ARCHITECTURE ET PROGRAMMATION DES MECANISMES DE BASE D'UN SYSTÈME INFORMATIQUE

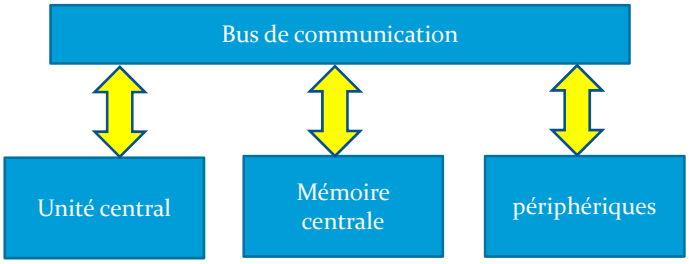
Proramme

- →L'Unité Centrale du microprocesseur
- →La Mémoire Centrale du microprocesseur
- →Les Périphériques et interfaces associés au microprocesseur
- →Le logiciel de base : assembleur

Architecture de Von Neuman

Un ordinateur comporte:

- **une unité centrale (UC)**
- **des mémoires (contenant données et programmes)**
- **des périphériques**
- **bus de communication**

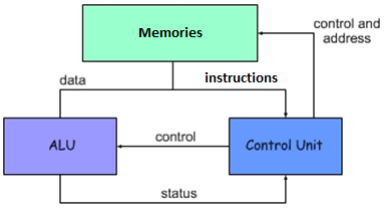


Architecture de Von Neuman

❖**Unité central** (ou microprocesseur) est composée de deux unités :

→**l'UC** (Control Unit) : lire et décoder les instructions (front end)

→**l'UAL** (Arithmetic and Logic Unit) : réaliser les calculs (+,-,/,*,AND,OR,NOT)



❖A fin de pouvoir stocker les opérandes et les résultats des calculs en sortie de l'UAL, le microprocesseur est doté de **registres**.

Un registre est un espace mémoire à l'intérieur du processeur → stocker des données temporaires qui sont généralement les opérandes ou le résultat d'un calcul de l'UAL.

Par exemple sur le 8086 d'Intel on trouve :

- 8 registres principaux (généraux) : AX, BX, CX, DX, BP, SP, SI, DI
- 4 registres de segment : CS, DS, ES, SS
- 1 registre (compteur de programme) : IP
- 1 registre d'état : FLAGS

Mémoire centrale

→**Stockage des** données **et des** programmes
→**codés par des suites de 0 et de 1**
→**Chaque** cellule **mémoire est désignée par son** adresse
→**Toutes les cellules ont la même taille (**mot**), exprimée en nombre de** bits **ou d'**octets**.**

Opérations sur une cellule:

lecture du contenu ***Adresse*** ⟶ ***Valeur***
écriture d'une ***Valeur*** **à une** ***Adresse***

Mémoire centrale

➢la lecture et l'écriture d'informations

➢RAM (Random Access Memory) qui fournit rapidement les données au processeur et qui perte toutes les données mémorisées dès que l'ordinateur cesse d'être alimenté en électricité ⇒ mémoires vives (dans laquelle le microprocesseur place les données lors de leur traitement)

- Les *RAM statiques* → l'information mémorisée est garantie aussi longtemps que l'alimentation électrique est maintenue sur la mémoire

- Les *RAM dynamiques* → l'information mémorisée diminue avec le temps
  ⇒ les rafraîchir une fois toutes les quelques millisecondes → disposer d'un dispositif interne auto rafraîchissement: *les mémoires quasi-statiques*.

→Static Random Access Memory SRAM, Dual Ported Random Access Memory DPRAM, Magnetic Random Access Memory MRAM, Phase-Charge Random Access Memory PRAM

➢ROM (Read-Only Memory) désignait une mémoire informatique non volatile (c'est-à-dire une mémoire qui ne s'efface pas lorsque l'appareil qui la contient n'est plus alimenté en électricité) ⇒ mémoire morte (dont le contenu était fixé lors de sa programmation, qui pouvait être lue plusieurs fois par l'utilisateur, mais ne pouvait plus être modifiée).

Mémoire centrale

❖**Les unités centrales réagissent beaucoup plus rapides que les mémoires principales. Ainsi, lorsque l'unité centrale sollicite la mémoire, elle passe une bonne partie de son temps à attendre que la mémoire réagisse.**

❖**Le microprocesseur doit donc "attendre" la mémoire vive à chaque accès, on dit que l'on insère des "Wait State" dans le cycle d'horloge d'un micro.**

Ex*: *un 386 DX cadencé à 33 MHz ne peut fonctionner sans état d'attente que si les RAM ont un temps d'accès de 40 ns.

❖**le problème** **technologique : NON**
économique : Oui

*Il est possible de construire des mémoires aussi rapides que les unités centrales mais leur coût, pour des capacités de plusieurs mégaoctets serait prohibitif***.**

❖**Les choix :** *à l'exception des superordinateurs (performance>coût)*
→ **disposer d'une faible quantité de mémoire rapide associée à une quantité importante de mémoire relativement plus lente. Cette mémoire plus rapide est appelée mémoire cache, cache ou antémémoire.**

Mémoire centrale

❖**mémoire cache → mémoire vive statique**
15 ns à 20 ns de temps d'accès
s'insère entre le processeur et la RAM dynamique.

❖**Un contrôleur de mémoire cache est chargé de recopier les instructions et les données les plus fréquemment utilisées par le processeur dans le cache.**

❖**Le principe du cache repose sur deux constatations:**
- en cours d'exécution d'un programme, lorsque le microprocesseur va chercher une instruction en mémoire il y a statistiquement de fortes chances pour que celle-ci se trouve à proximité de l'instruction précédente.
- de plus, les programmes contiennent un grand nombre de structures répétitives de sorte qu'ils utilisent souvent les mêmes adresses.

❖**Gestion de la mémoire cache**
le contrôleur du cache intercepte les adresses émises par le microprocesseur et dans un premier temps recopie le contenu d'un bloc entier de mémoire dans le cache de sorte qu'il y ait une forte probabilité que la mémoire cache contienne les prochaines instructions.
Ce principe permet au microprocesseur, via le contrôleur de cache, d'avoir 90% de chance d'obtenir l'information dans la mémoire cache donc sans "Wait State".

Mémoire centrale

Principe de fonctionnement

❖Le dispositif est constitué de deux éléments principaux:

-la mémoire cache, constituée de RAM
-le contrôleur de mémoire cache comprenant
- **•la gestion du cache**
- **•un index d'adresses stockant les adresses des blocs contenus dans le cache**
- **•un comparateur qui met en correspondance les adresses émises par le microprocesseur et celles contenues dans l'index quand il y a correspondance**

❖l'information est prise dans le cache sinon elle est transférée de la RAM et le contrôleur recopie tout un bloc dans le cache.

Mémoire centrale

Différentes étapes du fonctionnement d'un cache:

1. Le microprocesseur demande une information (instruction ou donnée) I1 située à l'adresse A1 de la mémoire vive
2. Le contrôleur de mémoire cache intercepte la demande et examine sa table d'index pour vérifier si A1 y est présente, et donc si une copie de l'information se trouve dans le cache.
3. Si c'est le cas, l'information est délivrée au microprocesseur depuis la mémoire cache sans temps d'attente
4. Dans le cas contraire, le contrôleur de cache accède à l'information de A1 de la RAM, la délivre à l'unité centrale avec plusieurs temps d'attente et simultanément la recopie en mémoire cache en même temps qu'un bloc d'informations contiguës, parmi lesquelles I2, I3, I4 etc... et actualise sa table d'index.
5. Le microprocesseur réclame l'information suivante, il y a statistiquement 90% de chance pour que ce soit I2, donc présente dans le cache et délivrée dans l'UC sans temps d'attente.

Mémoire centrale

Exemple: le cache du 486 (cache interne, cache externe)

Avec la génération 80486 et 68040 le contrôleur et la mémoire cache sont intégrés dans le processeur, on parle de cache premier niveau. Un second cache, de deuxième niveau, peut lui être associé en externe.

Le 486 possède un cache interne de 8 ko fonctionnant à la même vitesse que le processeur, ce qui n'est pas le cas par rapport aux caches externes.

Power PC 620: (64 ko cache interne) peut gérer 1 Mo de cache externe

Périphériques

Les périphériques

Claviers, écrans, souris, crayons optiques, lecteurs de codes à barre, capteurs, synthétiseurs vocaux / musicaux, lecteurs de disquettes, numériseurs (scanners), modems, imprimantes, unités de disques, bandes magnétiques, disques optiques, traceurs, réseaux, tablettes de projection, etc.

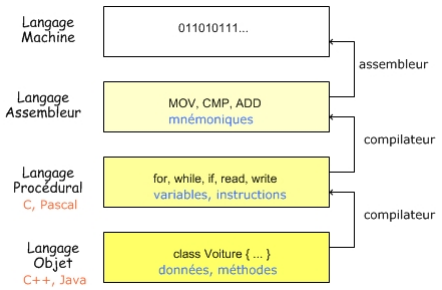
Classification possible :

- **périphériques d'entrée**
- **périphériques de sortie**
- **périphériques de stockage**

Langages

Langage machine, langage assembleur

- Le langage machine est le langage compris par le microprocesseur.
- Ce langage est difficile à maitriser puisque chaque instruction est codée par une séquence propre de bits.
- Afin de faciliter la tâche du programmeur, on a créé le langage assembleur qui utilise des **mnémoniques** pour le codage des instructions



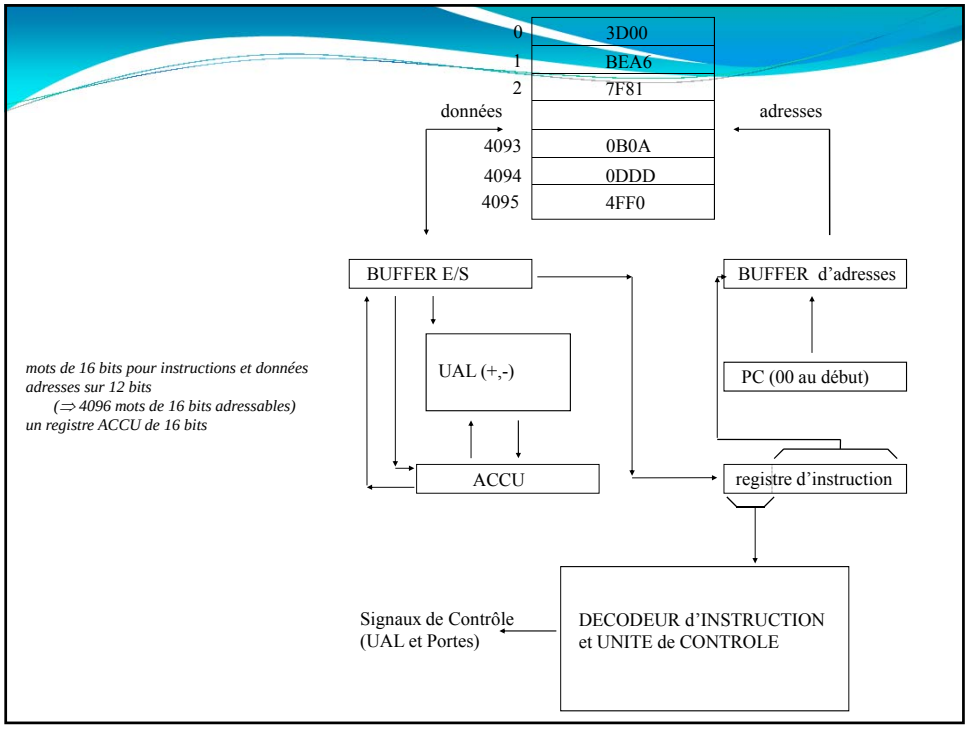
Langage machine	Langage assembleur	Pascal

A1 00 01
8B 1E 02 01
01 D8
A3 04 01
	MOV AX,[100h]
	MOV BX,[102h]
	ADD AX,BX
	MOV [104h],AX
		var a,b,c : integer;
		
		c:=a+b;

Contexte

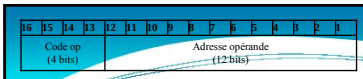
- Machine à mots de 16 bits
- Nombres en binaire complément à deux
- Instructions sur 16 bits
- Architecture Von Newmann (les données et les instructions sont dans le même espace mémoire)
- Adresses sur 12 bits (capacité d'adressage : 4096 mots de 16 bits)
- Compteur ordinal 16 bits
- Accumulateur 16 bits
- Opérations arithmétiques : addition et soustraction
- Adressage direct et indirect

Simulateur sur le site http://www.labri.fr/perso/billaud/WebSim16/



Jeux d'instructions

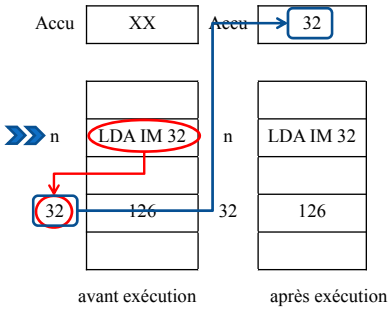
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Code op (4 bits)				Adresse opérande (12 bits) codé en binaire naturel et écrit sous format hexadécimal											



Jeux d'instructions

Code	Mnémonique	Description	Action	Cp = (compteur de programme)
0	**loadi** ***imm12***	chargement immédiat	Acc = ext(imm12)	Cp + 1
1	**load** ***adr12***	chargement direct	Acc = M[adr12]	Cp + 1
2	**loadx** ***adr12***	chargement indirect	Acc = M[M[adr12]]	Cp + 1
3	**store** ***adr12***	rangement direct	M[adr12] = Acc	Cp + 1
4	**storex** ***adr12***	rangement indirect	M[M[adr12]] = Acc	Cp + 1
5	**add** ***adr12***	addition	Acc += M[adr12]	Cp + 1
6	**sud** ***adr12***	soustraction	Acc -=M[adr12]	Cp + 1
7	**jmp** ***adr12***	saut inconditionnel		adr12
8	**jneg** ***adr12***	saut si négatif		Acc<0 ? Adr12 : Cp+1
9	**jzero** ***adr12***	saut si zéro		Acc ==0 ? Adr12 : Cp+1
A	**jmpx** ***adr12***	saut indirect		M[adr12]
B	**call** ***adr12***	Appel	M[adr12] = Cp+1	M[adr12] + 1
C	**halt 0**	arrêt		
D		op. illégale	erreur	
E		op. illégale	erreur	
F		op. illégale	erreur	

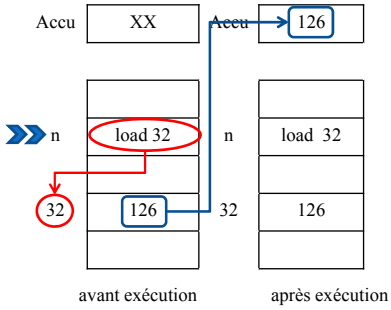
chargement



Adressage immédiat

Code	Mnémonique	Description	Action	Cp = (compteur de programme)
0	**loadi** ***imm12***	chargement immédiat	Acc = ext(imm12)	Cp + 1

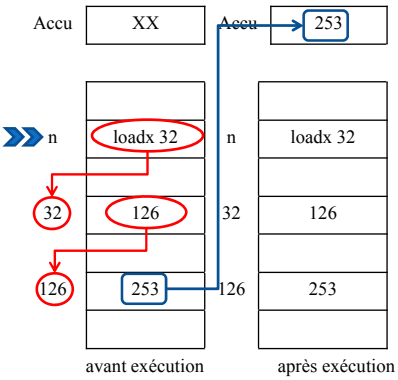
chargement



Adressage direct

Code	Mnémonique	Description	Action	Cp = (compteur de programme)
1	**load** ***adr12***	chargement direct	Acc = M[adr12]	Cp + 1

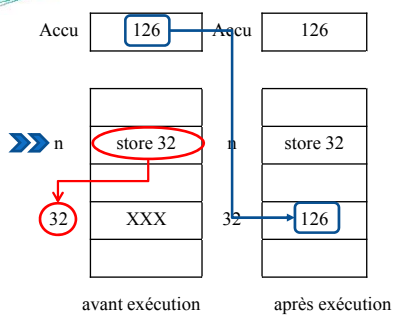
chargement



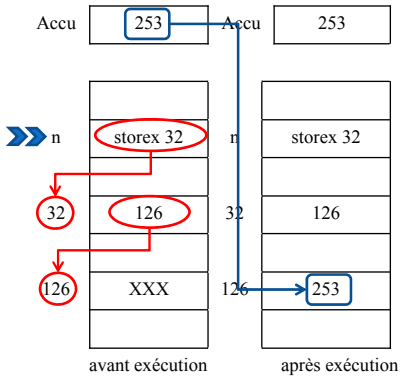
Adressage indirect

Code	Mnémonique	Description	Action	Cp = (compteur de programme)
2	**loadx** ***adr12***	chargement indirect	Acc = M[M[adr12]]	Cp + 1

rangements



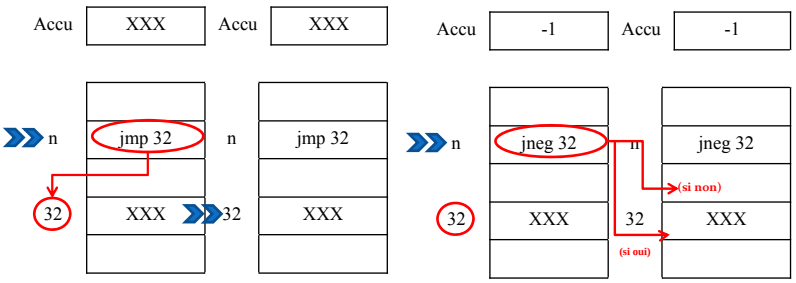
Adressage direct



Adressage indirect

Code	Mnémonique	Description	Action	Cp = (compteur de programme)
3	**store *adr12***	rangement direct	M[adr12] = Acc	Cp + 1
4	**storex *adr12***	rangement indirect	M[M[adr12]] = Acc	Cp + 1

sauts



Code	Mnémonique	Description	Action	Cp = (compteur de programme)
7	**jmp *adr12***	saut inconditionnel		adr12
8	**jneg *adr12***	saut si négatif		Acc<0 ? Adr12 : Cp+1

sauts

Code	Mnémonique	Description	Action	Cp = (compteur de programme)
9	**jzero *adr12***	saut si zéro		Acc ==0 ? Adr12 : Cp+1
A	**jmpx *adr12***	saut indirect		M[adr12]

Code	Mnémonique	Description	Action	Cp = (compteur de programme)
B	**call *adr12***	Appel	M[adr12] = Cp+1	M[adr12] + 1
C	**halt 0**	arrêt		

Commentaires

- *adr* →adresse codée sur 12 bits *imm12* →valeur immédiate codée sur 12 bits
- Chargement immédiat (loadi) : ⇒ *imm12* doit être écrit dans l'accumulateur de 16 bits

Exemple : loadi -3
Code de complément à 2 de -3 pour 12 bits : 0b1111 1111 1101 = 0xFFD
Code de l'instruction sur 16 bits : 0b0000 1111 1111 1101= 0x0FFD

Exemple : loadi 1024
Code de complément à 2 de -1 pour 12 bits : 0b0100 0000 0000 = 0x400
Code de l'instruction sur 16 bits : 0b0000 0100 0000 0000=0x0400

- Opérations indirectes : loadx, storex et jmpx

le contenu à l'adresse adr12 est un mot de 16 bits dont 12 bits d'adresse
erreur est détectée si les 4 bits de poids fort ne sont pas nuls ⇒ Arrêt du processeur

- Ajout possible de trois nouvelles opérations

Exemple de programmation

- Ex 1 : Le programme est écrit en hexadécimal

  0x0009
  0x5005
  0x6006
  0x3007
  0xC000

Simulateur utilisé pour la suite :
http://www.labri.fr/perso/billaud/WebSim16/

Exemple de programmation

- Le programme est écrit en hexadécimal

Status	**Memory**	
Line	Addr.	Source
1	0x000	0009
2	0x001	5005
3	0x002	6006
4	0x003	3007
5	0x004	C000
6	0x005	(vide)

Mot de 16bits	Code	Opérant	Mnémonique	Description	Action
0009	0	9	Loadi	chargement immédiat	Acc = 9
5005	5	5	Add	addition	Acc += M[5]
6006	6	6	Sub	soustraction	Acc -= M[6]
3007	3	7	Store	rangement direct	M[7] = Acc
C000	C	0	Halt	arrêt	

Exemple de programmation

- Le programme est écrit en hexadécimal

Status	**Memory**		Programmation
Line	Addr.	Source	sous assembleur
1	0x000	0009	loadi 9
2	0x001	5005	add 5
3	0x002	6006	sub 6
4	0x003	3007	store 7
5	0x004	C000	halt 0
6	0x005	(vide)	

Remarque:
Les valeurs préenregistrées aux adresses 5 et 6 ⇒ ***Exercice***

Exemple de programmation

```
loadi 5
store 5
loadi 3
store 6
loadi 9
add 5
sub 6
store 7
halt 0
```

Line	Addr.	Source	**Adress:contenu**
1	0x000	loadi 5	000:0005
2	0x001	store 5	001:3005
3	0x002	loadi 3	002:0003
4	0x003	store 6	003:3006
5	0x004	loadi 9	004:0009
6	0x005	add 5	005:5005
7	0x006	sub 6	006:6006
8	0x007	store 7	007:3007
9	0x008	halt 0	008:c000
10	0x009		009:0000
			00a:0000
			00b:0000
			00c:0000

Affectation des valeurs aux adresses 5 et 6 grâce à l'exécution du programme

M[007] = 9+5-3 = 11 (décimal) ⇒ M[007] = 000B (hexadécimal)

déroulement pas à pas

Line	Addr.	Source
1	0x000	loadi 5
2	0x001	store 5
3	0x002	loadi 3
4	0x003	store 6
5	0x004	loadi 9
6	0x005	add 5
7	0x006	sub 6
8	0x007	store 7
9	0x008	halt 0
10	0x009	

000:0005	000:0005	000:0005	000:0005	000:0005	000:0005	000:0005
001:3005	001:3005	001:3005	001:3005	001:3005	001:3005	001:3005
002:0003	002:0003	002:0003	002:0003	002:0003	002:0003	002:0003
003:3003	003:3003	003:0003	003:0003	003:0003	003:0003	003:0003
004:0009	004:0009	004:0009	004:0009	004:0009	004:0009	004:0009
005:5005	005:0005	005:0005	005:0005	005:0005	005:0005	005:0005
006:6006	006:6006	006:0003	006:0003	006:0003	006:0003	006:0003
007:3007	007:3007	007:3007	007:3007	007:3007	007:3007	007:0003
008:c000	008:c000	008:c000	008:c000	008:c000	008:c000	008:c000
009:0000	009:0000	009:0000	009:0000	009:0000	009:0000	009:0000
00a:0000	00a:0000	00a:0000	00a:0000	00a:0000	00a:0000	00a:0000
00b:0000	00b:0000	00b:0000	00b:0000	00b:0000	00b:0000	00b:0000
00c:0000	00c:0000	00c:0000	00c:0000	00c:0000	00c:0000	00c:0000
Acc = 0005	Acc = 0003	Acc = 0003	Acc = 0009	Acc = 0005	Acc = 0003	Acc = 0003

Remarque : Ajout des instructions vont décaler les variables ⇒ modification du codage !

Valeurs temporaires doivent enregistrées dans la zone hors programme de la mémoire

- **Etiquettes : noms utilisés pour désigner les adresses**

```
 loadi 9              1 0x000 loadi 9
 add premier          2 0x001 add premier
 sub second           3 0x002 sub second
 store 10             4 0x003 store 10
 halt 0               5 0x004 halt 0
premier word 5        6 0x005 premier word 5
second  word 3        7 0x006 second word 3
```

```
000:0009
001:5005
002:6006
003:300a
004:c000
005:0005
006:0003
007:0000
008:0000
009:0000
```

La *directive* **word** indique qu'un mot mémoire est réservé
L'étiquette est facultative et doit commencer en colonne 1 si elle est présente
Si elle est absente, il faut au moins un espace avant la mnémonique ou la directrice word

```
 loadi 9
 add premier
 sub second
 store resultat
 halt 0
premier word 5
second  word 3
resultat word 0
```

⇒ Résultat se trouve à l'adresse 007 avec une initialisation à 0

- étiquette peut être seule sur la ligne. Elle se réfère alors au prochain mot : donnée ou instruction
- chaque ligne de commentaires est commencée par #

Exercice

- Ex 2 :Traduire les affectations
  - A = B
  - A = A + 1
  - A = B + C -1

Tests et décisions

- Deux instructions :

  jzero ⇒ le contenu de l'accumulateur est nul ?
  jneg ⇒ le contenu de l'accumulateur est négatif ?

- Sauts conditionnels

  Si « la condition » désignée est vraie
  le déroulement se poursuit à l'adresse désignée
  Sinon
  on passe à l'instruction suivante

Tests et décisions

- Exemple : retourner la valeur absolue de X

Réflexion

```
X : nombre    //entrée
R : nombre    //retour
Si X ≥ 0
            R = X
Sinon
            R = -X
```

- **jneg ⇒ le contenu de l'accumulateur est négatif ?**
- **Si « le contenu de l'accumulateur est négatif ? » est vraie**
  le déroulement se poursuit à l'adresse désignée
  Sinon
  on passe à l'instruction suivante

```
X : nombre    //entrée
R : nombre    //retour
Si X < 0  aller à OPPOSE
            R = X
            aller à SUITE
OPPOSE :
            R = -X
SUITE :
```

Tests et décisions

- Exemple : retourner la valeur absolue de X

```
X : nombre    //entrée
R : nombre    //retour
Si X ≥ 0
            R = X
Sinon
            R = -X
```

```
X : nombre    //entrée
R : nombre    //retour
Si X < 0  aller à OPPOSE
            R = X
            aller à SUITE
OPPOSE :
            R = -X
SUITE :
```

```
            load X
            jneg OPPOSE

            load X
            store R

            jmp SUITE
OPPOSE
            loadi 0
            sub X
            store R

SUITE
            halt 0

X     word  -3
R     word  0
```

Exercice : optimisation du programme

Tests et décisions – optimisation du programme

```
1 0x000                 load X
2 0x001                  jneg OPPOSE
3 0x002
4 0x002                 load X  ←accumulateur contient encore X
5 0x003                 store R
6 0x004
7 0x004                 jmp SUITE
8 0x005    OPPOSE
9 0x005                 loadi 0
10 0x006                sub X
11 0x007                store R
12 0x008
13 0x008   SUITE
 14 0x008                halt 0
15 0x009
16 0x009   X word -3
17 0x00a   R word 0
18 0x00b
```

Tests et décisions – optimisation du programme

```
1 0x000                 load X
2 0x001                  jneg OPPOSE
3 0x002
4 0x002                 load X  ←accumulateur contient encore X
5 0x003                 store R
6 0x004
7 0x004                 jmp SUITE
8 0x005    OPPOSE
9 0x005                 loadi 0
10 0x006                sub X
11 0x007                store R
12 0x008
13 0x008   SUITE
 14 0x008                halt 0
15 0x009
16 0x009   X word -3
17 0x00a   R word 0
18 0x00b
```

```
1 0x000                 load X
2 0x001                 jneg OPPOSE
3 0x002
4 0x002                 jmp SUITE
5 0x003    OPPOSE
6 0x003                  loadi 0
7 0x004                 sub X
8 0x005    SUITE
9 0x005                 store R
10 0x006                halt 0
11 0x007
12 0x007   X word 5
13 0x008   R word 0
14 0x009
15 0x009
```

Tests et décisions – exercices

Ex 3 :
Ecrire la séquence d'instructions en pseudo-code qui identifie le maximum de deux nombres A et B
Traduire ensuite en instructions machine.
Remarque : l'étude de la différence sera faite pour l'action de comparer

Ex 4 :
Ecrire le programme qui ordonne deux nombres A et B.
Après l'exécution , A et B contiendront respectivement le max et le min des deux valeurs initiales.

Boucles

- Exemple : somme des entiers de 1 à N

```
N : nombre    //donnée
S : nombre    //résultat
K : nombre    //variable
S = 0
K = 1
Tant que K≤ N
            S = S + K
            K = K + 1
Fin de tant que
```

```
N : nombre    //donnée
S : nombre    //résultat
K : nombre    //variable
S = 0
K = 1
BOUCLE
Si N-K <0  aller à SUITE
            S = S + K
            K = K + 1
Aller à BOUCLE
SUITE
```

Réflexion

- **jneg ⇒ le contenu de l'accumulateur est négatif ?**
  Accumulateur ← K-N ou N-K ?
- **Si « le contenu de l'accumulateur est négatif ? »est vraie**
  le déroulement se poursuit à l'adresse désignée
  Sinon
  on passe à l'instruction suivante

Boucles

- Exemple : somme des entiers de 1

```
N : nombre    //donnée
S : nombre    //résultat
K : nombre    //variable
S = 0
K = 1
BOUCLE
Si N-K <0  aller à SUITE
           S = S + K
           K = K + 1
Aller à BOUCLE
SUITE
```

```
        loadi 0
        store S     #S=0
        loadi 1
        store K     #K=1
BOUCLE              #si K>N aller à la SUITE
        load N      #calcul N-K
        sub K
        jneg SUITE
        load S      #S=S+K
        add K
        store S
        loadi 1     #K=K+1
        add K
        store K
        jmp BOUCLE
SUITE
        halt 0
N   word  5
S   word  0
K   word  0
```

Boucles

- Exemple : somme des entiers de 1 à N

```
        loadi 0
        store S
        loadi 1
        store K
BOUCLE
        load N
        sub K
        jneg SUITE
        load S
        add K
        store S
        loadi 1
        add K
        store K
        jmp BOUCLE
SUITE
        halt 0
N   word  5
S   word  0
K   word  0
```

Le résultat S est le même?
Sinon pourquoi ?

```
        loadi 0
        store S
        loadi 1
        store K
BOUCLE
        load S
        add K
        store S
        load N
        sub K
        jneg SUITE
        loadi 1
        add K
        store K
        jmp BOUCLE
SUITE
        halt 0
N   word  5
S   word  0
K   word  0
```

Boucles

- Exemple : somme des entiers de 1 à N

```
        loadi 0
        store S
        loadi 1
        store K
BOUCLE
        load N
        sub K
        jneg SUITE
        load S
        add K
        store S
        loadi 1
        add K
        store K
        jmp BOUCLE
SUITE
        halt 0
N   word  5
S   word  0
K   word  0
```

S=0x000f=0d15

S=0x0015=0d21

un tour de plus avant de sortir la boucle

```
        loadi 0
        store S
        loadi 1
        store K
BOUCLE
        load S
        add K
        store S
        load N
        sub K
        jneg SUITE
        loadi 1
        add K
        store K
        jmp BOUCLE
SUITE
        halt 0
N   word  5
S   word  0
K   word  0
```

Boucles– exercices

Ex 5 :
Ecrire la séquence d'instructions pour multiplier deux valeurs (additions successives)

Ex 6 :
Ecrire la séquence d'instructions pour diviser deux valeurs (soustractions successives) et fournir le quotient et le reste.

Ex 7 :
Ecrire la séquence d'instructions pour calculer le factoriel d'un nombre

Ex 8 :
Ecrire la séquence d'instructions pour déterminer le plus petit diviseur non trivial d'un nombre (plus grand que 1)

Tableaux

- Deux instructions :
  - loadx ⇒ chargement indirect
  - storex ⇒rangement indirect
  - à une adresse qui est définie par une variable en mémoire

Exemple :
loadx 23 même effet que load 42 → 156

Le mot d'adresse 23 (c-à-d 42) est considéré comme un *pointeur* vers la donnée effectivement chargée

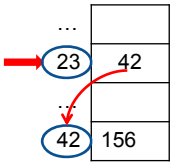


tableau	mémoire	adresse
T[0]		T
T[1]		T+1
T[k-2]		T+k-2
T[k-1]		T+k-1

initialisation

```
T   word  0   #T[0]
    word  0   #T[1]
    word  0   #T[2]
    word  0   #T[3]
…
```

Adressage :

```
loadi  T
add    K
store  PTR //pointeur
```

Accès à la case T+K par pointeur

```
loadx PTR
storex PTR
```

Tableaux

- Exemple : remplissage d'un tableau de 5 éléments avec les nombres de 0 à 4

```
T[5]: tableau de 5 nombres
k: : nombre //variable
Pour k de 0 à 4 par pas de 1
T[k]=k
```

Tableaux

- Exemple : remplissage d'un tableau de 5 éléments avec les nombres de 0 à 4

T[5]: tableau de 5 nombres
k: : nombre //variable
Pour k de 0 à 4 par pas de 1
T[k]=k

```
Début
   k = 0
   Début de la BOUCLE
      calculer N-k
      si N-k ≠ 0
          PTR = adress[T]+k
          Mot[PTR]=k
          k=k+1
      sinon //optionnel
          Fin de la BOUCLE
Fin
//données
N = 5
k = 0
PTR=0
T = 0
```

```
   loadi 0 #optionnel
   store k #optionnel
BOUCLE
   load N
   sub k
   jzero FIN
   loadi T
   add k
   store PTR
   load k
   storex PTR
   loadi 1
   add k
   store k
   jmp BOUCLE
FIN
   halt 0
#donnée
N   word  5
k   word  0
PTR  word 0
T   word   0
```

Tableaux

- Exemple : remplissage d'un tableau de 5 éléments avec les nombres de 0 à 4

Line	Addr.	Source	
1	0x000	BOUCLE	
2	0x000		load N
3	0x001		sub k
4	0x002		jzero FIN
5	0x003		loadi T
6	0x004		add k
7	0x005		store PTR
8	0x006		load k
9	0x007		storex PTR
10	0x008		loadi 1
11	0x009		add k
12	0x00a		store k
13	0x00b		jmp BOUCLE
14	0x00c	FIN	
15	0x00c		halt 0
16	0x00d	#donnée	
17	0x00d	N word 5	
18	0x00e	k word 0	
19	0x00f	PTR word 0	
20	0x010	T word 0	

Status memory

avant	après
000:100d	000:100d
001:600e	001:600e
002:900c	002:900c
003:0010	003:0010
004:500e	004:500e
005:300f	005:300f
006:100e	006:100e
007:400f	007:400f
008:0001	008:0001
009:500e	009:500e
00a:300e	00a:300e
00b:7000	00b:7000
00c:c000	00c:c000
00d:0005	00d:0005
00e:0000	00e:0005
00f:0000	00f:0014
010:0000	010:0000
011:0000	011:0001
012:0000	012:0002
013:0000	013:0003
014:0000	014:0004
015:0000	015:0000

Tableaux - Exercices

- Ex 9:

Ecrire un programme qui calcule la somme des éléments d'un tableau

- Ex 10:

Ecrire un programme qui détermine le minimum des éléments d'un tableau

- Ex 11:

Ecrire un programme qui trie les éléments d'un tableau dans l'ordre croissant

Sous-programmes

- Deux instructions :

  call ⇒ appel de sous-programme

  jmpx ⇒retours de sous-programme

Exemple : call *adr12* → M[*adr12*] = Cp+1: la mémoire à l'adresse *adr12* contient l'adresse de l'instruction suivante Cp+1

→continue à l'adresse *adr12*+1

- Le premier mot d'un sous-programme est donc réservé, et contiendra l'adresse à la quelle le sous-programme devra revenir, par l'intermédiaire de l'instruction jumpx

Sous-programmes

- Exemple : le plus grand valeur de deux nombres

```
#programme principal
        load X
        store MaxA
        load Y
        store MaxB
        call Max
        load MaxRes
        store R
        halt 0

#sous-programme de calcul du
#max de 2 nombres
Max     word 0
        load MaxA
        sub MaxB
        jneg MaxL1
        load MaxA
        jmp MaxL2

MaxL1   load  MaxB
MaxL2   store MaxRes
        jmpx Max
MaxA    word 0
MaxB    word 0
MaxRes word 0
#fin de sous-programme

X  word 9
Y  word 6
R  word 0
```

Sous-programmes

- Exemple : le plus grand valeur de deux nombres

```
#programme principal
        load X
        store MaxA      MaxA=X
        load Y
        store MaxB      MaxB = Y
        call Max        appel de ss-prg Max
        load MaxRes
        store R         R = MaxRes
        halt 0          arrêt du programme

#sous-programme de calcul du
#max de 2 nombres
Max     word 0
        load MaxA       accu = MaxA - MaxB
        sub MaxB        Si accu <0 (c-à-d MaxA <MaxB) aller à MaxL1
        jneg MaxL1      Sinon (MaxA>MaxB)
        load MaxA           accu = MaxA  et   aller à MaxL2
        jmp MaxL2

MaxL1   load  MaxB      accu = MaxB
MaxL2   store MaxRes    MaxRes = accu
        jmpx Max        retourner au programme principal via l'adresse Max
MaxA    word 0
MaxB    word 0
MaxRes word 0
#fin de sous-programme

X  word 9
Y  word 6
R  word 0
```

Sous-programmes - exercices

- Ex 12 :
  Ecrire un sous programme de multiplication par additions successives
- Ex 13 :
  Ecrire un sous programme de calcul factorielle
- Ex 14:
  Ecrire un sous programme de division par soustractions successives
- Ex 15:
  Ecrire un sous programme de calcul de coefficients binomiaux $\frac{n!}{p!(n-p)!}$

Passage de pointeurs

- Action, écrite sous forme d'un sous-programme, qui doit modifier ses paramètres en modifiant les adresses des données

Exemple : échanger les contenus de deux variables

```
#programme principal
        loadi X
        store SwapA
        loadi Y
        store SwapB
        call Swap
        halt 0

#sous-programme pour permuter
#max de 2 nombres
Swap    word 0
        loadx SwapA
        store SwapTemp
        loadx SwapB
        storex SwapA
        load SwapTemp
        storex SwapB
        jmpx Swap
SwapA   word 0
SwapB   word 0
SwapTemp word 0
#fin de sous-programme

X  word 9
Y  word 6
```

Passage de pointeurs

- Action, écrite sous forme d'un sous-programme, qui doit modifier ses paramètres en modifiant les adresses des données

Exemple : échanger les contenus de deux variables

```
#programme principal
        loadi X
        store SwapA        M[&SwapA] = &X
        loadi Y
        store SwapB        M[&SwapB] = &Y
        call Swap           appel de ss-prg Swap
        halt 0              arrêt du programme

#sous-programme pour permuter
#max de 2 nombres
Swap    word 0
        loadx SwapA        accu = M[M[&SwapA]] = M[ &X]
        store SwapTemp     SwapTemp = accu = M[&X]
        loadx SwapB        accu = M[M[&SwapB]] = M[ &Y]
        storex SwapA       M[M[&SwapA]] = accu = M[&Y]
        load SwapTemp      accu =  M[ &X]
        storex SwapB       M[M[&SwapB]] = accu = M[&X]
        jmpx Swap          retourner au programme principal via l'adresse Swap
SwapA   word 0
SwapB   word 0
SwapTemp word 0
#fin de sous-programme

X  word 9
Y  word 6
```

Contenu de la mémoire dont l'adresse donnée par le contenu de SwapA = contenu de Y

Passage de pointeurs

```
#programme principal
        loadi X
        store SwapA
        loadi Y
        store SwapB
        call Swap
        halt 0

#sous-programme pour permuter
#max de 2 nombres
Swap    word 0
        loadx SwapA
        store SwapTemp
        loadx SwapB
        storex SwapA
        load SwapTemp
        storex SwapB
        jmpx Swap
SwapA   word 0
SwapB   word 0
SwapTemp word 0
#fin de sous-programme

X  word 9
Y  word 6
```

000:0011	000:0011
001:300e	001:300e
002:0012	002:0012
003:300f	003:300f
004:b006	004:b006
005:c000	**005:c000**
006:0000	006:0005
007:200e	007:200e
008:3010	008:3010
009:200f	009:200f
00a:400e	00a:400e
00b:1010	00b:1010
00c:400f	00c:400f
00d:a006	00d:a006
00e:0000	00e:0011
00f:0000	00f:0012
010:0000	010:0009
011:0009	011:0006
012:0006	012:0009
013:0000	013:0000
014:0000	014:0000
015:0000	015:0000

Utilisation d'une pile

La technique précédente ne convient pas aux fonctions qui appellent elles-mêmes, directement ou indirectement

La pile est une zone de mémoire permettant de stocker et retrouver rapidement des valeurs pour :

•Transmettre les arguments à un sous-programme.
•Placer des variables locales dans un sous-programme
•Sauvegarder l'adresse de retour
•Mettre le résultat dans l'accumulateur

Utilisation d'une pile

- Exemple : Calcul de fibonacci récursif

La **suite de Fibonacci** est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Elle commence généralement par les termes 0 et 1 (parfois 1 et 1) et les autres termes sont définis tels que :

$$F_n = F_{n-1} + F_{n-2}$$

	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}
dec	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377
hex	0	1	1	2	3	5	8	D	15	22	37	59	90	E9	179

Utilisation d'une pile

- Exemple : Calcul de fibonacci récursif

Programme principal

```
Enregistrer l'adresse de pile de mémoire dans cadre          loadi pile
                                                             store cadre
Pour tmp de 1 à N par pas de 1 (voir la valeur imposée de tmp plus tard)   start
                                                               load N
                                                               sub tmp
   si tmp ≤ N                                                  jneg fin
     acc = tmp                                                 load tmp
     appeler le sous-programme  fib (N)                        call fib
     enregistrer le contenu de acc dans la mémoire où l'adresse est indiqué par cadre   storex cadre
                                                               loadi 1
                                                               add tmp
     tmp = tmp +1                                             store tmp
                                                               jmp start
Fin de pour                                                  fin
arrêt du programme                                             halt 0
```

Utilisation d'une pile

- Exemple : Calcul de fibonacci récursif

Sous-programme fib(N)

```
Début de sous programme qui indique l'adresse de retour      fib word 0
                                                               store tmp
                                                               loadi 1
                                                               sub tmp
                                                               jneg calcul
enregistrer la valeur d'itération (qui est dans l'accu) dans tmp   loadi 1
Si itération ≤ 1                                               add cadre
    cadre=cadre +1 : pour stocker l'adresse de pile[1]         store cadre
    accu = itération → mettre le résultat dans l'accu          load tmp
                                                               jmpx fib
Sinon  itération > 1 (faire calcul)                          calcul
      cadre = cadre +1                                         loadi 1
      ptr = cadre -1                                           add cadre
                                                               store cadre
                                                               loadi -1
                                                               add cadre
                                                               store ptr
      moins1 contient le contenu de fib[itération-1]           loadx ptr
      ptr = cadre -2                                           store moins1
      accu contient le contenu de fib[itéraion-2]              loadi -2
      accu contient la somme des contenus de fib[itération-1] et fib[itéraion-2]   add cadre
                                                               store ptr
                                                               loadx ptr
Fin de sous-programme avec le résultat dans l'accu             add moins1
                                                               jmpx fib
```

Utilisation d'une pile

- Exemple : Calcul de fibonacci récursif

Déclaration des variables

```
R word 0
N word 14
ptr word 0
tmp word 1
moins1 word 0
cadre word 0
pile word 0
```

tmp word 1 pour démarrer la boucle à 1 car F_1 est initialement égal à 0

Utilisation d'une pile

- Exemple : Calcul de fibonacci récursif

```
Line    Addr.   Source
1       0x000    loadi pile
2       0x001   store cadre
3       0x002   start
4       0x002   load N
5       0x003   sub tmp
6       0x004   jneg fin
7       0x005   load tmp
8       0x006   call fib
9       0x007   storex cadre
10      0x008   loadi 1
11      0x009   add tmp
12      0x00a   store tmp
13      0x00b   jmp start
14      0x00c   fin
15       0x00c  halt 0
16      0x00d
17      0x00d   fib word 0
18      0x00e   store tmp
19      0x00f   loadi 1
20      0x010   sub tmp
21      0x011   jneg calcul
22      0x012   loadi 1
23      0x013   add cadre
24      0x014   store cadre
25      0x015   load tmp
26      0x016    jmpx fib
27      0x017
28      0x017   calcul
29      0x017   loadi 1
30      0x018   add cadre
31      0x019   store cadre
32      0x01a   loadi -1
33      0x01b   add cadre
34      0x01c   store ptr
35      0x01d   loadx ptr
36      0x01e   store moins1
37      0x01f   loadi -2
38      0x020   add cadre
39      0x021   store ptr
40      0x022   loadx ptr
41      0x023   add moins1
42      0x024   jmpx fib
43      0x025
44      0x025
45      0x025   R word 0
46      0x026   N word 14
47      0x027   ptr word 0
48      0x028   tmp word 1
49      0x029   moins1 word 0
50      0x02a   cadre word 0
51      0x02b   pile word 0
52 0x02c

Symbol  Value   Line
------ ----- ----
start   0x002   3
fin     0x00c   14
fib     0x00d   17
calcul  0x017   28
r       0x025   45
n       0x026   46
ptr     0x027   47
tmp     0x028   48
moins1  0x029   49
cadre   0x02a   50
pile    0x02b   51
------ ----- ----
```

Utilisation d'une pile

- Exemple : Calcul de fibonacci récursif

000:002b	020:502a	040:0000	060:0000
001:302a	021:3027	041:0000	061:0000
002:1026	022:2027	042:0000	062:0000
003:6028	023:5029	043:0000	063:0000
004:800c	024:a00d	044:0000	064:0000
005:1028	025:0000	045:0000	065:0000
006:b00d	026:000e	046:0000	066:0000
007:402a	027:0000	047:0000	067:0000
008:0001	028:0001	048:0000	068:0000
009:5028	029:0000	049:0000	069:0000
00a:3028	02a:0000	04a:0000	06a:0000
00b:7002	02b:0000	04b:0000	06b:0000
00c:c000	02c:0000	04c:0000	06c:0000
00d:0000	02d:0000	04d:0000	06d:0000
00e:3028	02e:0000	04e:0000	06e:0000
00f:0001	02f:0000	04f:0000	06f:0000
010:6028	030:0000	050:0000	070:0000
011:8017	031:0000	051:0000	071:0000
012:0001	032:0000	052:0000	072:0000
013:502a	033:0000	053:0000	073:0000
014:302a	034:0000	054:0000	074:0000
015:1028	035:0000	055:0000	075:0000
016:a00d	036:0000	056:0000	076:0000
017:0001	037:0000	057:0000	077:0000
018:502a	038:0000	058:0000	078:0000
019:302a	039:0000	059:0000	079:0000
01a:0fff	03a:0000	05a:0000	07a:0000
01b:502a	03b:0000	05b:0000	07b:0000
01c:3027	03c:0000	05c:0000	07c:0000
01d:2027	03d:0000	05d:0000	07d:0000
01e:3029	03e:0000	05e:0000	07e:0000
01f:0ffe 03f			

MICROPROCESSEURS 80x86 et Assembleur

Microprocesseur 80x86

Le processeur 8086 d'Intel est à la base des processeurs Pentium actuels.

Les processeurs successifs (de PC) se sont en effet construits petit à petit en ajoutant à chaque processeurs des instructions et des fonctionnalités supplémentaires. Mais en conservant à chaque fois les spécificités du processeur précédent.

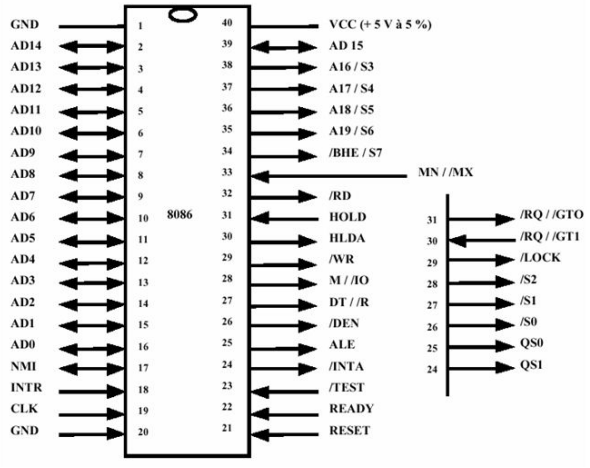
C'est cette façon d'adapter les processeurs à chaque étape qui permet qu'un ancien programme écrit pour un 8086 fonctionne toujours sur un nouvel ordinateur équipé d'un microprocesseur plus récent.

http://www.technologuepro.com/microprocesseur/chap2_microprocesseur.htm

Microprocesseur 80x86

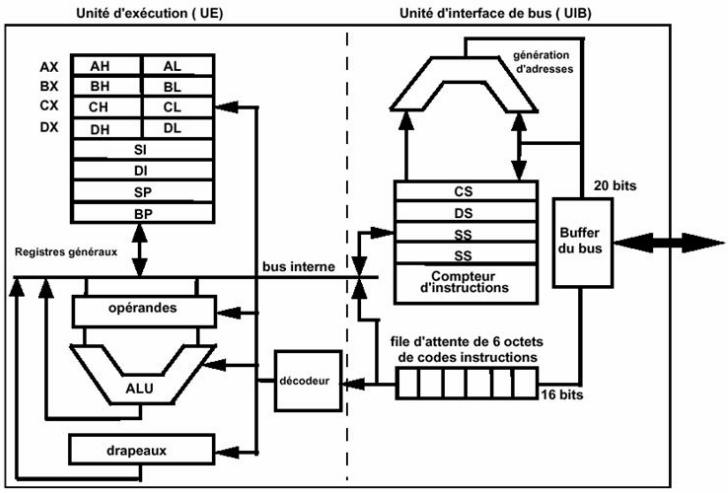
Le 8086 est un circuit intégré de forme DIL de 40 pattes
Le 8086 (développé en 1978) est le premier microprocesseur de type x86

- bus de données de 16 bits
- bus d'adresses de 20 bits
- fonctionne à des fréquences diverses selon plusieurs variantes: 5, 8 ou 10 MHz.



Microprocesseur 80x86

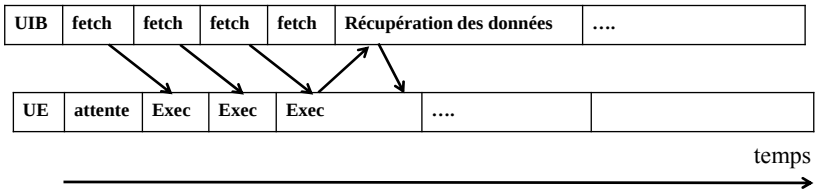
→deux unités internes distinctes:
•l'UE (Unité d'Exécution)
•l'UIB (Unité d'Interfaçage avec le Bus)
→Lorsque l'exécution d'une instruction est terminée, l'UE reste inactif un court instant, pendant que l'UIB extrait l'instruction suivante



Microprocesseur 8086/8088

Architecture « pipeline »

→ Pour remédier à ce temps d'attente, le *prétraitement* ou *traitement pipeline* a été introduit dans le 8086/8088. Pendant que l'UE exécute les informations qui lui sont transmises, l'instruction suivante est chargée dans l'UIB. Les instructions qui suivront sont placées dans une file d'attente. Lorsque l'UE a fini de traiter une instruction l'UIB lui transmet instantanément l'instruction suivante, et charge la troisième instruction en vue de la transmettre à l'UE. De cette façon, l'UE est continuellement en activité.



Microprocesseur 8086/8088

4 REGISTRES GENERAUX ou de travail (op' arithm.) sur 16 bits (DATA REGISTERS):

Registre AX : (Accumulateur)
•les opérations de transferts de données avec les entrées-sorties
•le traitement des chaînes de caractères
•les opérations arithmétiques et logiques.
•les conversions en BCD du résultat d'une opération arithmétique (addition, soustraction, multiplication et la division)

Registre BX : (registre de base)
Il est utilisé pour l'adressage de données dans une zone mémoire différente de la zone code : en général il contient une adresse de décalage par rapport à une adresse de référence.). De plus il peut servir pour la conversion d'un code à un autre.

Microprocesseur 8086/8088

4 REGISTRES GENERAUX ou de travail (op' arithm.) sur 16 bits (DATA REGISTERS):

Registre CX : (Le compteur)
Lors de l'exécution d'une boucle on a souvent recours à un compteur de boucles pour compter le nombre d'itérations, le registre CX a été fait pour servir comme compteur lors des instructions de boucle.

Registre DX :
On utilise le registre DX pour les opérations de multiplication et de division mais surtout pour contenir le numéro d'un port d'entrée/sortie pour adresser les interfaces d'E/S.

Microprocesseur 8086/8088

4 REGISTRES GENERAUX ou de travail (op' arithm.) sur 16 bits (DATA REGISTERS):

chaque registre peut être divisé en deux registres de 8 bits (AH,AL,BH,BL,CH,CL,DH et DL)

	7 0	7 0	
ax	**ah**	**al**	**accumulateur**
bx	**bh**	**bl**	**base**
cx	**ch**	**cl**	**compteur**
dx	**dh**	**dl**	**données**
	h=high	**l=low**	

Microprocesseur 8086/8088

4 REGISTRES SEGMENTS :

➢Quatre **registres segments** de 16 bits chacun :
- •CS (code segment)
- •DS (Data segment)
- •ES (Extra segment)
- •SS (stack segment)

	15 0
Code segment	CS
Data segment	DS
Stack segment	SS
Extra segment	ES

➢ces registres sont chargés de sélectionner les différents segments de la mémoire en pointant sur le début de chacun d'entre eux. Chaque segment de mémoire ne peut excéder les 65535 octets(2^{16}).

⇒***l'espace mémoire*** **est divisé en *4 segments* de capacité maximale 64 K octets**

Microprocesseur 8086/8088

4 REGISTRES SEGMENTS :

Le registre CS (code segment) :
Il pointe sur le segment qui contient les codes des instructions du programme en cours.
Le registre DS (Data segment) :
Le registre segment de données pointe sur le segment des variables globales du programme
Le registre ES (Extra segment) :
Le registre de données supplémentaires ES est utilisé par le microprocesseur lorsque l'accès aux autres registres est devenu difficile ou impossible pour modifier des données, de même ce segment est utilisé pour le stockage des chaînes de caractères.
Le registre SS (Stack segment) :
Le registre SS pointe sur la pile : la pile est une zone mémoire où on peut sauvegarder les registres (ou les adresses ou les données) pour les récupérer après l'exécution d'un sous programme ou l'exécution d'un programme d'interruption. En général il est conseillé de ne pas changer le contenu de ce registre car on risque de perdre des informations très importantes (exemple les passages d'arguments entre le programme principal et le sous programme)

Microprocesseur 8086/8088

4 REGISTRES GENERAUX d'ADRESSAGE dans un ESPACE D'ADRESSAGE / SEGMENT sur 16 bits (POINTER & INDEX REGISTERS) :

→spécialement adaptés au traitement des éléments dans la mémoire. Ils sont généralement munis de propriétés d'incrémentation et de décrémentation.

L'indexe SI : (source indexe) :
Il permet de pointer la mémoire et forme en général un décalage (un offset) par rapport à une base fixe (le registre DS), il sert aussi pour les instructions de chaîne de caractères, en effet il pointe sur le caractère source.

L'indexe DI : (Destination indexe) :
Il permet aussi de pointer la mémoire il présente un décalage par rapport à une base fixe (DS ou ES), il sert aussi pour les instructions de chaîne de caractères, il pointe alors sur la destination

Microprocesseur 8086/8088

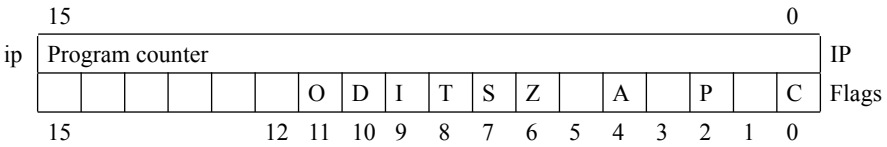
Les pointeurs SP et BP : (Stack pointer et base pointer)

•Ils pointent sur la zone pile (une zone mémoire qui stocke l'information avec le principe FILO, ils présentent un décalage par rapport à la base (le registre SS)

•Pour le registre BP il a un rôle proche de celui de BX, mais il est généralement utilisé avec le segment de pile.

Microprocesseur 8086/8088

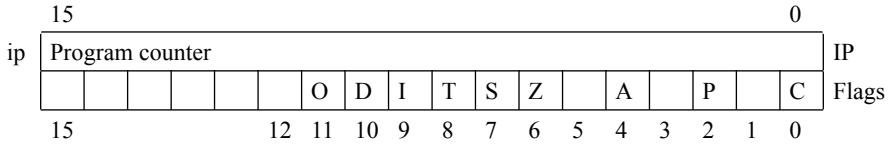
COMPTEUR ORDINAL (IP)



Instruction Pointer *(IP)* ou **Compteur de Programme** (PC), contient l'adresse de l'emplacement mémoire où se situe la prochaine instruction à exécuter. Autrement dit, il doit indiquer au processeur la prochaine instruction à exécuter. Le registre IP est constamment modifié après l'exécution de chaque instruction afin qu'il pointe sur l'instruction suivante.

Microprocesseur 8086/8088

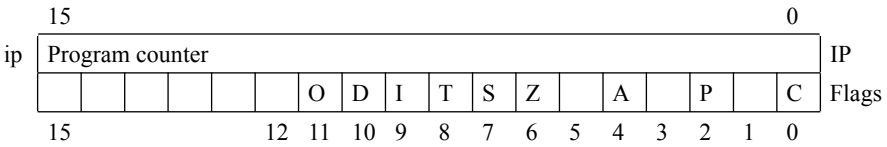
COMPTEUR ORDINAL (IP)



Le registre d'état FLAG sert à contenir l'état de certaines opérations effectuées par le processeur. Par exemple, quand le résultat d'une opération est trop grand pour être contenu dans le registre cible (celui qui doit contenir le résultat de l'opération), un bit spécifique du registre d'état (le bit OF) est mis à 1 pour indiquer le débordement.

Microprocesseur 8086/8088

REGISTRES D'ETAT (FLAGS)



O : Overflow Flag
Débordement : si on a un débordement arithmétique ce bit est mis à 1.c a d le résultat d'une opération excède la capacité de l'opérande (registre ou case mémoire), sinon il est à 0.

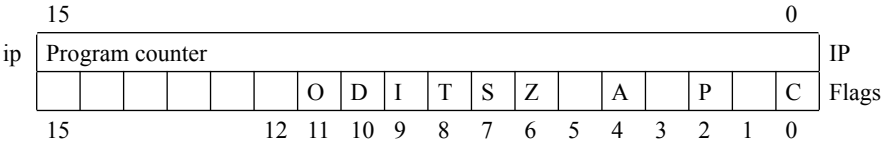
D : Direction Flags (Strings)
Auto Incrémentation/Décrémentation : utilisée pendant les instructions de chaîne de caractères pour auto incrémenter ou auto décrémenter.

I : Interrupt Flag
Masque d'interruption : pour masquer les interruptions venant de l'extérieur
ce bit est mis à 0, dans le cas contraire le microprocesseur reconnaît l'interruption de l'extérieur.

T : Trap (single step) Flag
Piége : pour que le microprocesseur exécute le programme pas à pas.

Microprocesseur 8086/8088

REGISTRES D'ETAT (FLAGS)



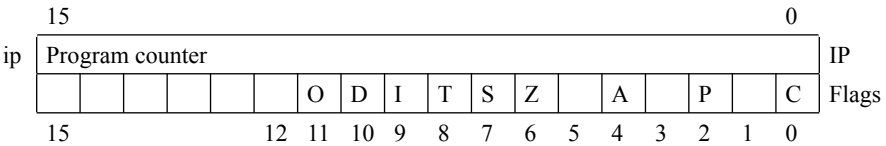
S : Sign Flag

SF est positionné à 1 si le bit de poids fort du résultat d'une addition ou soustraction est 1 ; sinon SF=0. SF est utile lorsque l'on manipule des entiers signés, car le bit de poids fort donne alors le signe du résultat. Exemples (sur 8 bits) :

```
    10010110          11011001
  + 01010100        + 01010010
SF=1 11101010     SF=0 00101011
```

Microprocesseur 8086/8088

REGISTRES D'ETAT (FLAGS)



Z : Zero Flag

Zéro : Cet indicateur est mis à 1 quand le résultat d'une opération est égal à zéro. Lorsque l'on vient d'effectuer une soustraction (ou une comparaison), ZF=1 indique que les deux opérandes étaient égaux. Sinon, ZF est positionné à 0.

A : Auxiliary Carry (BCD)

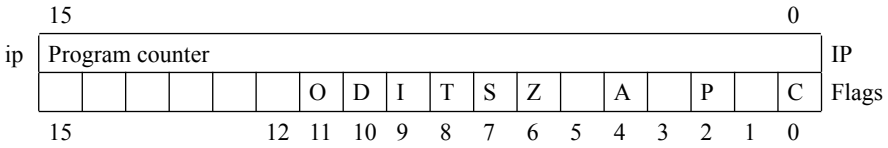
Demie retenue : Ce bit est égal à 1 si on a une retenue du quarter de poids faible dans le quarter de poids plus fort.

P : Parity Flag

Parité : si le résultat de l'opération contient un nombre pair de 1 cet indicateur est mis à 1, sinon zéro.

Microprocesseur 8086/8088

REGISTRES D'ETAT (FLAGS)



C : Carry Flag
Retenue : cet indicateur et mis à 1 lorsque il y a une retenue du résultat à 8 ou 16 bits.
Il intervient dans les opérations d'additions (retenue) et de soustractions (borrow) sur des entiers naturels . Il est positionné en particulier par les instructions ADD, SUB et CMP(comparaison entre deux valeurs).
CF = 1 s'il y a une retenue après l'addition ou la soustraction du bit de poids fort des opérandes.
Exemples (sur 8 bits pour simplifier) :

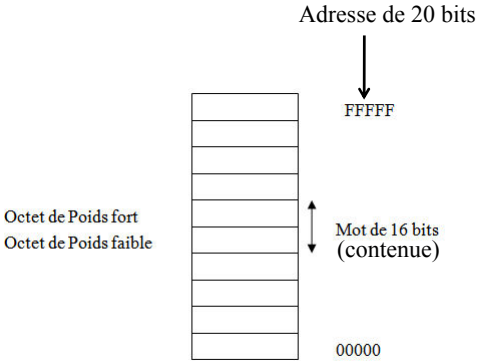
```
      10010110          11011001
    + 01010100        + 01010010
CF=0  11101010    CF=1  00101011
```

Microprocesseur 8086/8088

ORGANISATION DE LA MEMOIRE

Organisation logique

Le microprocesseur 8088 est processeur 16 bits (bus de données de 16 bits) → possibilité d'accéder en même temps à deux cases mémoires de 8 bits.



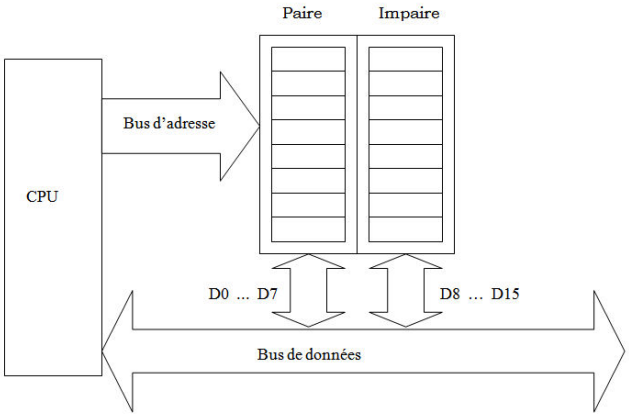
Microprocesseur 8086/8088

ORGANISATION DE LA MEMOIRE

Organisation physique

⇒la mémoire est organisée en deux Banks chacun de 512 Koctet)

•un bank pair
•un bank impair



Microprocesseur 8086/8088

ORGANISATION DE LA MEMOIRE

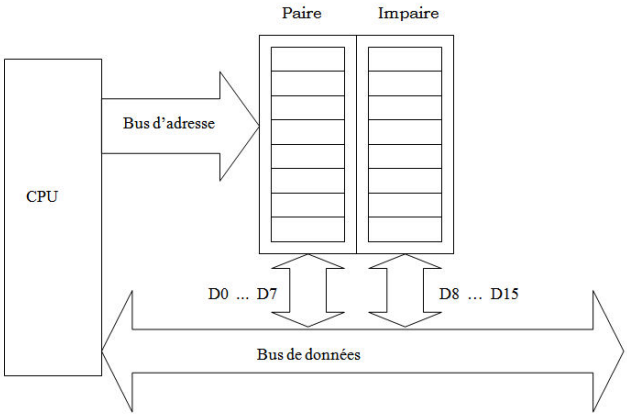
Organisation physique

bits D0..D7 →partie base
bits D8..D15 →partie haute
⇒Le microprocesseur peut charger :

•un octet (8 bits) →adresse de l'octet

•un mot (16 bits) →cherche l'octet du poids faible à l'adresse donnée et l'octet du poids le plus fort à l'adresse qui suit

•un double mot (32 bits)



Microprocesseur 8086/8088

ORGANISATION DE LA MEMOIRE

Organisation physique

Rangement des cases

- Un mot de 8 bits
- Un mot de 16 bits
  - un cycle (ex : octet2 et octet 3)
  - deux cycles (ex : octet1 et octet2)

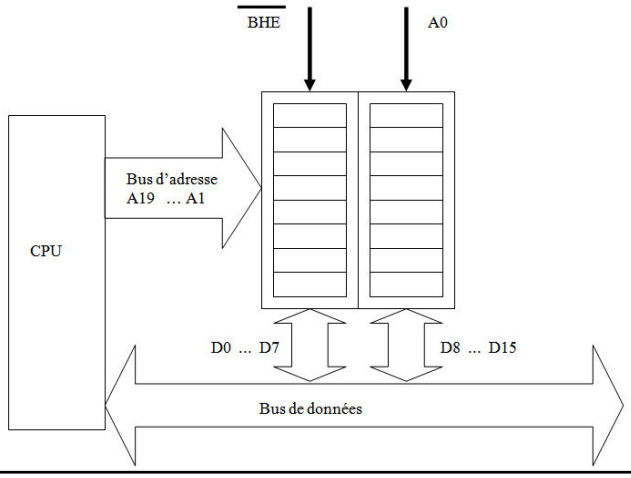
Paire	Impaire
octet 6	octet 7
octet 4	octet 5
octet 2	octet 3
octet 0	octet 1

Microprocesseur 8086/8088

ORGANISATION DE LA MEMOIRE

Organisation physique

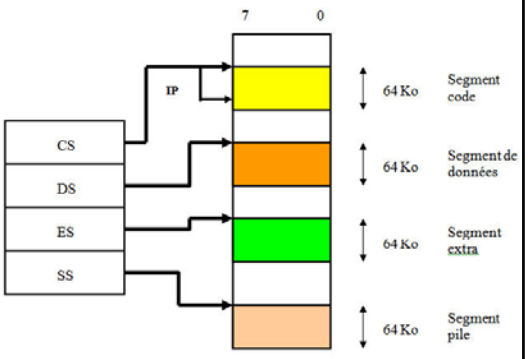
•sélectionner les banks pair et impair → le microprocesseur utilise deux signaux
- BHE (BHE = 0)
- A0 : le premier bit du bus d'adresse (A0=1)



Microprocesseur 8086/8088

GESTION DES ADRESSES MÉMOIRE

- **L'accès direct et simultané à ces espaces**
- **le compteur de programme est de 16 bits →possibilité d'adressage est de 2^{16} = 64 Ko → segments logiques de 64 Ko**
- **L'espace mémoire adressable : 1 méga = 2^{20} → 20 bits du bus d'adresse**
- **4 segments logiques ne couvre pas la totalité de la mémoire, →utilisation de deux registres pour indiquer une adresse au processeur**



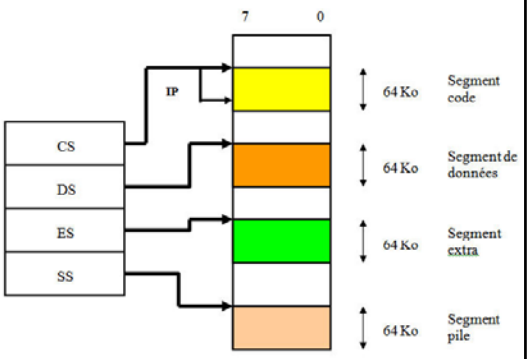
- **Chaque segment débute à l'endroit spécifié par le registre segment. Le déplacement (offset) à l'intérieur de chaque segment se fait par un registre de décalage qui permet de trouver une information à l'intérieur du segment. Exemple la paire de registre CS:IP : pointe sur le code d'une instruction (CS registre segment et IP Déplacement)**

Microprocesseur 8086/8088

GESTION DES ADRESSES MÉMOIRE

- **Pas de problème si :**
  - taille progr. ≤64Ko et
  - données ≤ 128 Ko
  (sécurité : code et données séparés)

- **Si la taille du programme est > 64Ko, le programmeur segmente (découpe) son application en autant de segments qu'il le souhaite, dans la limite du Mo disponible**



Microprocesseur 8086/8088

GESTION DES ADRESSES MÉMOIRE

Adresse physique (Segmentation de la mémoire) :

•les données → regroupées dans une zone mémoire nommée ***segment de données* DS**
•les instructions → placées dans un ***segment d'instructions* CS**(de même pour le segment pile et segment de données supplémentaires).
⇒Ce partage se fonde sur la notion plus générale de ***segment de mémoire***, qui est à la base du mécanisme de gestion des adresses par les processeurs 80x86.
•le registre IP de 16 bits → adressage de 64 Ko
•Le bus d'adresses du 8086 possède 20 bits.
⇒adresse de 20 bits est formée par la juxtaposition d'un registre segment (16 bits de poids fort) et d'un déplacement *(offset*, 16 bits de poids faible).
Adresse physique= Base * 16+ offset

Bits	20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
	Base	0 0 0 0
+	0 0 0 0	Offset
=	Adresse physique	

Microprocesseur 8086/8088

GESTION DES ADRESSES MÉMOIRE

Adresse physique (Segmentation de la mémoire) :

Bits	20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
	Base	0 0 0 0
+	0 0 0 0	Offset
=	Adresse physique	

Exemples : **Mots écrits avec le code hexadécimal*

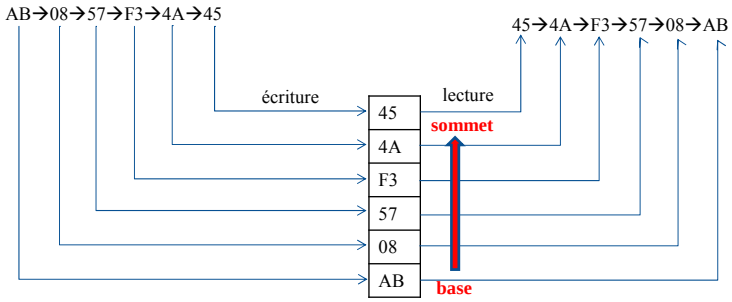
Adresse logique ou virtuel	CS→1000	10000	CS→12C4	12C40
	IP→2006	+2006	IP→0022	+0022
Adresse physique →		12006		12C62

Microprocesseur 8086/8088

IMPLEMENTATION DE LA PILE

Une pile est un ensemble de données placées en mémoire de manière à ce que seulement la donnée du "dessus" soit disponible à un instant donné.

La pile est de type LIFO (Last In First Out) : première valeur empilée sera la dernière sortie



Microprocesseur 8086/8088

IMPLEMENTATION DE LA PILE

Le pointeur de pile « Offset » SP (Stack Pointer) en combinaison avec le segment de pile « Base » SS (Stack Segment) pointe vers le dessus de la pile (TOS : top of stack) en mémoire.

SP pointe vers le sommet c'est-à-dire sur le dernier bloc occupé par la pile

Lorsqu'on ajoute un élément dans la pile, l'adresse contenue dabs SP est décrémentée de 2 octets (car un emplacement de la pile fait 16 bits de longueur). Quand on parcourt la pile de la base vers le sommet, les adressent décroissent

Microprocesseur 8086/8088

IMPLEMENTATION DE LA PILE

Le rôle du pointeur de pile (et de la pile vers laquelle il pointe) :

Quand un processeur exécute une instruction, il est possible qu'il soit interrompu par une "Interruption" Il doit alors arrêter de s'occuper de l'instruction qu'il traite présentement pour s'occuper de l'interruption.
Quand l'interruption sera traitée, il retournera à l'instruction qu'il traitait quand il a été interrompu.

⇒ il doit se rappeler de cette instruction ainsi que de l'état de certains registres au moment où il traitait l'instruction.
⇒ Donc pour ne pas les perdre, il les placera temporairement dans une pile (à l'intérieur de la mémoire RAM par exemple) et pourra les récupérer une fois l'interruption traitée. Le pointeur de pile (SP) donne donc l'adresse en mémoire de cette pile temporaire.

Les *piles* offrent un nouveau moyen d'accéder à des données en mémoire principale, qui est très utilisé pour stocker temporairement des valeurs

Microprocesseur 8086/8088

INTERRUPTION

❖deux types : **→externes : générées par l'extérieur (périphérique,...)**
→internes au microprocesseur 8086 (trap, déroutement, ...)

❖3 interruptions *externes* : **RESET : réinitialise le microprocesseur**
NMI : Non Masquable Interrupt
INTR : interruption prise en compte ou non selon l'état du bit I (Interrupt Enable Flag) du registre d'état

❖2 sortes d'interruptions *internes* : **les automatiques (div./0, d.d.c., pas à pas)**
les logicielles (les autres) : SVC

❖Toute interruption acceptée provoque l'exécution d'un sous-programme (*handler d'interruption*) démarrant à une adresse spécifique à chaque interruption et contenue dans une table
1 élément de la table = 4 octets
CS ←les deux premiers octets (adresse de base)
IP ←les deux derniers octets (déplacement)

❖Les adresses 00000H à 003ffH sont réservées à la table , dite table des interruptions (256 interruptions * 4 octets = 1Ko)

IP = Instruction Pointer PC = Compteur Ordinal

jeu d'instructions du 8086/8088 d'Intel

Les instructions de transfert de données

Usage	Nom	Fonction
Général	MOV	Transfert d'octets ou de mots
	PUSH	Chargement de la pile
	POP	Déchargement de la pile
	PUSHA	Chargement de tous les registres dans la
	POPA	pile Déchargement de tous les registres
	XCHG	dans la pile Echange d'octet ou de mot
	XLAT	Translation d'octet
Entrées-sorties	IN	Entrée de mot ou d'octet
	OUT	Sortie de mot ou d'octet
Adresses	LEA	Chargement de l'adresse effective
	LDS	Chargement du pointeur avec DS
	LES	Chargement du pointeur avec ES
Indicateurs	LAHF	Transfert des indicateurs dans AH
	SAHF	Rangement de AH dans les indicateurs
	PUSHP	Chargement des indicateurs dans la pile
	POPF	Déchargement des indicateurs de la pile.

Instructions arithmétiques :

Les instructions arithmétiques peu vent manipuler quatre types de nombres :

*Les nombres binaires non signés
*Les nombres binaires signés.
*Les nombres décimaux codés binaires (DCB), non signés.
*Les nombres DCB non condensés, non signés.

Usage	Nom	Fonction
Addition	ADD	Addition sur un octet ou un mot
	ADC	Addition sur un octet ou un mot avec retenue
	INC	Incrémentation de 1
	AAA	Ajustement ASCII
	DAA	Ajustement décimal
Soustraction	SUB	Soustraction sur un octet ou un mot
	SBB	Soustraction sur un octet (mot) avec retenue
	DEC	Décrémentation de 1
	NEG	Mêtre un octet ou un mot en négatif
	CMP	Comparaison d'octet ou mot
	AAS	Ajustement ASCII
	DAS	Ajustement décimal
Multiplication	MUL	Multiplication d'octet ou de mot **non signée**
	IMUL	Multiplication d'octet ou de mot **signée**
	AAM	Ajustement ASCII
Division	DIV	Division d'octet ou de mot **non signée**
	IDIV	Division d'octet ou de mot **signée**
	AAD	Ajustement ASCII
	CBW	Conversion d'un octet en un mot
	CWD	Conversion d'un mot en double mots

Les instructions logiques (de bits)

Usage	Nom	Fonction
Logique	NOT	Inversion logique sur un octet ou un mot
	AND	Et logique
	OR	Ou logique
	XOR	Ou exclusif
	TEST	Et logique sans résultat, affecte uniquement les indicateurs du registre des flags.
Décalages	SHL	Décalage logique à gauche
	SAL	Décalage arithmétique à gauche
	SHR	Décalage logique à droite
	SAR	Décalage arithmétique à droite
Rotation	ROL	Rotation à gauche
	ROR	Rotation à droite
	RCL	Rotation à gauche à travers le bit de retenue
	RCR	Rotation à droite à travers le bit de retenue

Instructions de sauts de programme

Elles permettent de faire des sauts dans l'exécution d'un programme (rupture de séquence)

Remarque :

Ces instructions n'affectent pas les Flags. Dans cette catégorie on trouve toutes les instructions de branchement, de boucle et d'interruption après un branchement

Type	Nom	Fonction
Branchements inconditionnels	CALL	Appel à un sous programme
	RET	Retour d'un sous programme
	JMP	Saut
Branchements conditionnels (arithmétique non signée)	JA/JNBE	Si supérieur / Si non inférieur ou non égal
	JAE/JNB	Si supérieur ou égal/ Si non inférieur
	JB/JNAE	Si inférieur/si non supérieur ni égal
	JBE/JNA	Si inférieur ou égal/si non supérieur.
Branchements conditionnels (arithmétique signée)	JG/JNLE	Si plus grand/si pas inférieur ni égal
	JGE/JNL	Si plus grand ou égal/Si pas inférieur
	JL/JNGE	Si moins que/Si pas plus grand ni égal
	JLE/JNG	Si moins que ou égal/Si pas plus grand
Branchement conditionnels (flags)	JC	Si retenue
	JE/JZ	Si égal/Si zéro
	JNC	Si pas de retenue
	JNE/JNZ	Si non égal / Non zéro
	JNO	Si pas de débordement
	JNP/JPO	Si pas de parité/ Si parité impaire
	JNS	Si pas de signe
	JO	Si débordement
	JP/JPE	Si parité / Si parité paire
	JS	Si signe (négatif)
Boucles	LOOP	Boucle
	LOOPE/LOOPZ	Boucle si égal/Si zéro
	LOOPNE/LOOPNZ	Boucle si différent/si diff 0
	JCXZ	Branchement si CX=0
Interruptions	INT	Interruption
	INTO	Interruption si débordement
	IRET	Retour d'interruption.

Les instructions de chaînes de caractères

Nom	Fonction
REP	Préfixe de répétition
REPE/REPZ	Répétition tant qu'égal à zéro
REPNE/REPNZ	Répétition tant que différent de zéro
MOVS	Déplacement de chaîne
MOVSB/MOVSW	Déplacement de chaîne
CMPS	Comparaison de chaînes
INS	Entrée (de port d'E/S)
OUTS	Sortie (vers un port d'E/S)
SCAS	Balayage d'une chaîne
LODS	Chargement de chaîne
STOS	Rangement de chaînes

Les instructions de commande du processeur

Type	Nom	Fonction
Indicateur (FLAGS)	STC	Met à 1 la retenue CF
	CLC	MET à 0 la retenue CF
	CMC	Complémente la retenue
	STD	Met à 1 la direction DF
	CLD	Met à 0 la direction DF
	STI	Met à 1 l'autorisation d'interruption
	CLI	Met à 0 l'autorisation d'interruption
Synchronisation	HLT	Halte jusqu'à interruption ou RESET
	WAIT	Attente jusqu'à broche TEST passe à 0
	ESC	Pour un coprocesseur
	LOCK	Verrouillage des bus pendant la prochaine instructions
Sans opération	NOP	Pas d'opération

Microprocessor Simulator

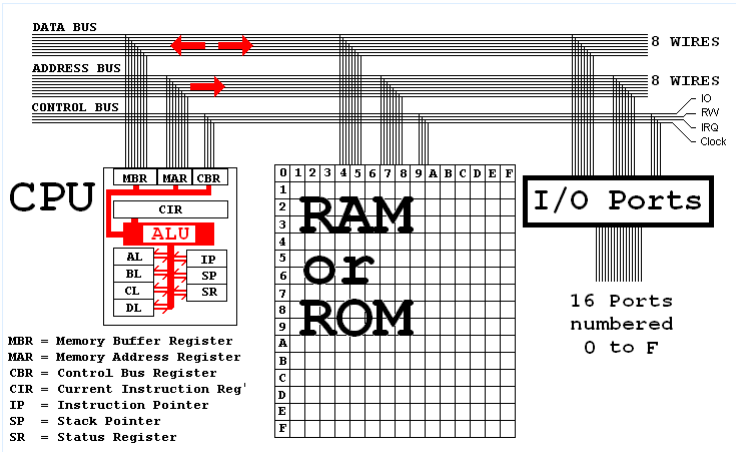
Ce simulateur simule

- une CPU à 8 bits qui est similaire aux huit bits de poids faible de la famille 80x86
- 256 octets de RAM sont simulées.
- 16 ports entrées-sorties dont les périphériques simulés sont sur les ports de 0 à 5.
- Langage de programmation : assembleur
- Lancements : étapes par étapes ou en continu
- Interrupt 02 est déclenché par une horloge matérielle (simulé) dont la vitesse peut être modifiée.
- Clavier synchronise l' Interrupt 03

Le simulateur est distribué sous licence GNU / GPL rend librement à la disposition des étudiants et des établissements d'enseignement

Microprocessor Simultor

architecture



Microprocessor Simultor

CPU

- Quatre registres : AL(code machine en hex 00), BL(01), CL(02) et DL(03) de huit bits
- Nombres non signées de 0 à 255
- Nombres signés de -128 à +127
- Trois registres IP, SR et SP
- Registre SR contient des indicateurs qui rendent compte de l'état de la CPU
  - Flag Z = 1 si le calcul donne un résultat nul.
  - Flag S = 1 si le calcul a donné un résultat négatif.
  - Flag O = 1 si le résultat était trop gros pour tenir dans un registre.
  - Flag I = 1 si les interruptions sont activées. Voir CLI et des IST.
- Le pile du simulateur commence à l'adresse BF juste en dessous de la RAM utilisée pour l'affichage vidéo. La dimension du pile grandit vers l'adresse zéro

RAM

- 256 octets
- Adresses de [00] à [FF] en hexadécimal
- Appel des adresses par le nombre en hexadécimal

Microprocessor Simultor

Périphériques

Keyboard
Port 07
Interrupt INT 03

- Pour rendre le clavier visible, utiliser OUT 07.
- Chaque fois qu'une touche est pressée, une interruption matérielle, INT 03 est généré. Par défaut, la CPU va ignorer cette interruption.
- Pour traiter l'interruption, au début du programme, utilisez la commande STI pour définir le Flag «I» dans le registre d'état SR. Utilisez CLI pour effacer l'indicateur Flag «I» .
- Placez un vecteur d'interruption à l'adresse 03 de la RAM.
- Cela doit pointer vers le code de votre gestionnaire d'interruption. Le gestionnaire d'interruption utilisera IN 07 pour lire le communiqué clé dans le registre AL.

Une fois STI a mis 1 au Flag « I » dans le registre d'état (SR), des interruptions du temps seront également générées. Ces interruptions génèrent INT 02. Pour traiter cette interruption, placez un vecteur d'interruption à l'emplacement 02 de la RAM. Cela doit pointer vers le code du gestionnaire d'interruption de la minuterie. Le code de la minuterie peut être aussi simple que IRET. Cela entraînera un retour d'interruption sans faire tout autre traitement.

Microprocessor Simultor

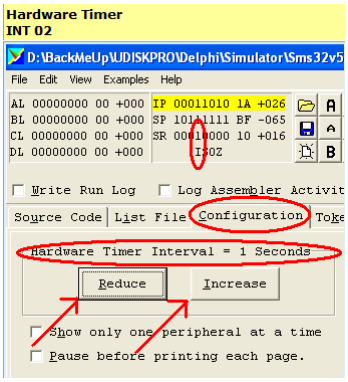
Périphériques

Numeric keypad
Port 08
Interrupt INT 04

- Pour rendre le clavier visible, utiliser OUT 08.
- Chaque fois qu'une touche est pressée, une interruption matérielle, INT 04 est généré. Par défaut, la CPU va ignorer cette interruption.
- Pour traiter l'interruption, au début du programme, utilisez la commande STI pour définir le Flag «I» dans le registre d'état SR. Utilisez CLI pour effacer l'indicateur Flag «I» .
- Placez un vecteur d'interruption à l'adresse 04 de la RAM.
- Cela doit pointer vers le code de votre gestionnaire d'interruption. Le gestionnaire d'interruption utilisera IN 08 pour lire le communiqué clé dans le registre AL.

Microprocessor Simultor

Périphériques



Hardware Timer
Interrupt INT 02

Le temporisateur matériel génère INT 02 à des intervalles de temps réguliers .

L'intervalle de temps peut être modifié en utilisant l'onglet Configuration , comme indiqué dans l'image

Par défaut CPU ignorera INT 02. Il faut mettre à 1 le Flag «I »

Si l'horloge est trop lent , une nouvelle INT 02 peut se produire avant que la précédente a été traitée . Ce n'est pas nécessairement un problème tant que le CPU finalement rattrape . Pour permettre que cela fonctionne, il est essentiel que le gestionnaire d'interruption sauvegarde et restaure tous les registres qu'il utilise. Utilisez PUSH et PUSF pour sauver registres . Utilisez POPF et POP pour restaurer les registres .

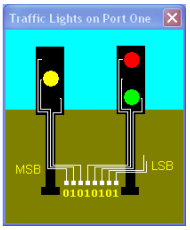
Si le CPU est trop lent et ne rattrape pas , la pile va progressivement grandir et envahir toute la RAM disponible . Finalement, la pile va écraser le programme causer un accident . Un vrai problème

Microprocessor Simultor

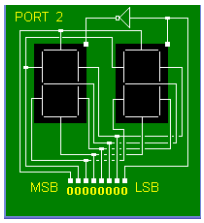
Périphériques

Le Visual Display Unit (VDU) affiche le text selon le code ASCII sur 16 colonnes et 4 lignes. Les positions de l'écran correspondent aux emplacements de RAM de C0 à FF.

Les feux de circulation (Traffic Lights) sont connectés à Port 01. Si un octet de données est envoyé à ce port, là où il y a un, le feu de circulation correspondant s'allume.



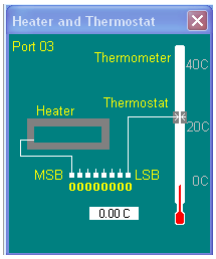
Les sept segments d'affichage (Seven Segment display) sont connectés à Port 02. Si un octet de données est envoyé, le segment correspondant à 1 est allumé. Si le bit le moins significatif (LSB) est égal à zéro, les segments de gauche sera active. Si le bit le moins significatif (LSB) est l'un, les bons segments seront actifs. Voici un extrait de code.



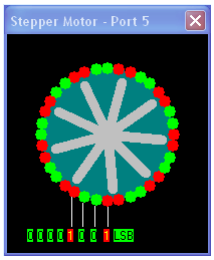
Microprocessor Simultor

Périphériques

Le système de chauffage-thermostat (Heater and Thermostat) est connecté au port 03.
Envoyer 00 au port 3 pour éteindre le radiateur.
Envoyer 80 vers le port 03 pour démarrer le chauffe sur.



Moteur pas à pas (Stepper Motor) est connecté au port 05



Microprocessor Simultor

jeu d'instructions

Move Instructions. Flags NOT set.			
Assembler	**Machine Code**		**Explanation**
MOV AL,15	**D0 00 15**	AL = 15	**Copy 15 into AL**
MOV BL,[15]	**D1 01 15**	BL = [15]	**Copy RAM[15] into BL**
MOV [15],CL	**D2 15 02**	[15] = CL	**Copy CL into RAM[15]**
MOV DL,[AL]	**D3 03 00**	DL = [AL]	**Copy RAM[AL] into DL**
MOV [CL],AL	**D4 02 00**	[CL] = AL	**Copy AL into RAM[CL]**

Microprocessor Simultor

jeu d'instructions

Ex 16 : 03MOVE.ASM

- Observation des chargements direct et indirect
- Ecrire le programme qui affiche les lettres H, E, L, L et O sur l'afficheur VDU sachant que ces lettres doivent être écrire dans les mémoires RAM de [C0], [C1], [C2], [C3] et [C4]

Direct Arithmetic and Logic. Flags are set.		
Assembler	Machine Code	
ADD AL,BL	A0 00 01	AL = AL + BL
SUB BL,CL	A1 01 02	BL = BL - CL
MUL CL,DL	A2 02 03	CL = CL * DL
DIV DL,AL	A3 03 00	DL = DL / AL
INC DL	A4 03	DL= DL + 1
DEC AL	A5 00	AL = AL - 1
AND AL,BL	AA 00 01	AL = AL AND BL
OR CL,BL	AB 03 02	CL = CL OR BL
XOR AL,BL	AC 00 01	AL = AL XOR BL
NOT BL	AD 01	BL = NOT BL
ROL AL	9A 00	Rotate bits left. LSB = MSB
ROR BL	9B 01	Rotate bits right. MSB = LSB
SHL CL	9C 02	Shift bits left. Discard MSB.
SHR DL	9D 03	Shift bits right. Discaed LSB.

Immediate Arithmetic and Logic. Flags are set.

Assembler	Machine Code	
ADD AL,12	**B0 00 12**	AL = AL + 12
SUB BL,15	**B1 01 15**	BL = BL - 15
MUL CL,03	**B2 02 03**	CL = CL * 3
DIV DL,02	**B6 03 02**	DL = DL / 2
AND AL,10	**BA 00 10**	AL = AL AND 10
OR CL,F0	**BB 02 F0**	CL = CL OR F0
XOR AL,AA	**BC 00 AA**	AL = AL XOR AA

Microprocessor Simultor

jeu d'instructions

Ex 17 : 04INCJMP.ASM

objectif

- Utilisation du registre BL
- Bits " S " (signé), "Z" (zéro) et " O " (overload) du registre SR

travail à effectuer

- écrire le programme pour décompter la valeur préenregistrer dans BL → observer les bits "Z" et "S"
- écrire le programme pour calculer coefficients de Fibonnaci en utilisant les mémoires RAM → observer bit "O" en choisissant la simulation par "Step"

Microprocessor Simultor

jeu d'instructions

Input Output Instructions. Flags NOT set.		
Assembler	**Machine Code**	**Explanation**
IN 07	F0 07	Data input from I/O port 07 to AL.
OUT 01	F1 01	Data output to I/O port 07 from AL.

Compare Instructions. Flags are set.		
Assembler	**Machine Code**	**Explanation**
CMP AL,BL	**DA 00 01**	Set 'Z' flag if AL = BL. Set 'S' flag if AL < BL.
CMP BL,13	**DB 01 13**	Set 'Z' flag if BL = 13. Set 'S' flag if BL < 13.
CMP CL,[20]	**DC 02 20**	Set 'Z' flag if CL = [20]. Set 'S' flag if CL < [20].

Microprocessor Simultor

jeu d'instructions

Branch Instructions. Flags NOT set.

Depending on the type of jump, different machine codes can be generated. Jump instructions cause the instruction pointer (IP) to be altered. The largest possible jumps are +127 bytes and -128 bytes.
The CPU flags control these jumps.
The 'Z'flag is set if the most recent calculation gave a Zero result.
The 'S' flag is set if the most recent calculation gave a negative result.
The 'O' flag is set if the most recent calculation gave a result too big to fit in the register.

Assembler	**Machine Code**	**Explanation**
JMP HERE	**C0 12** **C0 FE**	Increase IP by 12 Decrease IP by 2 (twos complement)
JZ THERE	**C1 09** **C1 9C**	Increase IP by 9 if the 'Z' flag is set. Decrease IP by 100 if the 'Z' flag is set.
JNZ A_Place	**C2 04** **C2 F0**	Increase IP by 4 if the 'Z' flag is NOT set. Decrease IP by 16 if the 'Z' flag is NOT set.
JS STOP	**C3 09** **C3 E1**	Increase IP by 9 if the 'S' flag is set. Decrease IP by 31 if the 'S' flag is set.
JNS START	**C4 04** **C4 E0**	Increase IP by 4 if the 'S' flag is NOT set. Decrease IP by 32 if the 'S' flag is NOT set.
JO REPEAT	**C5 09** **C5 DF**	Increase IP by 9 if the 'O' flag is set. Decrease IP by 33 if the 'O' flag is set.
JNO AGAIN	**C6 04** **C6 FB**	Increase IP by 4 if the 'O' flag is NOT set. Decrease IP by 5 if the 'O' flag is NOT set.

Microprocessor Simultor

jeu d'instructions

Ex 18 : 05KEYBIN.ASM

objectif

- acquisition avec l' élément extérieur : Clavier

travail à effectuer

- Ecrire le programme pour enregistrer le text dans le RAM grâce au clavier et afficher ensuite le text dans l'écran VDU une fois qu'on a tapé "Entrer"
- Comme exercice ci-dessus mais on affiche le text dans le sens inverse. Utilisation de la pile est conseillée.

Microprocessor Simultor

jeu d'instructions

Procedures and Interrupts. Flags NOT set.

CALL, RET, INT and IRET are available only in the registered version.

Assembler	Machine Code	Explanation
CALL 30	**CA 30**	Save IP on the stack and jump to the procedure at address 30.
RET	**CB**	Restore IP from the stack and jump to it.
INT 02	**CC 02**	Save IP on the stack and jump to the address (interrupt vector) retrieved from RAM[02].
IRET	**CD**	Restore IP from the stack and jump to it.

Stack Manipulation Instructions. Flags NOT set.

Assembler	Machine Code	Explanation
PUSH BL	**E0 01**	BL is saved onto the stack.
POP CL	**E1 02**	CL is restored from the stack.
PUSHF	**EA**	SR flags are saved onto the stack.
POPF	**EB**	SR flags are restored from the stack.

Microprocessor Simultor

jeu d'instructions

Miscellaneous Instructions. CLI and STI set I flag.		
Assembler	**Machine Code**	**Explanation**
CLO	FE	Close visible peripheral windows.
HALT	00	Halt the processor.
NOP	FF	Do nothing for one clock cycle.
STI	FC	Set the interrupt flag in the Status Register.
CLI	FD	Clear the interrupt flag in the Status Register.
ORG 40	Code origin	Assembler directive: Generate code starting from address 40.
DB "Hello"	Define byte	Assembler directive: Store the ASCII codes of 'Hello' into RAM.
DB 84	Define byte	Assembler directive: Store 84 into RAM.

Microprocessor Simultor

jeu d'instructions

Ex 19 : 06PROC.ASM
objectif → procédure
travail à effectuer
Modifier le programme pour contrôler le temps de deux feux tricolores

Ex 20 : 08TABLE.ASM
objectif → déclaration du tableau
travail à effectuer
Ecrire le programme pour allumer correctement les deux feux tricolores sans le contrôle de temps
(F1:F2) : (R:V)→(R,J:J)→(V:R)→(J:R,J)→(R:V)→ (R,J:J)→(V:R)→…

Ex 21 : 09PARAM.ASM
objectif → pile
travail à effectuer
Ecrire le programme qui calcule le factoriel de N en utilisant le procédure et la pile. Le résultat sera enregistré dans un registre

Ex 22 : 11HWINT.ASM
objectif → interruption
travail à effectuer
Concevoir un chronomètre de 3 boutons
B pour démarrer et continuer,
A pour arrêter
C pour remettre à zéro)
en utilisant le clavier numérique et afficheur de 7 segments

Programmer en C

Historique

- Le langage C a été développé en 1972 par Denis Ritchie et Ken Thompson des laboratoires Bell
- Système d'exploitation UNIX
- Le C est normalisé en 1988 par l'American National Standard Institute

⇒Langage de bas niveau qui permet de comprendre et de maîtriser les mécanismes d'accès à la mémoire et de communication avec le système d'exploitation

logiciel utilisé

- CodeBlocks
- Document à étudier pour démarrer :

http://wiki.codeblocks.org/index.php?title=Creating_a_new_project

Premier exemple

```
#include <stdio.h>-
#include <stdlib.h>

int main()

{
   char nom[100];
   int annee;

   printf("Quel est votre nom ? \n");
   scanf("%s",nom);
   printf("Et votre année de naissance ? \n");
   scanf("%d",&annee);
   printf("Bonjour %s, vous avez %d ans !\n", nom, 2014-annee);


   return EXIT_SUCCESS;
}
```

Premier exemple

```
#include <stdio.h>- Inclure les bibliothèques
#include <stdlib.h>          nécessaires

int main() L'exécution d'un programme commence par l'appel de sa fonction main()

{Début
   char nom[100]; chaîne de 100 caractères      Réserver l'emplacement pour
   int annee; donnée « entier »                 deux données en mémoire

   printf("Quel est votre nom ? \n"); écriture sur l'écran → fonction printf
   scanf("%s",nom); lecture au clavie → scanf   l'identificateur « nom » présente l'adresse de la zone
   printf("Et votre année de naissance ? \n");
   scanf("%d",&annee);          l'identificateur « annee » désigne la donnée dont l'adresse est &annee
   printf("Bonjour %s, vous avez %d ans !\n", nom, 2014-annee);
écriture sur l'écran avec les données nom et nombre d'année → demande les formats des données %s,
%d
   return EXIT_SUCCESS; retourner le macro qui indique le succès du programe
} Fin
```

Bibliothèques

- <assert.h> : pour un diagnostic de conception lors de l'exécution (assert)
- <ctype.h> : tests et classification des caractères (isalnum, tolower)
- <errno.h> : gestion minimale des erreurs (déclaration de la variable errno)
- <math.h> : fonctions mathématiques de base (sqrt, cos) ; nombreux ajouts en C99
- <signal.h> : gestion des signaux (signal et raise)
- <stddef.h> : définitions générales (déclaration de la constante NULL)
- <stdio.h> : pour les entrées/sorties de base (printf, scanf)
- <stdlib.h> : fonctions générales (malloc, rand)
- <string.h> : manipulation des chaînes de caractères (strcmp, strlen)
- <time.h> : manipulation du temps (time, ctime)

Fonction printf()

Spécificateurs :
%d → un nombre entier en décimal
%s → une chaîne de caractères
%c → le caractère correspondant à un nombre (ex : « A » a pour valeur 65)
%f → un nombre en virgule flottante (float ou double)
%p → un pointeur
%u → un nombre non signé
%x →un nombre entier en hexadécimal
%u → un nombre entier en octal

Format : spécificateur peut être précédé d'indicateurs et de champs pour préciser la présentation voulue
%4d → affichage d'un nombre sur au moins 4 caractères. Ex : «42» → « 42»
%04d → un nombre sur 4 chiffres avec les zéros en tête. Ex : «42» → «0042»
%10s → une chaîne cadrée à gauche sur 10 caractères. Les caractères manquantes sont remplacés par des espaces
%-10s→ cadrage à droite
%10.3f→un nombre réel sur 10 caractères dont 3 après la virgule

Longeur de la donnée

hh	pour les char (signés ou pas)
h	pour les short int
l	pour les long int
ll	pour les long long int
z	pour les size_t

Exemple : %lc : pour les caractères larges (wchar_t) %ls : chaîne de caractères larges (chaîne de caractères «multibyte» UTF-8

Paramètres de la ligne de commande

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int i;
   printf("Ceci est mon programme %s avec %d arguments. \n", argv[0], argc);

   for( i=0;i<argc;i++)
   printf("argv[%d] = %s\n", i, argv[i]);

   return EXIT_SUCCESS;
}
```

- Lancer : à partir de windows
Démarrer/Tous programme/Accessoires/Invite de commandes

- Entrer dans la répertoire qui contient le fichier *.exe
  Regarder dans Build log de Code::blocks une fois compiler

- Lancer *.exe bienvenue chez les programmeurs C [Entrée]

Paramètres de la ligne de commande

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int i;
   printf("Ceci est mon programme %s avec %d arguments. \n", argv[0], argc);

   for( i=0;i<argc;i++)
   printf("argv[%d] = %s\n", i, argv[i]);

   return EXIT_SUCCESS;
}
```

```
C:\Users\pham\code_block_pham\PUF\bin\Debug>puf.exe bienvenue chez les programme
urs C
Ceci est mon programme puf.exe avec 6 arguments.
argv[0] = puf.exe
argv[1] = bienvenue
argv[2] = chez
argv[3] = les
argv[4] = programmeurs
argv[5] = C
```

Opérateurs

Opérateur numérique	Dénomination	Effet
+	opérateur d'addition	Ajoute deux valeurs
-	opérateur de soustraction	Soustrait deux valeurs
*	opérateur de multiplication	Multiplie deux valeurs
/	opérateur de division	Divise deux valeurs
%	opérateur modulo	Donne le reste de la division entière
Opérateur d'assignation		
=	Affecte une valeur (à droite) à une variable (à gauche)	
+=	additionne deux valeurs et stocke la somme dans la variable	
-=	soustrait deux valeurs et stocke la différence dans la variable	
*=	multiplie deux valeurs et stocke le produit dans la variable	
/=	divise deux valeurs et stocke le quotient dans la variable	
%=	divise deux valeurs et stocke le reste dans la variable	
Opérateur d'incrémentation		
++	Incrémentation	Augmente d'une unité la variable
--	Décrémentation	Diminue d'une unité la variable

Opérateurs

Opérateur de comparaison	Dénomination	Effet
==	opérateur d'égalité	Compare deux valeurs et vérifie leur égalité
<	opérateur d'infériorité stricte	Vérifie qu'une variable est strictement inférieure à une valeur
<=	opérateur d'infériorité	Vérifie qu'une variable est inférieure ou égale à une valeur
>	opérateur de supériorité stricte	Vérifie qu'une variable est strictement supérieure à une valeur
>=	opérateur de supériorité	Vérifie qu'une variable est supérieure ou égale à une valeur
!=	opérateur de différence	Vérifie qu'une variable est différente d'une valeur
Opérateur logique		
||	OU logique	Vérifie qu'une des conditions est réalisée
&&	ET logique	Vérifie que toutes les conditions sont réalisées
!	NON logique	Inverse l'état d'une variable booléenne
Opérateur bit-à-bit		
&	ET bit-à-bit	Retourne 1 si les deux bits de même poids sont à 1 Ex : 9 & 12 (1001 & 1100)→8 (1000)
|	OU bit-à-bit	Retourne 1 si l'un ou l'autre des deux bits de même poids est à 1 (ou les deux) Ex : 9 | 12 (1001 | 1100→13 (1101)
^	OU bit-à-bit exclusif	Retourne 1 si l'un des deux bits de même poids est à 1 (mais pas les deux) Ex : 9 ^ 12 (1001 ^ 1100)→5 (0101)

Opérateurs

Opérateur de décalage de bit	Dénomination	Effet
<<	Décalage à gauche	Décale les bits vers la gauche (multiplie par 2 à chaque décalage). Les zéros qui sortent à gauche sont perdus, tandis que des zéros sont insérés à droite Ex : 6 << 1 (110 << 1) → 12 (1100)
>>	Décalage à droite avec conservation du signe	Décale les bits vers la droite (divise par 2 à chaque décalage). Les zéros qui sortent à droite sont perdus, tandis que le bit non nul de poids plus fort est recopié à gauche. Ex : 6 >> 1 (0110 >> 1)→3 (0011)

Priorité des opérateurs											
++++++++++++	()	[]									
++++++++++++	--	++	!	~	-						
++++++++++	*	/	%								
++++++++++	+	-									
+++++++++	<<	>>									
++++++++	<	<=	>=	>							
+++++++	==	!=									
++++++	&										
+++++	^										
++++	|										
+++	&&	||									
++	?	:									
+	=	+=	-=	*=	/=	%=	<<=	>>=	&=	^=	|=

Structure alternative

Les structures de contrôle définissent la suite dans laquelle les instructions sont effectuées.

```
if ( <expression> )
            <bloc d'instructions 1>
else        <bloc d'instructions 2>
```

<expression> → variable d'un type numérique
→une expression fournissant un résultat numérique
<bloc d'instructions>
→un bloc d'instructions compris entre accolades
→une seule instruction terminée par « ; »
La partie « else » est facultative

```
if ( <expr1> ) <bloc1>
else if (<expr2>) <bloc2>
    else  if (<expr3>) <bloc3>
        else  if (<exprN>) <blocN>
            else <blocN+1>
```

Les opérateurs conditionnels
<expr1> ? <expr2> : <expr3>
- *Si <expr1> fournit une valeur différente de zéro, alors la valeur de <expr2> est fournie comme résultat*
- *Si <expr1> fournit la valeur zéro, alors la valeur de< expr3> est fournie comme résultat*

Exemple
La suite d'instructions
if (A>B) MAX=A; else MAX=B;
peut être remplacée par:
MAX = (A > B) ? A : B;

Structures répétitives

Les boucles, dans un programme, permettent de répéter certaines instructions plusieurs fois sans avoir à recoder plusieurs fois ces instructions. En C il existe trois types de boucles, nous parlerons de chacune d'elle. L
•for
•while
•do / while

1. La boucle for teste une condition avant d'exécuter les instructions qui lui sont associées.
   for (expression1 ; expression2 ; expression3) { instructions à réaliser }
   expression1 sera une initialisation d'une variable
   expression2 sera justement la condition pour que la boucle s'exécute.
   expression3 modifier la valeur de la variable de contrôle à chaque passage de la boucle.

2. La boucle while, tout comme la boucle for ne permet l'exécution d'instructions que si une condition vérifiée, avant l'exécution des instructions, est vrai (TRUE).
   while (condition) { instructions(s) à réaliser}

3. La boucle do / while diffère des autres dans le sens où les instructions qui lui sont rattachées s'exécutent au moins une fois.
   do { instruction(s) à réaliser }while (condition);

Structures répétitives

Exemple : un programme qui se chargera d'afficher 3 fois le terme "Hello world!"

1. La boucle for :

for (expression1 ; expression2 ; expression3)
{ instructions à réaliser }

```
int main()
{
int i = 0; /* voilà notre variable de contrôle */
for ( i = 0 ; i < 3 ; i++)
                { printf("Hello World !\n"); }
 return 0;
 }
```

2. La boucle while :

while (condition) { instructions(s) à réaliser}

```
int main()
{ int i = 0;
 while ( i < 3 )
        { printf("Hello World !\n");
          i++; }
return 0;
 }
```

3. La boucle do / while

do { instruction(s) à réaliser }while (condition);

```
int main()
{ int i = 0;
do   { printf("Hello World !\n");
        i++; }
while ( i < 3 );
 return 0;
 }
```

Boucle infinie

```
while (1)
{ printf("Boucle infinie !"); } /* 1 est toujours vrai */

for ( i = 0 ; i >-1 ; i++)
{ printf("Boucle infinie !"); } /* i vaut 0 au départ donc
sera TOUJOURS supérieur a -1 dans cette boucle ! */

do { printf("Boucle infinie !"); }
while (4 < 5); /* 4 est toujours inférieur à 5 */
```

Structures répétitives – exercices

Ex 23 :

Ecrire le programme qui vérifie si un nombre est premier, et affiche le résultat à l'écran.

Ce nombre, une variable nommée *number*, qui sera acquis au clavier par la fonction scanf.

Information :

•un nombre x n'est pas divisible par un nombre y si x % y est différent de zéro → fonction %
•un nombre premier est divisible uniquement par 1 et par lui-même.

Ex 24 :

Ecrire le programme pour calculer la somme de tous les nombres compris entre 1 et N.

Information :

la somme sera 1+2+3+...+(N−2)+(N−1)+N .

Structures répétitives – exercices

Ex 25 :

Ecrire un programme qui affiche la table de multiplication

		1	2	3	4	5	…	9
		--	--	--	--	--		--
1	:	1	2	3	4	5	…	9
2	:	2	4	6	8	10		18
…		…..						
9	:	9	18	27	36	45	…	81

Ex 26 :

Calcul du PGCD de deux nombres

Le PGCD de deux nombres a et b est le plus grand nombre qui peut diviser à la fois a et b.

Remarque :

En supposant a(dividende) > b (diviseur)

si a = b.q0 → pgcd = b

si a = b.q0+r0

et b = r0.q1→a = r0.q1.q0+r0→a = r0 (q0.q1+r1)→ pgcd = r0

si a = b.q0+r0

et b = r0.q1+r1

r0 = r1.q2→b = r1(q1.q2+1)

et a = (r0.q1+r1).q0+r0 = r0.q1.q0+r1.q0+r0 = r0(q1.q0+1)+r1.q0 = r1.q2(q1.q0+1)+r1.q0

= r1(q2.q1.q0+q2+q0)

→ pgcd = r1

donc pgcd est le dernier diviseur de la division entière où le reste vaut 0.

Branchements inconditionnels

- Les instructions qui vont faire reprendre notre programme
  - à un autre endroit si une condition est vraie, et qui ne font rien sinon.
  - à un endroit bien précis, quelle que soit la situation.

- Il existe ainsi trois grands branchements inconditionnels en C :
  - continue → permet de passer au tour de boucle suivant, sans finir celui en cours
  - break → permet (entre autres) de quitter la boucle en cours
  - goto → permet de sauter carrément dans un autre morceau de code

Programme :

```
int i;
for (i = 0 ; i < 10 ; i++)
{ if (i == 5)  break;
  printf("i = %d\n", i);
}
```

Affichage :

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

Programme :

```
int i;
for (i = 0 ; i < 10 ; i++)
{if (i == 5)       continue;
 else printf("i = %d\n", i);
}
```

Affichage :

```
i = 0
i = 1
i = 2
i = 3
i = 4
 i = 6
i = 7
i = 8
i = 9
```

Programme :

```
int i;
for (i = 0 ; i < 10 ; i++)
{ if (i == 5) goto Erreur;
printf("i = %d\n", i);
}
Erreur:
puts("\ni vaut 5");
```

Affichage :

```
i = 0
i = 1
i = 2
i = 3
i = 4

i vaut 5
```

Procédure avec passage de paramètres

```
#include <stdio.h>
#include <stdlib.h>
int somme(const int x, const int y)
{
   return x+y;
}
int main(int argc, char *argv[])
{
   int a,b ;
   printf ("Donnez deux nombres : ");
   scanf ("%d %d" , &a, &b);
   printf("la somme %d + %d vaut %d \n", a, b, somme(a,b));
   return EXIT_SUCCESS;
}
```

- Définition de la fonction somme() est comme printf() ou main()

- La fonction scanf() sert à la lecture des données a et b
- Comme printf(), elle demande une chaîne de format de lecture et une indication des variables à remplir

- La fonction scanf() utilise un passage par adresse → &a, &b

Adresses et pointeurs

Les ***pointeurs*** sont des variables susceptibles de contenir des adresses. On les déclare en précédant leur nom par une étoile.

```
int a, *pa;                    pa : pointeur vers un entier
char message[10], *ptr;        ptr : pointeur vers des caractères
```

Affectation :

```
pa = &a;                  adresse de la variable a est dans la contenue de pa
ptr = & (message[3]); adresse du 3è caractère de la chaîne « message» est dans la contenue de ptr
        Remarque :    la chaîne « message» commence à l'adresse pointée par message
                      le 3è caractère de message est pointé à message+3
```

Remarque :

```
        scanf ("%d %d" , &a, &b);
```

peut être remplacé par :

```
        int pa = &a, pb = &b ;
        scanf ("%d %d" , pa, pb);
```

Déréférencement :

Quand un pointeur contient l'adresse d'une donnée, on peut le déférencer, c-à-d accéder à la donnée pointée par le pointeur directement. Le déférencement, ou indirection, se note par une étoile.

*pa → accéder à la donnée a

Adresses et pointeurs

```
#include <stdio.h>
 #include <stdlib.h>
void saisirDeuxNombres(int *px, int *py)
{
                printf ("Donnez deux nombres : ");
                 scanf ("%d %d" , px, py);
}
void calculerSomme(const int x, const int y, int *pz)
{
                *pz = x+y;
}
void afficherSomme(const int x, const int y, const int z)
{
   printf("la somme %d + %d vaut %d \n", x, y, z);
}
// programme principal-----------------------------------
int main (void)
 {
int a, b ,c;
saisirDeuxNombres(&a, &b);
calculerSomme(a, b, &c);
afficherSomme(a, b, c);
return EXIT_SUCCESS;
}
```

Quand une action doit modifier un de ses paramètres, cette action se traduit par une fonction C qui reçoit comme paramètre l'adresse de la donnée

Adresses et pointeurs - exercice

Ex 27 :

Écrire un programme qui :

- demande trois nombres entiers
- Appelle une fonction qui ordonne ces trois nombres
- Les affiche dans l'ordre (du petit au plus grand)

Chaînes de caractères

- Les chaînes → suites de caractères consécutifs en mémoire terminées par un caractère nul '\0'.
- Elles sont désignées par l'adresse de leur premier caractère

→Une chaîne est un tableau

Exemple : la chaîne 'abc' →une suite de 4 octets : 'a', 'b', 'c' et le caractère nul '\0'

```
char message[] = " Hello,world ! "
for (int i=0; message[i] != '\0'; i++)
{                                          par l'indice
    printf(" %c "  , message [i]);
}
```

```
char *p;
 char message[] = "Hello, world !";
 for (p = message; *p !='\0'; p++)          par les pointeurs
{
   printf(" %c ",*p);
}
```

Chaînes de caractères

Opérations sur les chaînes :
se font par l'intermédiaire de fonctions de bibliothèque <string.h>

- **size_t strlen(const char *s)** *retourne la longueur de la chaîne s. Le type de retour est size_t ce qui correspond à des entiers non signés*
- **char *strcpy(char *dest, const char *src)** *copie les octets de src, jusqu'au caractère nul final (compris), dans dest*
- **char *strcat(char *dest, const char *src**) *détermine la position du caractère nul de c, et copie à partir de là les octets de src jusqu'au caractère nul final (compris)*
- **int strcmp(const char*s1, const char *s2)** *compare les chaînes s1 et s2 et renvoie*
  - *Un nombre négatif si s1 précède s2*
  - *0 si les chaînes sont égales et détermine la position du caractère nul de c*
  - *Un nombre positif si s1 est après s2*

Attention : Les deux fonctions strcpy et strcat demandent la réservation des tableaux assez grande pour y loger le résultat

Opération sur les caractères

Les fonctions de *<ctype.h>*
- servent à classifier et à convertir des caractères. Les symboles nationaux (é, è, ä, ü, ß, ç, ...) ne sont pas considérés.
- sont indépendantes du code de caractères de la machine et favorisent la portabilité des programmes.

Dans la suite, <c> représente une valeur du type **int** qui peut être représentée comme caractère.

Les fonctions de ***classification*** suivantes fournissent un résultat du type **int**
- différent de zéro, si la condition respective est remplie,
- sinon zéro.

La fonction:	retourne une valeur différente de zéro,
isupper(<c>)	si< c> est une majuscule **('A'...'Z')**
islower(<c>)	si< c> est une minuscule **('a'...'z')**
isdigit(<c>)	si< c> est un chiffre décimal **('0'...'9')**
isalpha(<c>)	si **islower(<c>)** ou **isupper(<c>)**
isalnum(<c>)	si **isalpha(<c>)** ou **isdigit(<c>)**
isxdigit(<c>)	si< c> est un chiffre hexadécimal ('**0**'...'**9**' ou '**A**'...'**F**' ou '**a**'...'**f**')
isspace(<c>)	si< c> est un signe d'espacement (' ', '**\t**', '**\n**', '**\r**', '**\f**')

Les fonctions de ***conversion*** suivantes fournissent une valeur du type **int** qui peut être représentée comme caractère, la valeur originale de <c> reste inchangée:

tolower(<c>)	retourne< c> converti en minuscule si <c> est une majuscule
toupper(<c>)	retourne< c> converti en majuscule si <c> est une minuscule

Opération sur les caractères - exercice

Ex 28

Ecrire un programme qui
- lit une chaîne de caractères "Hello, World !"
- convertit toutes les majuscules dans des minuscules et vice-versa
- affiche la chaîne obtenue

Lecture et écriture dans des fichiers de textes

fscanf → lire le fichier texte

fscanf(<FP>, "<Form1>\n", <Adr1>);

ou bien

fscanf(<FP>,"<Form1>\n<Form2>\n...\n<FormN>\n", <Adr1>, <Adr2>, ... , <AdrN>);

fprintf → écrire dans un fichier texte

fprintf(<FP>, "<Form1>\n", <Expr1>);

ou bien

fprintf(<FP>,"<Form1>\n<Form2>\n...\n<FormN>\n", <Expr1>, <Expr2>, ... , <ExprN>);

- **<FP>** est un pointeur du type **FILE*** qui est relié au nom du fichier cible.
  FILE *<FP> = fopen (" <NOM> ", " <mode d'accès> ");
  - déclarer un pointeur du type **FILE*** pour chaque fichier dont nous avons besoin,
  - affecter l'adresse retournée par **fopen** à ce pointeur
  - employer le pointeur à la place du nom du fichier dans toutes les instructions de lecture ou d'écriture,
  - libérer le pointeur à la fin du traitement à l'aide de **fclose (<FP>);**
  - <mode d'accès> au fichier : " w " pour écriture (write) et " r" pour lecture (read)
- **<Form1>, <Form2>, ... , <FormN>** →les spécificateurs de format pour l'écriture et pour la lecture
- **<Adr1>, <Adr2>, ... , <AdrN>** →les adresses des variables qui reçoivent les valeurs lues à partir du fichier.
- **<Expr1>, <Expr2>, ... , <ExprN>** → les valeurs respectives à écrire dans le fichier.

Lecture et écriture dans des fichiers de textes

Lors de la fermeture d'un fichier ouvert en écriture, la fin du fichier est marquée automatiquement par le symbole de fin de fichier **EOF** *(End Of File).*
Lors de la lecture d'un fichier, la fonction **feof(<FP>)** nous permettent de détecter la fin du fichier:

Elle retourne une valeur différente de zéro, si la tête de lecture du fichier référencé par <FP> est arrivée à la fin du fichier, sinon la valeur du résultat est zéro.
Pour que la fonction **feof** détecte correctement la fin du fichier, il faut qu'après la lecture de la dernière donnée du fichier, la tête de lecture arrive jusqu'à la position de la marque **EOF**.
Nous obtenons cet effet seulement si nous terminons aussi la chaîne de format de **fscanf** par un retour à la ligne '\n' (ou par un autre signe d'espacement).

Lecture et écriture dans des fichiers de textes

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   float somme = 0,0;
   int nombre = 0;
   float note;

   FILE *fnotes = fopen("notes.txt", "r");

   while ( fscanf (fnotes, "%f", &note) == 1)
                {somme += note;
                nombre +=1;
                }
   fclose (fnotes);
   printf ("%d notes lues, total = %.2f, moyenne = %05.2f\n", nombre, somme, somme/nombre);


   return EXIT_SUCCESS;
}
```

Lecture et écriture dans des fichiers de textes

Exemple :

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   float somme = 0,0;
   int nombre = 0;
   float note;

   FILE *fnotes = fopen("notes.txt", "r");    //fichier notes.txt est ouvert pour la lecture

   while ( fscanf (fnotes, "%f", &note) == 1) // la boucle est répétée tant que fscanf() réussit à lire un nombre sur fnotes
                {somme += note;
                nombre +=1;
                }
   fclose (fnotes); //libérer le pointeur
   printf ("%d notes lues, total = %.2f, moyenne = %05.2f\n", nombre, somme, somme/nombre);
   // affichage de la somme se fait avec 2 chiffres après la virgule, celui de la moyenne sur 5 chiffres, dont 2 après la
   virgule, avec un ou plusieurs zéros non significatifs en tête
   return EXIT_SUCCESS;
}
```

fscanf(stdin,…) → scanf(…) fprintf(stdout,…) → printf(…)
FILE *stdin, *stdout //correspondent aux flots d'entrée et de sortie standards
la sortie d'erreur est *stderr*

Saisie de caractère et de chaîne

#include <stdio.h>

int fgetc (FILE *_stream_);
char * fgets (char * _s_, int _size_, FILE * _stream_);
int getc (FILE * _stream_);
int getchar (void);
char * gets (char * _s_);
int ungetc (int _c_, FILE * _stream_);

DESCRIPTION
fgetc() lit le caractère suivant depuis le flux *stream*
renvoie sous forme d'un **unsigned char**, transformé en **int**, ou **EOF** en cas d'erreur ou de fin de fichier.
getc() est équivalent à **fgetc**() sauf qu'il peut être implementé sous forme de macro, qui évalue l'argument *stream* plusieurs fois.
getchar() est équivalent à **getc(***stdin***).**

fgets() lit (*size – 1)* caractères depuis *stream* et les place dans le tampon pointé par *s*
elle renvoit **NULL** en cas d'erreur, ou si la fin de fichier est atteinte avant d'avoir pu lire au moins un caractère
La lecture s'arrête après **EOF** ou un retour-chariot.
Si un retour-chariot (newline) est lu, il est placé dans le tampon. Un caractère nul « **\0** » est placé à la fin de la ligne.
gets() lit une ligne depuis *stdin* et la place dans le tampon pointé par *s* jusqu'à atteindre un retour-chariot, ou **EOF**, qu'il remplace par « **\0** ».
Il n'y a pas de vérification de débordement de tampon.
ungetc() replace le caractère *c* dans le flux *stream*, en le transformant en **unsigned char**, ou **EOF** en cas d'erreur

Remarque
UTILISEZ TOUJOURS **fgets**() À LA PLACE DE **gets**() à cause du contrôle de nombres de caractères

Saisie de caractère et de chaîne - exercice

Ex 29 :
Ecrire un programme qui fait afficher, ligne par ligne, un fichier texte dont le nombre est passé en paramètre. Les lignes apparaîtront numérotées dur 4 chiffres, en commençant par 0001.
Prévoir le cas des lignes trop longues qui apparaîtront sans numérotation.

0001 ceci est une première ligne
0002 ceci est une seconde ligne qui e
st trop longue pour apparaître e
n une seule fois
0003 ceci est la troisième ligne
…

Structure

Une structure est un mécanisme permettant de grouper un certain nombre de variables de types différents au sein d'une même entité → une simple collection de champs

Exemple :
Mon profil doit avoir : nom, prénom,age, sexe, cursus scolaire, travail, loisir,…

Structure

Première méthode
struct personne { char nom[20]; char prenom[20]; int no_employe; };
l'identificateur (ou étiquette de structure) personne →structure composée
d'un tableau nom de 20 caractères, d'un tableau prenom de 20 caractères et d'un entier no_employe
déclarer des variables, de la manière suivante :
struct personne p1,p2; → deux variables de type struct personne de noms p1 et p2;

Deuxième méthode : *déclaration des variables de type structure sans utiliser d'étiquette de structure*
struct { char nom[20]; char prenom[20]; int no_employe; } p1,p2;
Deux variables p1 et p2 comme étant deux structures de trois membres, mais la structuree n'a pas un identificateur.
L'inconvénient de cette méthode est qu'il sera par la suite impossible de déclarer une autre variable du même type.
En effet, si plus loin on écrit : struct { char nom[20]; char prenom[20]; int no_employe; } p3;
→deux structures ont le même nombre de champs, avec les mêmes noms et les mêmes types,
mais elles seront considérées de types différents. Il sera impossible en particulier d'écrire p3 = p1;.

Troisième méthode *combinaison de déclaration d'étiquette de structure et celle de variables*
struct personne { char nom[20]; char prenom[20]; int no_employe; } p1,p2;
déclare les deux variables p1 et p2 et donne le nom personne à la structure.
→utiliser le nom struct personne pour déclarer d'autres variables : struct personne p1, p2, p3;
→la première qui est recommandée, car elle permet de bien séparer la définition du type structure de ses utilisations.

http://c.developpez.com/cours/bernard-cassagne/node80.php#SECTION00821000000000000000

Structure - initialisation

•Une structure peut être **initialisée** par une liste d'expressions constantes à la manière des initialisations de tableau.
Exemple :
struct personne p = {"Jean", "Dupond", 7845};

struct personne p = {→nom = "Jean", →prenom = "Dupond", .no_employe = 7845};

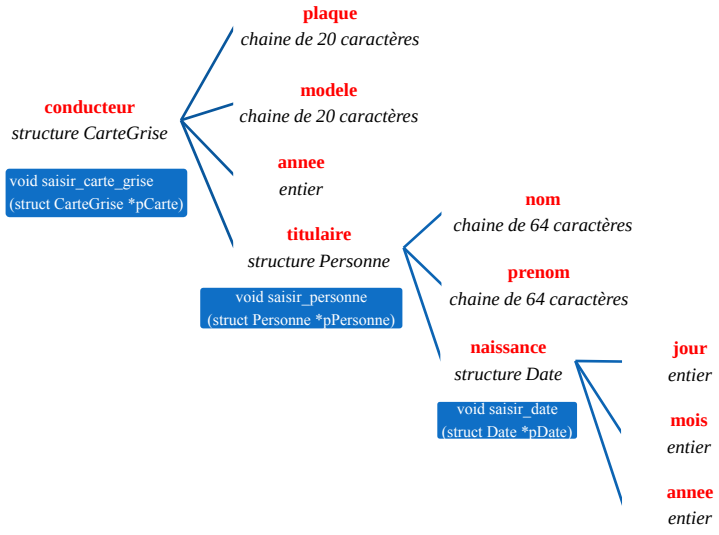
Les champs manquants sont initialisés à 0.

•**Affectation, passage de paramètres**

```
/déclaration de structure
struct Date
{int jour, mois, annee;}
/initialisation
struct Date noel = {.mois = 12, .jour = 25}
/déclaration de variable
struct Date anniversaire ;
/affectation
anniversaire = noel ;
```

Structure - exercice

Saisir la Carte Grise d'un conducteur



Structure - exercice

```
#include <stdio.h>
#include <stdlib.h>

struct Date {int jour, mois, annee;};
struct Personne
{char nom[64],
prenom[64];
 struct Date naissance;};
struct CarteGrise
{char plaque[20],
 modele[20];
 int annee;
struct Personne titulaire;};

void saisir_date(struct Date *pDate)
{
   printf("Jour mois annee ? ");
   scanf("%i", &(pDate->jour));
   scanf("%i", &(pDate->mois));
   scanf("%i", &(pDate->annee));
}
void saisir_personne(struct Personne *pPersonne)
{
   printf("Nom : ");
   scanf("%s",pPersonne->nom);
   printf("Prenom : ");
   scanf("%s",pPersonne->prenom);
   printf("Date de naissance : ");
   saisir_date(&(pPersonne->naissance));
}
```

```
void saisir_carte_grise(struct CarteGrise *pCarte)
{
   printf("numero de plaquessss : ");
   scanf("%s",pCarte->plaque);
   printf("modele : ");
   scanf("%s",pCarte->modele);
   printf("annee : ");
   scanf("%i",&(pCarte->annee));
   saisir_personne(&(pCarte->titulaire));
}
int main()
{
   struct CarteGrise conducteur;
   saisir_carte_grise(& conducteur);
  printf("%s\n", conducteur.titulaire.nom);
   printf("%s\n", conducteur.titulaire.prenom);
   printf ("%i/",conducteur.titulaire.naissance.jour);
   printf ("%i/",conducteur.titulaire.naissance.mois);
   printf ("%i\n",conducteur.titulaire.naissance.annee);
   printf ("%s\n",conducteur.plaque);
   printf ("%s\n",conducteur.modele);
   return 0;
}
```

Structure - union

```
struct  EvenementClavier
{ int type; //type =1
   char touche;
};

struct EvenementSouris
{ int type; //type = 2
int x, y; //coordonnées
int etat; //état des boutons
};

union Instruction {
int type;
struct EvenementClavier ec;
struct EvenementSouris es;
};
```

- un entier appelé type
- EvenementClavier appelé ec
  - → ec.type
  - → ec.touche
- EvenementSouris appelé es
  - → es.type
  - → es.x et es.y
  - → es.etat

```
void traiterEvenement (union Evenement e)
{
switch (e.type)
            {
              case 1 : //evenement souris
                        if (e.ec.touche ==‘\e’ // touche escape
                              SuspendrePartie():
                        else if (e.ec.touche == ‘\n’)
                                    TraiterSaisie();
                              else
                                    AjouterCaractere(e.ec.touche);
              break;
              case 2 : //evenement clavier
                        DeplacerPointeur(e.es.x, e.es.y);
                        ….
              break;
              …
            }
}
```

Une **union** est une donnée en mémoire qui a plusieurs membres. Chaque membre constitue une façon de voir le contenu de cette zone de données.
A noter que le champ type se retrouve, au même endroit, dans les 3 membres.

Structure – union - initialisation

L'initialisation d'une union peut se faire selon la même syntaxe que pour les structures, avec laquelle elle se combine.

```
union Evenement e =     {
                        .ec =   {       .type = 1;
                                        .touche = '@';};
                        }
```

Structure – définition de noms de types

Le mot clé **typedef** permet de définir des noms de types, au lieu de définit des noms de variables

```
struct Point {
   int x, y;
} p, **pp;
```

→variable p de type struct Point
→variable pp de type struct Point *

```
void  tracerTrait (struct Point A, struct Point B)
{…}
```

```
typedef  struct   Point {
                    int x, y;
                  } POINT, *PPOINT;
```

→ POINT est synonyme de struct Point
→PPOINT synonyme de struct Point *

```
void tracerTrait (POINT A, POINT B)
{…}
```

Préprocesseur : usage des macros

➢#include
cette directive sert à inclure le contenu d'un autre fichier
#include <nom de fichier> →pour fichier d'entête de bibliothèques système
#include "nom de fichier" →pour fichier personnel qui est dans le même répertoire que la source. La compilation demande le paramètre –I

➢#ifdef…#endif
cette directive permet de définir une variable par le préprocesseur

```
#include <stdio.h>
#include <stdlib.h>
#define USA 1
//#define EUP 1 : para EUP n'est pas défini

#include <stdio.h>
#ifdef USA
#define currency_rate 46
#endif

#ifdef EUP
#define currency_rate 100
#endif


int main()
{
int rs;
   rs = 10 * currency_rate;
   printf ("%d\n", rs);

   return 0;
}
```

Préprocesseur : usage des macros

➢#define
cette directive est utilisée pour définir des constantes ou de l'ensemble des informations alphanumérique quelconque

```
#define TAILLE_MAXIMUM  1000
#define VERSION "Truc 3.14.16"
#define SI if(
#define ALORS ){
#define SINON } else{
```

Le préprocesseur remplace les identificateurs par la chaîne correspondante qui va jusqu'à la fin de la ligne.
printf("VERSION =%s\n", "Truc 3.14.16");
ou printf("VERSION =%s\n", VERSION);
ou printf("VERSION =" VERSION"\n");

les macros ne sont pas des fonctions

```
#include <stdio.h>
#include <stdlib.h>
#define CARRE(n) ((n)*(n))

int main()
{
   printf ("2*2= %d\n", CARRE(2));
   printf ("(1+1)*(1+1)= %d\n", CARRE(1+1));

   return 0;
}
```

```
2*2= 4
(1+1)*(1+1)= 4
```

```
#include <stdio.h>
#include <stdlib.h>
#define CARRE(n) n*n

int main()
{
   printf ("2*2= %d\n", CARRE(2));
   printf ("(1+1)*(1+1)= %d\n", CARRE(1+1));

   return 0;
}
```

```
2*2= 4
(1+1)*(1+1)= 3
```

pb : 1+1*1+1=3

Allocation dynamique : malloc, free

- la fonction **malloc()** → demander au système de réserver (allouer) en mémoire un espace d'une certaine taille. La fonction retourne un pointeur ptr vers cet espace.
  Une allocation dynamique qui est différente de :
  *allocation statique des variables globales
  *allocation automatique des variables locales

- la fonction **free(ptr)** → restituer l'espace demandé sinon l'espace est restitué à la fin du programme.

- la fonction **realloc()** → changer la taille mémoire d'une zone allouée

```
#include <stdlib.h>
struct Date {
int jour, mois, annee;
};
void truc() {
struct Date *ptr;
…
ptr = (struct Date *) malloc(sizeof(structDate));
}
```

```
struct TableauExtensible {
int taille;
int *t;
};

void initialiser (struct TableauExtensible *tab, int tailleInitiale)
{
 tab→taille = tailleInitiale;
tab→t = (int*) malloc(tailleInitiale * sizeof(int));
//allocation pour 1 tableau de tailleInitiale dimension
//t est un pointeur de type entier
}

void affecter(struct TableauExtensible *tab, int indice, int
valeur)
{tab→t[indice] = valeur;
}

void valeur(structTableauExtensible *tab,int indice)
{return tab→t[indice];
}

void redimensionner(struct TableauExtensible *tab, int taille)
{tab→taille = taille;
tab→t  = (int *) realloc (tab,taille*sizeof(int));
```

Traitement d'exceptions : setjump/longjump

- la fonction setjmp sert à effectuer la sauvegarde de la pile
  int setjmp(jmp_buf env);
  → sauvegarder le contenu des registres, compteur,… dans le tampon "env" et retourne la valeur 0

- la fonction longjmp sert à restaurer la pile
  void longjmp(jmp_buf env, int val);
  →retourner au "point de retour" indiqué dans "env" et transmettre la valeur "val"

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
int main()
{
  jmp_buf env;
  int value = -1;
  int errCode = setjmp(env);          →   errCode = 0
  if(errCode == 0)
                {
                printf("Appeler setjmp\n");
                longjmp(env,value);
→ errCode = -1 et retourner  au point d'appel setjmp()
                return 0;
                }
  else
                {
                printf("Longjmp est appelé\n");
                return -1;
                }
}
```

```
Appeler setjmp
Longjmp est appele
```

Traitement d'exceptions : setjump/longjump

- la fonction setjmp sert à effectuer la sauvegarde de la pile
  int setjmp(jmp_buf env);
  → sauvegarder le contenu des registres, compteur,… dans le tampon "env" et retourne la valeur 0

- la fonction longjmp sert à restaurer la pile
  void longjmp(jmp_buf env, int val);
  →retourner au "point de retour" indiqué dans "env" et transmettre la valeur "val"

```
1er retour de setjmp: i=0
2eme retour de setjmp:   i=1
```

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

jmp_buf env;
int i = 0;

int g()
{
   longjmp(env, 1);
   /*NOTREACHED*/
}

int main ()
{
   if(setjmp(env) != 0) → setjmp(env) = 0
                {
                printf("2eme retour de setjmp:   i=%d\n", i);
                return 0;
                }
   printf("1er retour de setjmp: i=%d\n", i);
   i = 1;
   g(); → retourner au point d'appel setjmp() en mettant 1
   return -1;
}
```