données.....	14
1.8 Exécution d'une instruction.....	15
1.9 Langages et exécution	15
2 Architecture des Ordinateurs: généralités et rappels	16
2.1 Usages	16
2.2 Composants d'un ordinateur	16
2.3 Fonctions de la machine.....	16
2.4 Le processeur.....	16
2.4.1 Famille de processeurs	17
2.4.2 Famille x86.....	17
2.5 Représentation des informations.....	17
2.5.1 Les bases.....	18
2.5.2 Parler binaire, octal et hexadécimal.....	18
2.5.3 Changement de base.....	18
2.6 Notation des entiers	19
2.6.1 Positif et négatif	19
2.6.2 Complément à deux	19
3 Présentation du 8086	20
3.1 Vue externe du 8086	20
3.1.1 Le packaging	20
3.1.2 Les différentes broches	20
3.1.3 Les différentes broches	20

3.2	Vue interne du 8086.....	21
3.2.1	Le schéma bloc	21
3.2.2	Les registres et le bus interne de données.....	22
3.3	Types de registres.....	22
3.3.1	Les registres généraux.....	23
3.3.2	Les registres d'index	23
3.3.3	Les registres de segment	24
3.3.4	Les registres pointeurs	24
3.3.5	Le registre IP	25
3.3.6	Le registre indicateur.....	25
4	Présentation de l'environnement Emu8086	27
4.1	L'environnement de programmation.....	27
4.1.1	Le démarrage.....	27
4.1.2	Le premier programme assembleur.....	27
4.1.3	Description du premier code.....	29
4.2	L'environnement d'exécution	29
4.2.1	L'exécution du premier programme	29
4.2.2	L'adressage du code objet en mémoire	30
4.3	Assemblage et la notion d'opcode	32
4.3.1	La notion de mnémonique et d'opcode.....	32
4.3.2	Instruction à à dressage immédiat et à registre.....	32
4.3.3	Extrait du databook de l'Intel 8086	32
4.3.4	Description du code.....	33
4.4	Exécution	34
4.4.1	Exécution de la première instruction (MOV).....	34
4.4.2	Exécution de la deuxième instruction (MOV)	35
4.4.3	Exécution de la troisième instruction (ADD)	35
4.4.4	Exécution de la quatrième instruction (SUB)	36
4.4.5	Exécution de la cinquième instruction (RET).....	37
4.4.6	Exécution de la sixième instruction (INT)	37
5	Manipulation de base.....	39
5.1	Opérations logiques	39
5.1.1	AND, OR, NOT et XOR.....	39
5.1.2	Décalage à gauche et décalage à droite.....	39
5.1.3	Rotation à gauche et rotation à droite.....	40

5.2	Opérations arithmétiques	41
	5.2.1 Addition et soustraction sans retenue	41
	5.2.2 Incrémentation et décrémentation.....	41
	5.2.3 L'addition sur 32 bits	42
	5.2.4 Multiplication et division.....	43
5.3	L'adressage	43
	5.3.1 Adressage immédiat et adressage par registre.....	43
	5.3.2 L'adressage des instructions dans CS.....	44
	5.3.3 L'adressage des données dans DS.....	44
	5.3.4 L'adressage des données dans SS et ES.....	45
	5.3.5 Adressage en mode mémoire avec ADD, SUB et ADC.....	45
	5.3.6 Adressage en mode mémoire avec MUL, IMUL, DIV et IDIV.....	46
	5.3.7 Spécification de la taille des données	46
5.4	TD2 : Manipulation de base	47
	5.4.1 Exercice1: Opérations logiques : AND, OR, NOT et XOR	47
	5.4.2 Exercice2: Opérations logiques : Décalage et Rotation.....	47
	5.4.3 Exercice3: Addition sur 32 bits (Utilisation du bit CF)	47
	5.4.4 Exercice4: Multiplication et division.....	48
	5.4.5 Exercice5: Adressage direct & Adressage par registre.....	48
6	Les sauts	49
	6.1 La notion d'étiquette.....	49
	6.1.1 Le principe	49
	6.1.2 La déclaration d'un label	49
	6.2 Les sauts inconditionnels.....	49
	6.2.1 Le mnémonique JMP	49
	6.3 La comparaison	50
	6.3.1 Le mnémonique CMP	50
	6.4 Les sauts conditionnels.....	51
	6.4.1 Le principe	51
	6.4.2 Flags (Rappel)	51
	6.4.3 Les mnémoniques de sauts conditionnels:	51
	6.4.4 Les mnémoniques de sauts conditionnels	53
	6.4.5 La limitation dans l'amplitude des sauts	53
	6.4.6 Exercice d'application.....	53
	6.5 Les boucles	54

6.5.1	Le registre CX comme compteur	54
6.5.2	Syntaxe d'une boucle	54
6.6	TD3 : Les Sauts et les Boucles.....	56
6.6.1	Exercice1: Les sauts inconditionnels	56
6.6.2	Exercice2: Les sauts conditionnels	56
6.6.3	Exercice3: Les boucles	57
7	La manipulation des variables.....	58
7.1	La notion de variable	58
7.1.1	La règle de nommage	58
7.2	Les variables numériques	58
7.2.1	La création de la variable par DB et DW.....	58
7.2.2	L'utilisation de la variable.....	58
7.2.3	L'adresse d'une variable	59
7.3	Les tableaux.....	59
7.3.1	La création d'un tableau et l'accès aux données	59
7.3.2	La création d'un tableau par répétition de valeurs.....	60
7.4	Les chaînes de caractères.....	60
7.4.1	La création	60
7.5	Les constantes numériques	61
7.5.1	La création et l'utilisation	61
7.6	TD4 : Les Variables.....	62
7.6.1	Exercice 1.....	62
7.6.2	Exercice 2.....	62
7.6.3	Exercice 3.....	62
7.6.4	Exercice 4.....	62
7.6.5	Exercice 5.....	62
8	Les procédures	63
8.1	Introduction.....	63
8.2	Les directives PROC et ENDP.....	63
8.2.1	Spécifier à l'assembleur la présence d'une procédure	63
8.2.2	Le squelette d'une procédure	63
8.3	Les instructions CALL et RET.....	63
8.4	La sauvegarde temporaire des données.....	64
8.4.1	Le contexte d'exécution	64
8.4.2	Les types d'empilement	64

8.4.3	Les types de dépilement.....	65
8.5	Passage de paramètres inter-procédures	66
8.5.1	Passage de paramètres par registre.....	66
8.5.2	Passage de paramètres dans la pile	66
8.6	TD : Les procédures	69
8.6.1	Exercice 1 : Appel de sous-programme sans passage de paramètres.....	69
8.6.2	Exercice 2: Appel de sous-programme sans passage de paramètres	69
8.6.3	Exercice 3 : Appel de sous-programme avec passage de paramètres par registre	
	69	
8.6.4	Exercice 4 : Appel de sous-programme avec passage de paramètres dans la pile	
	69	
9	Les interruptions	70
9.1	Le principe.	70
9.1.1	Le mécanisme des interruptions matérielles	70
9.1.2	Le contrôleur d'interruption.....	70
9.1.3	Le cas du clavier.....	70
9.1.4	Le traitement de l'IRQ	72
9.1.5	Synthèse	74
9.2	Les types des interruptions	74
9.2.1	Les interruptions matérielles.....	74
9.2.2	Les interruptions logicielles.....	74
9.2.3	Taxinomie des interruptions	75
9.2.4	Les interruptions matérielles externes.....	75
9.2.5	Le masquage des interruptions	76
9.3	Gestion de l'écran.....	76
9.3.1	Le principe	76
9.3.2	L'affichage d'un caractère	76
9.3.3	L'affichage d'une chaîne de caractères	76
9.4	Gestion du clavier.....	77
9.4.1	Le démarrage.....	77
9.4.2	La lecture du clavier	77
9.4.3	Le test du clavier.....	77
9.5	Le déroutage des interruptions.....	78
9.5.1	L'attente active.....	78
9.5.2	Le déroutement des interruptions	79
9.6	TD : Les interruptions	80

9.6.1 Exercice 1 :	80
9.6.2 Exercice 2 :	80
9.6.3 Exercice 3 :	80
10 Références.....	82
10.1 Publications	82
10.2 Sites web	82
10.3 Ma Page web.....	83

Liste des Figures

Figure 1. Structure d'un ordinateur.....	14
Figure 2. Chemin de données.....	15
Figure 3. Processeur 8086	20
Figure 4. 8086, représentation matérielle.....	20
Figure 5. 8086, représentation fonctionnelle.....	21
Figure 6. le schéma bloc de 8086	22
Figure 7. Registres généraux	23
Figure 8. Segmentation de la mémoire	24
Figure 9. Registre indicateur.....	26
Figure 10. Écran de démarrage de l'EMU8086.....	27
Figure 11. EMU8086- choix du type de fichier	28
Figure 12. Code d'un fichier .com vide	28
Figure 13. EMU8086-Les fenêtres de l'exécution.....	30
Figure 14. Adressage du code en mémoire.....	31
Figure 15. Codage du premier exemple en mémoire.....	31
Figure 16. Extrait du databook de l'Intel 8086	32
Figure 17. Correspondance opcode – mnémonique	34
Figure 18. EMU: Exécution de la première instruction	35
Figure 19. EMU: Exécution de la deuxième instruction	35
Figure 20. EMU: Exécution de la troisième instruction	36
Figure 21. EMU: Exécution de la quatrième instruction	36
Figure 22. EMU: Exécution de la cinquième instruction	37
Figure 23. EMU: Exécution de l'instruction INT 020h.....	38
Figure 24. Addition sur 32 bits.....	42

1 Architecture des ordinateurs

1.1 Ordinateur et logiciel

Les technologies numériques sont maintenant omniprésentes. Elles sont le moteur et l'objet de ce qu'on appelle la « révolution numérique ».

Elles sont basées sur l'interaction entre :

- Des programmes, aussi appelés logiciels, décrivant des processus de traitement de l'information : biens immatériels
- Des ordinateurs, capables d'exécuter ces programmes : biens matériels

1.2 Représentation de l'information

L'information est représentée au sein des composants de l'ordinateur sous forme de différents états de la matière :

- « Trou » ou « pas trou » sur la surface d'un cédérom ou DVD
- Orientation nord ou sud d'un matériau magnétique
- Lumière ou absence de lumière émise par un laser
- Courant électrique ou non
- Ce sont souvent des représentations à deux états, c'est-à-dire « binaires »

1.3 Constituants élémentaires

Presque tous les ordinateurs sont construits à base de circuits électroniques, ces circuits électroniques sont réalisés au moyen de transistors.

Les transistors sont des composants élémentaires, dont le courant de sortie dépend de deux valeurs d'entrée. Un transistor a donc trois « pattes » appelées : base, émetteur et collecteur. (Voir la figure). Analogique à un « robinet à électricité » : plus il arrive de courant sur la base, plus le courant circule de l'émetteur vers le collecteur.



Dans les ordinateurs, on utilise les transistors en mode saturé, c'est-à-dire « tout ou rien ». Ce mode de fonctionnement est analogue à celui d'un interrupteur, pour une éventuelle représentation des valeurs binaires « 0 » et « 1 ».

- Robinet fermé ou ouvert en grand
- Soit le courant passe, soit il ne passe pas du tout.

En combinant plusieurs transistors, on peut effectuer des calculs complexes sur la base de montages en série ou en parallèle. Le regroupement des transistors se fait au sein de « circuits intégrés ».

1.4 Performance

Les calculs des ordinateurs sont cadencés par une horloge. Plus la fréquence de l'horloge est élevée, et plus l'ordinateur pourra effectuer d'opérations par seconde (s'il n'est pas ralenti par autre chose...).

On mesure la fréquence d'une horloge en Hertz (Hz), ce qui représente le nombre de battements par seconde :

- 1 kHz (kilo Hertz) = 10^3 Hz
- 1 MHz (méga Hertz) = 10^6 Hz
- 1 GHz (giga Hertz) = 10^9 Hz
- 1 THz (téra Hertz) = 10^{12} Hz

En fait, ce qui importe aux usagers, c'est le nombre d'opérations (plus généralement, « d'instructions ») qu'un ordinateur est capable d'effectuer par seconde. On la mesure en MIPS, pour « Millions d'Instructions Par Seconde ».

On pense souvent que la puissance d'un ordinateur dépend de sa fréquence de fonctionnement, c'est loin d'être toujours vrai !

1.5 Évolutions architecturales

- 1946 : Ordinateur ENIAC
 - Architecture à base de lampes et tubes à vide
 - 30 tonnes, 170 m² au sol, 5000 additions par seconde
 - 0,005 MIPS, dirons-nous...
- 1947 : Invention du transistor
- 1958 : Invention du circuit intégré sur silicium
 - Multiples transistors agencés sur le même substrat
- 1971 : Processeur Intel 4004
 - 2300 transistors dans un unique circuit intégré
 - Fréquence de 740 kHz, 0,092 MIPS
- ...40 ans d'une histoire très riche..., puis :
- 2011 : Processeur Intel Core i7 2600K
 - Plus de 1,4 milliards de transistors
 - Fréquence de 3,4 GHz
 - 4 cœurs, 8 threads

- 128300 MIPS

Pour résumer l'évolution architecturale entre le 4004 et le Core i7 2600K, La fréquence a été multipliée par 4600, et la puissance en MIPS a été multipliée par 1,4 million.

En outre, la puissance d'un ordinateur ne dépend clairement pas que de sa fréquence !, d'où l'intérêt d'étudier l'architecture des ordinateurs pour comprendre :

- Où les gains se sont opérés
- Ce qu'on peut attendre dans le futur proche

1.6 Barrières à l'évolution

L'évolution architecturale connaît aujourd'hui une sorte de stagnation suite à des limites atteintes.

1.6.1 Barrière de la chaleur

Plus on a de transistors par unité de surface, plus on a d'énergie à évacuer. La dissipation thermique évolue de façon proportionnelle.

La tension de fonctionnement des circuits a été abaissée de 5V pour les premières générations à 0,9V maintenant. Il n'est plus vraiment possible de diminuer cette tension avec les technologies actuelles, car le bruit thermique causerait trop d'erreurs.

Par conséquent, la fréquence ne peut raisonnablement augmenter au-delà des 5 GHz. Actuellement la tendance d'évolution est plutôt focalisée pour la réduction :

- « Green computing »
- On s'intéresse maintenant à maximiser le nombre d'opérations par Watt

Mais on veut toujours plus de puissance de calcul !...

1.6.2 Barrière de la complexité

À surface constante, le nombre de transistors double tous les 2 ans, « Loi de Moore », du nom de Gordon Moore, cofondateur d'Intel, énoncée en 1965.

Avec la diminution continue de la taille de gravure des transistors et circuits sur les puces de silicium, on grave actuellement avec un pas de 20 nm. En revanche, les limites atomiques bientôt atteintes...

- Donc plus possible d'intégrer plus.

Mais on veut toujours plus de puissance de calcul !...

Que faire de tous ces transistors ?

On ne voit plus trop comment utiliser ces transistors pour améliorer individuellement les processeurs. On se retrouve avec des processeurs trop complexes consommant trop d'énergie sans aller beaucoup plus vite.

Actuellement la seule solution adoptée est de faire plus de processeurs sur la même puce :

- Processeurs bi-cœurs, quadri-cœurs, octo-cœurs,... déjà jusqu'à 128 cœurs !

Mais comment les programmer efficacement ?!

L'architecture des ordinateurs a été l'un des secteurs de l'informatique qui a fait le plus de progrès. Les ordinateurs d'aujourd'hui sont très complexes. Pour le Core i7 8 cœurs on a 1,4 milliards de transistors.

D'où la nécessité d'étudier leur fonctionnement à différents niveaux d'abstraction

- Du composant au module, du module au système,
- Multiples niveaux de hiérarchie.

1.7 Structure d'un ordinateur

Un ordinateur est une machine programmable de traitement de l'information. Pour accomplir sa fonction, il doit pouvoir :

- Acquérir de l'information de l'extérieur
- Stocker en son sein ces informations
- Combiner les informations à sa disposition entre elles
- Restituer ces informations à l'extérieur

L'ordinateur doit donc posséder :

- Une ou plusieurs unités de stockage : pour mémoriser le programme en cours d'exécution ainsi que les données qu'il manipule
- Une unité de traitement : permettant l'exécution des instructions du programme et des calculs sur les données qu'elles spécifient
- Différents dispositifs « périphériques » : servant à interagir avec l'extérieur : clavier, écran, souris, carte graphique, carte réseau, etc.

Les constituants de l'ordinateur sont reliés par un ou plusieurs bus, ensembles de fils parallèles servant à la transmission des adresses, des données et des signaux de contrôle (voir Figure 1. Structure d'un ordinateur) :

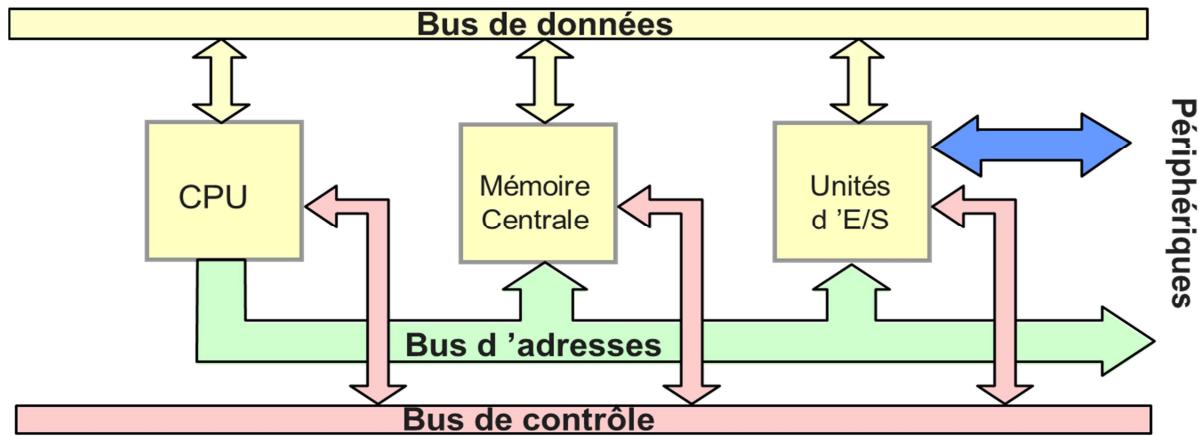


Figure 1. Structure d'un ordinateur

1.7.1 Unité Centrale de Traitement (CPU)

L'unité centrale de traitement (ou CPU, pour « Central Processing Unit »), aussi appelée « processeur », est le cœur de l'ordinateur. Elle exécute les programmes chargés en mémoire centrale en extrayant l'une après l'autre leurs instructions, en les analysants et en les exécutants.

L'unité centrale de traitement est composée de plusieurs sous-ensembles distincts :

- **L'unité de contrôle:** qui est responsable de la recherche des instructions à partir de la mémoire centrale et du décodage de leur type
- **L'unité arithmétique et logique (UAL):** qui effectue les opérations spécifiées par les instructions
- **Un ensemble de registres:** zones mémoires rapides servant au stockage temporaire des données en cours de traitement par l'unité centrale

1.7.2 Chemin de données

Le chemin de données représente la structure interne de l'unité centrale de traitement. Il comprend les registres, l'UAL, et un ensemble de bus internes dédiés.

L'UAL peut posséder ses propres registres destinés à mémoriser les données d'entrées afin de stabiliser leurs signaux pendant que l'UAL réalise des calculs.

Le chemin des données conditionne fortement la puissance des machines : Pipe-line, superscalarité, ...

- La Figure 2, illustre le chemin de données d'une machine de type «Von Neumann».

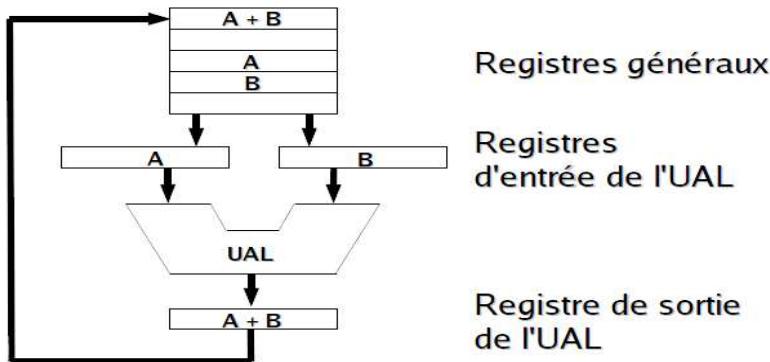


Figure 2. Chemin de données

1.8 Exécution d'une instruction

L'exécution d'une instruction par l'unité centrale de traitement s'effectue selon les étapes suivantes :

1. Charger la prochaine instruction à exécuter depuis la mémoire vers le registre d'instruction
2. Décoder (analyser) l'instruction venant d'être lue
3. Faire pointer le compteur ordinal vers l'instruction suivante
4. Localiser en mémoire les données nécessaires
5. Charger si nécessaire les données dans l'UAL
6. Exécuter l'instruction, puis recommencer

1.9 Langages et exécution

Les processeurs ne comprennent que le langage machine, ce langage est trop élémentaire et trop long à programmer. D'où la motivation de construire des programmes à un niveau d'abstraction plus élevé pour une meilleure expressivité et généricité et avec moins de risques d'erreurs.

Besoin de passer d'un langage humainement compréhensible à une exécution machine !

2 Architecture des Ordinateurs: généralités et rappels

2.1 Usages

L'assembleur est langage de bas niveau caractérisé par la rapidité de son exécution, par conséquent on utilise l'Assembleur (entre autre) pour :

- Ecrire un compilateur.
- Environnements embarqués, micro-controleurs.
- Systèmes temps-réels durs.

L'assembleur a plusieurs avantages, mais principalement on va l'étudier pour une meilleure compréhension à la fois des langages et des machines.

2.2 Composants d'un ordinateur

L'ordinateur possède plusieurs composants et la plupart d'entre eux sont placés sur la carte mère. Ces composants sont :

- Processeur (CPU)
- Mémoire
- Stockage (Disque dur, CD, carte mémoire...)
- Périphériques de sortie (carte vidéo, écran, carte son...)
- Périphériques d'entrée (clavier, souris, capteur ...)
- ...

2.3 Fonctions de la machine

Une machine (un ordinateur) a deux fonctions primordiales :

- Calculer : rôle du (micro)-processeur
- Stocker : rôle de la mémoire

Une machine possède d'une part, un langage spécifique par processeur, appelé jeu d'instructions, et d'autre part, une référence commune appelée le langage binaire.

2.4 Le processeur

Le processeur c'est le cœur de l'ordinateur, Il applique un programme sur des données, avec :

- Le programme est une suite d'instructions
- Les données sont situées en mémoire

Le principe général du processeur consiste :

- Lecture d'une instruction en mémoire

- Lecture des paramètres nécessaires en mémoire
- Exécution de l'instruction
- Stockage du résultat en mémoire

2.4.1 Famille de processeurs

On classe les processeurs à des familles en prenant en considération le :

- Taille des données qu'ils peuvent manipuler
 - 4 bits, 8, 16, 32, 64....
- Instructions qu'ils peuvent exécuter
 - x86 (dans les PCs, xbox...)
 - PowerPC (anciens Mac, PS3)
 - ARM (PDA,...)

Chaque famille comporte énormément de modèles différents

2.4.2 Famille x86

Il s'agit d'une famille de processeurs à compatibilité ascendante, c.-à-d., que le code des plus anciens est toujours compris et correctement exécuté par les plus récents.

- Le 8086 dispose d'un bus de 16 bits pour les données. Sa capacité d'adressage est de 1Mo et il est cadencé à 4,77 ou 8 Mhz
- Le 8088 est la version 8 bits du 8086. Il a les mêmes caractéristiques, mais un assembleur plus réduit.
- Le 80286 est une machine 16 bits, pouvant adresser 8 Mo de code et de données, cadence à 6, 8, 10, 12 ou 16 Mhz. Il introduit deux modes d'adressage (réel ou protégé).
- Le 80386 est un faux 32 bits, adressant 4 Go et cadencé à 16, 20 ou 25 Mhz.
- Le 80486 est un vrai 32 bits doté d'une mémoire cache intégrée et d'une unité de calcul en virgule flottante.
- Le 80586, appelé pentium pour des raisons de protection commerciale, dispose d'un bus de données de 64 bits et est muni d'un dispositif de prévision des branchements. Il est constitué de 2 processeurs en pipe-line parallèles lui permettant d'exécuter deux instructions en même temps. Son cadencement est envisagé (en 1994) jusqu'à 150 MHz.

Dans la suite nous nous intéressons qu'à la famille 8086 des processeurs

2.5 Représentation des informations

L'unité de base d'un ordinateur est le « bit » et il ne peut prendre que 2 valeurs: 0 ou 1. Les bits sont regroupés par 8 bits pour former des octets.

2.5.1 Les bases

La même information peut être représentée sur plusieurs bases différentes. Parmi ces bases on trouve :

- Le binaire (base 2) → 0,1
- L'Octale (base 8) → 0,1,2,3,4,5,6,7
- Le décimal (base 10) → 0,1,2,3,4,5,6,7,8,9
- L'hexadécimal (base 16) → 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Le code ASCII → codage des caractères (sur un octet)

2.5.2 Parler binaire, octal et hexadécimal

Pour préciser la base de présentation, on utilise la syntaxe suivante :

- Pour la base binaire : **01010b**, en ajoutant **0** au début et **B** à la fin
- Pour la base Octal : **012o**, en ajoutant **0** au début et **O** à la fin
- Pour la base Hexadécimal : **0AFh**, en ajoutant **0** au début et **H** à la fin
- Pour la base décimal: **12**, en ajoutant rien des tous.

Ces différentes bases sont utilisées (entre autre) pour :

- Masques de bits
- Permissions
- Adresses mémoire

2.5.3 Changement de base

Un nombre entier X dans une base b, s'écrit comme une suite de symboles

- $X = a_0 \dots a_n$ avec $a_i \in [0 \dots b-1]$

2.5.3.1 De la base b à la base 10

Pour couverture un entier de la base b à la base 10, on utilise la multiplication et additions. Autrement dit le développement polynomial.

$$x = \sum_{i=0}^n a_i * b^i$$

Par exemple, si $X=01010b$ dans la base binaire, alors la valeur de X dans la base décimale est :

$$X = 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 8 + 2 = 10$$

2.5.3.2 De la base 10 à la base 2

Pour couverture un entier de la base 10 à la base b, on utilise la divisions successives avec la prise en compte du reste.

Cette méthode consiste à diviser l'entier sur la base b et tant que le quotient de la division est non nul on le devise par b jusqu'à avoir un quotient nul. A la fin, on récupère les restes de l'ensemble des opérations de dévions dans le sens inverse (de bas vers le haut).

Par exemple, on cherche à calculer la valeur du 44 dans la base binaire :

- 44 div 2=22 reste=0=a₀ bit de poids faible
-
- 1 div 2=0 reste=1=a₅ bit de poids fort

$$\Rightarrow (44)_{10} = (0010\ 1100)_2 = (2C)_{16} = (054)_8$$

2.6 Notation des entiers

2.6.1 Positif et négatif

Suivant le signe de l'entier nous avons deux présentations différentes :

- Les entiers positifs sont stockés en binaire par conversion simple.
- Les entiers négatifs sont stockés en binaire par complément à deux.

Cette méthode permet des opérations arithmétiques sans corrections.

2.6.2 Complément à deux

Le complément à deux se calcule en deux étapes : on inverse d'abord les bits (complément à un) puis, on ajoute 1 au résultat.

Par exemple : on a 13 se note en binaire : 0000 1101 sur 8 bits, alors :-13 se note 1111 0011 sur 8 bits

L'opposé d'un entier est son complément à deux. En Assembleur, on le calcule avec le mnémonique neg.

3 Présentation du 8086

3.1 Vue externe du 8086

3.1.1 Le packaging

Le 8086 se présente sous la forme d'un boîtier.



Figure 3. Processeur 8086

3.1.2 Les différentes broches

Ce processeur est doté d'un bus d'adresse et d'un bus de données qui sont **multiplexés** (certaines pattes transmettent à certains moments un bit d'adresse et à d'autres moments un bit de donnée).

Nous avons 16 broches qui permettent de transporter les données (AD0 ... AD15), la largeur du bus de données est de 16 bits.

Nous avons 20 broches qui permettent de véhiculer les adresses (AD0 ... A19), le 8086 peut donc adresser 2^{20} (=1048576) positions mémoire différentes contenant chacune 1 octet (8 bits) : la mémoire a donc au plus une capacité de 1 Mo.

3.1.3 Les différentes broches

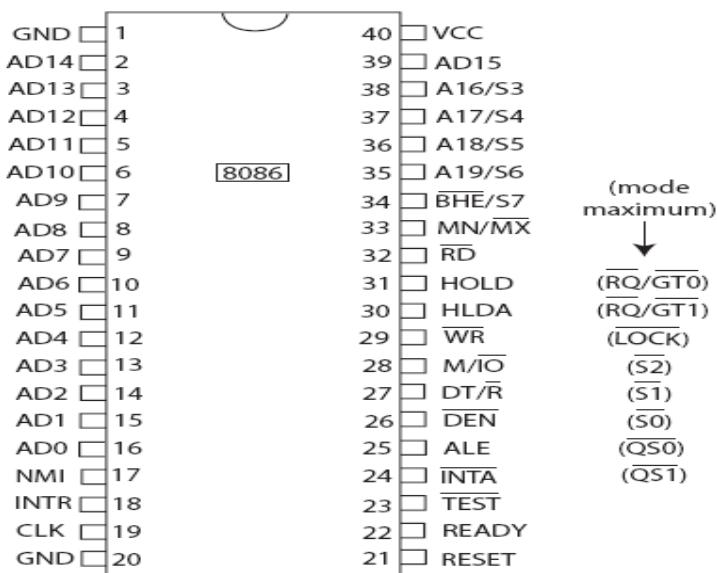


Figure 4. 8086, représentation matérielle

Le 8086 peut également être représenté d'une manière fonctionnelle. Cela permet de mettre en évidence les différents signaux en entrée et en sortie du processeur.

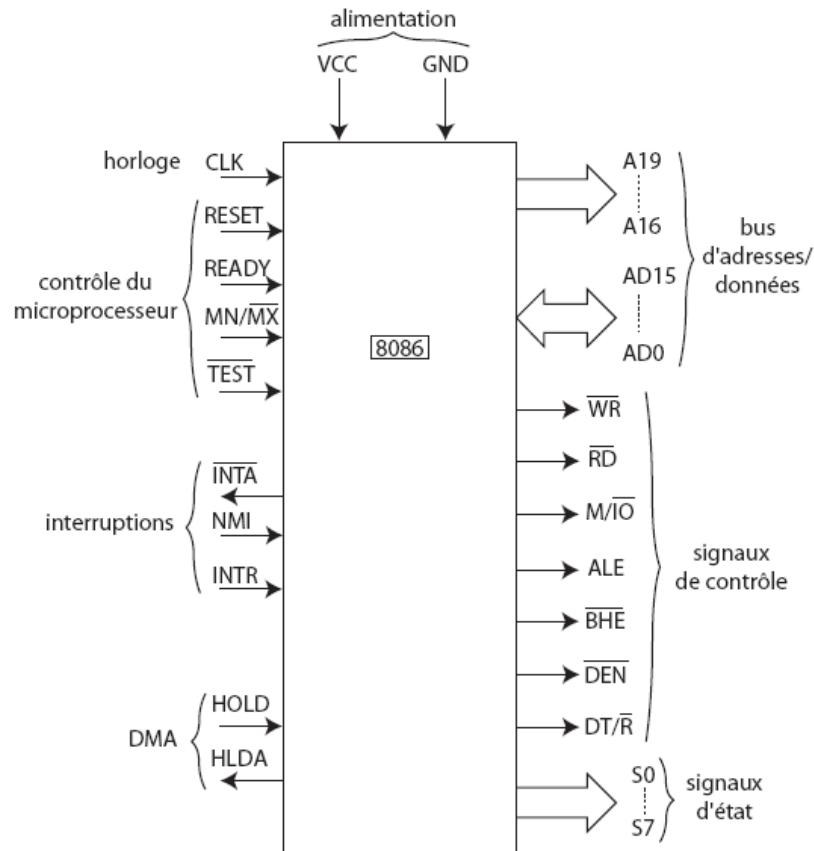


Figure 5. 8086, représentation fonctionnelle

3.2 Vue interne du 8086

3.2.1 Le schéma bloc

Dans les *databooks* qui décrivent les processeurs, ces derniers sont souvent représentés sous forme de schémas bloc afin de mettre en évidence :

- Les gros ensembles de composants ;
- Les bus qui relient ces gros ensembles.

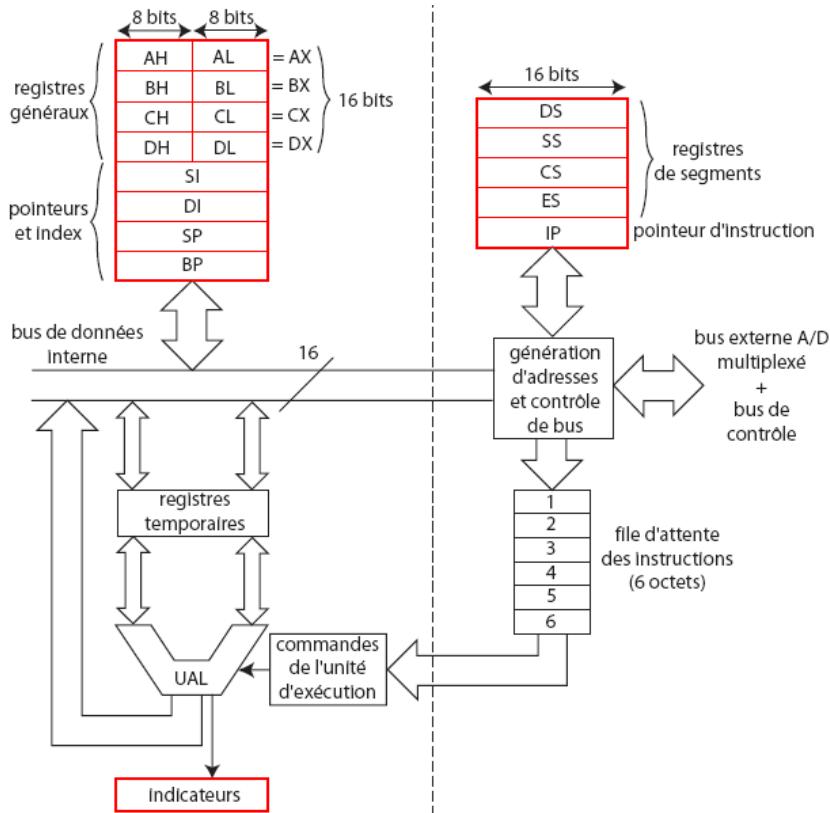


Figure 6. le schéma bloc de 8086

3.2.2 Les registres et le bus interne de données

L'analyse du schéma bloc montre que le 8086 est organisé autour d'un bus interne de données d'une largeur de 16 bits. Autour de ce bus, nous trouvons :

- Des registres de 16 bits permettant de stocker des données ;
- Des registres de 16 bits permettant de stocker une part d'une adresse (qui est codée sur 20 bits) ;
- Des registres temporaires qui débouchent sur une ALU ;
- Un bloc qui permet de former une adresse sur 20 bits à partir de morceaux d'adresse codés sur 16 bits.

3.3 Types de registres

Le 8086 possède six types de registres

1. Les registres généraux.
2. Les registres d'index.
3. Les registres de segment.
4. Les registres pointeurs.
5. Le registre IP.
6. Le registre d'état.

3.3.1 Les registres généraux

3.3.1.1 Définition

Le 8086 dispose de registres généraux qui peuvent être vus de deux manières :

- Si nous manipulons des nombres codés sur 16 bits (ce que nous désignons sous le terme de **mot**), nous utilisons les registres en mode 16 bits et nous disposons alors de 4 registres appelés

AX, BX, CX et DX.

- Si nous manipulons des nombres codés sur 8 bits (ce que nous désignons sous le terme d'**octet**), nous utilisons les registres en mode 8 bits et nous disposons de 8 registres appelés

AH, AL, BH, BL, CH, CL, DH et DL.

Le H est mis pour "High" et le L pour "Low" afin de rappeler que AH contient l'octet de poids fort du mot de 16 bits contenu dans AX et que AL contient l'octet de poids faible de ce même mot de 16 bits.

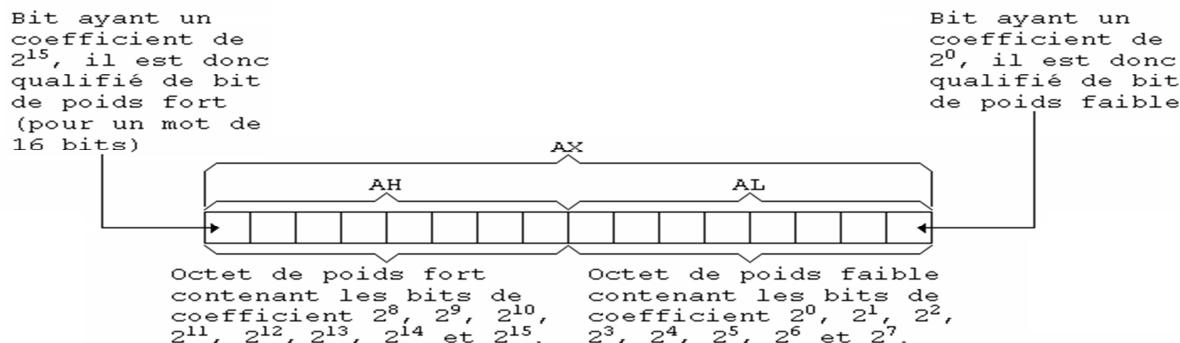


Figure 7. Registres généraux

3.3.1.2 Les deux modes d'utilisation

Ces registres sont qualifiés de généraux car on peut les utiliser de manière indifférenciée pour les calculs arithmétiques et logiques mais ils ont un rôle bien précis dans d'autres circonstances comme nous le verrons un peu plus loin :

- **AX**, appelé l'**accumulateur**, est utilisé pour stocker les résultats de certains calculs arithmétiques et logiques ;
- **BX**, appelé la **base**, est utilisé comme registre de base pour des données dans le segment pointé par le registre de segment ES ;
- **CX**, appelé le **compteur**, est utilisé pour les boucles ;
- **DX**, appelé le **registre de données**, contient l'adresse des ports d'entrée/sortie (pour les instructions IN et OUT) et sert également d'extension à AX lorsqu'on manipule des données sur 32 bits.

3.3.2 Les registres d'index

Le 8086 possède deux registres d'index, notés SI et DI, qui sont utilisés pour indexer les éléments d'un tableau.

Dans les instructions de mouvement de chaînes d'octets, ils sont utilisés simultanément :

- **SI (Source Index)** indexe les caractères de la chaîne émettrice ;
- **DI (Destination Index)** indexe les caractères de la chaîne de destination.

3.3.3 Les registres de segment

Le 8086 peut adresser une mémoire de 1 Mo grâce à son bus d'adresse de 20 bits de large. Cette mémoire est découpée en tranches de 64 Ko. maximum, appelées **segments** et référencées par 4 registres.

Les 4 registres segment sont les suivants :

- **CS (Code Segment)** pointe sur la base du segment qui contient le code (les instructions que doit exécuter le processeur) ;
- **DS (Data Segment)** pointe sur la base du segment contenant les données (variables, tableaux ...) ;
- **SS (Stack Segment)** pointe sur la base du segment qui contient la pile gérée par les registres SP et BP ;
- **ES (Extra Segment)** pointe sur la base d'un segment supplémentaire qui est généralement utilisé pour compléter le segment de données.

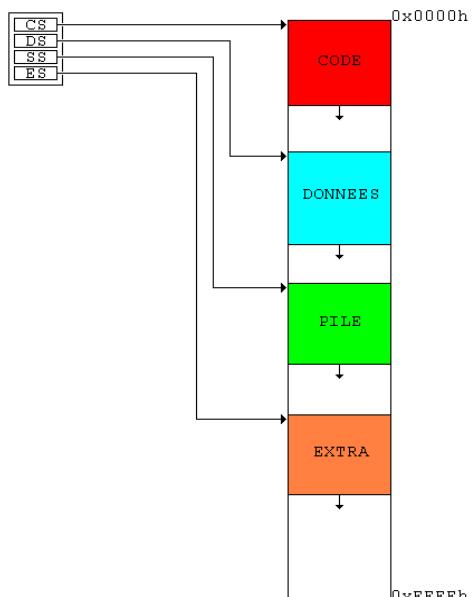


Figure 8. Segmentation de la mémoire

3.3.4 Les registres pointeurs

Le 8086 possède deux registres pointeurs, notés SP et BP, qui sont utilisés pour gérer la pile :

- **SP (Stack Pointer)** pointe sur le sommet de la pile et il est mis à jour automatiquement par les instructions d'empilement et de dépilement ;
- **BP (Base Pointer)** pointe la base de la région de la pile contenant les données accessibles (variables locales, paramètres,...) à l'intérieur d'une procédure. Il doit être mis à jour par le programmeur.

La pile est une zone de mémoire qui permet de conserver de manière temporaire des données (par exemple, l'état des registres lors d'un appel de procédure).

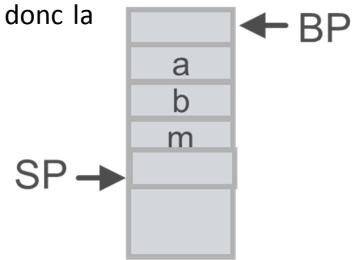
Cette pile est de type LIFO (*Last In First Out*) et elle est manipulé par des instructions telles que :

- PUSH pour empiler une valeur de 16 bits (SP est décrémenté automatiquement de 2) ;
- POP pour dépiler une valeur de 16 bits (SP est incrémenté automatiquement de 2).

La pile permet également de conserver les arguments d'une procédure. On utilise alors le registre BP (géré par le programmeur) pour accéder à ces arguments (cela permet de se déplacer dans la pile pour accéder aux informations sans avoir à modifier SP).

Si on considère la procédure min (int a, int b, int m), on a donc la représentation suivante en mémoire:

- a est en SP+6
- b est en SP+4
- m est en SP+2



3.3.5 Le registre IP

Le registre IP (*Instruction Pointeur*) est appelé « pointeur d'instruction » ou « compteur ordinal ».

Comme son nom l'indique, il permet de pointer une case mémoire dans le segment de code afin que le 8086 puisse charger la prochaine instruction à exécuter.

3.3.6 Le registre indicateur

Le registre indicateur contient des drapeaux (*flags*) qui sont mis à 0 ou 1 par le processeur (à la suite de l'exécution d'une instruction) ou par le programmeur pour modifier le mode de fonctionnement du processeur :

- **CF:** Le bit **Retenue** (*Carry*), encore appelé **Dépassement**,
 - Il est mis à 1 par l'ALU lorsqu'une addition de deux nombres de 16 bits (respectivement de 8 bits) produit un résultat sur 17 bits (respectivement sur 9 bits).
 - Il peut être utilisé pour effectuer des additions sur 32 bits (voir les prochaines sections).
- **PF:** Le bit **Parité** (*Parity*) est mis à 1 par l'ALU lorsque le résultat d'une opération contient un nombre pair de 1.
- **AF:** Le bit **Retenue Intermédiaire** (*Auxiliary carry*) est mis à 1 par l'ALU lorsqu'il y a une retenue du quartet de poids faible (8^{ème} bit) vers le quartet de poids fort (9^{ème} bit)
- **ZF:** Le bit **Zéro** (*Zero*) est mis à 1 par l'ALU quand le résultat d'une opération est 0.
- **SF:** Le bit **Signe** (*Signe*) est mis à 1 par l'ALU lorsque le bit de poids fort du résultat d'une opération est égal à 1.
- **OF:** Le bit **Débordement** (*Overflow*) est mis à 1 par l'ALU lorsqu'un débordement de la capacité de stockage du résultat est survenu et qu'un chiffre significatif a été perdu.
- **IF:** Le bit **Interruptions externes** (*Interrupt Enable*) est mis à 1 par le programmeur pour spécifier au processeur qu'il doit tenir compte du signal d'interruption envoyé par les autres circuits électroniques de la carte mère.

- **DF:** Le bit **Direction** (*Direction*) est mis à 1 par le programmeur pour indiquer que le traitement des blocs de données s'effectuera par ordre décroissant des adresses (par exemple, la recopie d'une chaîne de caractères).
- **TF:** Le bit **Pas-à-pas** (*Trap*) est mis à 1 par le programmeur pour forcer le processeur à fonctionner en mode pas-à-pas (pour mettre au point des cartes électroniques par exemple).

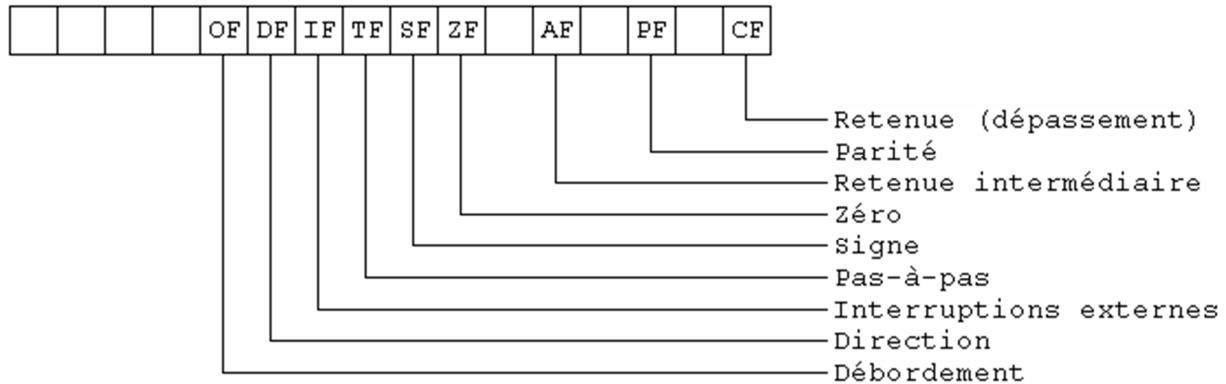


Figure 9. Registre indicateur

Remarque concernant le bit CF :

On peut utiliser le bit CF (retenue, dépassement) pour effectuer des additions sur 32 bits. Si on considère deux nombres A et B codés sur 32 bits, on va additionner ces deux nombres en procédant de la manière suivante :

- Addition des 16 bits de poids faible de A et de B ;
- Addition des 16 bits de poids fort de A et de B avec le bit CF.

Ce bit CF permet donc de propager la retenue entre le bit n°16 et le bit n°17 de l'addition.

4 Présentation de l'environnement Emu8086

4.1 L'environnement de programmation

4.1.1 Le démarrage

Lorsqu'on démarre Emu 8086, l'application propose une interface comparable à celle de la figure ci-dessous.

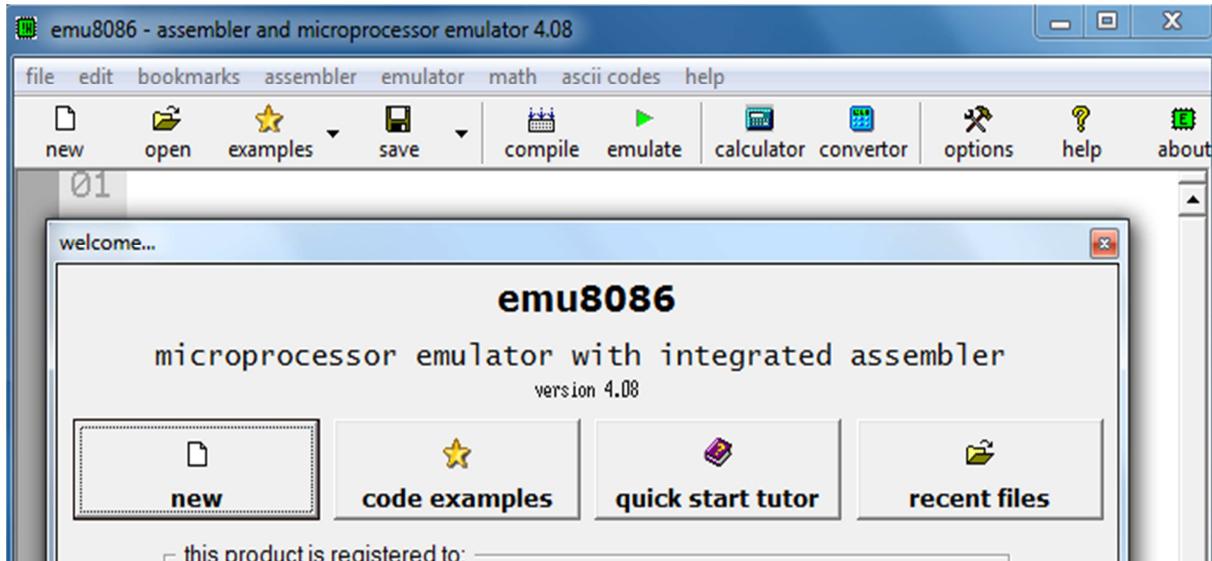


Figure 10. Écran de démarrage de l'EMU8086

4.1.2 Le premier programme assembleur

Si nous appuyons sur le bouton "Code Examples", nous avons accès à une liste d'exemple. Le programme Hello World étant trop complexe à ce stade (il faut d'abord décrire le fonctionnement de la carte vidéo pour comprendre cet exemple), nous allons nous orienter vers le programme "Add / Subtract".

Appuyer sur le bouton « new », puis choisir le type de fichier « COM »

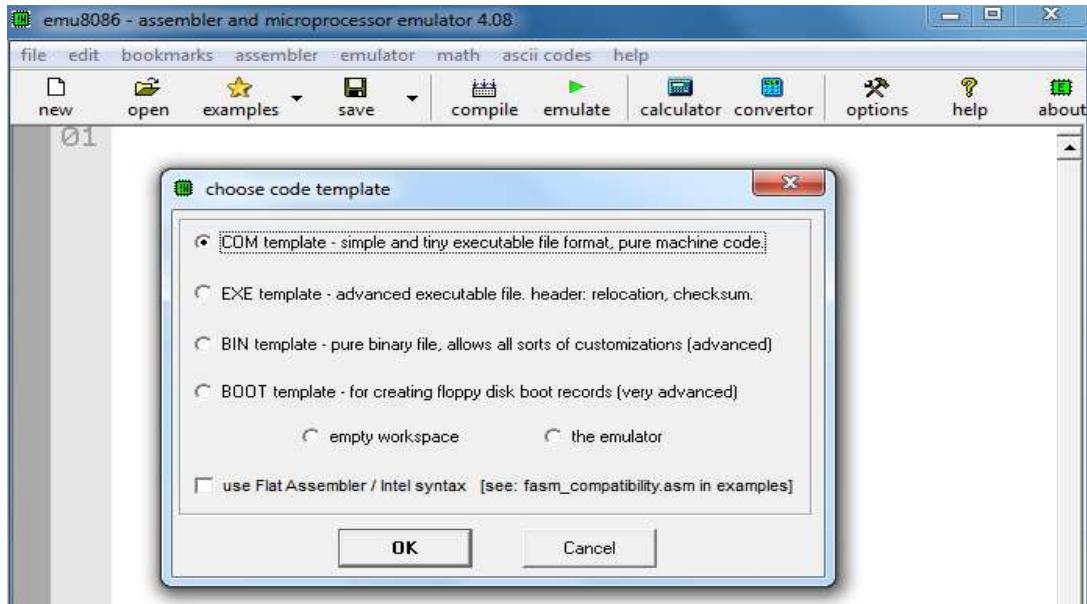


Figure 11. EMU8086- choix du type de fichier

Le code du programme apparaît dans la fenêtre principale.

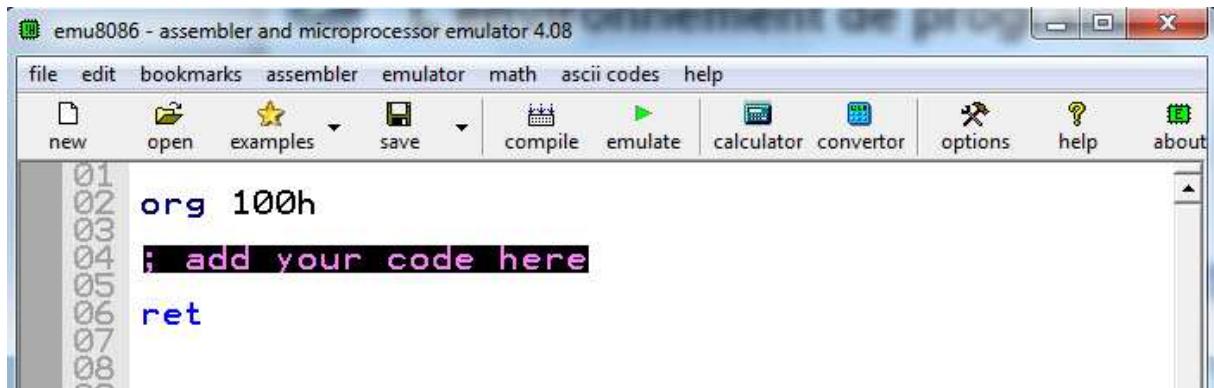
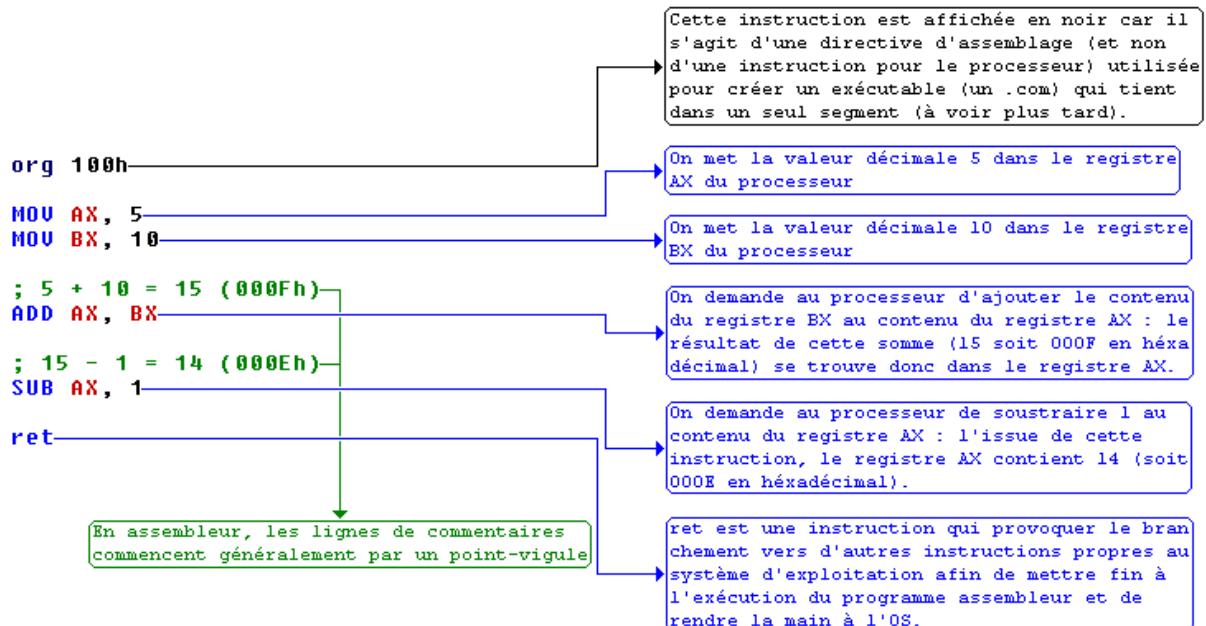


Figure 12. Code d'un fichier .com vide

Compléter le fichier .com par le code Assembleur suivant:

```
org 100h
MOV AX,5
MOV BX,10
ADD AX,BX
SUB AX,1
ret
```

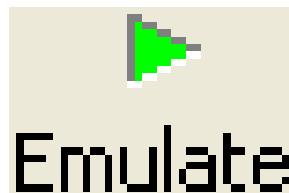
4.1.3 Description du premier code



4.2 L'environnement d'exécution

4.2.1 L'exécution du premier programme

Nous pouvons appuyer sur le bouton "Emulate" afin de compiler le programme (créer un exécutables - un .com - qui se trouve dans le répertoire MyBuild, sous le répertoire d'installation d'Emu 8086) et de lancer le module de simulation.



La fenêtre de gauche apparaît, nous pouvons appuyer sur les boutons du bas afin de faire apparaître d'autres fenêtres qui nous aiderons à comprendre le fonctionnement de ce programme au niveau du pseudo-processeur.

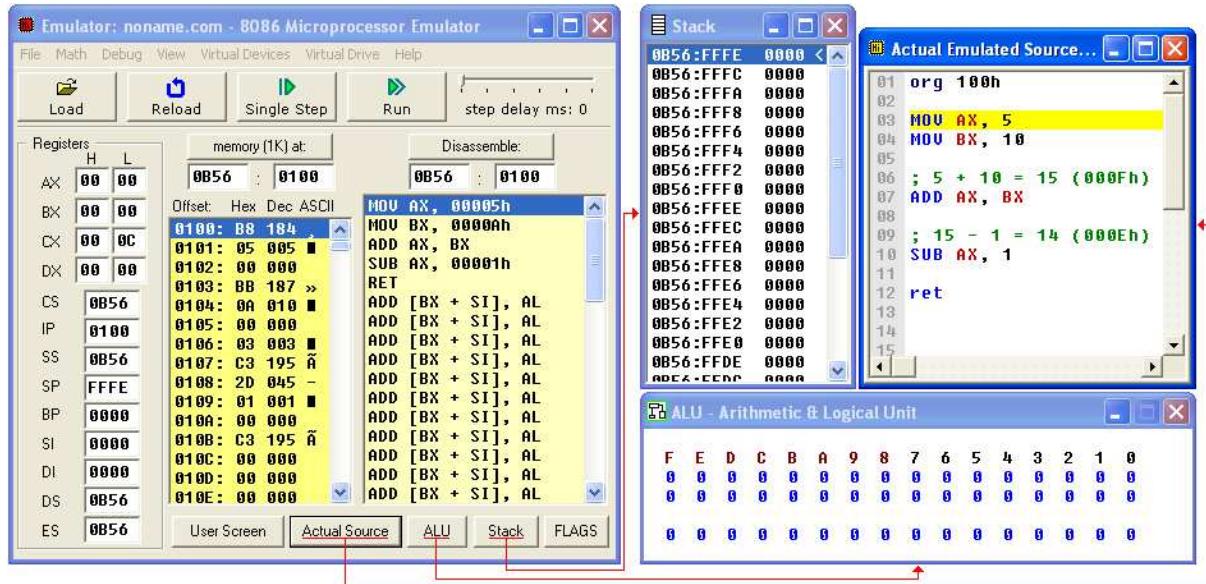


Figure 13. EMU8086-Les fenêtres de l'exécution

4.2.2 L'adressage du code objet en mémoire

La partie gauche de la fenêtre principale comporte les différents registres (AX, BX, IP ...) et une représentation du programme assemblé en mémoire vive.

Nous constatons que sous le bouton "memory (1K) at", nous avons deux nombres hexadécimaux :

- le premier est le nombre stocké dans le registre CS
- le second à celui stocké dans IP.

Pour désigner une instruction contenue dans une case de la mémoire de 1 Mo., nous avons besoins d'une adresse codée sur 20 bits (cela dépasse les capacités des registres interne du 8086).

Pour contourner ce problème, l'adresse est codée avec deux registres :

- le registre CS contient l'adresse de la base du segment de code
- le registre IP contient une adresse relative à cette base (on parle de déplacement ou d'offset).

L'adresse absolue est obtenue en faisant **CS * 16 + IP**.

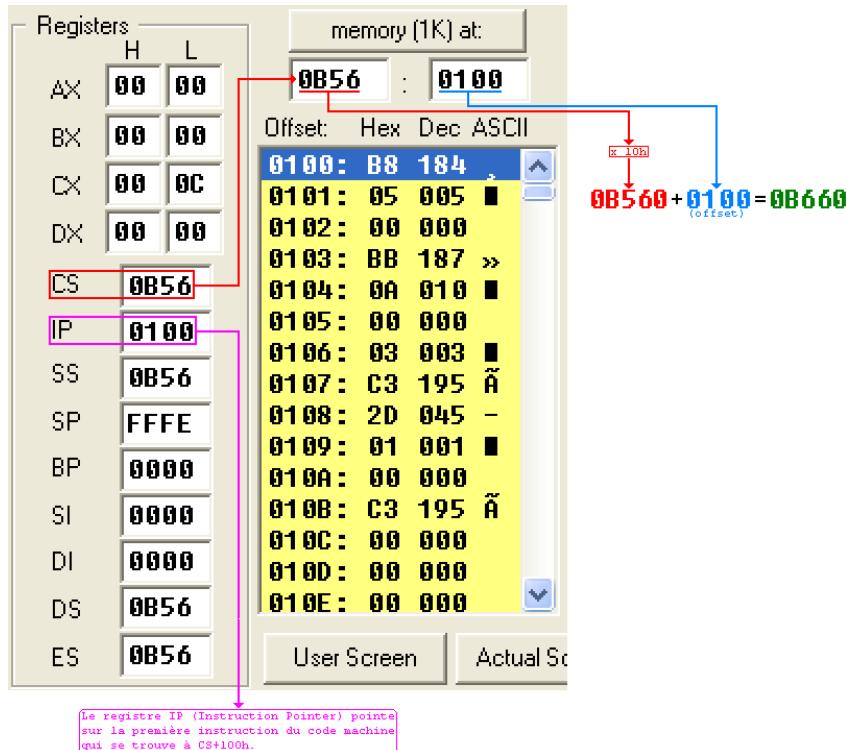


Figure 14. Adressage du code en mémoire

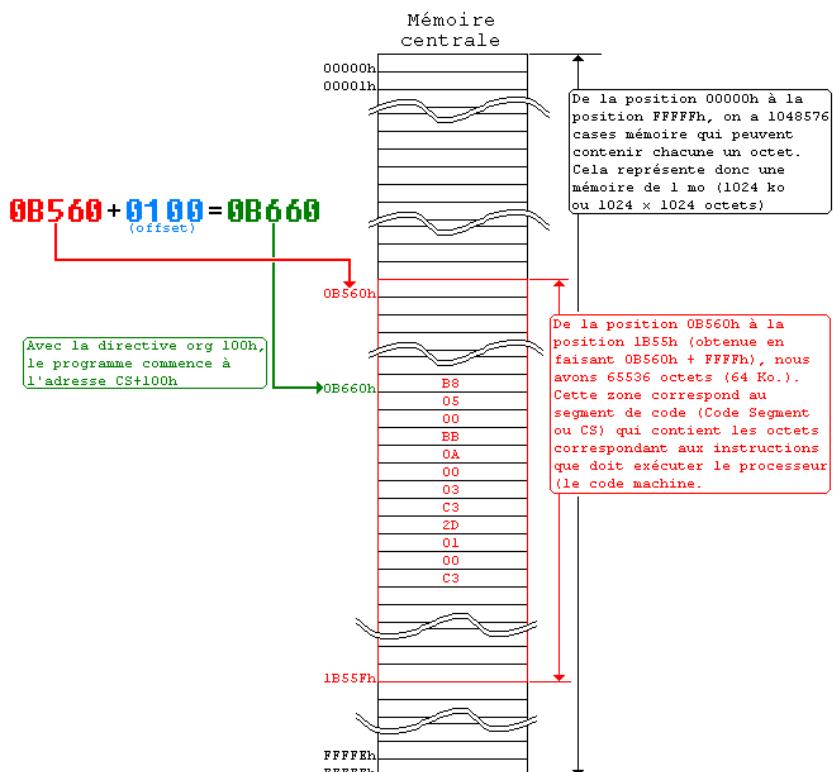


Figure 15. Codage du premier exemple en mémoire

4.3 Assemblage et la notion d'opcode

4.3.1 La notion de mnémonique et d'opcode

Chaque instruction assembleur (mnémonique) possède une correspondance en langage machine, appelée **code opérateur** ou **opcode**, qui peut être codé sur un ou plusieurs octets selon le besoin.

Cette correspondance est spécifiée par le constructeur - Intel - dans son *databook* ce qui permet au programmeur de développer leur assembleur et/ou leur émulateur.

Par exemple l'opcode des registres généraux donné par Intel est :

- AX: 000
- BX:011
- CX:001
- DX:010

4.3.2 Instruction à dressage immédiat et à registre

Les instructions **opération registre, valeur** sont dites à **adressage immédiat** car on indique "en dur" la valeur qui doit être chargée dans le registre

Les instructions **opération registre, registre** sont dites **de registres** car les données, sur lesquelles les opérations sont effectuées, sont contenues dans les registres.

Dans ces deux cas de figures, il n'y a pas de construction d'adresses sur 20 bits puisqu'on ne fait pas intervenir la mémoire, ces instructions sont donc rapides à exécuter.

4.3.3 Extrait du databook de l'Intel 8086

D'après le *databook* (écrit par Intel) qui décrit le fonctionnement du processeur 8086, nous avons les correspondances suivantes entre les mnémoniques et les octets (ou les groupes d'octets) :

```
* Un mov immédiat vers un registre se traduit par un bloc de deux octets si w=0 ou de trois octets si w=1
  1011'w.reg  [data]  [data si w=1]
  avec reg=000 pour AL (si w=0) ou AX (si w=1)
  et reg=011 pour BL (si w=0) ou BX (si w=1)

* Un add entre deux registres se traduit par un bloc de deux octets
  000000'd'w  mod[reg'r/m]
  avec d=1 (si c'est vers la destination) ou d=1 (si c'est depuis une source). w=0 (addition sur 8 bits) ou w=1 (addition sur 16 bits). mod=11 car l'addition fait intervenir deux registres : reg est égal au code du premier registre et r/m au code du deuxième registre

* Un sub immédiat depuis l'accumulateur (le registre AX) se traduit par un bloc de deux octets si w=0 ou de trois octets si w=1
  0010110'w  [data]  [data si w=1]

* Un ret (un retour d'appel de programme, de fonction ou de procédure) se traduit par un octet si le saut s'effectue dans le segment CS
  11000011
```

Figure 16. Extrait du databook de l'Intel 8086

4.3.4 Description du code

4.3.4.1 OpCode de l'Instruction MOV

Pour la première instruction du programme : **MOX AX,5**, signifie qu'on transfert de façon immédiate le nombre 5 vers le registre AX (l'accumulateur).

Comme AX est un registre sur 16 bits, cela signifie qu'on manipule un mot donc : w=1 et cette instruction occupera trois octets d'après de databook. En outre, comme il s'agit du registre AX, nous avons reg=000 (cf. databook). Enfin, comme les processeurs Intel sont de type « **big endian** », l'octet de poids fors est placé après l'octet de poids faible : le nombre 5, qui aurait dû être codé par 0005h, est codé par 0500h.

Pour résumer le MOV AX , 5 se traduit par :

- 1011 w reg | data |data si w=1 avec W=1 et Reg=000

Donc l'opcode de l'instruction est 10111000 00000101 00000000 soit B8 05 00

De même façon, on déduit l'opcode de l'instruction **MOV BX, 10** par :

- 10111011 00001010 00000000 soit BB 0A 00
- Car on a w=1 et reg = 011

4.3.4.2 OpCode de l'Instruction ADD

L'instruction **ADD AX, BX** signifie qu'on fait une addition sur 16 bits d'un registre BX vers un autre registre AX, nous avons donc w=1 et d=1. Comme il s'agit d'une addition entre deux registres, nous avons le champ **mod** égale à 11 et le champ **r/m** (qui correspond à la destination) est égale au code du registre AX soit 000. Enfin le champ **reg** est égal au code du registre BX soit 011.

Cette instruction se traduit donc par 00000011 11000011 soit 03 C3.

4.3.4.3 OpCode de l'Instruction SUB

L'instruction **SUB AX, 1** signifie qu'on fait une soustraction immédiate sur 16 bits depuis l'accumulateur (le registre AX), nous avons donc w=1 et l'instruction qui est codé sur 3 octets. Comme le 8086 est un processeur « big endian », 1 se traduit par 0100h au lieu de 0001h.

Cette instruction se traduit donc par 00101101 00000000 soit 2D 01 00.

4.3.4.4 OpCode de l'Instruction RET

L'instruction **RET** signifie qu'on fait un retour vers le programme qui appellé ce code machine (dans notre cas de figure, il s'agit de l'OS). Concrètement cela se traduit par un saut dans la pile de code CS (on reste dans la même pile) vers l'instruction qui rend la main à l'OS. Cette instruction se code donc sur un octet.

Cette instruction se traduit donc par 11000011 soit C3.

4.3.4.5 Correspondance opcode - mnémonique

Le programme assembleur ci-dessus sera traduit et met en mémoire comme la figure suivante:

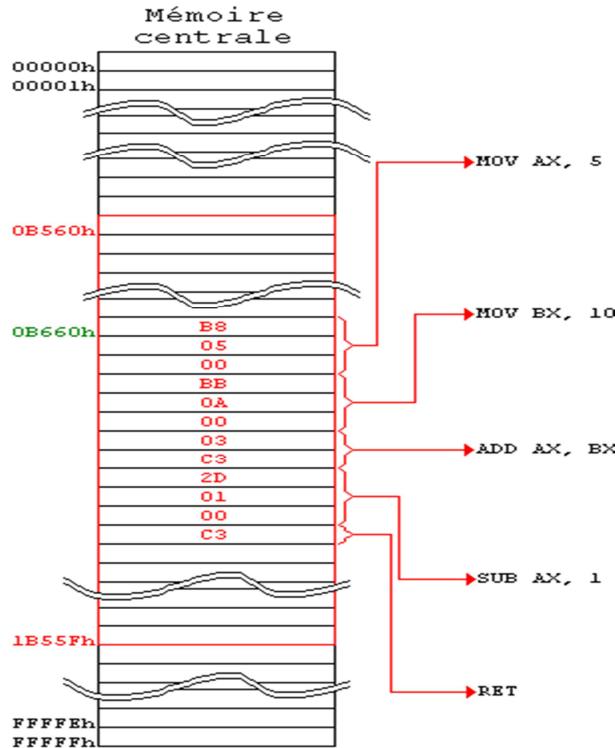


Figure 17. Correspondance opcode – mnémonique

4.4 Exécution

4.4.1 Exécution de la première instruction (MOV)

Nous pouvons alors appuyer sur le bouton "Single Step" afin de lancer une exécution pas-à-pas du programme.

La première instruction (MOV AX, 5) est chargée, décodée et exécutée par le processeur. Les changements sont les suivants :

- Le registre AX contient la valeur 5 ;
- Le registre IP contient la valeur 103h (IP passe de 100h à 103h car MOV AX, 5 est codé sur 3 octets) ;
- La prochaine instruction pointée par IP (MOV BX, 10) est surlignée dans les différents écrans.

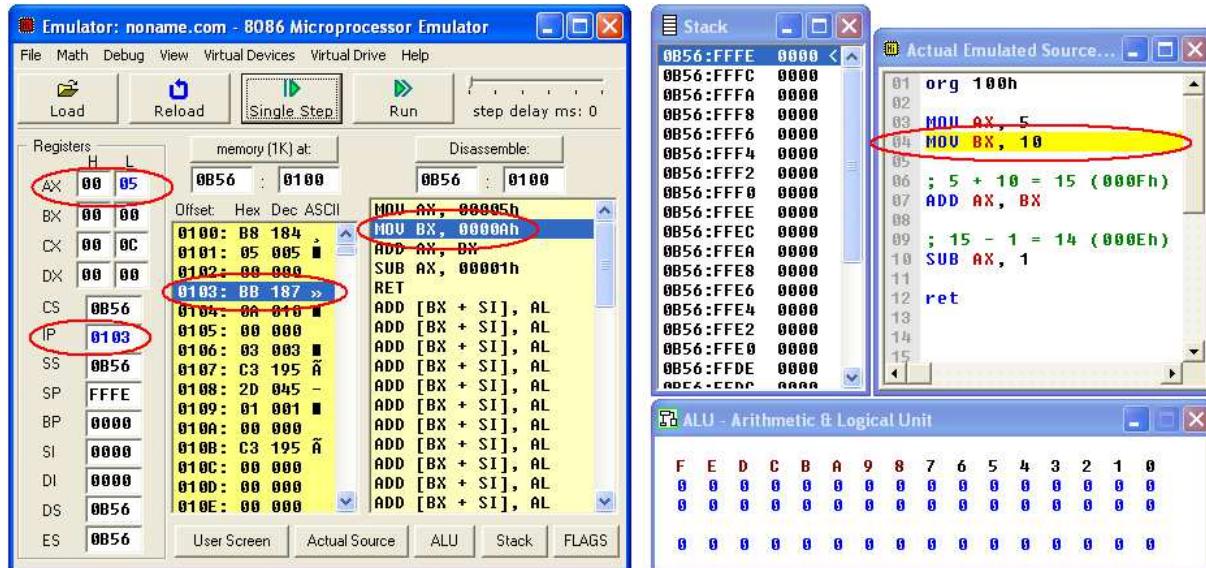


Figure 18. EMU: Exécution de la première instruction

4.4.2 Exécution de la deuxième instruction (MOV)

Nous appuyons sur le bouton "Single Step" pour exécuter la deuxième instruction (MOV BX, 10). A l'issue de cette exécution, nous constatons les changements suivants :

- Le registre BX contient la valeur 10 ;
- Le registre IP contient la valeur 106h (IP passe de 103h à 106h car MOV BX, 10 est codé sur 3 octets) ;
- La prochaine instruction pointée par IP (ADD AX, BX) est surlignée dans les différents écrans.

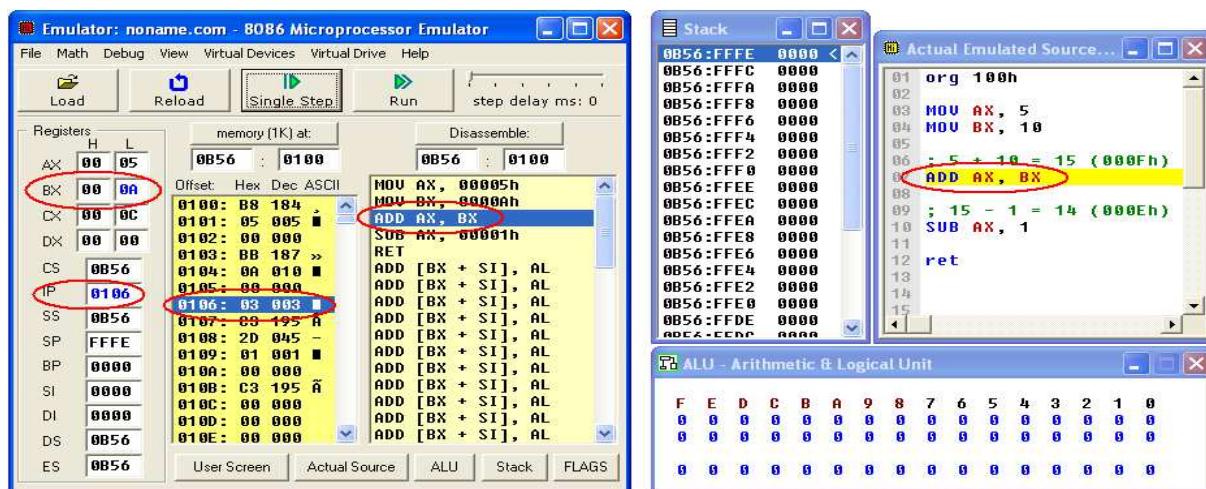


Figure 19. EMU: Exécution de la deuxième instruction

4.4.3 Exécution de la troisième instruction (ADD)

Nous exécutons la troisième instruction (ADD AX, BX), ce qui entraîne les changements suivants :

- La fenêtre ALU affiche le contenu des deux registres temporaires d'entrées et le résultat de l'opération ;
- Le registre AX contient la valeur 0Fh (05h + 0Ah = Fh) ;

- Le registre IP contient la valeur 108h (IP passe de 106h à 108h car ADD AX, BX est codé sur 2 octets) ;

La prochaine instruction pointée par IP (SUB AX, 1) est surlignée dans les différents écrans.

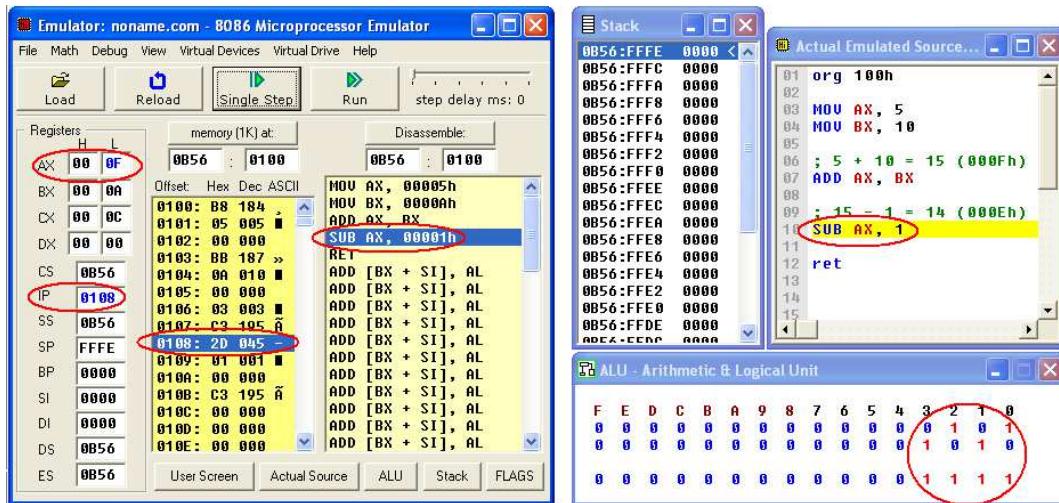


Figure 20. EMU: Exécution de la troisième instruction

4.4.4 Exécution de la quatrième instruction (SUB)

Nous exécutons la quatrième instruction (SUB AX, 1), ce qui entraîne les changements suivants :

- La fenêtre ALU affiche le contenu des deux registres temporaires d'entrées et le résultat de l'opération ;
- Le registre AX contient la valeur 0Fh (0Fh + 01h = Eh) ;
- Le registre IP contient la valeur 10Bh (IP passe de 108h à 10Bh car SUB AX, 1 est codé sur 3 octets) ;

La prochaine instruction pointée par IP (RET) est surlignée dans les différents écrans.

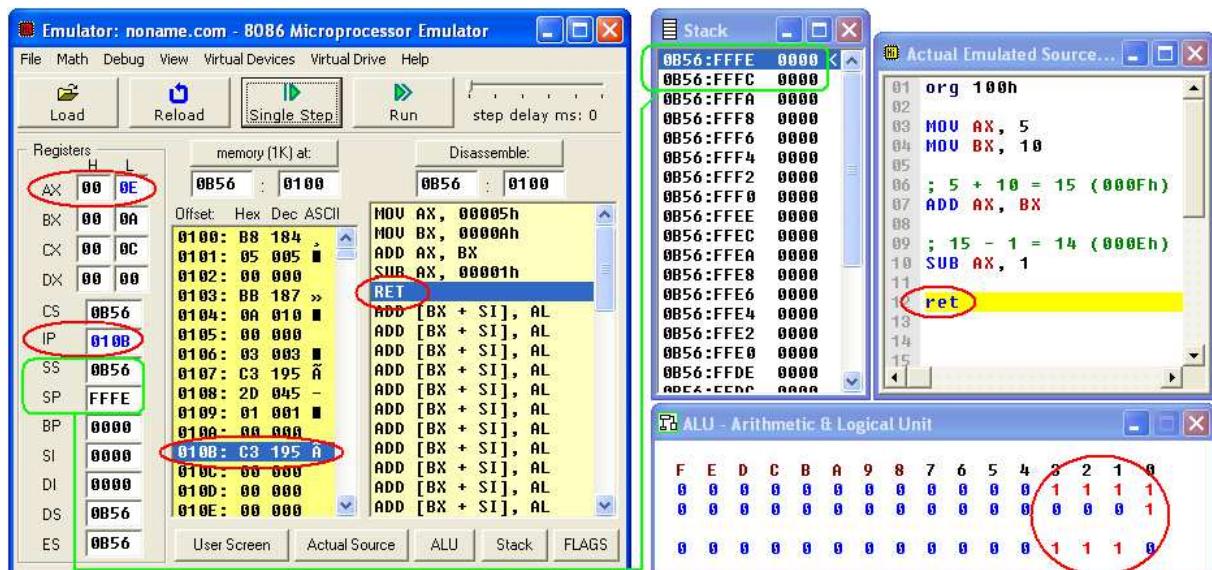


Figure 21. EMU: Exécution de la quatrième instruction

4.4.5 Exécution de la cinquième instruction (RET)

Nous exécutons la cinquième instruction (RET). Comme nous le verrons un peu plus loin, cette instruction est équivalente dans ce cas de figure à effectuer POP IP ce qui amène les deux traitements ci-dessous :

On dépile la valeur stockée dans la pile à la position SS:SP (autrement la case mémoire dont l'adresse est SS * 16 + SP) pour la charger dans le registre IP. Dans notre exemple, SS:SP vaut 0B56h:FFF Eh et **le mot** (autrement dit les 2 octets consécutifs) stockée à cette adresse valent 0000h.

A l'issue de ce traitement, IP est donc égale à 0000h et la prochaine instruction qui sera exécutée est située à l'adresse CS:IP soit 0B56h:0000h

Dans le même temps, on modifie la valeur de SP car on a déplié un élément : on ajoute 2 à SP donc SP vaut FFF Eh + 0002h = 10000h soit 0000h car SP est un registre 16 bits.

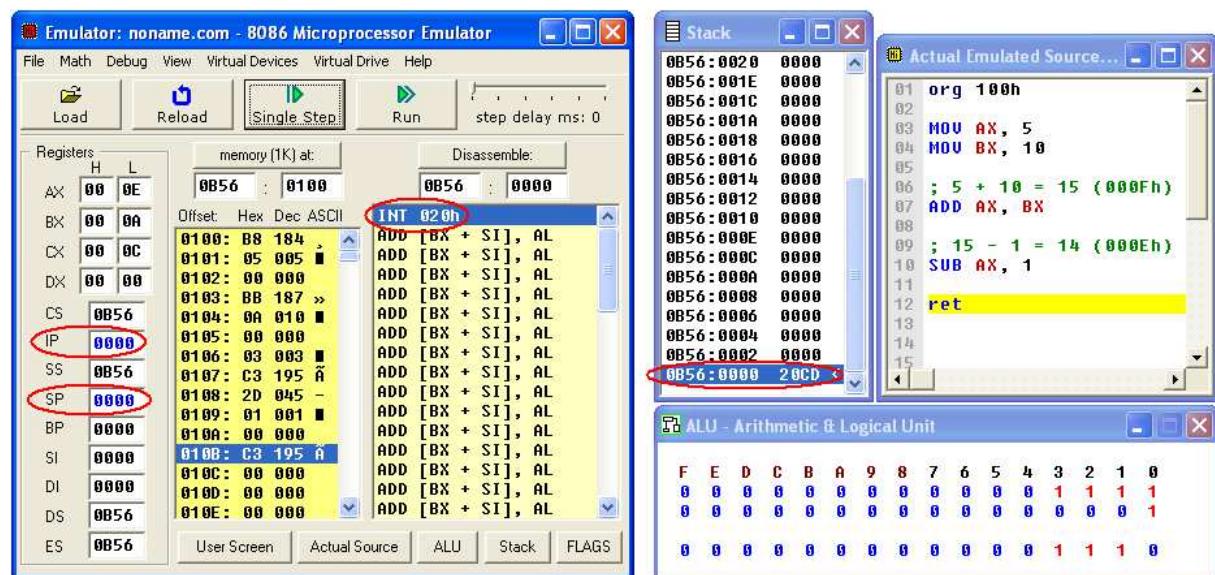


Figure 22. EMU: Exécution de la cinquième instruction

4.4.6 Exécution de la sixième instruction (INT)

Nous appuyons sur "single step" et nous exécutons une instruction que nous n'avons pas saisie dans notre programme (INT 020h) qui a été automatiquement ajoutée par le système d'exploitation (de même que la valeur qui était empilée et la valeur des registres SS et SP).

Nous décrirons le mécanisme des interruptions logicielles un peu plus loin. A notre niveau, nous devons comprendre que INT 020h donne l'ordre au processeur d'exécuter l'interruption (comprendre la routine ou la procédure) numéro 20h. A l'appel de cette procédure, le processeur modifie le contenu de ses registres car il saute dans une autre zone mémoire puis il exécute cette instruction qui consiste simplement à terminer le programme et à rendre la main au système d'exploitation (ici une émulation du DOS)

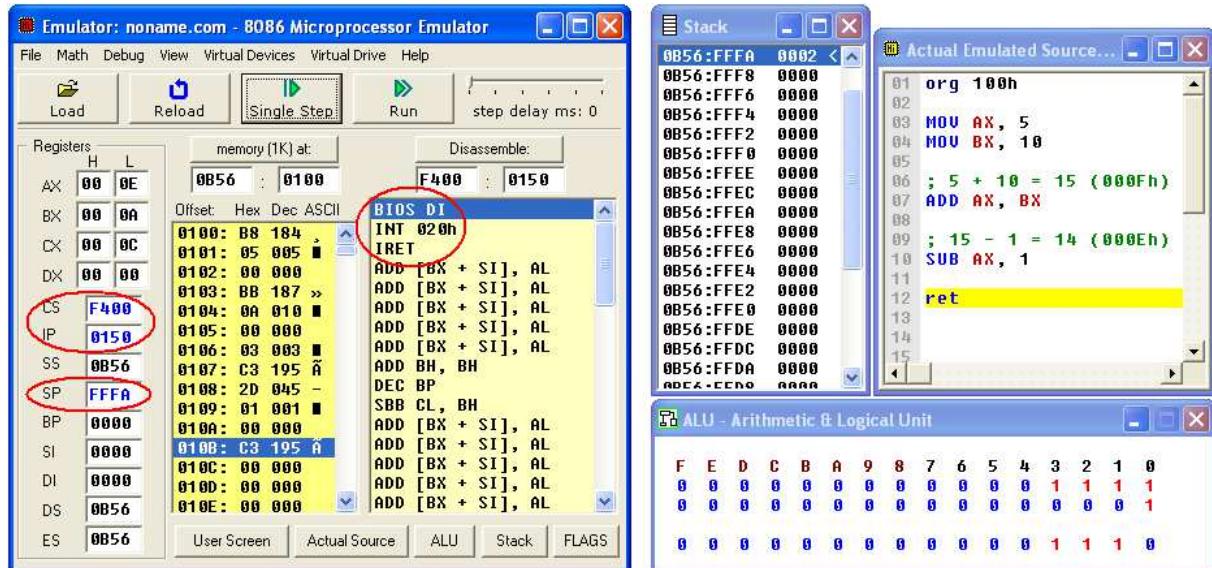


Figure 23. EMU: Exécution de l'instruction INT 020h

5 Manipulation de base

5.1 Opérations logiques

5.1.1 AND, OR, NOT et XOR

Le 8086 permet naturellement d'effectuer des opérations binaires classiques grâce au mnémoniques **AND**, **OR**, **NOT** et **XOR**. La syntaxe de ces opérateurs logiques est :

AND registre, nombre ou **AND** registre, registre

OR registre, nombre ou **OR** registre, registre

NOT registre

XOR registre, nombre ou **XOR** registre, registre

Par exemple :

- MOV AL, 01100001b
- MOV BL, AL
- AND AL, 11011111b ; On obtient AL=01000001
- MOV AL, BL
- OR AL, 01110110b ; On obtient AL=01110111
- MOV AL, BL
- NOT AL ; On obtient AL=10011110
- MOV AL, BL
- XOR AL, 01010101b ; On obtient AL=00110100

Des cas d'utilisation :

1. L'instruction **AND** permet de masquer des valeurs, par exemple :
 - MOV AL, 0C3h
 - AND AL, 00Fh ; On obtient AL=003h, donc on ne retient que les bits de poids faible
2. L'instruction **OR** peut être utilisée pour forcer des bits à 1, par exemple :
 - MOV AL, 0C3h
 - AND AL, 00Fh ; On obtient AL=00Fh, force les bits de poids faible à 1
3. L'instruction **XOR** permet d'inverser les bits d'un registre ou de les mettre plus rapidement à zéro que ne le fait un MOV
 - XOR AX, AX ; remise à 0 rapidement le AX.
 -
 - MOV BL, 0B1h
 - XOR BL, OFFh ;BL=04Eh, prend la valeur de sa négation, c.-à-d. NOT BL

5.1.2 Décalage à gauche et décalage à droite

Le 8086 permet d'effectuer des opérations de décalage sur des nombres signés ou non signés. Les bits qui sont expulsés sont stockés dans le bit CF :

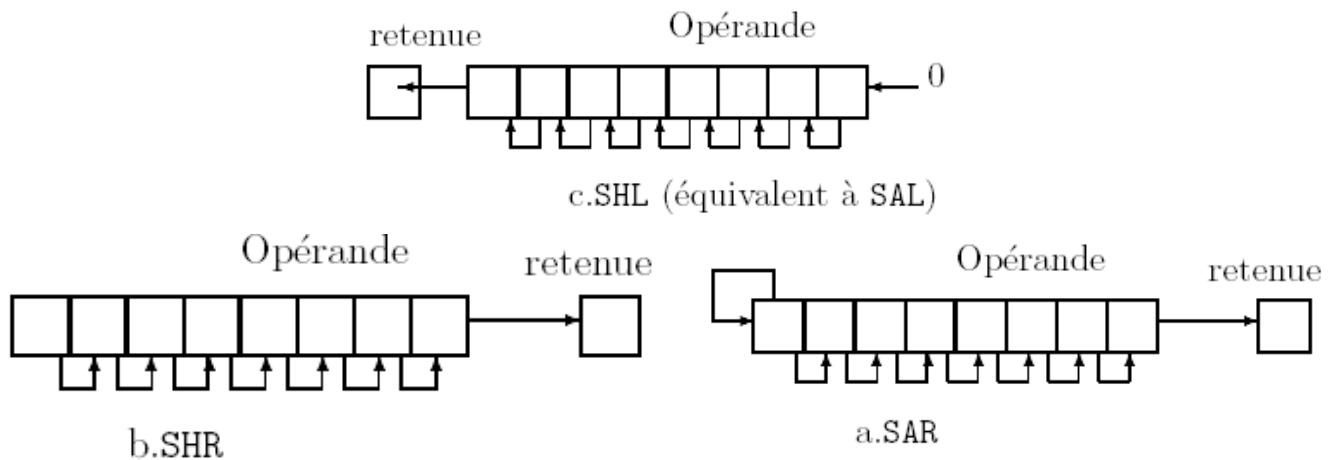
- **SHL** (shift left) et **SAL** (shift arithmetic left) ;

- **SHR** (shift right) et **SAR** (shift arithmetic right).

La différence entre les deux types de décalage est que dans la seconde version le bit de signe n'est pas concerné par ce décalage. Autrement dit, la première version traite des nombres non signés et la deuxième version permet de manipuler les nombres signés.

Cette opération est plus rapide à exécuter que l'opération de multiplication ou de division donc il est préférable d'utiliser un décalage pour effectuer une division ou une multiplication par 2.

- Le décalage logique à gauche (SHL) revient à réaliser une multiplication par 2 d'une valeur non signée.
- Le décalage logique à droite (SHR) revient à réaliser une division par 2 d'une valeur non signée.
- Le décalage arithmétique à gauche (SAL) revient à réaliser une multiplication par 2 d'une valeur non signée.
- Le décalage arithmétique à droite (SAR) revient à réaliser une division par 2 d'une valeur signée.



Le nombre de bits de décalage est indiqué dans la seconde opérande. La syntaxe d'utilisation est la suivante :

SHL/SAL/SHR/SAR registre, nombre de bits de décalage

Par exemple :

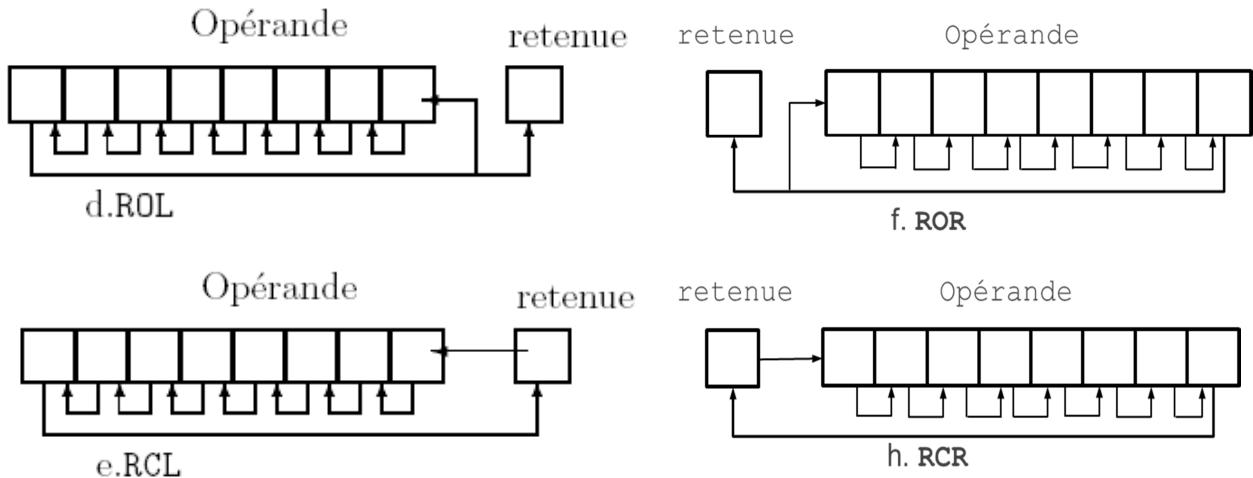
- MOV AL,11110000B
- SHL AL,1 ;On obtient AL=11100000 CF=1
- SAL AL,1 ;On obtient AL=11000000 CF=1
- SAR AL,1 ;On obtient AL=11100000 CF=0
- SHR AL,1 ;On obtient AL=01110000 CF=0

5.1.3 Rotation à gauche et rotation à droite

Le 8086 permet également de réaliser des opérations de rotation sur des nombres signés ou non signés. Les bits qui sont expulsés sont remplacés dans les trous formés par ce décalage :

- ROL (rotate left) et RCL (rotate through carry left) ;
- ROR (rotate right) et RCR (rotate through carry right).

La différence entre les deux types de décalage est que dans la seconde version le bit CF est utilisé en plus des bits à modifier (lors d'un mouvement, le trou est rempli avec une copie du bit CF et l'autre bit, qui est expulsé, est placé dans CF) (voir le schéma).



Le nombre de bits de décalage est indiqué dans la seconde opérande. La syntaxe d'utilisation est la suivante :

ROL/RCL/ROR/RCR registre, nombre de bits de décalage

Par exemple :

- MOV AL, 11110000b
- ROL AL,1 ;On obtient AL=11100001 CF=1
- RCL AL,1 ;On obtient AL=11000011 CF=1
- RCR AL,1 ;On obtient AL=11100001 CF=1
- ROR AL,1 ;On obtient AL=11110000 CF=1

5.2 Opérations arithmétiques

5.2.1 Addition et soustraction sans retenue

Nous avons vu dans la partie précédente que les mnémoniques ADD (**addition**) et SUB (**subtraction**) permettent d'effectuer des additions et des soustractions entre deux registres ou entre un registre et un nombre.

- **ADD AX, BX** effectue l'opération $AX \leftarrow AX + BX$
- **SUB AX, BX** effectue l'opération $AX \leftarrow AX - BX$

Ces opérations sont effectuées sans tenir compte de l'état initial du bit CF du registre d'état mais l'état de ce bit peut être modifié à l'issue de l'opération.

5.2.2 Incrémentation et décrémentation

Le 8086 propose deux instructions qui permettent d'incrémenter ou de décrémenter la valeur d'un registre :

- **INC AX** incrémente la valeur stockée dans AX
- **DEC BX** décrémente la valeur stockée dans BX

INC AX donne le même résultat que ADD AX, 1 mais le premier mnémonique correspond à un opcode de 1 octet alors que l'opcode correspondant au second mnémonique en occupe 3. On constate la même chose si on compare DEC BX et SUB BX, 1

5.2.3 L'addition sur 32 bits

Dans certains cas, il peut être nécessaire d'effectuer des additions sur 32 bits. Cependant, la taille des registres est limitée à 16 bits sur le 8086, nous devons donc scinder les nombres codés sur 32 bits en deux parties :

- 1 registre pour stocker les 16 bits de poids fort ;
- 1 registre pour stocker les 16 bits de poids faible.

Lorsqu'on effectue une addition sur 32 bits, on effectue d'abord l'addition entre les 16 bits de poids faible puis l'addition entre les 16 bits de poids fort en utilisant le bit CF pour propager la retenue entre les deux blocs de 16 bits.

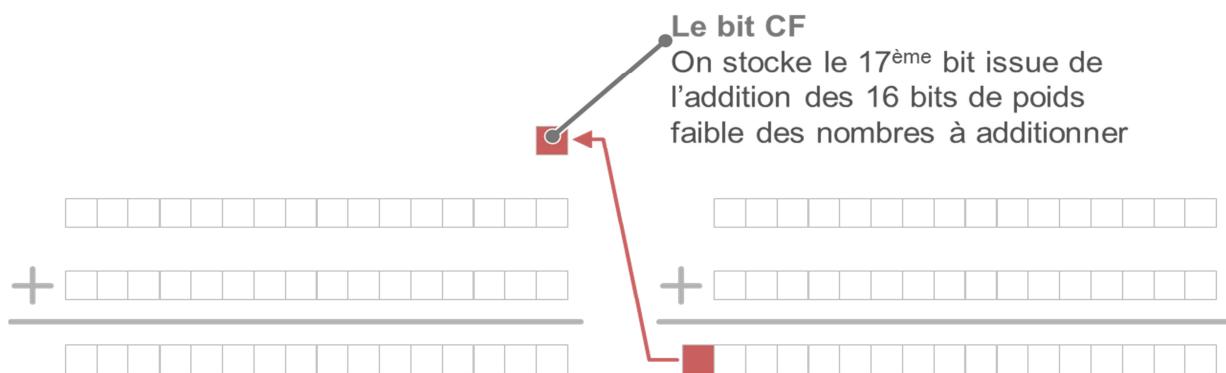


Figure 24. Addition sur 32 bits

5.2.3.1 Les mnémoniques ADC et SBB

Pour demander au processeur d'effectuer des additions qui tiennent compte du bit CF, nous devons utiliser une nouvelle mnémonique : **ADC** (*addition with carry*).

Il n'existe pas de mnémonique SUBC mais un mnémonique appelé **SBB** (*subtract with borrow*) qui effectue le même type d'opération mais en sens inverse.

Comme les opérations ADC et SBB modifient l'état du bit CF, on peut donc s'en servir pour effectuer des opérations sur 48 bits, 64 bits ... en stockant les résultats intermédiaires en mémoire : l'opération se fera donc par bloc de 16 bits.

5.2.3.2 Utilisation du bit CF : Les mnémoniques STC, CLC et CMC

Le 8086 offre 3 instructions qui permettent de modifier directement l'état du bit CF :

- **STC** (set carry flag) permet de mettre CF à 1 ;
- **CLC** (clear carry flag) permet de mettre CF à 0 ;
- **CMC** (complement carry flag) permet d'inverser l'état du bit CF.

Il est à remarquer que le 8086 offre 4 autres instructions qui permettent de modifier directement l'état des bits DF (direction) et IF (interruptions externes) :

- **STD** (set direction flag) permet de mettre DF à 1 ;
- **CLD** (clear direction flag) permet de mettre DF à 0 ;

- **STI** (set interrupt flag) permet de mettre IF à 1 ;
- **CLI** (clear interrupt flag) permet de mettre IF à 0.

Les mnémoniques STI et CLI sont utiles pour implanter au niveau assembleur des procédures qui ne peuvent pas être interrompues par des événements externes.

5.2.4 Multiplication et division

5.2.4.1 La multiplication non signée

Le 8086 utilise le mnémonique **MUL** pour effectuer des multiplications mais son mode de fonctionnement est très différent par rapport à celui de l'addition et la soustraction.

Si nous voulons multiplier un nombre N par un nombre M, nous devons d'abord copier le nombre N dans le registre AL (ou AX) puis invoquer l'instruction MUL M :

- si l'opérande M est un octet, on calcule **AL * opérande** et le résultat est stocké dans **AX** ;
- si l'opérande M est un mot, on calcule **AX * opérande** et le résultat est stocké dans les registres **DX et AX** (DX contenant la partie de poids fort).

5.2.4.2 La multiplication signée

Le 8086 propose un autre mnémonique pour effectuer les multiplication entre deux nombres signées.

Le mnémonique **IMUL** est utilisé de la même manière que le mnémonique MUL à la différence que les opérandes sont considérés comme des nombres signés (codés en complément à 2) et que le résultat est un nombre signé.

5.2.4.3 La division euclidienne non signée

Le mnémonique permettant d'effectuer une division euclidienne est DIV et son mode de fonctionnement est voisin de celui de MUL.

Dans le cas d'une division d'un nombre N par un nombre M, nous devons d'abord copier le nombre N dans le registre AX (ou AX et DX) puis invoquer l'instruction **DIV M** :

- si l'opérande est un octet, on calcule **AX / opérande** : le quotient est stocké dans AL et le reste dans AH ;
- dans le cas d'un mot, on calcule **(AX DX) / opérande** : le quotient est stocké dans **AX** et le reste dans **DX**.

5.2.4.4 La division euclidienne signée

A l'instar de la multiplication signée, le 8086 propose un autre mnémonique pour effectuer les divisions euclidiennes entre deux nombres signés.

Le mnémonique **IDIV** est donc utilisé de la même manière que le mnémonique DIV : les différences sont que les opérandes sont considérées comme des nombres signés (codés en complément à 2) et que le résultat est un nombre signé.

5.3 L'adressage

5.3.1 Adressage immédiat et adressage par registre

Jusqu'à présent, nous avons rencontré deux modes d'adresses :

- Le mode **immédiat** où un opérande est un nombre codé sur 8 ou 16 bits directement dans l'opcode. Par exemple : MOV AX, 10
- Le mode **par registre** qui fait intervenir deux registres. Par exemple : MOV AX, BX

Ces deux modes **ne faisaient pas intervenir la mémoire centrale**, mais nous allons maintenant introduire un autre mode qui manipule les données en mémoire.

5.3.2 L'adressage des instructions dans CS

Nous avons appris dans le deuxième partie que les instructions d'un programme sont pointées par le registre IP qui contient une adresse relative au segment de code (*Code Segment* ou CS).

- L'adresse de base est stockée dans CS.
- L'adresse absolue de l'instruction est donc CS:IP qui est égal à $CS \times 16 + IP$.

5.3.3 L'adressage des données dans DS

L'adressage des données dans le segment de données (Data Segment ou DS) peut s'effectuer de différentes manières (en utilisant différentes combinaisons) :

- DS : DEP
- DS : BX
- DS : SI
- DS : DI
- DS : BX+DEP
- DS : SI+DEP
- DS : DI+DEP
- DS : BX+SI
- DS : BX+DI
- DS : BX+SI+DEP
- DS : BX+DI+DEP

Avec, **DEP** est un nombre qui correspond à un décalage en mémoire, exprimé en octets

5.3.3.1 L'instruction MOV

D'un point de vue programmation, cet adressage des données peut être utilisé à l'aide de l'instruction **MOV**.

D'un point de vue syntaxique, on distingue ce type d'adressage de ceux évoqués précédemment grâce à l'utilisation des crochets [] qui entourent la seconde partie de l'adresse.

Par exemples :

- « **MOV DS:[BX], 10** » signifie qu'on stocker 10 dans la case dont l'adresse est DS:BX (soit $DS \times 16 + BX$).
- « **MOV AX, DS:[BX+DI+0010h]** » signifie qu'il faut copier le contenu de la case mémoire d'adresse DS:BX+DI+0010h dans le registre AX.

5.3.4 L'adressage des données dans SS et ES

On peut également adresser des données dans la pile (Stack Segment ou SS) ou dans le segment supplémentaire (Extra Segment ou ES). On a alors les combinaisons supplémentaires suivantes :

- ES : DEP
- ES : BX
- ES : SI
- ES : DI
- ES : BX+DEP
- ES : SI+DEP
- ES : DI+DEP
- ES : BX+SI
- ES : BX+DI
- ES : BX+SI+DEP
- ES : BX+DI+DEP
- SS : BP+DEP
- SS : BP+SI
- SS : BP+DI
- SS : BP+SI+DEP
- SS : BP+DI+DEP

Avec, **DEP** est un nombre qui correspond à un décalage en mémoire, exprimé en octets

5.3.4.1 L'instruction MOV - Remarque complémentaire

L'adressage des données dans ces segments s'effectue de la même manière que précédemment :

- « MOV ES:[BX+SI], DX » permet de stocker le contenu du registre DX dans la case mémoire d'adresse ES:[BX+SI].
- « MOV CX, SS:[BP+0002h] » permet de copier, dans le registre CX, le contenu dans la case mémoire d'adresse SS:[BP+0002h].

Si le segment concerné est DS, il n'est pas nécessaire de l'indiquer car il est sous-entendu :

- MOV [BX], 4 est équivalent à MOV DS:[BX], 4

5.3.5 Adressage en mode mémoire avec ADD, SUB et ADC

Jusqu'à présent, nous avons utilisé les instructions ADD, SUB et ADC en respectant l'une des deux syntaxes ci-dessous (qui correspondent respectivement au mode immédiat et au mode par registre) :

- ADD/SUB/ADC registre, immédiat
- ADD/SUB/ADC registre, registre

Ces instructions peuvent également s'appliquer en mode d'adressage en mémoire :

- ADD/SUB/ADC registre, mémoire
- ADD/SUB/ADC mémoire, registre
- ADD/SUB/ADC mémoire, immédiat

5.3.6 Adressage en mode mémoire avec MUL, IMUL, DIV et IDIV

Les instructions MUL, IMUL, DIV et IDIV que nous avons évoqués précédemment ne supportent pas le mode immédiat.

Ces instructions s'utilisent uniquement en mode par registre et en mode en mémoire. Nous avons donc les syntaxes suivantes :

- MUL/IMUL/DIV/IDIV registre
- MUL/IMUL/DIV/IDIV mémoire

5.3.7 Spécification de la taille des données

Il peut être utile de spécifier si le pointeur manipule un mot ou un octet (cela est particulièrement utile pour opération de division et de multiplication). Pour cela, nous devons ajouter une information supplémentaire devant le pointeur. Par conséquent, si le pointeur manipule un mot, nous le faisons précéder par « **WORD PTR** » ou « **w.** » :

- Par exemple : **MOV WORD PTR ES:[BX+DI+0010h], AX**

Dans le cas contraire, nous le faisons précéder par « **BYTE PTR** » ou « **b.** » :

- Par exemple : **DIV b.ES:[BX+DI+0010h]**

5.4 TD : Manipulation de base

5.4.1 Exercice1: Opérations logiques : AND, OR, NOT et XOR

Ecrire des programmes Assembleur pour réaliser les opérations suivantes:

1. 01100001b **AND** 11011111b
2. 01100001b **OR** 01110110b
3. **NOT** 01100001b
4. 01100001b **XOR** 01010101b

5.4.2 Exercice2: Opérations logiques : Décalage et Rotation

Soit 11010110b la valeur du registre AL. Trouver la nouvelle valeur de ce registre après les opérations suivantes:

1. Décalage à gauche avec un bit ;
2. Décalage à droite avec un bit ;
3. Décalage arithmétique à droite avec un bit ;
4. Décalage arithmétique à droite avec deux bits ;
5. ROL AL, 1 ;
6. RCL AL, 1 ; (on suppose CF=0)
7. RCR AL, 1 ; (on suppose CF=1)
8. ROR AL, 1 ;

5.4.3 Exercice3: Addition sur 32 bits (Utilisation du bit CF)

1. Construire un programme assembleur qui effectue les traitements suivants :
 - addition de 10F0C0BBh et 0A10DDFFh en utilisant le mécanisme décrit dans le cours.
 - retour vers le programme appelant (l'OS).
2. Modifier le programme précédent :
 - placer l'instruction STC avant le premier ADC et l'instruction CLC entre les deux ADC ;
 - observer l'état du bit CF lorsqu'on exécute ces nouvelles instructions.
3. Modifier une nouvelle fois le programme :
 - modifier le programme de manière à remplacer les instructions STC et CLC par CMC;
 - observer l'état du bit CF lorsqu'on exécute ces nouvelles instructions.

5.4.4 Exercice4: Multiplication et division

Ecrire des programmes assembleurs qui effectuent les traitements suivants :

1. la multiplication 15×20 sur 8 bits ;
2. la multiplication 15×-20 sur 8 bits ;
3. la multiplication 115×-10 sur 8 bits ;
4. la multiplication 115×-10 sur 16 bits ;
5. la division $93 / 3$ sur 8 bits ;
6. la division $520 / -10$ sur 8 bits ;
7. la division $520 / -10$ sur 16 bits.

- Que constate-t-on au niveau du registre d'état pour les calculs 3 et 6 par rapport aux autres calculs ?

5.4.5 Exercice5: Adressage direct & Adressage par registre

Construire un programme effectuant les tâches suivantes :

1. chargement du mot $0010h$ dans la case mémoire DS:BX en mettant BX au préalable à $0020h$ (on ne touche pas à DS) ;
2. chargement du nombre $0003h$ dans le registre AX ;
3. division du contenu du registre AX par celui de la case mémoire DS:BX en utilisant la fonction DIV ;
4. multiplication du contenu du registre AX par celui de la case mémoire DS:BX en utilisant la fonction MUL ;
5. recopie du résultat dans la case mémoire située après celle(s) qui stocke(nt) le nombre de départ (attention à ne pas l'écraser).

- Une fois le programme implanté, effectuer le travail suivant :

1. lancer le simulateur et indiquer dans un tableau quel est le contenu des registres et des éléments mémoire impliqués dans le programme ;
2. modifier le programme de manière à ce que $0010h$ soit considéré comme un octet. Relancer alors le simulateur et reconstruire le tableau ;
3. modifier le programme de manière à manipuler le nombre $-0010h$ sous forme d'octet. Relancer alors le simulateur et reconstruire le tableau ;
4. modifier le programme de manière à manipuler le nombre $-0010h$ sous forme de mot. Relancer alors le simulateur et reconstruire le tableau.

6 Les sauts

6.1 La notion d'étiquette

6.1.1 Le principe

L'étiquette (*label* en anglais) est utilisée par les mnémoniques de contrôle du flux d'instructions comme JMP, JO, JNZ ... Elle permet de spécifier au programme à quel endroit continuer l'exécution. Par la suite l'étiquette est **convertie en une adresse** au moment de l'assemblage des instructions de saut.

6.1.2 La déclaration d'un label

Pour ajouter un label dans le programme, il suffit de mettre un nom (qui ne commence pas par un chiffre) suivi de « : ».

Par exemple :

```
etiq1: MOV AX, 10
        ADD AX, 20
        INC AX
etiq2: MOV BX, 20
        SUB BX, 30
        DEC BX
```

6.2 Les sauts inconditionnels

6.2.1 Le mnémonique JMP

Le branchement inconditionnel consiste simplement à sauter d'une position dans le code à une autre position pour continuer l'exécution.

Ce type de branchement est opposé aux branchements conditionnels (étudiés plus loin) qui réalisent des sauts en fonction de test effectués sur les bits du registre d'état.

L'instruction utilisée pour réaliser le saut inconditionnel est JMP suivi d'un nom de l'étiquette (ou d'un nombre représentant l'adresse de destination mais ce n'est pas conseillé).

Par exemple :

```
ORG 100h
        MOV AX, 5 ; assigner 5 à AX.
        MOV BX, 2 ; assigner 2 à BX.
JMP calc ; aller à "calc".
back: JMP stop ; aller à "stop".
```

```

calc: ADD AX, BX ; ajouter BX à AX.

JMP back ; aller à "back".

stop: RET      ; redonner la main à l'OS.

END

```

6.2.1.1 Exercice d'application

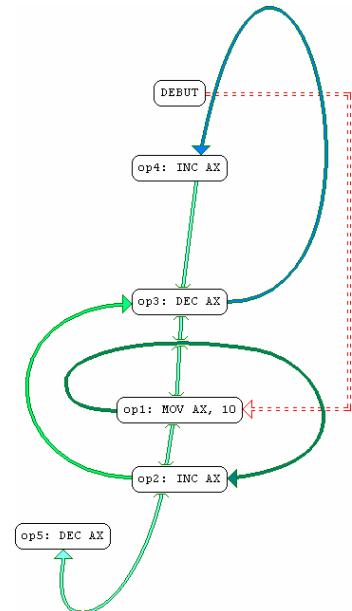
Etant donné le programme incomplet suivant :

```

ORG 100h
...
INC AX ; OP4
...
DEC AX ; OP3
...
MOV AX, 10 ; OP1
...
ADD AX, 6 ; OP2
...
DEC AX ; OP5
...
END

```

Ajouter les mnémoniques et les étiquettes de manière à obtenir un programme assembleur dont le fonctionnement corresponde à l'organigramme ci-contre.



6.3 La comparaison

6.3.1 Le mnémonique CMP

La comparaison de deux nombres s'effectue en faisant une pseudo-soustraction qui affecte les drapeaux ci-dessous dans le registre d'état :

- OF : Overflow Flag ;
- SF : Sign Flag (le bit de signe);
- ZF : Zero Flag ;(0 quand le résultat !=0)
- CF : Carry Flag ;(à 1 quand il y a une retenue).

Compte tenu de la « ressemblance » avec l'instruction SUB, les types des opérandes traitées sont les mêmes, on peut donc effectuer les comparaisons suivantes :

- CMP registre, immédiat
- CMP registre, registre
- CMP registre, mémoire
- CMP mémoire, registre
- CMP mémoire, immédiat

6.4 Les sauts conditionnels

6.4.1 Le principe

L'instruction CMP ainsi les instructions qui affectent les drapeaux du registre d'état sont souvent utilisées avec des instructions de branchement conditionnel afin d'implanter des boucles, des tests ...

A l'instar de l'instruction JMP, les instructions de branchement conditionnel s'utilisent avec un label. Elles doivent être placées juste après l'instruction (CMP, DEC, INC ...) qui modifient l'état du registre d'état sinon les autres mnémoniques placées entre cette instruction et le branchement conditionnel pourraient altérer les bits du registre d'état.

6.4.2 Flags (Rappel)

- OF: le bit d'overflow est positionné par la plupart des instructions arithmétiques pour indiquer s'il y a eu un débordement de capacité lors du calcul (un nombre trop grand ou trop petit)
- SF : pour indiquer le signe du résultat
- ZF : pour indiquer que le résultat du calcul est 0
- CF : le bit de carry (retenue), pour indiquer si le calcul a engendré une retenue qui devra être reportée sur les calculs suivants
- AF : le bit dit auxiliary carry (retenue auxiliaire), pour indiquer une retenue entre bits de poids faible et bits de poids forts d'un octet, d'un mot ou d'un double mot
- PF : le bit de parité. Il indique si les 8 bits de poids faible du résultat comportent un nombre pair de 1.

6.4.3 Les mnémoniques de sauts conditionnels:

6.4.3.1 Pour les valeurs non signées

Instruction	Conditions de saut	Indicateurs
JB / JNAE	Below / not above or equal	CF=1 & ZF=0
JAE / JNB	Above or equal / not below	CF=0 & ZF=1
JBE / JNA	Below or equal	CF=1 & ZF=1
JA / JNBE	Above / not below or equal	CF=0 & ZF=0

6.4.3.2 Pour les valeurs signées

Instruction	Conditions de saut	Indicateurs
JL / JNGE	Less than / not greater or equal	SF # OF
JGE / JNL	Greater or equal / not Less than	SF = OF
JLE / JNG	Less or equal / not greater than	SF # OF & ZF=1
JG / JNLE	greater than / not Less or equal	SF = OF & ZF=0

Par Example:

```
mov al,5
```

```
mov bl,-128
```

```
cmp al,bl
```

Cette comparaison consiste à faire une soustraction de 5 avec -128, donc l'opération génère la valeur de $5 - (-128) = 133 = (10000101)_2$. Donc :

- On a SF=1 car le bit le plus fort =1 et
- OF=1 car $133 > 127$ il s'agit d'un débordement de capacité.

```
mov al,-5
```

```
mov bl,-128
```

```
cmp al,bl
```

Cette comparaison consiste à faire une soustraction de -5 avec -128, donc l'opération génère la valeur de $-5 - (-128) = 123 = (01111011)_2$. Donc :

- On a SF=0 car le bit le plus fort =0 et
- OF=0 car $123 < 127$ il n'est pas de débordement de capacité.

6.4.4 Les mnémoniques de sauts conditionnels

Instruction	Conditions de saut	Indicateurs
JE / JZ	Equal / zero	ZF=1
JNE / JNZ	Not Equal / not zero	ZF=0
JP / JPE	Parity even (paire)	PF = 1
JNP / JNPO	Parity odd (impaire)	PF = 0
JS	Jump if Sign	SF = 1
JNS	Jump if Not Sign	SF = 0
JC	Jump if Carry	CF = 1
JNC	Jump if Not Carry	CF = 0
JO	Jump if Overflow	OF = 1
JNO	Jump if Not Overflow	OF = 0

6.4.5 La limitation dans l'amplitude des sauts

L'amplitude des sauts conditionnels est limitée à 127 octets vers l'avant et 128 octets vers l'arrière.

Pour contourner ce problème (nous affranchir de cette limite), nous pouvons coupler ce saut conditionnel avec un saut inconditionnel.

```

...
CMP AX, BX
JZ grandSaut
JMP loin1
grandSaut: JMP loin2
...
loin1: ...
loin2: ...

```

6.4.6 Exercice d'application

Traduire le code C ci-contre en assembleur avec :

```

bx=5;
while (bx>0)
{
    bx--;
}

```

6.5 Les boucles

6.5.1 Le registre CX comme compteur

Nous avons vu une première manière d'implanter des boucles en utilisant les mnémoniques de sauts conditionnels (JNZ, JMP ...) : cela se rapproche du « while » et du « do ... while ».

La seconde méthode (qui se rapproche plutôt du « for ») consiste à utiliser le registre CX comme un compteur par l'intermédiaire d'une des mnémoniques suivantes :

- LOOP ;
- LOOPE ;
- LOOPNE ;
- LOOPNZ ;
- LOOPZ ;
- JCXZ.

Instruction	Description
LOOP	Décrémente CX et saute vers le label si CX ≠ 0
LOOPE	Décrémente CX et saute vers le label si CX ≠ 0 et ZF = 1
LOOPNE	Décrémente CX et saute vers le label si CX ≠ 0 et ZF = 0
LOOPNZ	Décrémente CX et saute vers le label si CX ≠ 0 et ZF = 0
LOOPZ	Décrémente CX et saute vers le label si CX ≠ 0 et ZF = 1
JCXZ	Saute vers le label si CX = 0

Ces instructions, à l'exception de la dernière, décrémentent la valeur contenue dans le registre CX.

6.5.2 Syntaxe d'une boucle

La construction d'une boucle peut s'effectuer simplement en respectant quelques contraintes :

1. charger la valeur de la boucle dans le registre CX ;
2. faire précéder la première instruction de la boucle par une étiquette ;
3. terminer la boucle par une instruction de type LOOP qui pointe sur l'étiquette.

Nous obtenons alors une suite d'instructions qui peut ressembler à celle ci-dessous :

```

MOV CX, 5
boucle: INC AX
... etc
LOOP boucle ; Si CX != 0, on reboucle
...etc

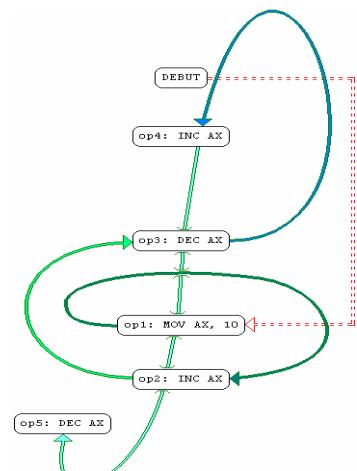
```


6.6 TD : Les Sauts et les Boucles

6.6.1 Exercice1: Les sauts inconditionnels

Etant donné le programme incomplet suivant :

```
ORG 100h
...
INC AX
...
DEC AX
...
MOV AX, 10
...
INC AX
...
DEC AX
...
RET
```



- Ajouter les mnémoniques et les étiquettes de manière à obtenir un programme assembleur dont le fonctionnement corresponde à l'organigramme ci-contre.

6.6.2 Exercice2: Les sauts conditionnels

- Traduire les codes C ci-contre en assembleur

1.

```
if (ax == 4h)
{
    bx = 10h;
}
else {
    bx = 20h;
    cx = 30h;
}
bx=bx+10
```

2.

```
bx=0;
for (cx=0; cx<10; cx++)
{
    bx += cx;
}
```

3.

```
Ax=0;
bx=10;

do
{
    ax++;
    bx--;
}
while (bx>0);
```

4.

```
switch (bx)
{
    case 1 : ax=1;
               break;
    case 2 : ax=5;
               break;
    case 3 : ax=9;
               break;
    default : ax=0;
}
```

5.

```
ax=0;
bx=5;
while (bx>0)
{
    ax++;
    bx--;
}
```

6.6.3 Exercice3: Les boucles

Refaire les questions 2, 3 et 4 de l'exercice 2 avec l'utilisation de la mnémonique LOOP.

7 La manipulation des variables

7.1 La notion de variable

7.1.1 La règle de nommage

Les variables sont considérées comme **des emplacements mémoire** qui peuvent être manipulés par l'intermédiaire de leur adresse ou de leur nom.

Le nom d'une variable peut être une combinaison de chiffres, de lettres et du caractère « _ » mais il doit respecter les contraintes suivantes :

- Ne jamais commencer par un chiffre ;
- Ne pas correspondre à un mot clé du langage Assembleur.

7.2 Les variables numériques

7.2.1 La création de la variable par DB et DW

L'association entre le nom et l'adresse s'effectue en utilisant les instructions **DB** (Define Byte) ou **DW** (Define Word) selon que la variable contient des octets ou des mots.

La valeur associée à la variable peut être un nombre (hexadécimal, décimal ou binaire) ou le symbole « ? » si la variable n'est pas initialisée.

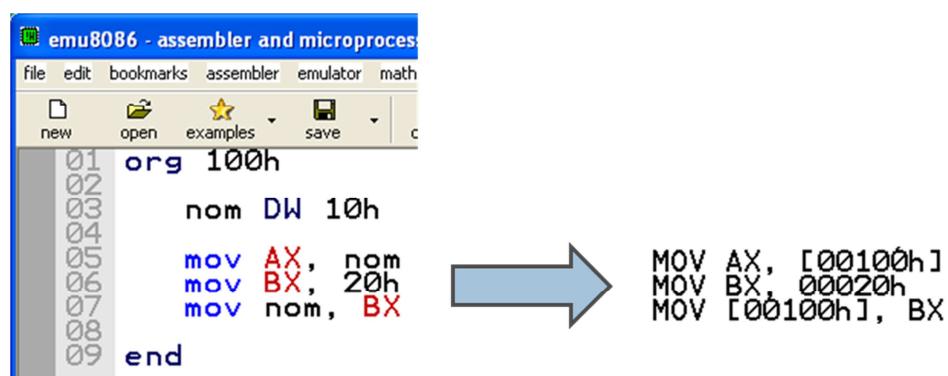
Par exemple :

- nom1 DB 10h
- nom2 DW 2030h
- nom3 DB ?
- nom4 DB ?

7.2.2 L'utilisation de la variable

Lorsque l'association entre le nom et la variable est créée, ce nom peut être utilisé pour désigner l'emplacement mémoire contenant l'information.

L'assembleur **traduit le nom en une adresse** lors du processus d'assemblage comme le montre l'exemple ci-dessous.



The screenshot shows the emu8086 assembler interface with the following assembly code:

```

01 org 100h
02
03 nom DW 10h
04
05 mov AX, nom
06 mov BX, 20h
07 mov nom, BX
08
09 end

```

An arrow points from the assembly code to the corresponding machine code:

```

MOV AX, [00100h]
MOV BX, 00020h
MOV [00100h], BX

```

Figure 25. Traduction du nom d'une variable en une adresse

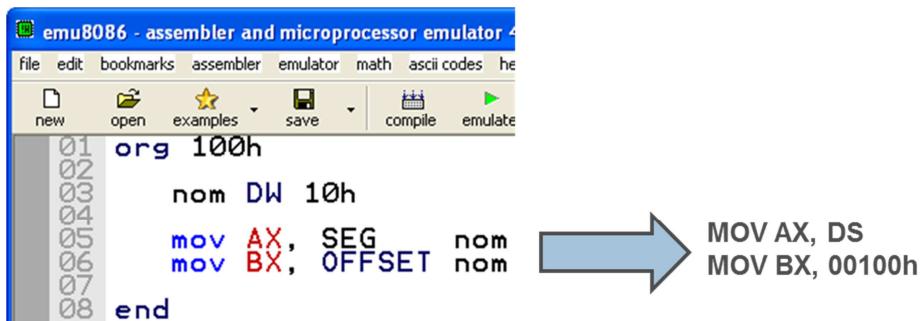
7.2.3 L'adresse d'une variable

Il peut être intéressant de connaître l'adresse d'une variable pour utiliser cette dernière comme argument des interruptions logicielles. Celle-ci se décompose en deux parties :

- L'adresse du segment, obtenue par l'opérateur **SEG** ;
- Le déplacement à l'intérieur du segment, accessible grâce à l'opérateur **OFFSET** ou l'instruction **LEA** (Load Effective Address).

Lors du processus d'assemblage, des transformations sont opérées au niveau du code :

- « SEG nom » est remplacé par le nom du registre associé au segment contenant la donnée ;
- « OFFSET nom » est remplacé par la valeur du déplacement.



```

emu8086 - assembler and microprocessor emulator
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate
01 org 100h
02
03 nom DW 10h
04
05 mov AX, SEG nom
06 mov BX, OFFSET nom
07
08 end

```

A blue arrow points from the assembly code to the resulting machine code:

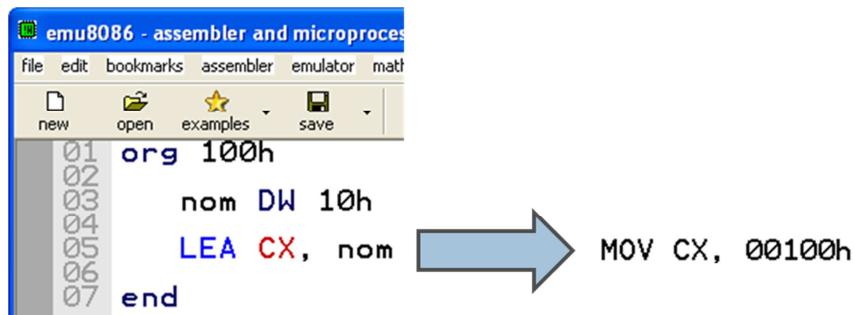
```

MOV AX, DS
MOV BX, 00100h

```

Figure 26. L'adresse d'une variable

L'instruction « **LEA CX, m** » est, quant à elle, traduite en « **MOV CX, 00100h** » (00100h étant la valeur du déplacement pour la variable m).



```

emu8086 - assembler and microprocessor emulator
file edit bookmarks assembler emulator math
new open examples save
01 org 100h
02
03 nom DW 10h
04
05 LEA CX, nom
06
07 end

```

A blue arrow points from the assembly code to the resulting machine code:

```

MOV CX, 00100h

```

Figure 27. L'instruction LEA

7.3 Les tableaux

7.3.1 La création d'un tableau et l'accès aux données

Un tableau peut être vu comme une liste de variables qui sont associées à un même nom. La création d'un tel tableau s'effectue alors de la manière suivante :

tableau DB 12h, 20h, 30h, 40h, 50h, 60h

Il est alors possible d'accéder à la première case du tableau grâce à la combinaison du SEG et OFFSET. Par exemple :

- **MOV CX, SEG tableau ; MOV CX, DS**
- **LEA BX, tableau ; MOV BX, 00100h**

■ MOV AL,[BX] ; MOV AL, 012h

Pour accéder aux autres cases du tableau en incrémentant la valeur du pointeur OFFSET par 1 (respectivement, par 2) s'il s'agit d'un tableau des octets (respectivement, des mots). Par exemple :

- LEA BX, tableau ; MOV BX, 00100h
- MOV DL,[BX+1] ; MOV DL, 020h
- MOV DH,[BX+2] ; MOV DH, 030h
-

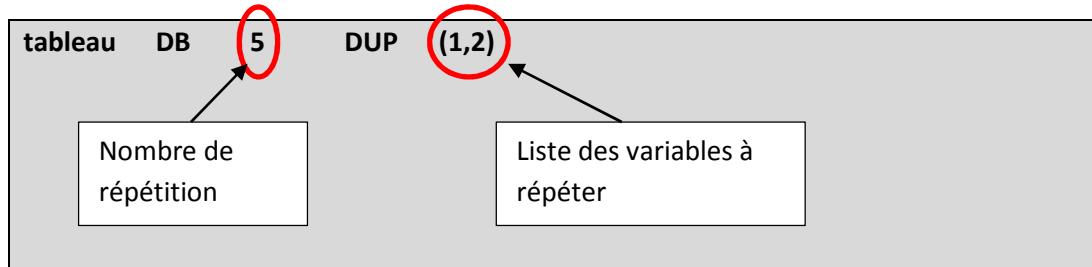
7.3.2 La création d'un tableau par répétition de valeurs

Si un tableau est construit en répétant une ou plusieurs valeurs, il peut être utile d'utiliser l'opérateur **DUP**.

L'opérateur DUP nécessite deux informations :

- Le nombre de répétition;
- La liste des variables à répéter, que nous plaçons entre parenthèses

La création d'un tel tableau s'effectue alors de la manière suivante :



7.4 Les chaînes de caractères

7.4.1 La création

Une chaîne de caractères peut être vue comme un tableau contenant une suite de code ASCII (correspondant chacun à une lettre) et se terminant par un « \$ » (pour marquer la fin de la chaîne de caractères).

Si on souhaite créer la chaîne de caractères « Bonjour », on peut alors écrire :

chaine DB 66, 111, 110, 106, 111, 117, 114, 36

Cette première écriture est fastidieuse car il faut connaître le code ASCII de chaque caractère. On peut demander à l'Assembleur de faire ce travail pour nous en plaçant les caractères entre guillemets (' '), on obtient alors :

chaine DB 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\$'

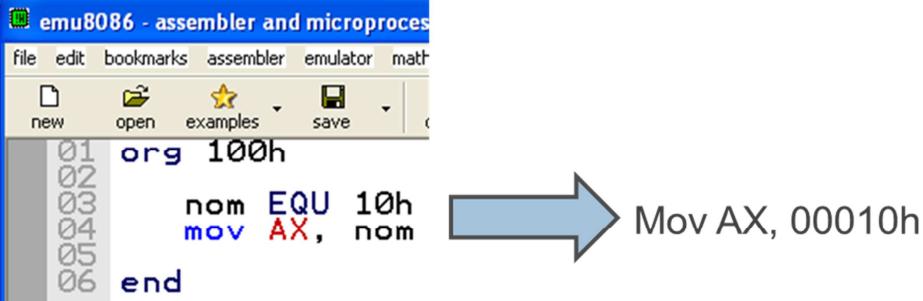
On peut utiliser une écriture encore plus rapide qui consiste simplement à placer la chaîne de caractères entre guillemets (sans oublier le caractère « \$ » à la fin de la chaîne de caractères). On obtient alors :

chaine DB 'Bonjour\$'

7.5 Les constantes numériques

7.5.1 La création et l'utilisation

Une constante numérique est définie grâce à la directive **EQU** qui indique à l'assembleur que le nom de cette constante doit être remplacée par sa valeur au moment de la création de l'exécutable.



The screenshot shows the emu8086 assembler interface. The assembly code is as follows:

```
01 org 100h
02 nom EQU 10h
03 mov AX, nom
04
05 end
06
```

A blue arrow points from the instruction `mov AX, nom` to the resulting assembly output: `Mov AX, 00010h`.

Figure 28. Création d'une constante avec EQU

7.6 TD : Les Variables

7.6.1 Exercice 1

Ecrire un programme Assembleur correspondant à l'algorithme suivant:

```
A=10: INT  
If (A>0) then  
    A=A+1  
Else  
    A=A+2  
End if
```

7.6.2 Exercice 2

Ecrire un programme Assembleur qui incrémente 10 fois un conteur initialisé à 0.

7.6.3 Exercice 3

Ecrire un programme Assembleur qui détermine la taille d'une chaîne **ch** initialisé à "Ceci est un test !\$" et ranger cette taille dans la case mémoire DS:BX en mettant BX au préalable à 0020h.

7.6.4 Exercice 4

Ecrire un programme Assembleur qui détermine la valeur la plus grande dans un tableau. Pour tester le programme utilisé le tableau suivant (12, 9, 50, 60, 59)

7.6.5 Exercice 5

Soit le tableau [34, 72, 48, 32, 56, 12, 8, 9, 45, 63, 80]. Ecrire un programme assembleur qui stocke dans le registre BX l'indice du premier élément du tableau ayant une valeur multiple de 9.

8 Les procédures

8.1 Introduction

Jusqu'à présent, nous avons construit des programmes assembleur qui ne comportent qu'une procédure principale.

Il peut être utile de placer dans des procédures distinctes des sections de code qui sont appelées plusieurs fois de manière à :

- Minimiser les recopies de code ;
- Diminuer la taille du code ;
- Augmenter la lisibilité du code.

8.2 Les directives PROC et ENDP.

8.2.1 Spécifier à l'assembleur la présence d'une procédure

A l'instar de sauts, la mise en place d'une procédure s'effectue en respectant quelques contraintes :

- Nous devons d'abord écrire son nom (une étiquette mais sans les deux points) suivi de la directive PROC pour signaler à l'assembleur qu'il s'agit d'une procédure ;
- Nous plaçons ensuite les instructions assembleur correspondant à l'implantation de la procédure (la dernière étant l'instruction RET) ;
- Nous terminons le code de la procédure par une ligne comportant le nom de la procédure suivi de la directive ENDP.

8.2.2 Le squelette d'une procédure

L'écriture d'une procédure peut donc se résumer par le squelette de code ci-dessous :

```

name PROC
...
RET
name ENDP

```

8.3 Les instructions CALL et RET.

L'appel d'une procédure et le retour de la procédure se fait grâce à deux instructions CALL et RET. (Voir la figure).

- Lorsqu'une procédure est implantée, elle peut être appelée par l'instruction **CALL** suivi du nom de la procédure.
- La procédure est déroulée jusqu'à l'instruction **RET** qui entraîne l'exécution de l'instruction qui suit **CALL**.

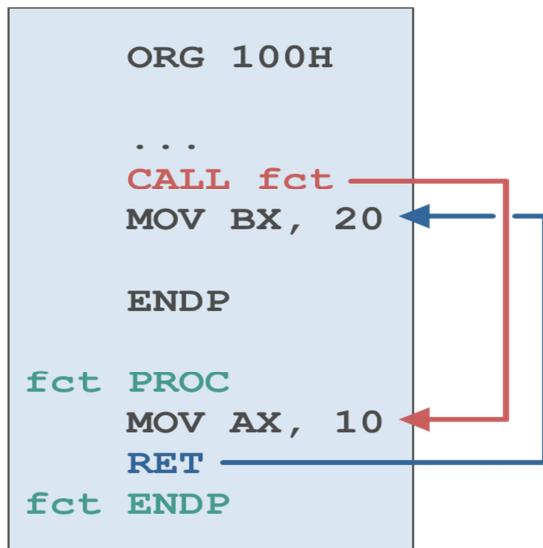


Figure 29. Les instructions CALL et RET

8.4 La sauvegarde temporaire des données.

8.4.1 Le contexte d'exécution

Lorsqu'on quitte une « procédure appelante » pour entrer dans une « procédure appelée », nous devons sauvegarder le **contexte d'exécution** de la « procédure appelante » afin de pouvoir reprendre son exécution lors du retour de la procédure appelée.

Ce contexte d'exécution de la « procédure appelante » est défini par **le contenu de l'ensemble des registres qu'elle utilise** :

- Les registres AX, BX ... ;
- Le registre d'état.

Si on ne conserve pas l'état de ces registres et si ces derniers sont modifiés par la « procédure appelée », le fonctionnement de la « procédure appelante » est corrompu.

Le moyen le plus couramment utilisé pour conserver ces données est la pile (le segment SS) que nous manipulons grâce aux instructions suivantes :

- **PUSH, PUSHA et PUSHF** ;
- **POP, POPA et POPF**.

8.4.2 Les types d'empilement

8.4.2.1 L'empilement d'une donnée de 16 bits

L'instruction **PUSH** permet de conserver une donnée à l'adresse SS:[SP] et de décrémenter SP de 2 (octets).

On peut réaliser 3 types de PUSH :

- Avec un nombre :
 - **PUSH 10h**
- Avec le contenu d'un registre :
 - **PUSH AX**
- Avec le contenu d'une case mémoire :

- **PUSH [BX]**

8.4.2.2 L'empilement d'un ensemble des registres

L'instruction **PUSHA** permet d'empiler le contenu des registres AX, CX, DX, BX, SP, BP, SI et DI.

Nous avons donc une équivalence entre l'instruction de droite et l'ensemble des instructions de gauche **pris dans cet ordre**.

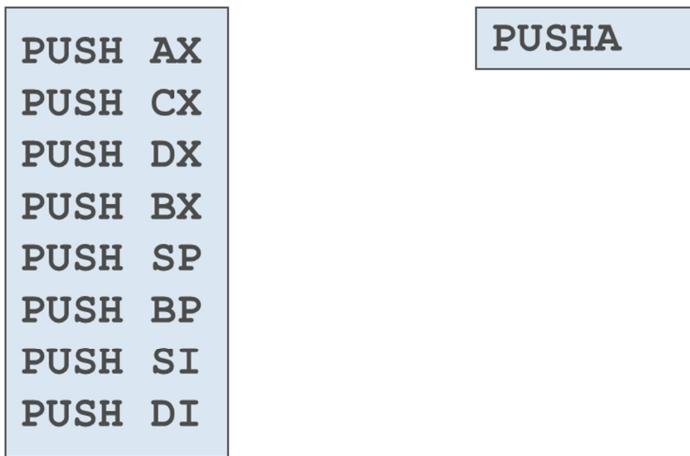


Figure 30. L'instruction PUSHA

8.4.2.3 L'empilement du contenu du registre d'état

L'instruction **PUSHF** permet d'empiler le registre d'état (FLAGS) à l'adresse SS:[SP].

Comme ce registre a une taille de 16 bits SP est décrémentée de 2 (octets) à l'issue de cette opération.



8.4.3 Les types de dépilement

8.4.3.1 Le dépilement d'une donnée de 16 bits

L'instruction **POP** permet de récupérer une donnée à l'adresse SS:[SP] et d'incrémenter SP de 2 (octets). On peut réaliser 2 types de POP :

- Pour stocker l'information récupérée dans un registre :

- **POP AX**

- Pour stocker l'information récupérée dans une case mémoire :

- **POP [BX]**

8.4.3.2 Le dépilement d'un ensemble des registres

L'instruction **POPA** effectue le traitement inverse de **PUSHA** : il récupère les informations stockées dans la pile pour les placer dans les registres AX, CX, DX, BX, SP, BP, SI et DI.

Comme précédemment, nous avons donc une équivalence entre l'instruction de droite et l'ensemble des instructions de gauche **pris dans cet ordre** (nous remarquons cette fois-ci que l'ordre est inversé).

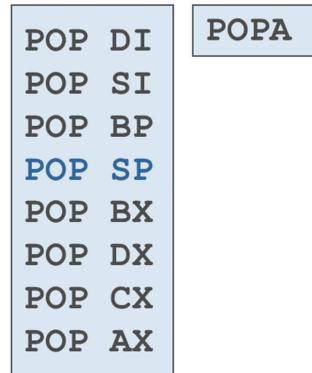


Figure 31. L'instruction POPA

L'instruction **POP SP** est ignorée pour ne pas perturber le fonctionnement de la pile.

8.4.3.3 Le dépilement du contenu du registre d'état

L'instruction **POPF** effectue le traitement inverse de **PUSHF**.

POPF permet donc de copier le contenu stocké à l'adresse SS:[SP] vers le registre d'état. Comme ce registre a une taille de 16 bits, SP est incrémentée de 2 (octets) à l'issue de cette opération.

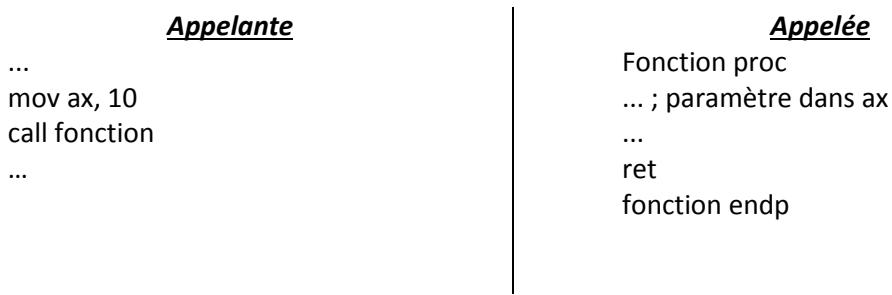


8.5 Passage de paramètres inter-procédures

8.5.1 Passage de paramètres par registre

Le mode de passage de paramètres par registre consiste à passer la valeur dans un registre. Le concept de ce mode est le suivant :

- La procédure appelante charge un registre avec la valeur à passer ; puis
- La procédure appelée récupère la valeur à partir de ce registre.



8.5.2 Passage de paramètres dans la pile

Il s'agit du mode de passage de paramètres que la plupart des langages de haut niveau passent leurs paramètres aux procédures et fonctions.

Le processeur utilise implicitement la pile pour y empiler l'adresse de retour des sous-programmes. Il doit impérativement trouver l'adresse de retour d'un sous-programme au sommet de la pile lorsqu'il exécute une instruction RET.

- Le processeur empile la valeur du registre IP quand il exécute une instruction CALL

Le sous-programme appelant est responsable du passage des paramètres, c'est-à-dire de leur empilage.

Pour le sous-programme appelé, la convention est d'utiliser le registre BP pour accéder à la valeur des paramètres. Donc le registre BP doit être sauvegardé par le sous-programme appelé.

La figure suivante illustre le concept de passage de paramètres en utilisant la pile.

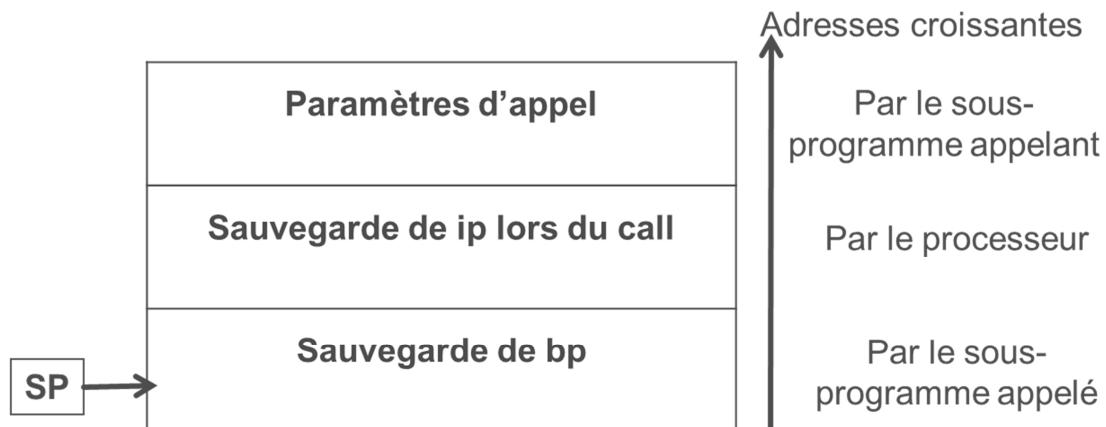
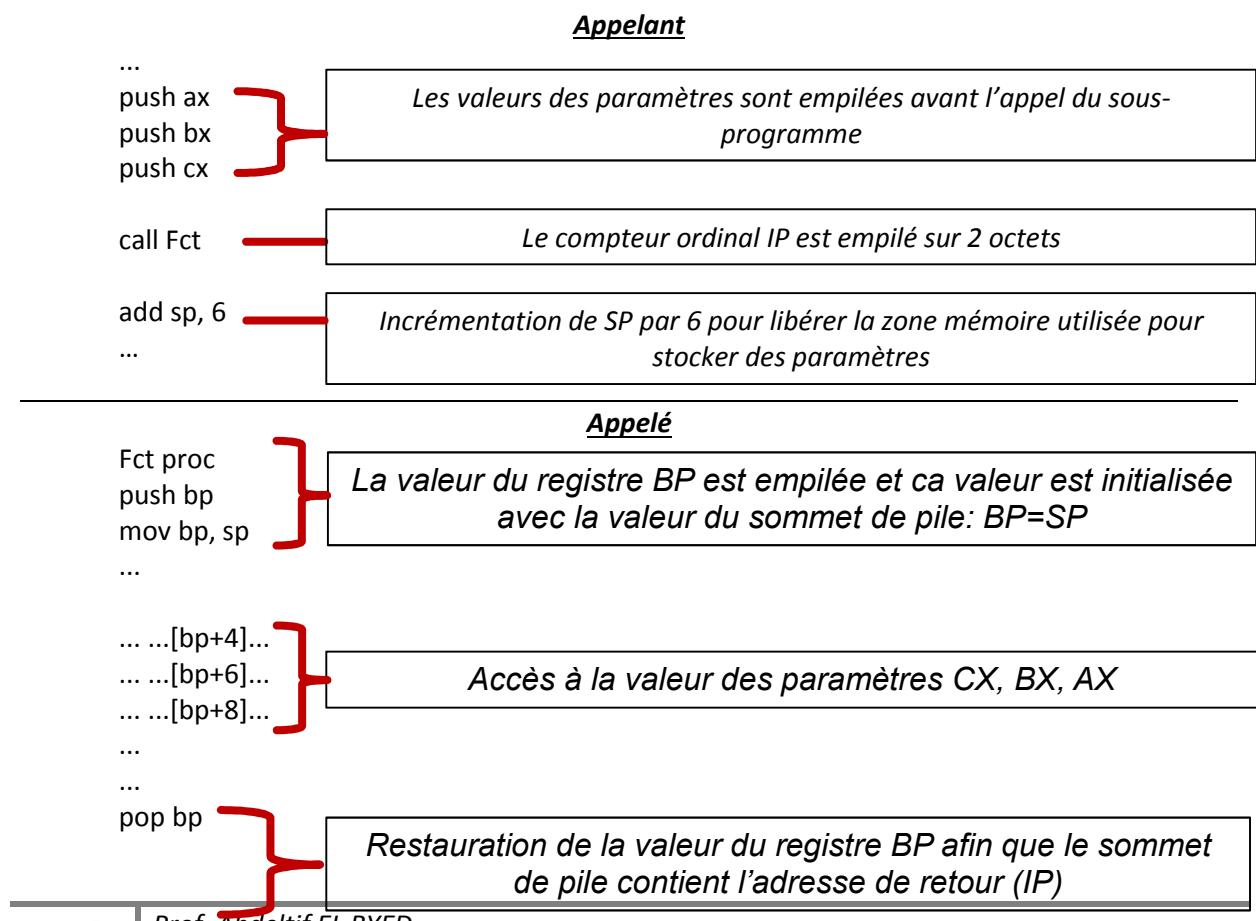


Figure 32. Passage de paramètres dans la pile

Ci-dessous un exemple d'un squelette d'un appel au sous-programme avec passage de paramètres. Dans l'exemple que nous cherchons à passer trois paramètres qui représentent les valeurs des registres AX, BX et CX.



ret

Fct endp

8.6 TD : Les procédures

8.6.1 Exercice 1 : Appel de sous-programme sans passage de paramètres

Ecrire un programme assembleur qui permet de calculer la fonction ci-dessous pour les nombres allant de 0 à 10.

$$y(x) = \sum_{i=0}^x i$$

Le programme doit comporter :

- Une procédure principale qui effectue une boucle de 0 à 10 et qui mets la valeur courante dans AH ;
- Une procédure de restitution du résultat dans la case mémoire DS :[300h]
- Une procédure *fct* qui :
 - Sauvegarde les registres ;
 - Appel à la procédure principale ;
 - Appel à la procédure de restitution du résultat
 - Restitue l'état initial des registres.

8.6.2 Exercice 2: Appel de sous-programme sans passage de paramètres

Ecrire un programme assembleur qui permet de calculer la fonction suivante:

$$F(X, n) = X^n$$

Le programme doit comporter :

- Une procédure fctMul qui effectue la multiplication de deux nombres A et B
- Une procédure principale qui fait l'appel n fois à la procédure fctMul.

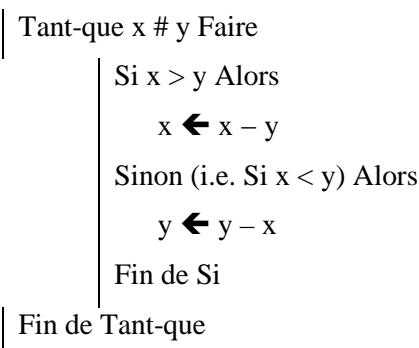
8.6.3 Exercice 3 : Appel de sous-programme avec passage de paramètres par registre

Ecrire un programme Assembleur calculant la factorielle d'un nombre dont la valeur est passé par le registre AX

$$F(X) = X!$$

8.6.4 Exercice 4 : Appel de sous-programme avec passage de paramètres dans la pile

Ecrire un programme Assembleur qui calcule le PGCD de deux nombres passés en paramètre. L'algorithme utilisé est le suivant : soit à calculer le PGCD de x et y :



9 Les interruptions

9.1 Le principe.

9.1.1 Le mécanisme des interruptions matérielles

Le processeur exécute un programme stocké dans le segment de code de la mémoire. Il doit cependant être capable de **réagir à des événements externes** en **interrompant le programme** en cours pour se brancher sur une **procédure de traitement** de l'événement.

Ce mécanisme nécessite l'adjonction d'un composant appelée « **contrôleur d'interruption** » sur lequel sont branchés les périphériques susceptibles de demander au processeur d'effectuer des traitements en réaction à leur fonctionnement (le clavier, la souris, le disque dur ...).

9.1.2 Le contrôleur d'interruption

Si nous rentrons dans les détails, le contrôleur d'interruption est souvent composé de deux circuits 8259A mis en cascade.

Chaque fil, appelé **IRQ** (*Interrupt ReQuest*) est donc raccordé à un périphérique (ou un ensemble de périphériques).

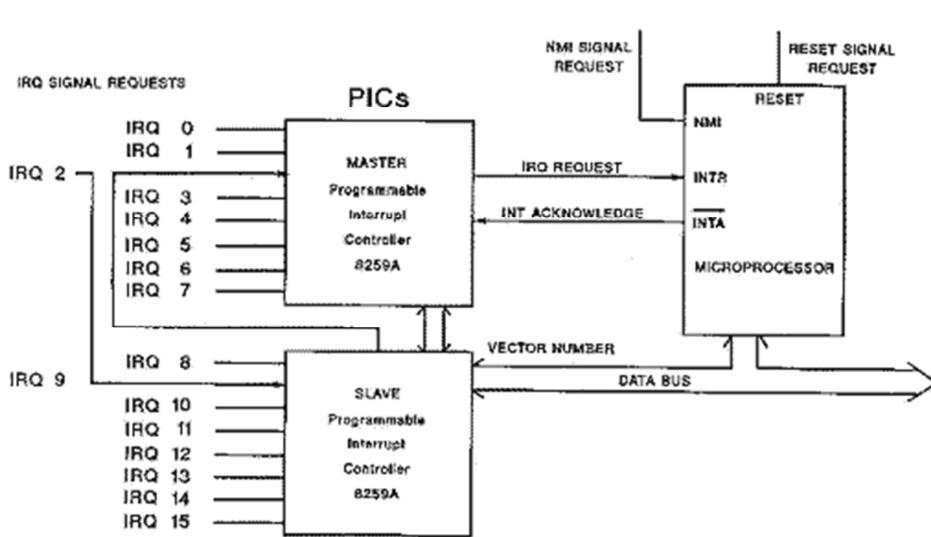
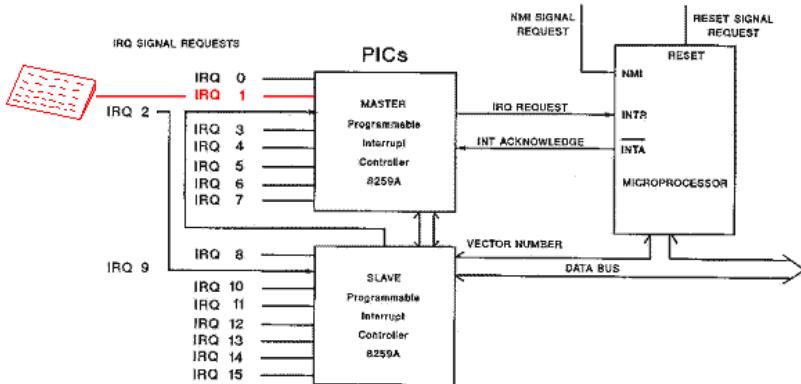


Figure 33. Circuits 8259A

9.1.3 Le cas du clavier

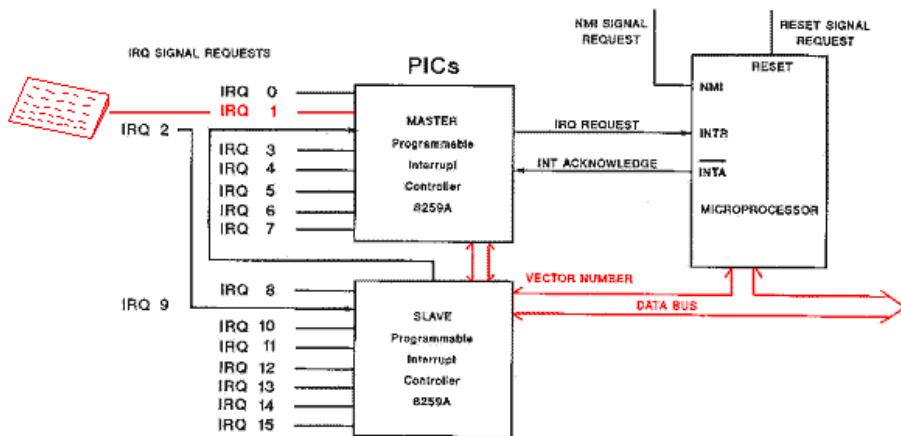
9.1.3.1 Le clavier sollicite le processeur

Si nous prenons l'exemple du clavier, celui-ci est raccordé à l'IRQ 1 : lorsque le clavier sollicite l'attention du processeur, celui-ci envoie un signal sur cette patte.



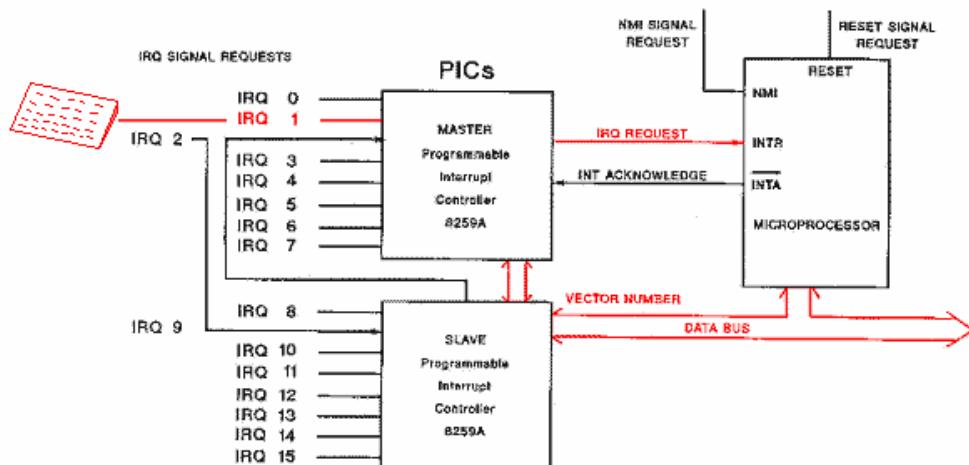
9.1.3.2 Le numéro de l'IRQ est sur le bus

Le numéro de l'interruption (le 1) est mis sur le bus de données pour être transmis vers le processeur.



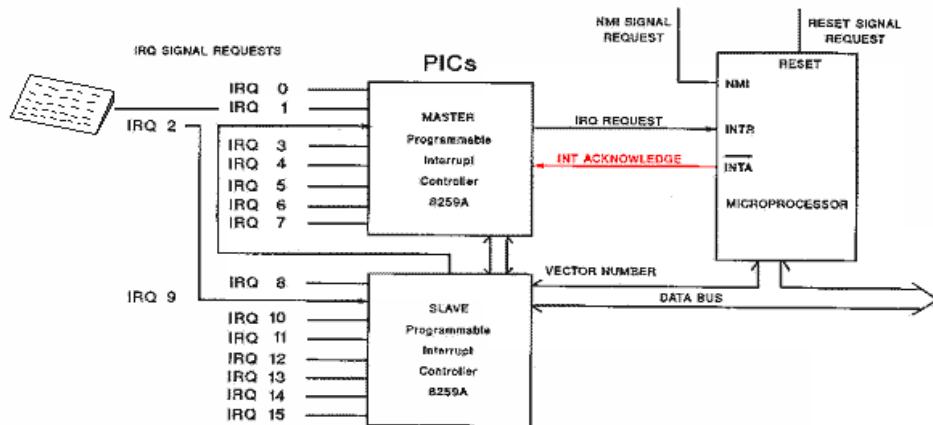
9.1.3.3 Le processeur est sollicité

Lorsque le bus de données est stable, le contrôleur d'interruption envoie un signal « IRQ Request » vers le processeur pour lui demander de tenir compte du clavier.



9.1.3.4 Pris en compte du signal

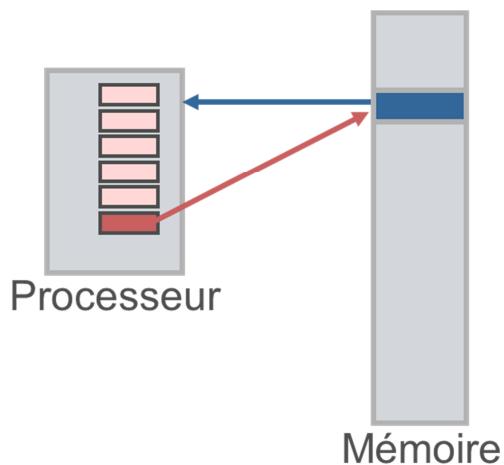
Le processeur indique par un « INT ACKNOWLEDGE » qu'il a prise en compte le signal d'interruption : le contrôleur d'interruption désactive alors ce signal pour pouvoir en reprendre en compte ultérieurement.



9.1.4 Le traitement de l'IRQ

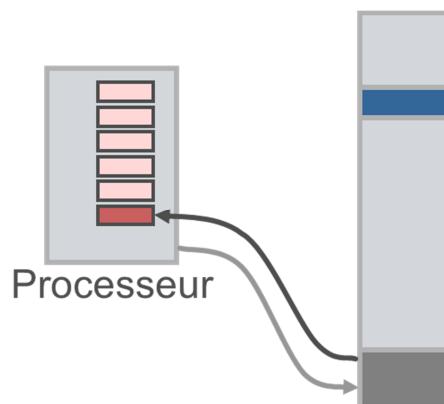
9.1.4.1 Le contexte d'exécution

Avant le traitement de l'IRQ, le processeur exécutait un programme : il existe donc un contexte d'exécution correspondant à l'état des différents registres au moment de cette interruption (en particulier le compteur ordinal qui pointe la prochaine instruction à exécuter).



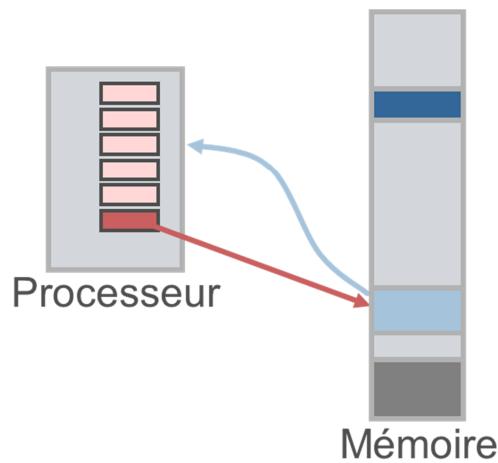
9.1.4.2 Lecture de la TVI

Le processeur lit d'abord une zone mémoire appelée « la table des vecteurs d'interruption » afin de connaître l'adresse de la procédure de traitement en fonction du numéro de l'interruption.



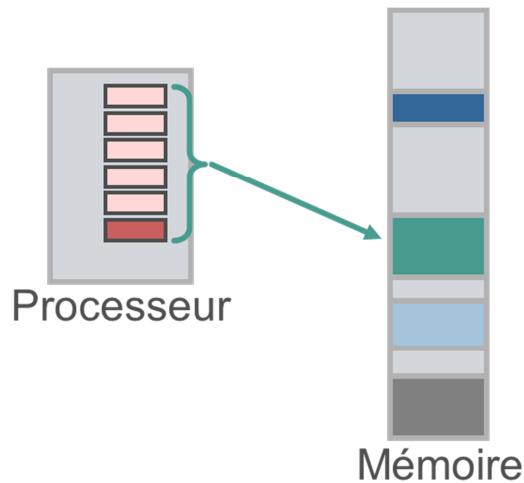
9.1.4.3 Exécution de la procédure

Le processeur exécute alors la première instruction de la procédure.



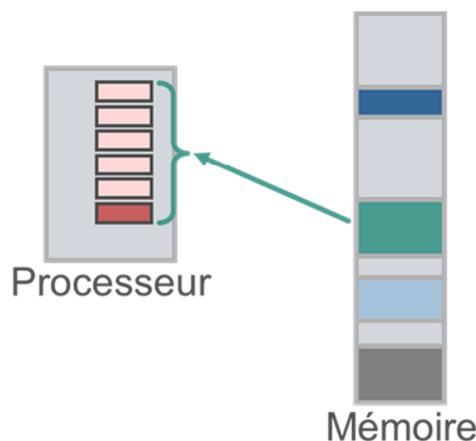
9.1.4.4 La sauvegarde du contexte

Les premières instructions consistent généralement à sauvegarder le contexte dans la pile avant d'exécuter la procédure de traitement de l'interruption proprement dite.



9.1.4.5 Retour à l'exécution normale

Lorsque le traitement proprement dit est terminé, on recharge le contexte d'exécution depuis la pile (le compteur ordinal pointe sur l'instruction du programme principal dont l'exécution pourra reprendre) et on exécute l'instruction IRET (*Interrupt RETurn*).



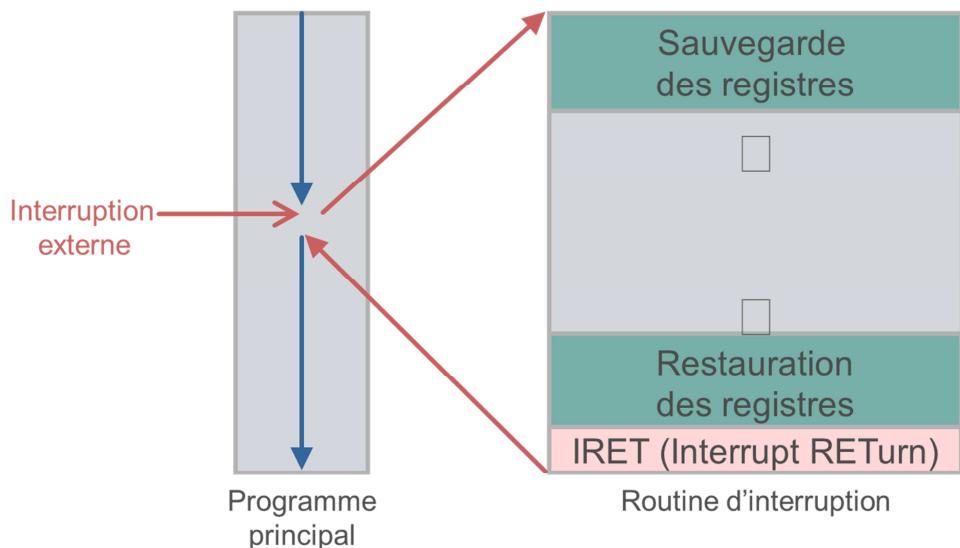
9.1.5 Synthèse

Le principe de fonctionnement des interruptions est donc :

1. Stopper le programme principal ;
2. Lire la table des vecteurs d'interruption pour connaître l'adresse de la procédure chargée de traiter l'interruption ;
3. Sauvegarder le contexte d'exécution ;
4. Exécuter la procédure ;
5. Recharger le contexte d'exécution afin de reprendre l'exécution du programme principal.

9.2 Les types des interruptions

9.2.1 Les interruptions matérielles



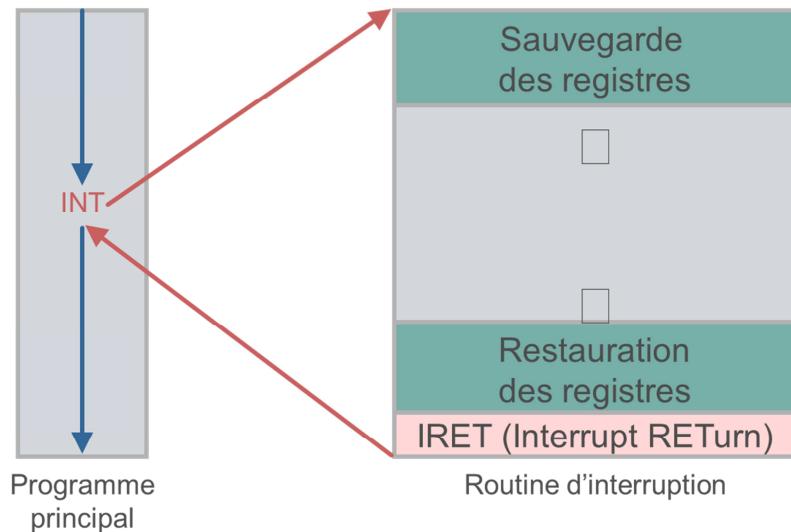
9.2.2 Les interruptions logicielles

Ce principe appliqué au traitement de signaux électroniques externes a été étendu au traitement des **signaux logiciels internes**.

Le programmeur a en effet la possibilité d'invoquer l'exécution de procédures toute faites (on parle d'interruption par abus de langage) en utilisant l'instruction **INT** suivie du numéro de l'interruption à exécuter. Par exemple :

INT 21h

Le processeur interrompt alors le programme principal et se branche sur la procédure de traitement selon le principe évoqué précédemment.



Les interruptions logicielles sont en réalité des procédures toutes faites, mises à la disposition des programmeurs pour faciliter le contrôle des organes de l'ordinateur (disque dur, clavier ...). On distingue entre deux types d'interruptions logicielles:

1. **les interruptions du BIOS** (Basic Input Output System) qui sont implantées par le constructeur de la carte mère ;
2. **les interruptions système** (DOS, Windows, Linux ...) qui sont chargées en mémoire lors du chargement du système d'exploitation.

9.2.3 Taxinomie des interruptions

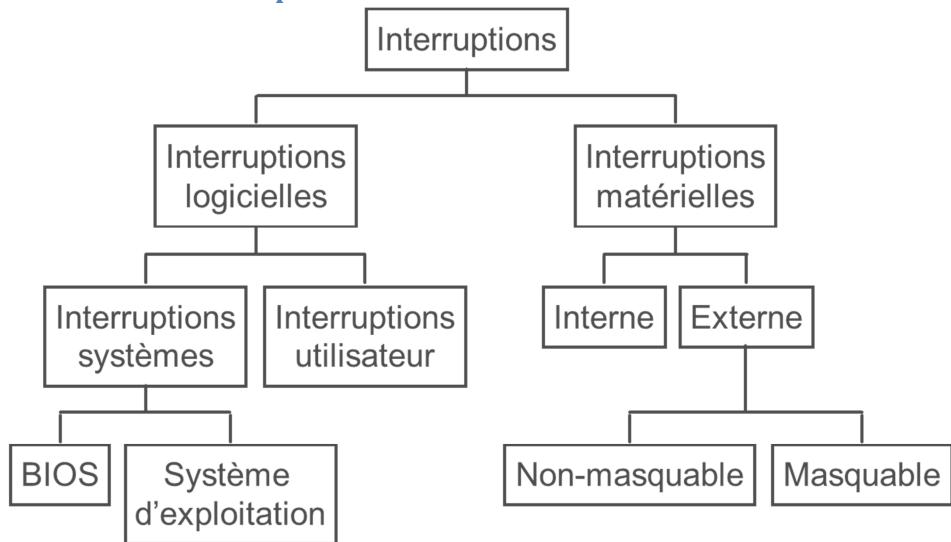


Figure 34. Taxinomie des interruptions

9.2.4 Les interruptions matérielles externes

Les interruptions matérielles externes sont les interruptions que nous avons décrites au début et qui sont issues du contrôleur d'interruption.

Elles permettent au processeur de réagir aux actions des périphériques externes comme le clavier, la souris ...

Certaines de ces interruptions (par exemple, le clavier) peuvent être masquées (le processeur peut ne pas en tenir compte) alors que d'autres (défaut de circuit RAM) ne peuvent l'être.

9.2.5 Le masquage des interruptions

Le masquage des interruptions peut s'effectuer de manière logicielle grâce à l'instruction **CLI** (*CLear Interrupt*) qui a pour effet de mettre à 0 le bit IF du registre d'état.

L'instruction **STI** (*SeT Interruption*) permet de rendre le processeur sensible aux interruptions masquables. Ce mécanisme est utilisé lorsque le processeur doit effectuer un traitement critique.

Par exemple: la commutation entre deux processus dans un système d'exploitation multitâches, des informations critiques doivent être sauvegardées et le processeur ne doit en aucun cas être dérangé pendant cette phase.

9.3 Gestion de l'écran

9.3.1 Le principe

L'écran peut être contrôlé par l'interruption **n°10h (BIOS)** ou l'interruption **n°21h (DOS)**. Nous allons décrire la seconde qui est émulé par EMU8086.

L'interruption 21h cache en fait un ensemble de fonction : le numéro de la fonction utilisée et les « paramètres » de la fonction doivent être placés dans certains registres avant l'instruction **INT 21h**.

9.3.2 L'affichage d'un caractère

Si on souhaite afficher un caractère à l'écran, nous utilisons la fonction **6h**. Par conséquent, nous devons effectuer les traitements ci-dessous avant d'appeler l'interruption 21h :

- Placer la valeur 6h dans le registre AH et
- Placer le code ASCII du caractère à afficher dans le registre DL.

Par exemple l'affichage du caractère 'A' ce fait :

```
MOV AH, 6h  
MOV DL, 'A'  
INT 21h
```

La **routine** de traitement lit d'abord le contenu de AH pour déterminer qu'il s'agit de la fonction 6h puis elle lit le registre DL pour déterminer le caractère qui doit être affiché à la position courante du curseur sur l'écran.

9.3.3 L'affichage d'une chaîne de caractères

Pour l'affichage **d'une chaîne de caractères**, il faut utiliser la fonction **9h**. Pour cela, il faut effectuer les affectations ci-dessous avant d'appeler l'interruption 21h :

- Placer la valeur 9h dans le registre AH
- Placer l'adresse de base du segment contenant la chaîne de caractères dans DS (inutile dans le cas d'un fichier .com)
- Placer l'offset de cette même chaîne de caractères dans DX.

La chaîne de caractères est alors affichée à la position courante du curseur sur l'écran. Par la suite un exemple du code assembleur pour afficher la chaîne de caractères 'Hello World'.

```

ORG 100h
JMP START
msg DB "Hello world $"
START:
MOV AH, 9h
MOV DS, SEG msg ; optionnel pour les fichiers .com
MOV DX, OFFSET msg
INT 21h
RET

```

9.4 Gestion du clavier

9.4.1 Le démarrage

Le clavier peut être contrôlé par l'interruption **n°16h** du BIOS ou l'interruption **n°21h** du DOS. Nous allons décrire la première qui est émulée par EMU8086.

L'interruption **16h** est également composée de plusieurs fonctions mais deux nous intéressent plus particulièrement :

- **La fonction 00h** qui permet de récupérer le code de la touche qui a été frappée au clavier;
- **La fonction 01h** qui permet de vérifier si une touche a été frappée au clavier.

9.4.2 La lecture du clavier

Pour lire la touche qui a été frappée sur le clavier, il faut utiliser la fonction 00h. Pour cela :

- On écrit la valeur 00h dans AH puis
- On exécute l'instruction « INT 16h ».

Lors de son exécution, cette routine **retire du buffer** du clavier, le code de la touche qui a été tapé, pour le mettre à notre disposition avec:

- le registre AL contient le code ASCII de la touche
- le registre AH contient le code « clavier » de la touche

9.4.3 Le test du clavier

Pour tester si une touche a été frappée sur le clavier, il faut utiliser la fonction 01h. Comme précédemment,

- On écrit la valeur 01h dans AH puis
- On exécute l'instruction « INT 16h ».

Si un caractère est présent dans le buffer, cette routine met le drapeau ZF à 0 (sinon il est à 1) puis elle **copie depuis le buffer** du clavier, le code de la touche qui a été tapé, pour le mettre à notre disposition :

- le registre AL contient le code ASCII de la touche ;
- le registre AH contient le code « clavier » de la touche.

9.4.3.1 Exercice d'application

Ecrire un petit programme qui :

- Affiche à l'écran, les caractères qui ont été tapés au clavier ;
- S'arrête lorsque le code ASCII de la lettre 'Q' a été détecté.

Quel est l'inconvénient de ce type de programme (pourquoi peut-on parler d'attente active)?

9.4.3.2 Solution de l'exercice d'application

```
org 100h

BCL:
    MOV AH,0
    INT 16H
    MOV AH,6
    MOV DL,AL
    INT 21H
    CMP DL,'Q'
    JNE BCL
    ret
```

9.5 Le déroutage des interruptions

9.5.1 L'attente active

Le petit programme précédent est une boucle qui teste de façon incessante si une touche a été frappée sur le clavier :

- Si une touche est effectivement frappée, on exécute la procédure correspondante
- Si ce n'est pas le cas, on retourne au début de la boucle et on recommence le test.

Ce type de programme fait de **l'attente active** : il teste de façon répétée si un événement est survenu pour déclencher une action au lieu d'être activée par l'événement lui-même.

9.5.2 Le déroutement des interruptions

Il est généralement conseillé d'éviter du programme avec une attente active, car elle conduit à gaspiller du temps CPU et par voie de conséquence à dégrader les performances d'un programme Assembleur.

Il est donc préférable d'adopter une technique de **programmation événementielle** où le programme réagit à des événements externes. Pour réaliser ce type de programmation en assembleur, il est nécessaire de **dérouter les interruptions**.

Le principe de ce déroutement s'effectue simplement en modifiant l'adresse contenue dans la table des vecteurs d'interruption.

Si nous prenons l'exemple du clavier, celui-ci est d'abord géré par l'interruption **9h** qui est appelée par le contrôleur d'interruption. Le traitement par défaut consiste à copier les codes concernant la touche qui a été frappée dans un buffer ;

En effet, il est possible d'implanter notre propre procédure de traitement puis de modifier l'adresse de la procédure associée à l'interruption 9h de manière à ce que notre procédure soit appelée lorsqu'une touche est saisie.

Cette technique était utilisée dans les anciens jeux vidéos pour gagner en fluidité dans le contrôle du jeu.

9.6 TD : Les interruptions

9.6.1 Exercice 1 :

Ecrire un petit programme qui :

- Affiche à l'écran, les caractères qui ont été tapés au clavier ;
- S'arrête lorsque le code ASCII de la lettre 'Q' a été détecté.

Quel est l'inconvénient de ce type de programme (pourquoi peut-on parler d'attente active)?

9.6.2 Exercice 2 :

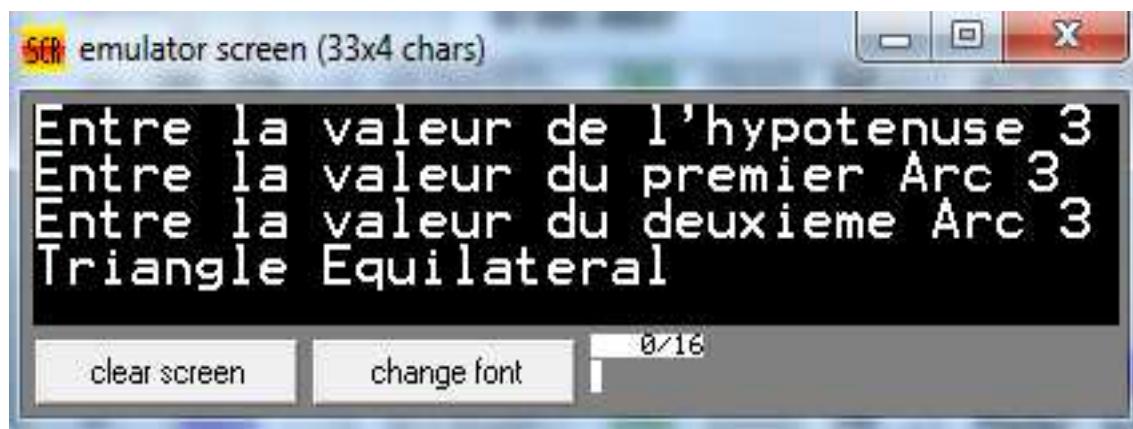
- Ecrire un programme Assembleur qui demande à l'utilisateur d'entrer la valeur de deux variables X et Y, en affichant les messages de la figure ci-dessous.



9.6.3 Exercice 3 :

Ecrire un programme Assembleur qui demande à l'utilisateur de saisir les valeurs des trois côtés d'un triangle (hypoténuse en premier). Le programme affiche à l'utilisateur, suivant le type du triangle, l'un des messages suivants :

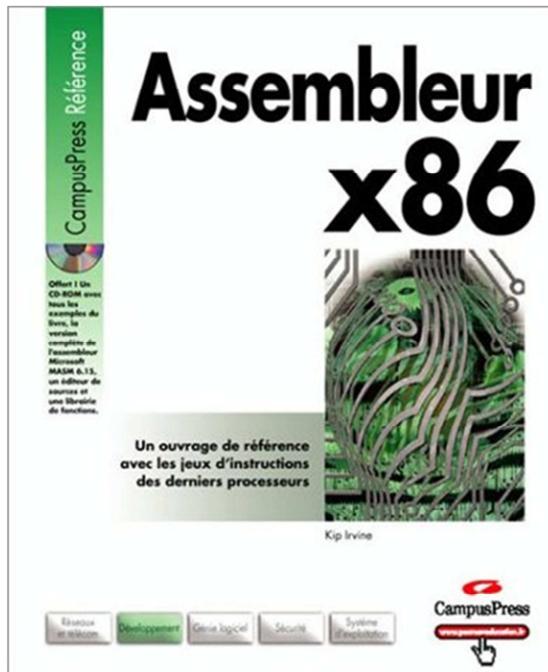
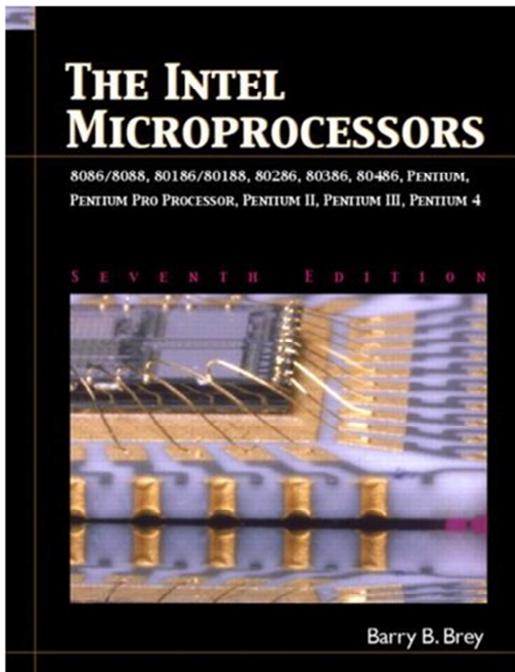
- « Triangle Rectangle », « Isocèle », « Isocèle rectangle », « Equilatéral » ou « Simple Triangle »
- La figure représente le résultat final de l'exécution du programme
- Hypothèse : la valeur des trois arcs est entre 0 et 9



10 Références

10.1 Publications

1. <http://www.amazon.fr/exec/obidos/ASIN/0131195069>
2. <http://www.amazon.fr/exec/obidos/ASIN/2744015342>



10.2 Sites web

1. <http://www.asmfr.com/>
2. <http://asm.developpez.com/>
3. <http://castevinz.free.fr/tutorial/>
4. <http://znssoft.free.fr/Developpement/Asm/>
5. <http://pdf1.alldatasheet.com/datasheet-pdf/view/130012/INTEL/8086.html>
6. <http://viers.free.fr/asm.pdf>

10.3 Ma Page web

L'ensemble des supports et des évènements sont publiés en ad hoc sur ma page web à l'adresse suivante:

- <https://sites.google.com/site/aelbyed/enseignement>



Figure 35. Capture de ma page web