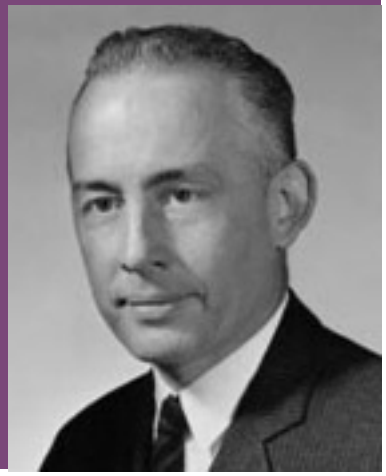
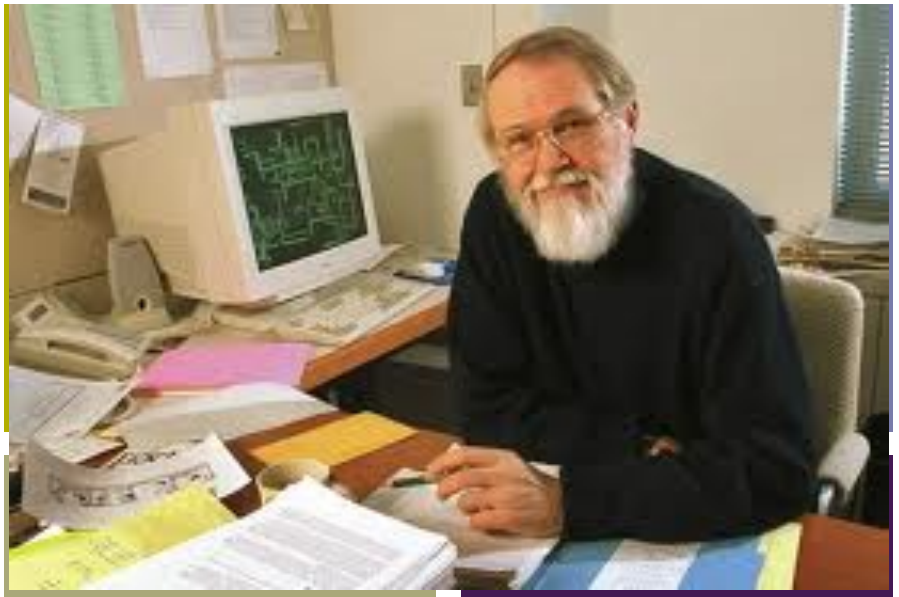




Le langage C pour microntrolleurs

Ressources

- Support de cours de Andrei Doncescu sur le site du LAAS : **homepages.laas.fr/adoncesc** (chercher dans les supports de cours TEACHING)

- Ouvrage de référence « The C programming language », publié en 1978 par Brian Kernighan et Dennie Ritchie, une seconde édition a été publiée en 1988.

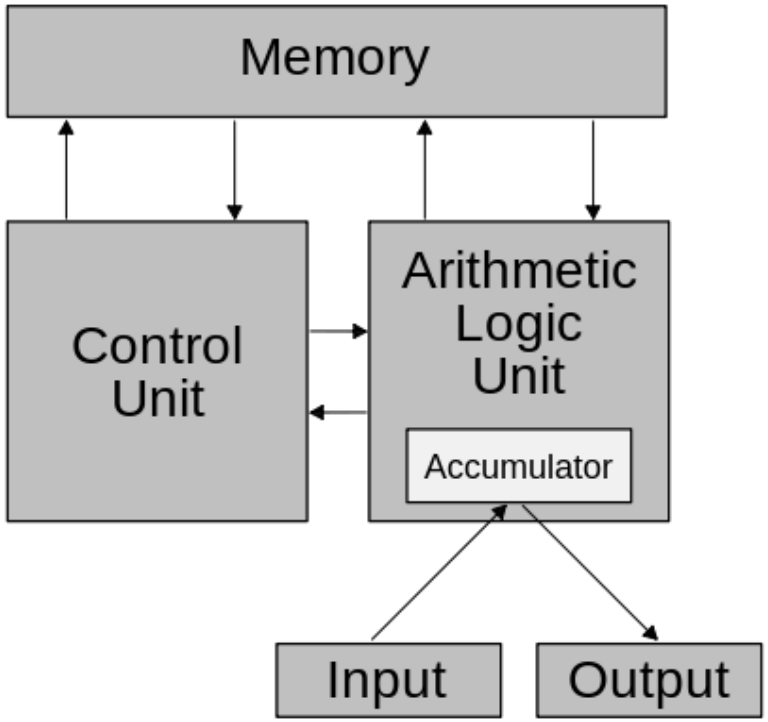
Généralités

- Langage successeur du langage B dans les années 60, premières normes en 1978 puis norme ANSI en 1989 et ISO en 1990.
- Langage de base sous UNIX
- **Langage généraliste de syntaxe « simple »**
- Langage compilé (et non interprété)
- **Usage des pointeurs, récursivité**

L'architecture de base des ordinateurs

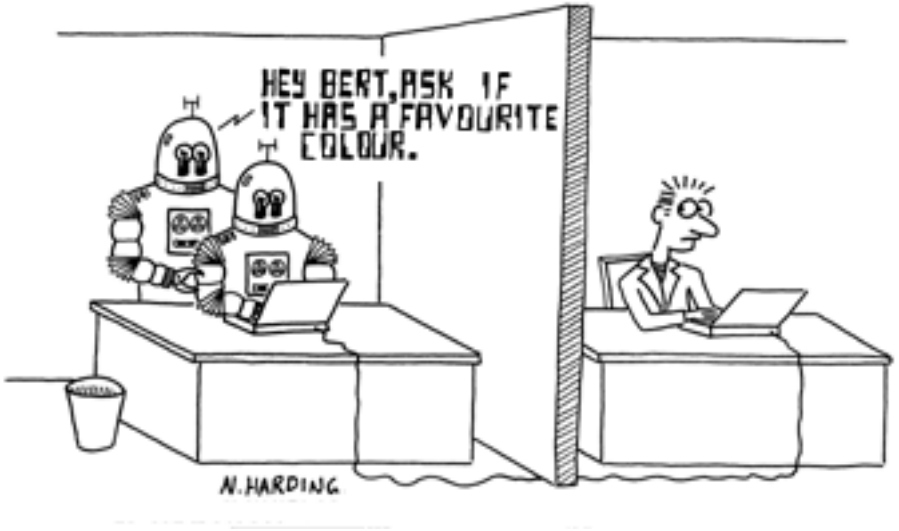
- **Introduction**
- **Architecture de base d'une machine**
  - **La Mémoire Centrale**
  - **UAL (unité arithmétique et logique)**
  - **UC (unité de contrôle ou de commande)**
- **Jeu d'instructions , Format et codage d'une instruction**
- **Modes d'adressage**
- **Étapes d'exécution d'un instruction**

Plan



- Architecture d'une machine von newman.
- Comprendre les étapes de déroulement de l'exécution d'une instruction.
- Le principe des différents modes d'adressage.

A Thinking Machine

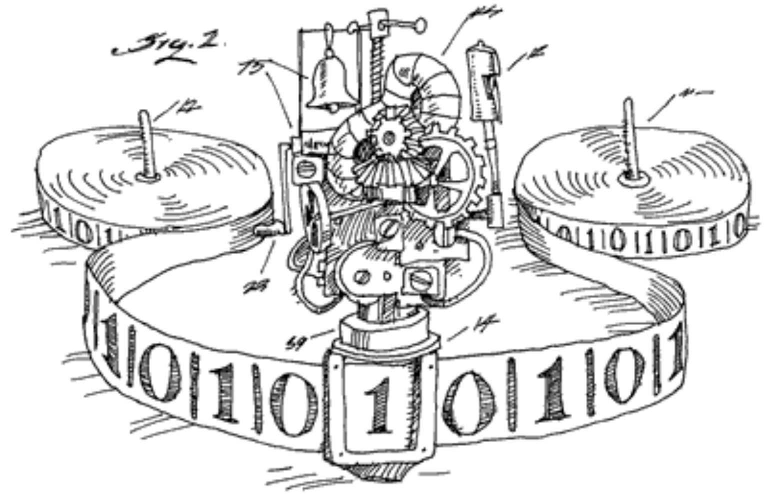


Le terme "computer" = *person* who's line of work is to calculate numerical quantities!

Le Concept de Turing: *Any "algorithm" can be carried out by one of his machines*

+

La Machine Savante



Le but est d'éviter les erreurs humaines

L'ordinateur suit un nombre fini de règles

Une seule règle peut être activée à la fois

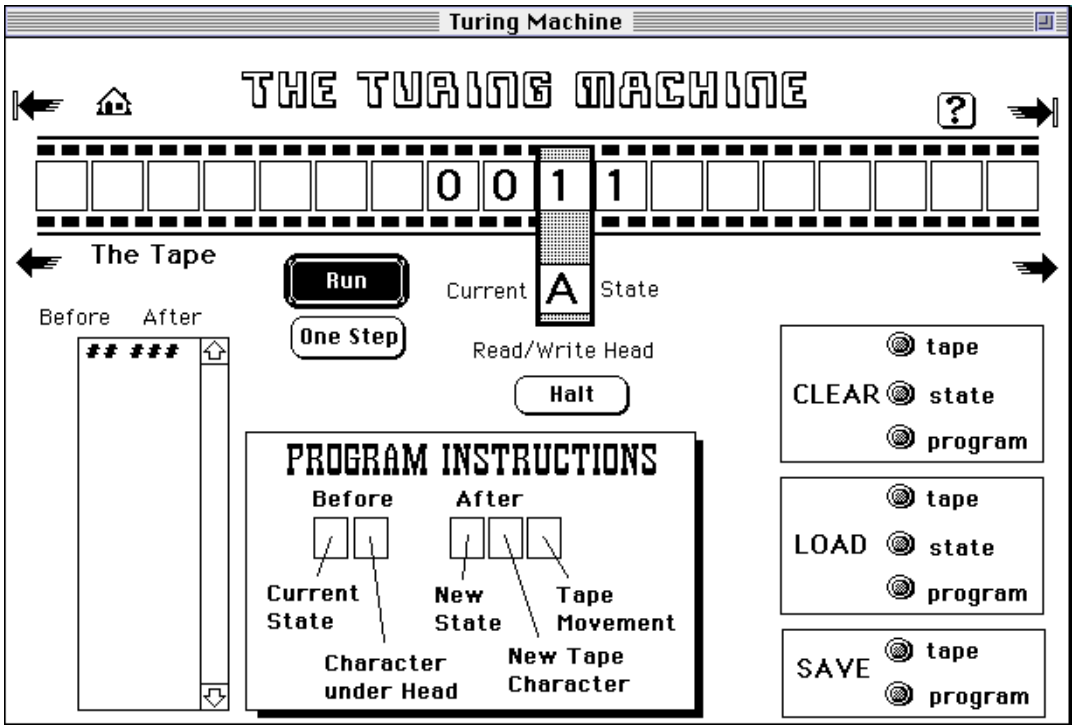
Chaque règle active une autre en fonction des conditions définies

A Thinking Machine
Le programme EG Successor

Des régles simples :

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

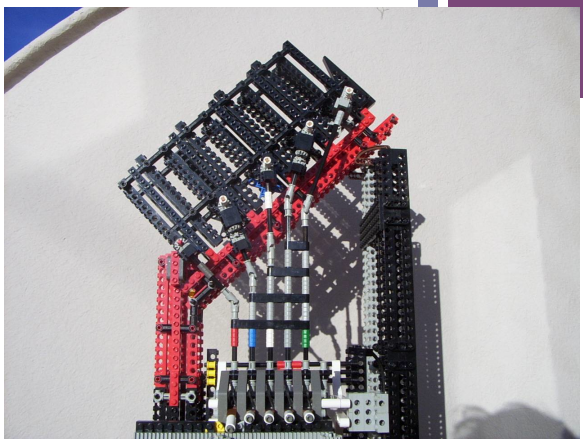


A Thinking Machine
Le programme EG Successor

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!



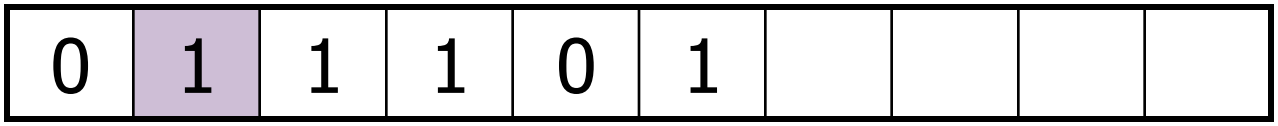
1	1	1	1	0	1				

A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

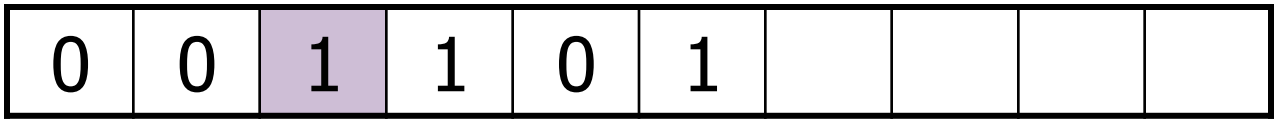


A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

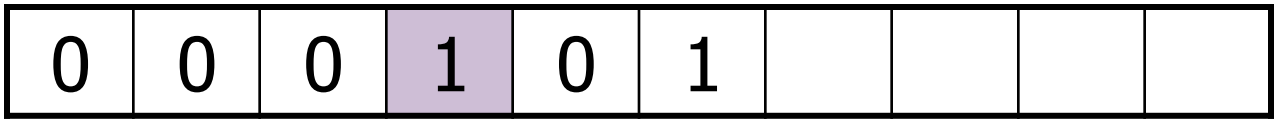


A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

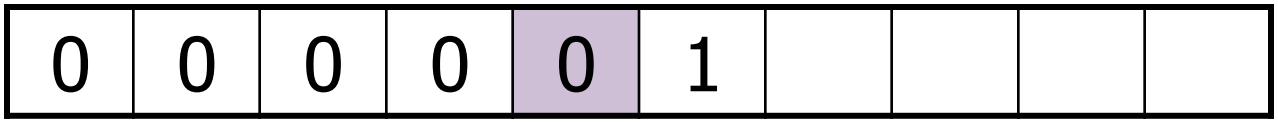


A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!



A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

0	0	0	0	1	1				

A Thinking Machine
Le programme EG `Successor`

Donc le succeseur de 111101 est 000011 qui le complement de 48

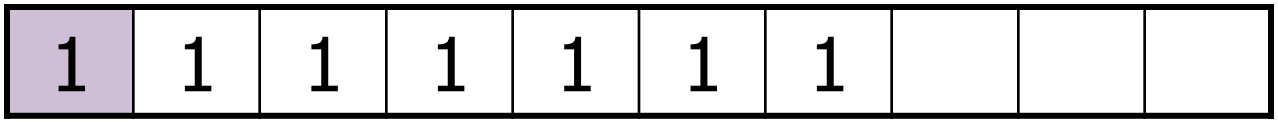
Donc le succeseur de 127 est 128 :

A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

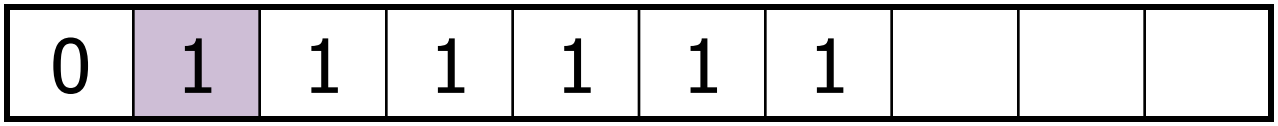


A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!



A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

0	0	1	1	1	1	1			

A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

0	0	0	1	1	1	1			

A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

0	0	0	0	1	1	1			

A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

0	0	0	0	0	1	1			

A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

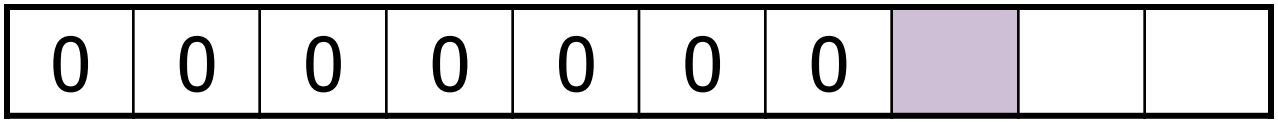
0	0	0	0	0	0	1			

A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!



A Thinking Machine
Le programme EG `Successor`

If read 1, write 0, go right, repeat.

If read 0, write 1, HALT!

If read , write 1, HALT!

0	0	0	0	0	0	0	1		

TM : Instructions

0. if read , go right (*dummy move*), ACCEPT

   if read 0, write $, go right, goto 1 // $ detects start of tape

   if read 1, write $, go right, goto 2

1. if read , go right, REJECT

   if read 0 or X, go right, repeat (= *goto* 1) // look for a 1

   if read 1, write X, go left, goto 3

2. if read , go right, REJECT

   if read 1 or X, go right, repeat // look for a 0

   if read 0, write X, go left, goto 3

3. if read $, go right, goto 4 // look for start of tape

   else, go left, repeat

4. if read 0, write X, go right, goto 1 // similar to step 0

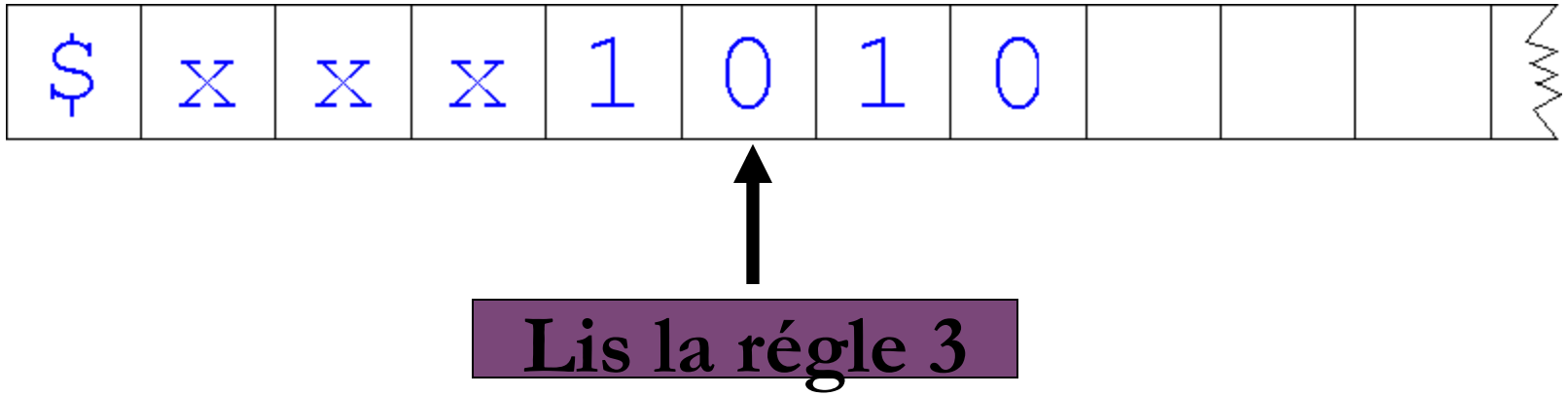
   if read 1, write X, go right, goto 2

   if read X, go right, repeat

   if read , go right, ACCEPT

TM : Notation

Exemple



S'écrit sous la forme:

$\$xxx1q_3010$

if read $, go right, goto 4 **// look for start of tape**
else, go left, repeat

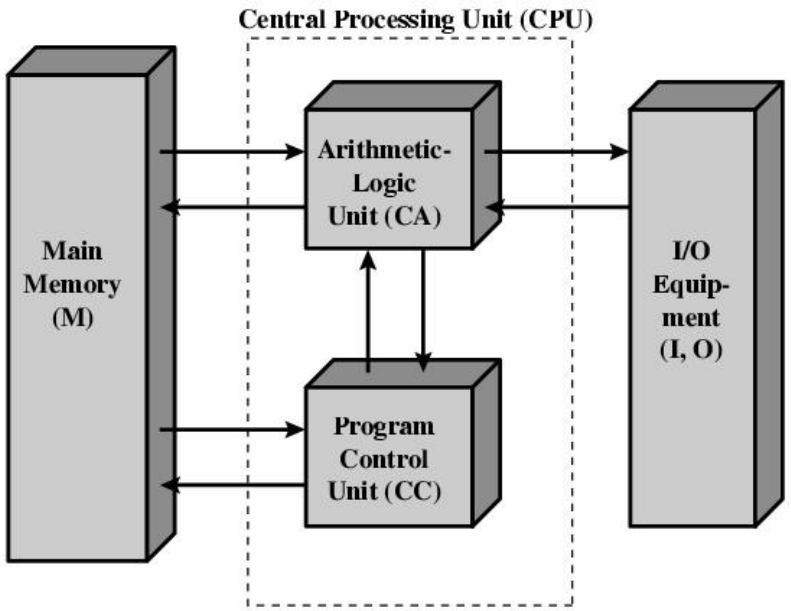
1. Introduction

- Un programme est un ensemble d'instructions exécutées dans un ordre bien déterminé.
- Un programme est exécuté par un processeur (machine).
- Un programme est généralement écrit dans un langage évolué (Pascal, C, VB, Java, etc.).
- Les instructions qui constituent un programme peuvent être classifiées en 4 catégories :
  - Les Instructions d'affectations : permet de faire le transfert des données
  - Les instructions arithmétiques et logiques.
  - Les Instructions de branchement (conditionnelle et inconditionnelle)
  - Les Instructions d'entrées sorties.

Exécution d'un Programme

- Pour exécuter un programme par une machine, on passe par les étapes suivantes :

1. **Édition :** on utilise généralement un éditeur de texte pour écrire un programme et le sauvegarder dans un fichier.
2. **Compilation** : un compilateur est un programme qui convertit le code source (programme écrit dans un langage donné) en un programme écrit dans un langage machine (binaire). Une instruction en langage évolué peut être traduite en plusieurs instructions machine.
3. **Chargement** : charger le programme en langage machine dans mémoire afin de l'exécuter .

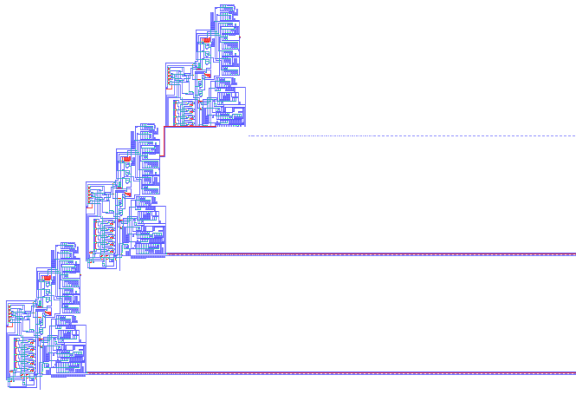


- **Comment s'exécute un programme dans la machine ?**

- Pour comprendre le mécanisme d'exécution d'un programme → il faut comprendre le mécanisme de l'exécution d'une instruction .

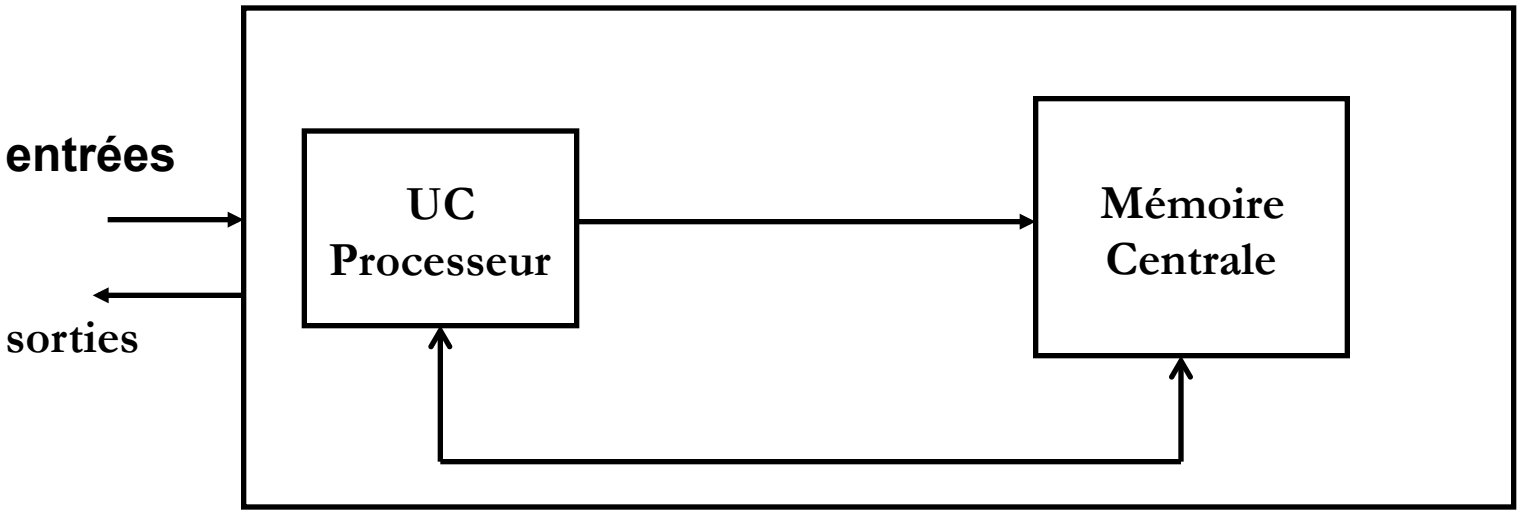
- Pour comprendre le mécanisme de l'exécution d'une instruction → il faut connaître l'architecture de la machine (processeur) sur la quelle va s'exécuter cette instruction.

Architecture matérielle d'une machine (architecture de Von Neumann)

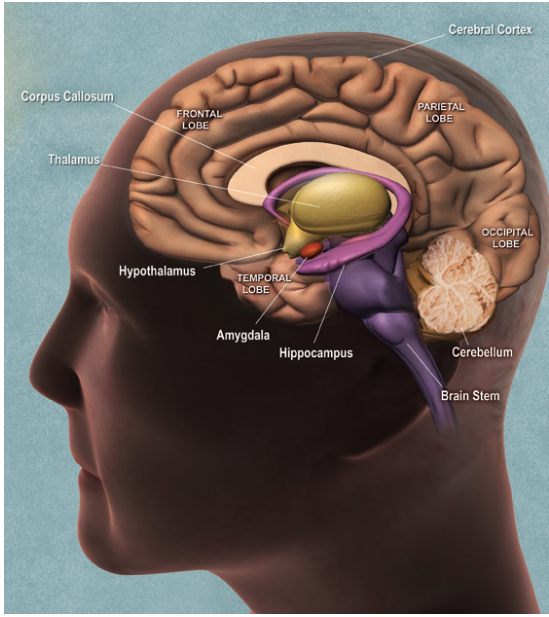


L'architecture de Von Neumann est composé

- D'une **mémoire centrale**,
- D'une **unité centrale** UC , CPU (Central Processing Unit), processeur , microprocesseur.
- D'un ensemble de **dispositifs d'entrées sorties** pour communiquer avec l'extérieur.

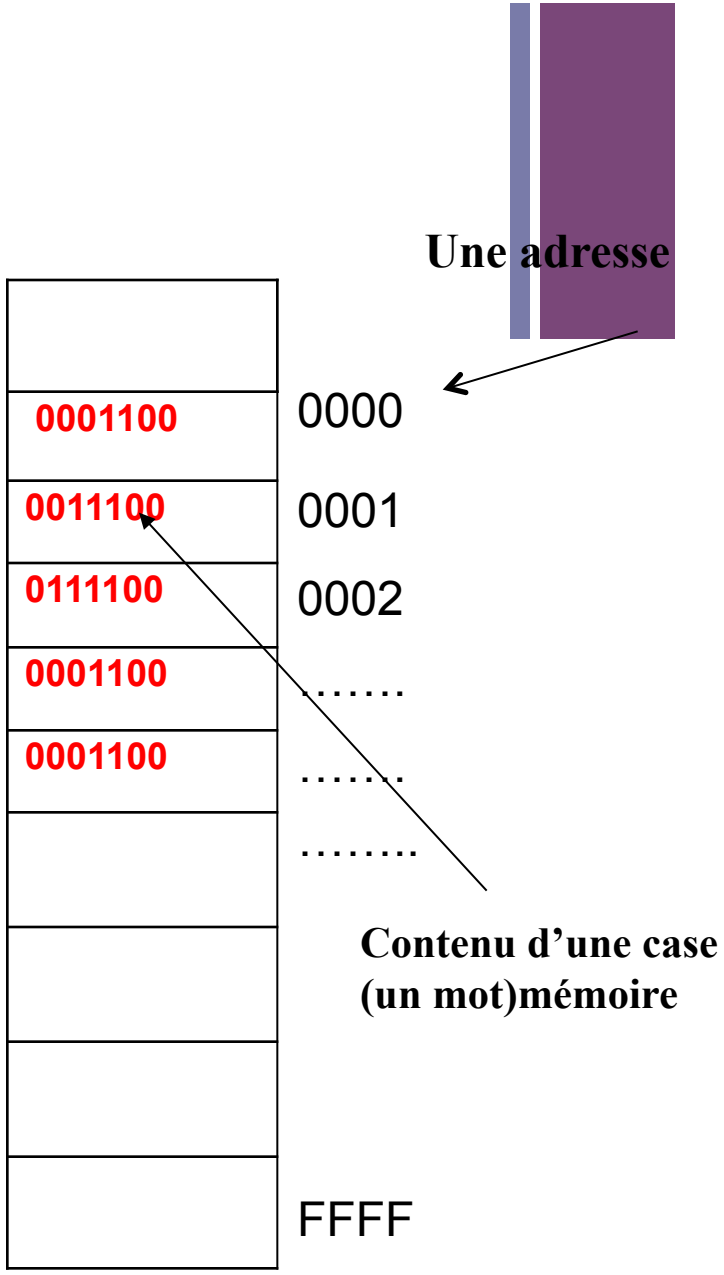


La mémoire centrale

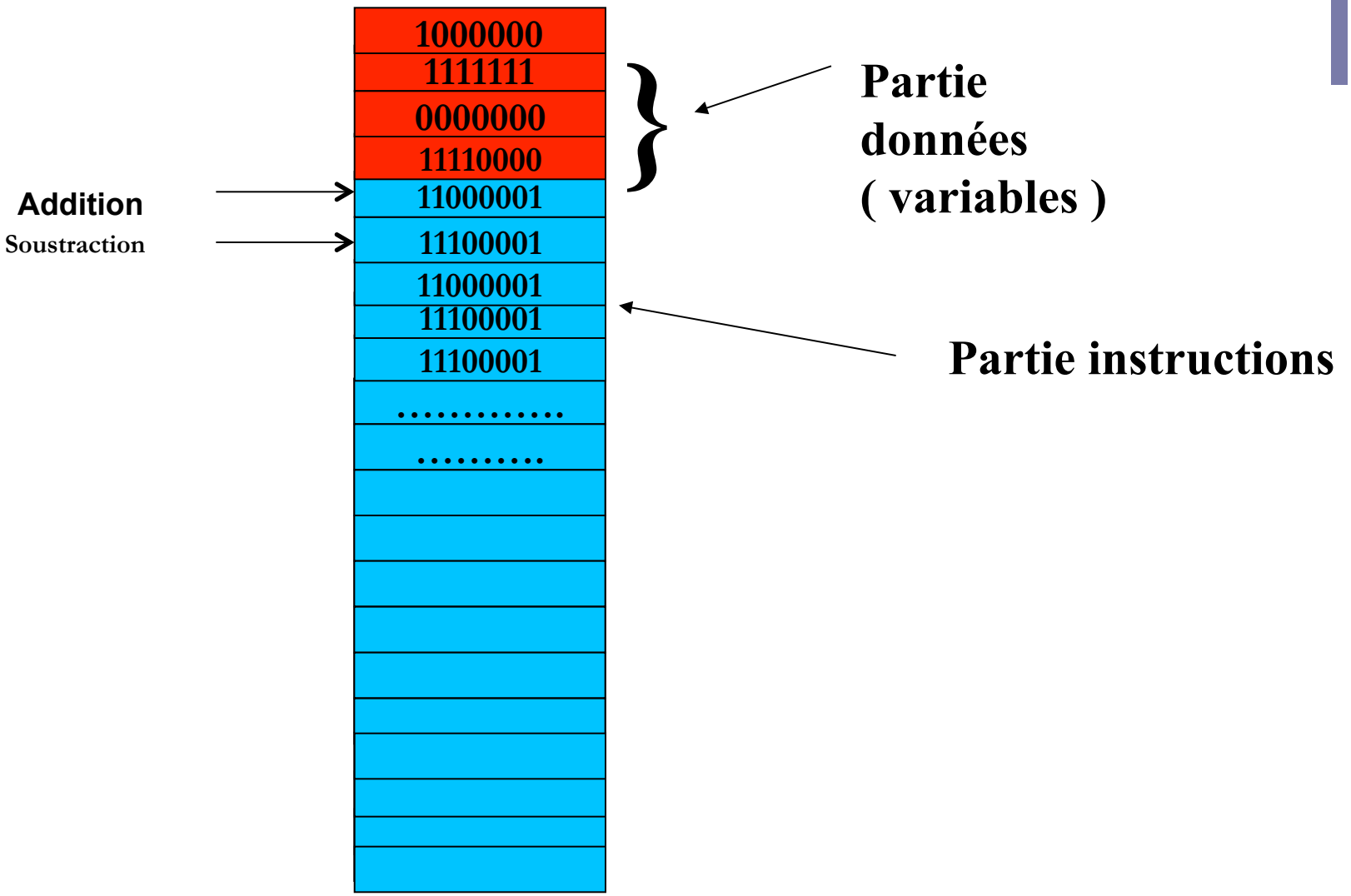


- La mémoire centrale (MC) représente l'espace de travail de l'ordinateur .
- C'est l'organe principal de rangement des informations utilisées par le processeur.
- Dans un ordinateur pour exécuter un programme il faut le charger (copier) dans la mémoire centrale .
- Le temps d'accès à la mémoire centrale et sa capacité sont deux éléments qui influent sur le temps d'exécution d'un programme (performances d'une machine).

- La mémoire centrale peut être vu comme un large vecteur (tableau) de mots ou octets.
- Un mot mémoire stocke une information sur n bits.
- Chaque mot possède sa propre adresse.
- La mémoire peut contenir des programmes et les données utilisées par les programmes.



Structure d’ un programme en MC

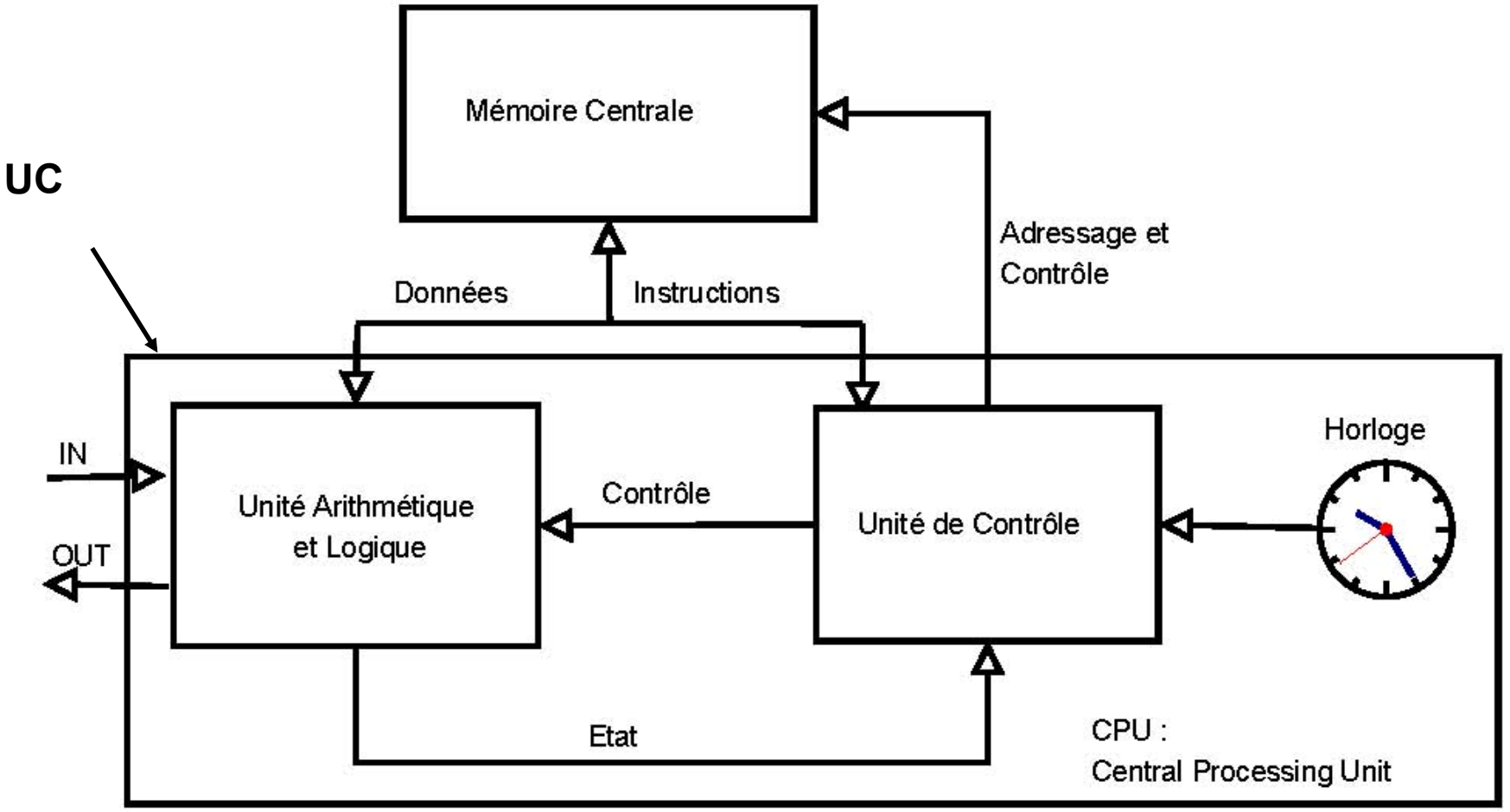


L'Unité Centrale (UC)

- L'unité centrale (appelée aussi processeur , microprocesseur) à pour rôle d'exécuter les programmes.

- L'UC est composée d'une unité arithmétique et logique (UAL) et d'une unité de contrôle.
  - L'unité arithmétique et logique réalise les opérations élémentaires (addition, soustraction, multiplication, . . .) .
  - L'unité de commande contrôle les opérations sur la mémoire (lecture/écriture) et les opérations à réaliser par l'UAL selon l'instruction en cours d'exécution.

Architecture matérielle d'une machine Von Neumann



L'UAL

- L'unité arithmétique et logique réalise une opération élémentaire (addition, ,soustraction, multiplication, . . .).

- L'UAL regroupe les circuits qui assurent les fonctions logiques et arithmétiques de bases (ET,OU,ADD,SUS,…..).

- L'UAL comporte un registre accumulateur (ACC) : c'est un registre de travail qui sert a stocker un opérande (données)au début d'une opération et le résultat à la fin.

- L'UAL comporte aussi un registre d'état : Ce registre nous indique l'état du déroulement de l'opération .
- Ce registre est composé d'un ensemble de bits. Ces bits s'appels indicateurs (drapeaux ou flags).
- Ces indicateurs sont mis à jours (modifiés)après la fin de l'exécution d'une opération dans l'UAL.
- Les principeaux indicateurs sont :
  - Retenue : ce bit est mis à 1 si l'opération génère une retenue.
  - Signe :ce bit est mis à 1 si l'opération génère un résultat négative.
  - Débordement :ce bit est mis à 1 s'il y a un débordement.
  - Zero : ce bit est mis à 1 si le résultat de l'opération est nul.

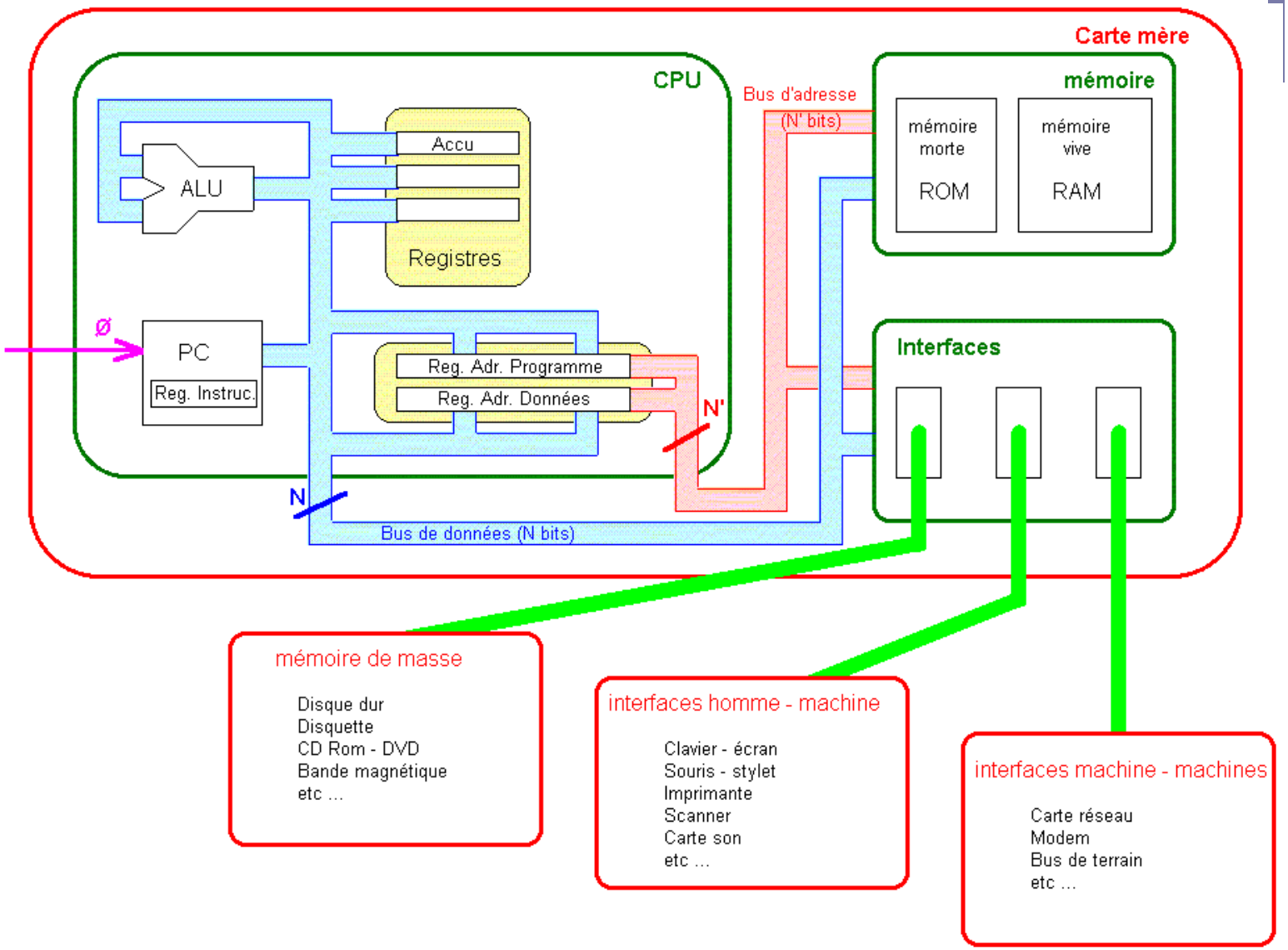
Unité de contrôle

- Le rôle de l'unité de contrôle (ou unité de commande) est de :
  - coordonner le travail de toutes les autres unités (UAL , mémoire, ….)
  - et d'assurer la synchronisation de l'ensemble.

- Elle assure :
  - la recherche (lecture) de l'instruction et des données à partir de la mémoire,
  - le décodage de l'instruction et l'exécution de l'instruction en cours
  - et prépare l'instruction suivante.

- L’unité de contrôle comporte :
  - Un registre instruction (RI) : contient l’instruction en cours d’exécution. Chaque instruction est décoder selon sont code opération grâce à un décodeur.
  - Un registre qui s’appel compteur ordinal (CO) ou le compteur de programme (CP) : contient l’adresse de la prochaine instruction à exécuter (pointe vers la prochaine instruction à exécuter). Initialement il contient l’adresse de le première instruction du programme à exécuter.
  - Un séquenceur : il organise (synchronise) l’exécution des instruction selon le rythme de l’horloge, il génère les signaux nécessaires pour exécuter une instruction.

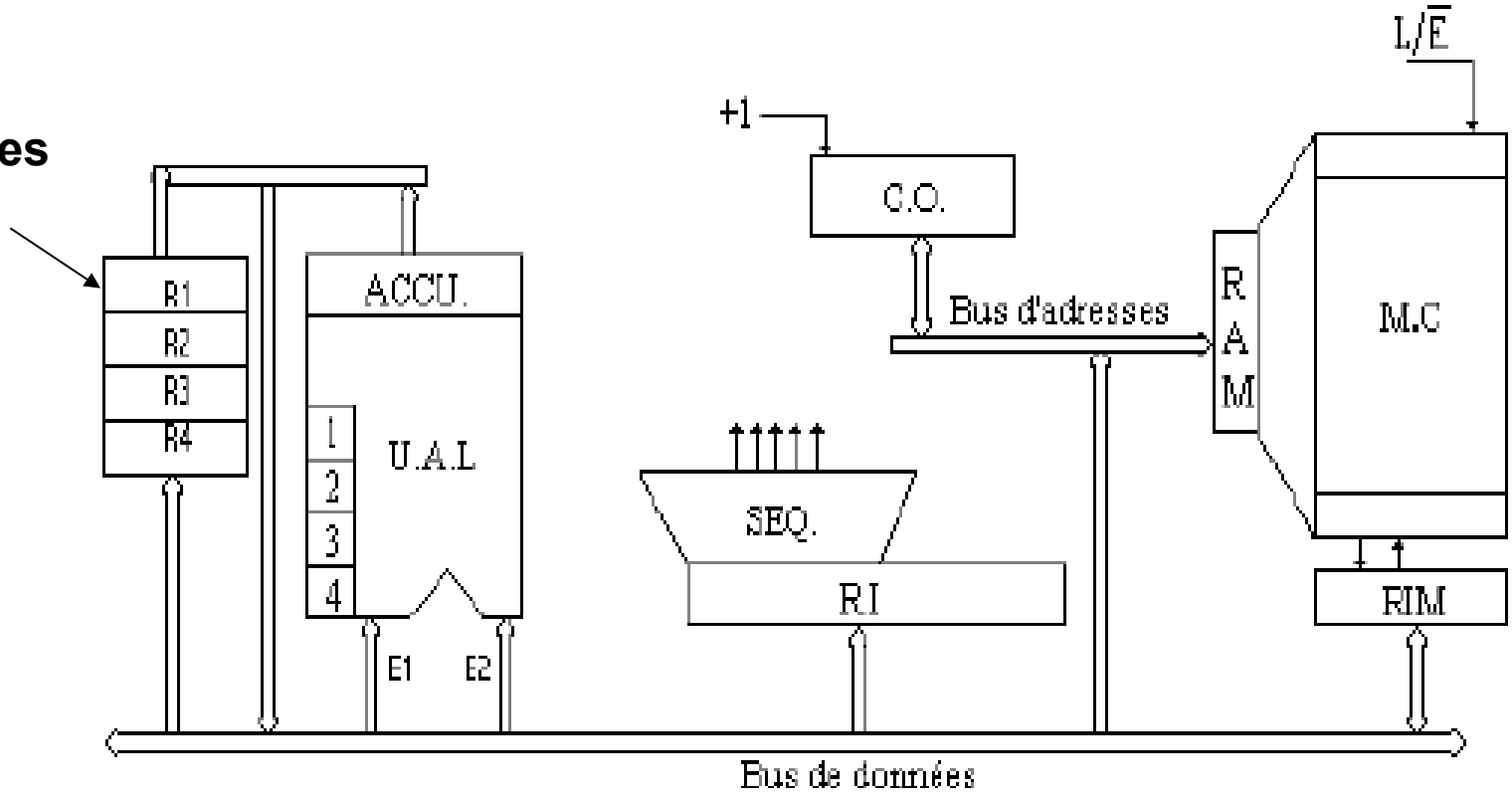
Schéma d'une UC



Remarque

- Le microprocesseur peut contenir d'autres registres autre que CO,RI et ACC.
- Ces registres sont considérés comme une mémoire interne (registre de travail) du microprocesseur.
- Ces registres sont plus rapide que la mémoire centrale , mais le nombre de ces registre est limité.
- Généralement ces registres sont utilisés pour sauvegarder les données avant d'exécuter une opération.
- Généralement la taille d'un registre de travail est égale à la taille d'un mot mémoire

Une machine avec des registres de travail

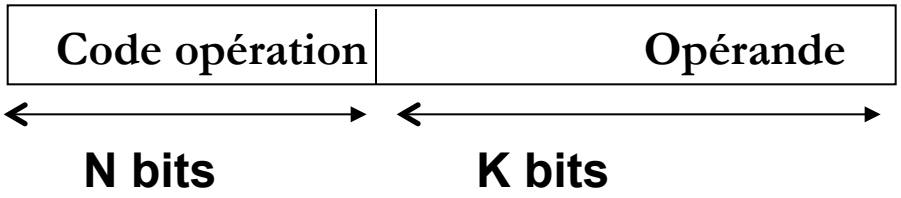


Jeu d'instructions

- Chaque microprocesseur possède un certain nombre limité d'instructions qu'il peut exécuter. Ces instructions s'appelles jeu d'instructions.
- Le jeu d'instructions décrit l'ensemble des opérations élémentaires que le microprocesseur peut exécuter.
- Les instructions peuvent être classifiées en 4 catégories :
  - Instruction d'affectation : elle permet de faire le transfert des données entre les registres et la mémoire
    - Écriture : registre → mémoire
    - Lecture : mémoire → registre
  - Les instructions arithmétiques et logiques (ET , OU , ADD,….)
  - Instructions de branchement (conditionnelle et inconditionnelle)
  - Instructions d'entrées sorties.

Codage d'une instruction

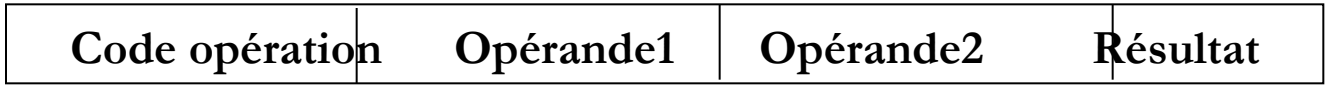
- Les instructions et leurs opérandes (données) sont stocké dans la mémoire.
- La taille d'une instruction (nombre de bits nécessaires pour la représenter en mémoire) dépend du type de l'instruction et du type de l'opérande.
- L'instruction est découpée en deux parties :
  - Code opération (code instruction) : un code sur N bits qui indique quelle instruction.
  - La champs opérande : qui contient la donnée ou la référence (adresse) à la donnée.



•Le format d'une instruction peut ne pas être le même pour toutes les instructions.
•Le champs opérande peut être découpé à sont tours en plusieurs champs

Machine à 3 adresses

- Dans ce type de machine pour chaque instruction il faut préciser :
  - l'adresse du premier opérande
  - du deuxième opérande
  - et l'emplacement du résultat



Exemple :

ADD A,B,C (C←B+C)

•Dans ce type de machine la taille de l'instruction est grand .

• Pratiquement il n'existent pas de machine de ce type.

Machine à 2 adresses

- Dans de type de machine pour chaque instruction il faut préciser :
  - l'adresse du premier opérande
  - du deuxième opérande ,
- l'adresse de résultat est implicitement l'adresse du deuxième opérande .

Code opération	Opérande1	Opérande2

Exemple :

ADD A,B **(B←A +B)**

Machine à 1 adresses

- Dans de type de machine pour chaque instruction il faut préciser uniquement l'adresse du deuxième opérande.
- Le premier opérande existe dans le registre accumulateur.
- Le résultat est mis dans le registre accumulateur.

Code opération	Opérande2

Exemple :

ADD A (ACC←(ACC) + A)

Ce type de machine est le plus utilisé.

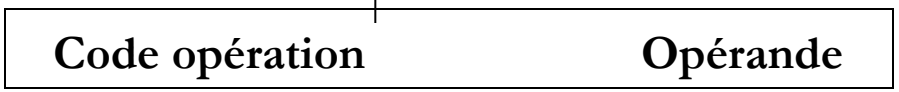
Mode d'adressage

- La champs opérande contient la donnée ou la référence (adresse) à la donnée.
- Le mode d'adressage définit la manière dont le microprocesseur va accéder à l'opérande.
- Le code opération de l'instruction comportent un ensemble de bits pour indiquer le mode d'adressage.

- Les modes d'adressage les plus utilités sont :
  - Immédiat
  - Direct
  - Indirect
  - Indexé
  - relatif

Adressage immédiat

- L'opérande existent dans le champs adresse de l'instruction



Exemple :
ADD 150

Cette commande va avoir l'effet suivant : ACC←(ACC)+ 150

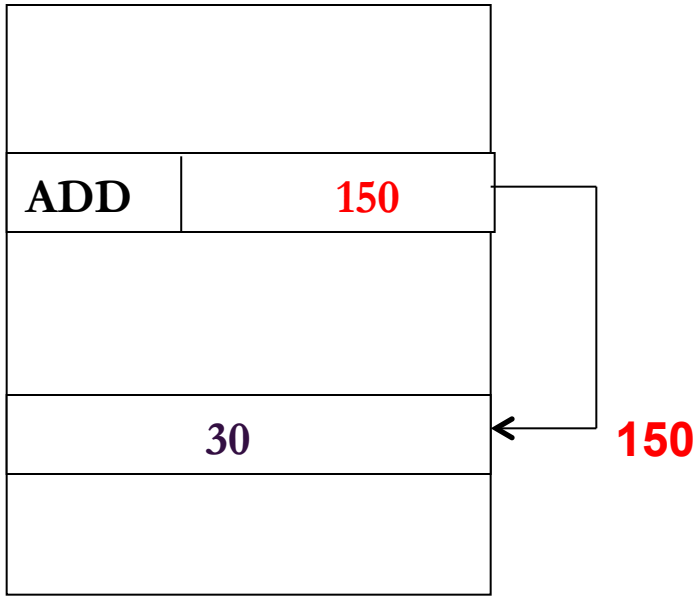
Si le registre accumulateur contient la valeur 200 alors après l'exécution son contenu sera égale à 350

Adressage direct

- Le champs opérande de l'instruction contient l'adresse de l'opérande (emplacement en mémoire)
- Pour réaliser l'opération il faut le récupérer (lire) l'opérande à partir de la mémoire. ACC ← (ACC)+ (ADR)

Exemple :
On suppose que l'accumulateur continent la valeur 20 .

A la fin de l'exécution nous allons avoir la valeur 50 (20 + 30)



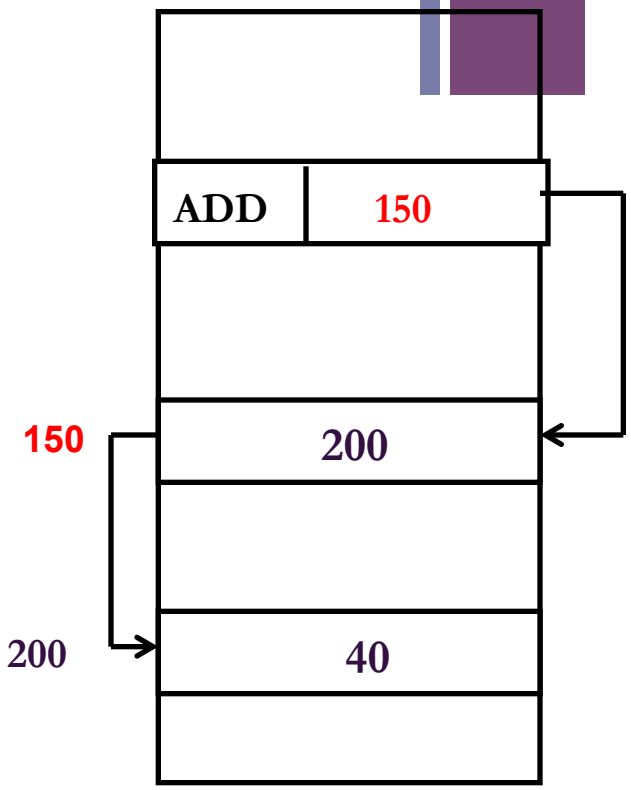
Adressage indirect

- La champs adresse contient l'adresse de l'adresse de l'opérande.
- Pour réaliser l'opération il faut :
  - Récupérer l'adresse de l'opérande à partir de la mémoire.
  - Par la suite il faut chercher l'opérande à partir de la mémoire.

ACC← (ACC)+ ((ADR))

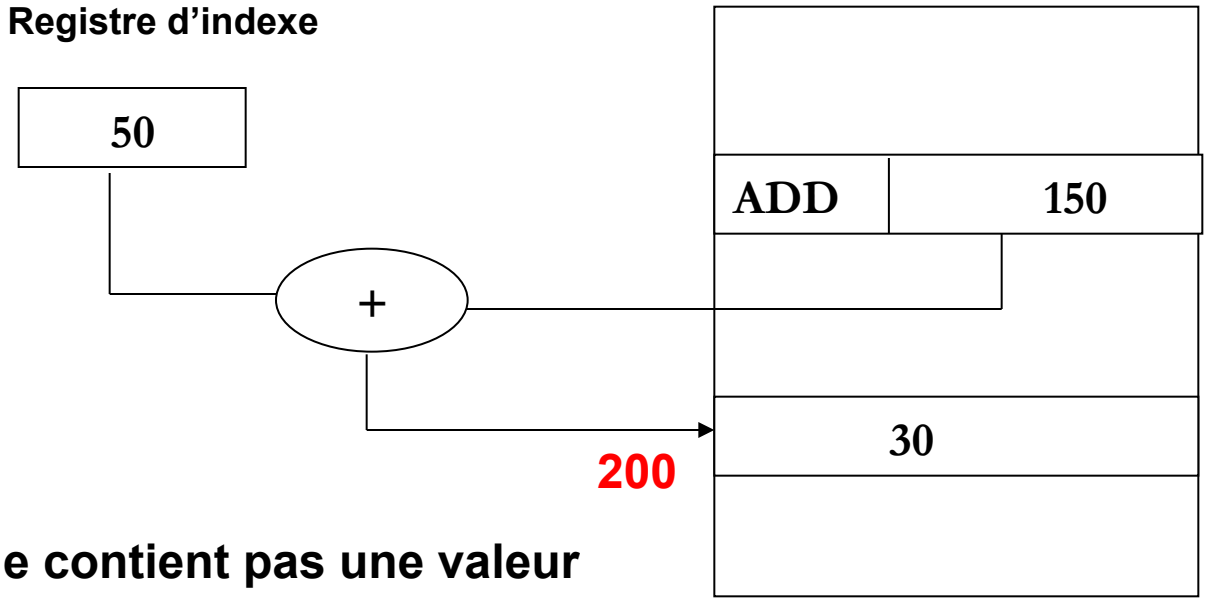
- Exemple :
- Initialement l'accumulateur contient la valeur 20
- Il faut récupérer l'adresse de l'adresse (150).
- Récupérer l'adresse de l'opérande à partir de l'adresse 150 (la valeur 200)
- Récupérer la valeur de l'opérande à partir de l'adresse 200 (**la valeur 40**)

Additionner la valeur 40 avec le contenu de l'accumulateur (20) et nous allons avoir la valeur **60**



Adressage indexé

- L'adresse effectif de l'opérande est relatif à une zone mémoire.
- L'dresse de cette zone se trouve dans un registre spécial (registre indexe).
- Adresse opérande = ADR + (X)



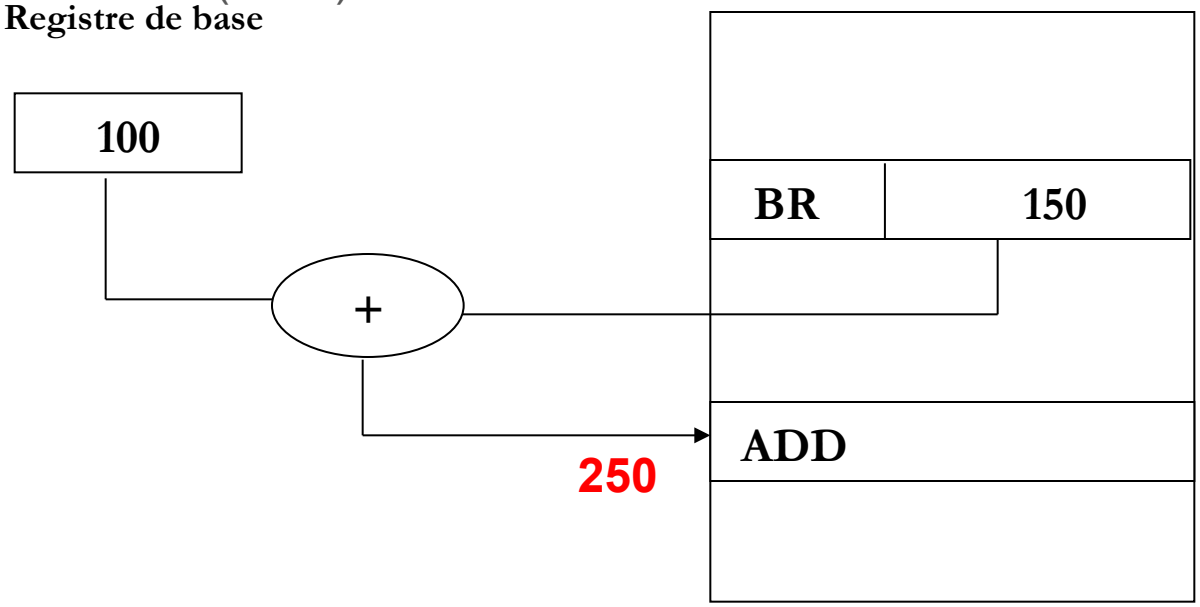
Remarque : si ADR ne contient pas une valeur immédiate alors

Adresse opérande = (ADR)+ (X)

Adressage relatif

- L'adresse effectif de l'opérande est relatif a une zone mémoire.
- L'dresse de cette zone se trouve dans un registre spécial (registre de base).
- Ce mode d'adressage est utilisée pour les instructions de branchement.

Adresse = ADR + (base)



Cycle d'exécution d'une instruction

- Le traitement d'une instruction est décomposé en trois phases :
  - Phase 1 : rechercher l'instruction à traiter et décodage
  - Phase 2 : rechercher de l'opérande et exécution de l'instruction
  - Phase 3 : passer à l'instruction suivante

- Chaque phase comporte un certain nombre d'opérations élémentaires (microcommandes) exécutées dans un ordre bien précis (elle sont générées par le séquenceur).

- La phase 1 et 3 ne change pas pour l'ensemble des instructions , par contre la phase 2 change selon l'instruction et le mode d'adressage

- Exemple1 : déroulement de l'instruction d'addition en mode immédiat ACC←(ACC)+ Valeur

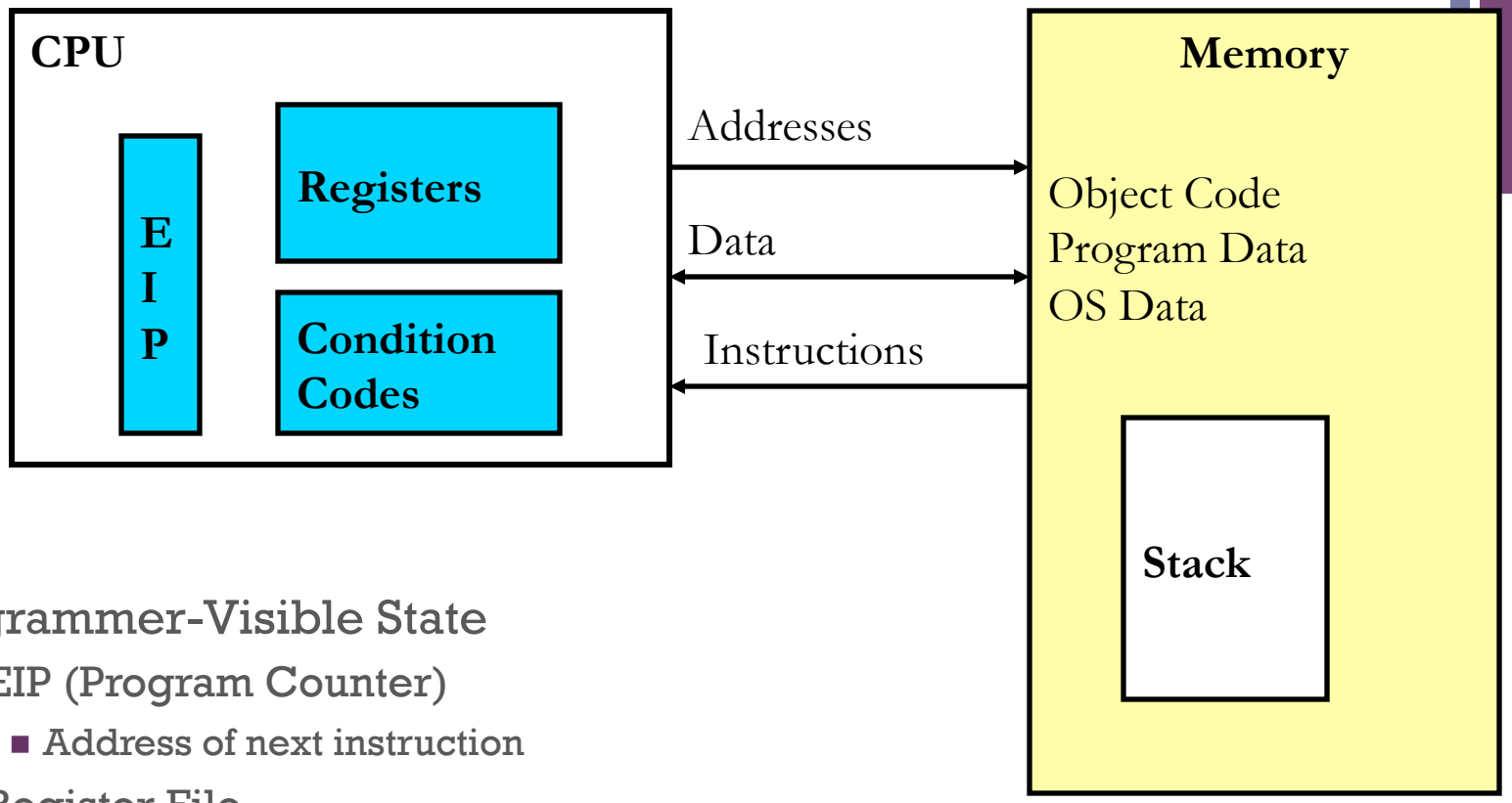
  - Phase 1 : (rechercher l'instruction à traiter)
    - Mettre le contenu du CO dans le registre RAM RAM← (CO)
    - Commande de lecture à partir de la mémoire
    - Transfert du contenu du RIM dans le registre RI RI← (RIM)
    - Analyse et décodage
  - Phase 2 : (traitement)
    - Transfert de l 'opérande dans l'UAL UAL ← (RI).ADR
    - Commande de l'exécution de l'opération (addition)
  - Phase 3 : (passer à l'instruction suivante)
    - CO ← (CO)+ 1

- Exemple 2 : déroulement de l'instruction d'addition en mode direct ACC←(ACC)+ (ADR)

  - Phase 1 : (rechercher l'instruction à traiter)
    - Mettre le contenu du CO dans le registre RAM RAM← (CO)
    - Commande de lecture à partir de la mémoire
    - Transfert du contenu du RIM dans le registre RI RI← (RIM)
    - Analyse et décodage
  - Phase 2 : (décodage et traitement)
    - Transfert de l'adresse de l 'opérande dans le RAM RAM← (RI).ADR
    - Commande de lecture
    - Transfert du contenu du RIM vers l'UAL UAL ←(RIM)
    - Commande de l'exécution de l'opération (addition)
  - Phase 3 : (passer à l'instruction suivante)
    - CO ← (CO)+ 1

- Exemple 3 : Déroulement de l'instruction d'addition en mode indirect ACC←(ACC)+ ((ADR))

  - Phase 1 : (rechercher l'instruction à traiter)
    - Mettre le contenu du CO dans le registre RAM RAM← (CO)
    - Commande de lecture à partir de la mémoire
    - Transfert du contenu du RIM dans le registre RI RI← (RIM)
    - Analyse et décodage
  - Phase 2 : (décodage et traitement)
    - Transfert de l'adresse de l 'opérande dans le RAM ← (RI).ADR
    - Commande de lecture /* récupérer l'adresse */
    - Transfert du contenu du RIM vers le RAM RAM←(RIM)
    - Commande de lecture /* récupérer l'opérande */
    - Transfert du contenu du RIM vers l'UAL UAL ← (RIM)
    - Commande de l'exécution de l'opération (addition)
  - Phase 3 : (passer à l'instruction suivante)
    - CO ← (CO)+ 1

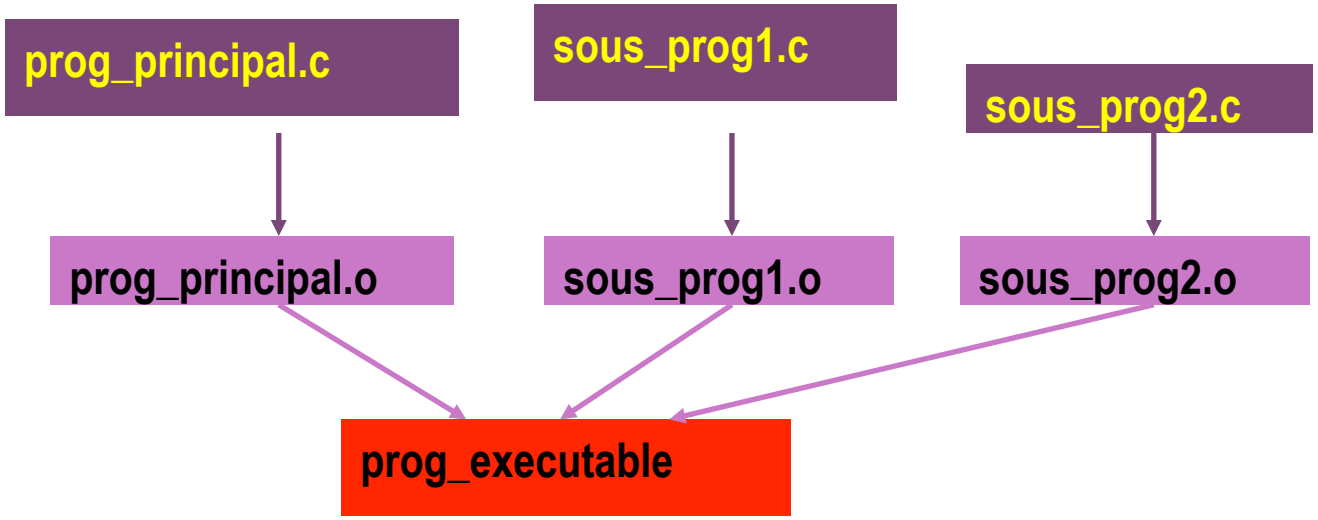


- Programmer-Visible State
  - EIP (Program Counter)
    - Address of next instruction
  - Register File
    - Heavily used program data
  - Condition Codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
  - Memory
    - Byte-addressable array
    - Code, user data, (most) OS data
    - Includes stack used to support procedures

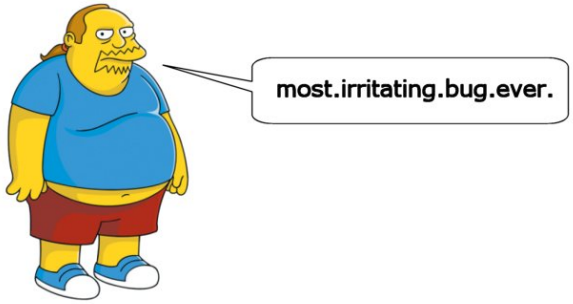
Comment créer un programme « exécutable » ?

0) Ecriture des fichiers sources *.c
1) Compilation : cc -c prog_principal.c (idem pour les sous programmes)
2) Liens : cc prog_principal.o sous_prog1.o sous_prog2.o -o prog_executable
3) Reste à lancer prog_executable …

La procédure de compilation est souvent automatisée à l 'aide d 'un fichier nommé Makefile



Structure d'un programme



- Un programme est composé de plusieurs fonctions qui échangent et modifient des variables
- Chaque fonction, y compris la fonction main, comprend les éléments suivants :
  - **Directives du préprocesseur**
  - **Déclaration d 'objets externes (variables, fonctions)**
  - **Interface de la fonction**
  - **Bloc d'instructions**

EXEMPLES

Système à base 10 → Système Décimal

$[72069.45]_{10} = 72069.45 = 7x10^4 + 2x10^3 + 0x10^2 + 6x10^1 + 9x10^0 + 4x10^{-1} + 5x10^{-2}$

Système à base 8 → Système Octal

$[726041.53]_8 = [7x8^5 + 2x8^4 + 6x8^3 + 0x8^2 + 4x8^1 + 1x8^0 + 5x8^{-1} + 3x8^{-2}]_{10} = [?]_{10}$

Système à base 16 → Système HexaDécimal

$[7A6C9D.F5]_{16} = [7x16^5 + Ax16^4 + 6x16^3 + Cx16^2 + 9x16^1 + Dx16^0 + Fx16^{-1} + 5x16^{-2}]_{10} = [?]_{10}$

Système à base 2 → *Système Binaire*

$[1001010.01]_2 = [1x2^6 + 0x2^5 + 0x2^4 + 1x2^3 + 0x2^2 + 1x2^1 + 0x2^0 + 0x2^{-1} + 1x2^{-2}]_{10} = [?]_{10}$

X=0

$=[?]_{10}$ ➔ 2^x → …, 0.125, 0.25, 0.5, 1, 2, 4, 8, 16, 32, 32, 64, 128, 256, 512, 1024, 2048, …

Note 1: 2^{10} = 1K (kilo), 2^{20} = 1M (Mega), 2^{30} = 1G (Giga), 2^{40} = 1T (Tira),
Note 2: $16^x = 2^{4x}$ et $8^x = 2^{3x}$ → d' ou l' importance des systèmes : Octal et Hexadécimale en microélectronique.

$[10\ 110\ 001\ 101\ 011\ .\ 111\ 100\ 110]_2 = [26153.746]_8 = [26153.746]_8 = [2C6B.F3]_{16}$

Conversion Décimale → RADIX (2, 8, 16)

CONVERTION DECIMALE → R (2, 8, 16)

***Règle*:**

$[Ne.Nf]_{10} = [?]_R$

Ne / R : Quotient → Ne0, et Reste0 → d_0
Ne0 / R : Quotient → Ne1, et Reste1 → d_1
Ne1 / R : Quotient → Ne2, et Reste1 → d_2
…, etc.

Nf X R : (Résultat1 - P_Ent1) → Nf1, et P_Ent1 → d_{-1}
Nf1 X R : (Résultat2 - P_Ent2) → Nf2, et P_Ent2 → d_{-2}
Nf3 X R : (Résultat2 - P_Ent2) → Nf3, et P_Ent2 → d_{-2}

…, etc.

Complément à ‘2’ et Complément à ‘1’

- Le complément à ‘*1*’ d’un nombre binaire consiste simplement à inverser ses bits.

Le complément à ‘*2*’ d’un nombre binaire consiste à ajouter 1 à son complément à 1

Exemple

C’ 1 de 010110 → 101001

C’ 2 de 010110 → 101010

Nombres Binaires Signés

Températures → +15°C, 0 °C, -21°C ➔ Nombres non signés

Avantage du C' 2 par rapport aux autres

Un seul '0' est représenté

	C' 2	***C' 1***	***Amplitude Signée***
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
- 0	-----	1111	1000
- 1	1111	1110	1001
- 2	1110	1101	1010
- 3	1101	1100	1011
- 4	1100	1011	1100
- 5	1011	1010	1101
- 6	1010	1001	1110
- 7	1001	1000	1111
- 8	1000	-----	-----

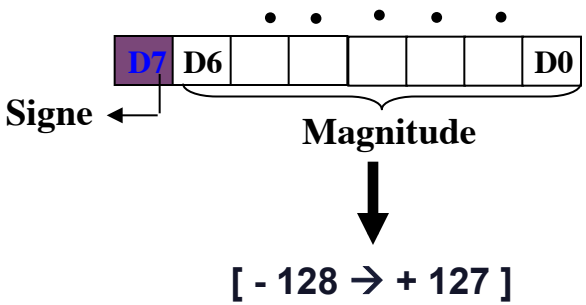
-8 est représenté →

L'ORDINATEUR représente les nombres en nombre limité de bits: 8, 16, 32, 64 bits

- Octet Signé →

+127	0111 1111
+126	0111 1110
…	…
+2	0000 0010
+1	0000 0001
0	0000 0000
-1	1111 1111
-2	1111 1110
…	…
-127	1000 0001
-128	1000 0000

Nombres positifs



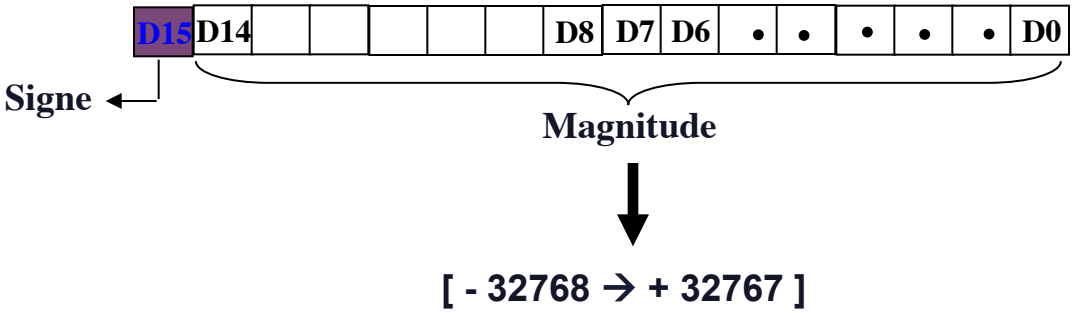
Nombres négatifs → '<u>C'2</u>' des magnitudes de leurs équivalents positifs

- n = NOT (+ n) + 1

<u>Exemple</u>: -32 → NOT (+32d = 00100000b) + 1 → 11011111 + 1 → 11100000b

+32767	0111 1111 1111 1111
+1	0000 0000 0000 0001
0	0000 0000 0000 0000
-1	1111 1111 1111 1111
-32768	1000 0000 0000 0000

- Mot Signé →



Code ASCII

Hex	*Symbol*	*Hex*	*Symbol*
41	A	61	a
42	B	62	b
43	C	63	c
44	D	64	d
…	…	…	…
59	Y	79	y
5A	Z	7A	z

Types de variables manipulées en C

- Toutes les variables doivent être explicitement typées (pas de déclaration implicite comme en fortran)

- Il y a globalement trois (quatre?) types de variables :
  - Les entiers : **int, short int, long int**
  - Les réels : **float, double, long double**
  - Les caractères : **char**
  - Rien ... **: void**

- exemples : short int mon_salaire;
  double cheese;
  char avoile;

- NB : la présence d 'une ou plusieurs étoiles devant le nom de la variables indique un pointeur, dans nos applications il s 'agira en général de tableaux.
  ex : double **mat permet de définir une matrice

Representation d'un nombre BINAIRE

En général un nombre dans un système à base R (Radix) est noté comme suit:

Partie Entière ***Partie Fractionnaire***

$$N = [d_m d_{m-1} \ldots d_1 d_0 \blacksquare d_{-1} d_{-2} \ldots d_{-n}]_R$$

avec *m* et *n* *étant entiers*

R* = *Radix (2, 8, 10, 16, etc.)

d_i ***entier = [0, 1,2, …, R-1]***

Soit:

$$N = d_m.R^m + d_{m-1}.R^{m-1} + \ldots + d_1.R + d_0 + d_{-1}.R^{-1} + \ldots d_{-n}.R^{-n}$$

Le programme principal

- La fonction « main » contient le programme principal
- Le programme exécutable binaire commence par exécuter les instructions de ce programme principal
- Sans fonction main, il est impossible de générer un programme exécutable

Directives du préprocesseur

- #include <math.h> : insère les interfaces des fonctions mathématiques comme par exemple ***fabs()*** qui évalue la valeur absolue d ’un réel

- #include <stdio.h> : entrées sorties standard

- #define chaine1 chaine2 : remplacement littéral de la chaîne de caractères chaine1 par chaine2

Les opérateurs arithmétiques

Addition +

```
a = 3;
c = a + 2;      // c = 5
```

Soustraction -

```
a = 3;
c = a - 2;      // c = 1
```

Multiplication *

```
a = 3;
b = 11;
c = a + b;    // c = 33
```

Division /

```
a = 75;
b = 3;
c = a / b;      // c = 25
```

Modulo %

Reste de la division

```
a = 75;
diz = a / 10;          // diz = 7
unite = a % 10      // unite = 5    c 'est à dire le reste
```

Qu'est-ce qu'un bloc d 'instructions ?

- Un bloc débute par une **accolade ouvrante** et se termine par une **accolade fermante**
- Il contient des déclarations de **variables internes** au bloc et des **instructions**
- Les instructions peuvent être elles aussi des blocs ou des commandes du langage C.
- Les lignes d 'instructions se terminent par des **points virgules**.

Les entrées sorties : printf, scanf

- Le prototype de ces fonctions est dans <stdio.h>
- printf permet d ’afficher du texte à l ’écran
- scanf permet d ’entrer du texte au clavier
- fprintf et fscanf permettent de lire et d ’écrire dans des fichiers

  Le programme de résolution de l’équation d’advection linéaire scalaire donne de nombreux exemples d’usage de ces fonctions

printf, scanf (suite)

- Ces fonctions utilisent des formats qui permettent de lire/écrire des variables de différents types :
  %e , %f : réels
  %le, %lf : réels de type double
  %d : entiers
  %ld : entiers long int
  %s : chaîne de caractères
  %c : un caractère

printf, scanf (suite)

- Le caractère « \ » (backslash) permet d ’utiliser certains caractères « non imprimables » , les plus utilisés sont les suivants :

  \n : fin de ligne
  \t : tabulation horizontale
  \a : sonnerie

printf, scanf : exemples

```
#include<stdio.h>
int i;
printf("  i = %d \n",i);

fscanf(infile,  "%d ",&i);
```

le fichier infile la valeur de i,
on passe à fscanf l 'adresse de i
c 'est à dire &i */

nb: Comme fscanf doit modifier la variable i, le paramètre passé à la fonction est l 'adresse de la variable i.

- La boucle **for** :
  for (initialisation ; test ; instruction)
  { instructions };

- Exemple : for (i = 0 ; i <= 50 ; i++) {
  printf(« i = %d\n »,i);
  }

  « Commencer à i =0, tant que i <= 50 , exécuter l ’instruction printf et incrémenter i »

Les tests

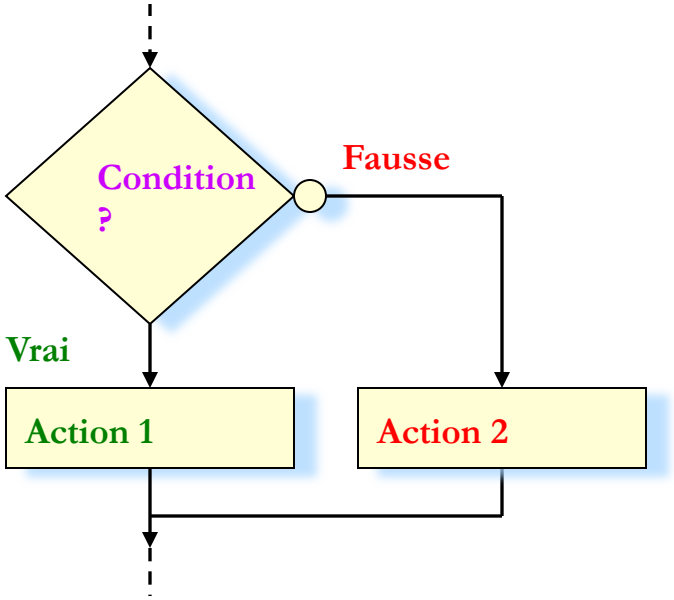
- Syntaxes :
  - if (expression_test)
    bloc d ’instructions 1

    Si expression_test est vraie on exécute le bloc d ’instructions 1, sinon on passe à la suite.

  - if (expression_test)
    bloc d ’instructions 1
    else
    bloc d ’instructions 2

    expression_test est vraie on exécute le bloc d ’instructions 1 sinon on exécute le bloc 2.

Instruction conditionnelle IF…. ELSE



Si la condition est vrai

- faire l 'action 1
- sinon faire action 2

```
if( condition )      action1;
else                 action2;
```

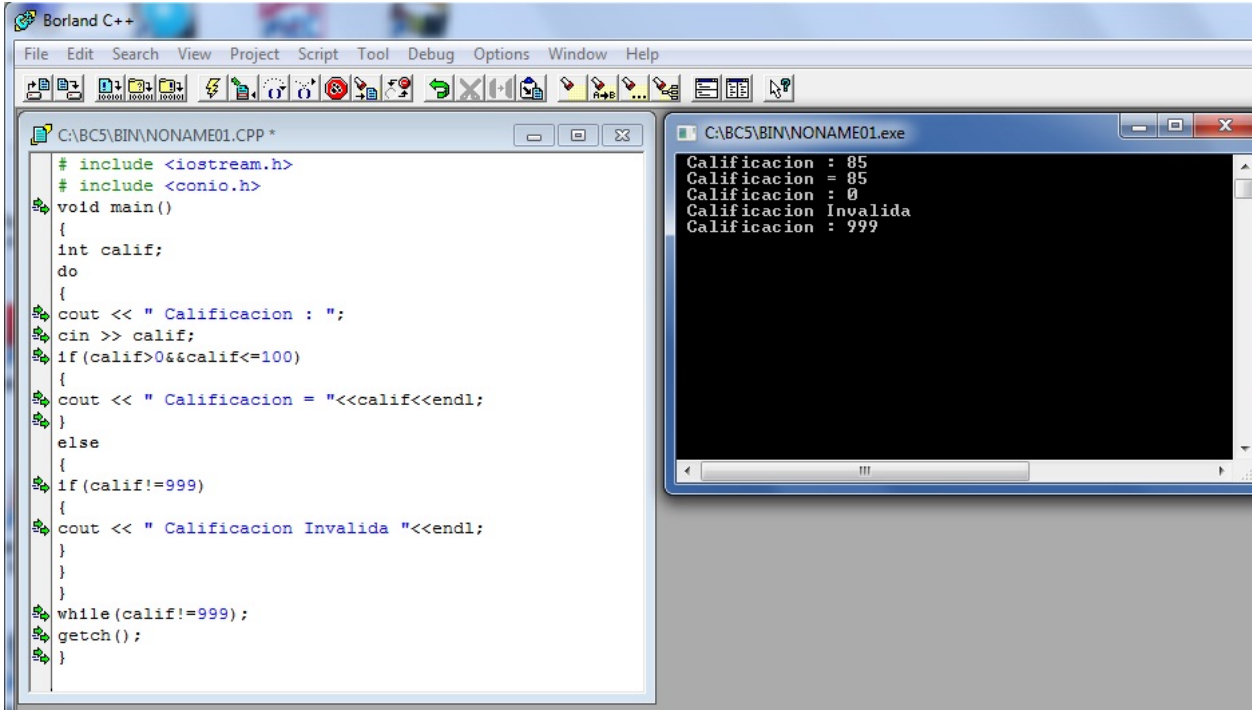
```
if( a > b )      c = c - a;
   else          c = c- b;
```

```
if( a > b )
{   c = c - a;
    d = c - a;
}
   else          {   c = c - b;
                     d = c - b;
                 }
```

```
if( a > b )
{   c = c - a;
    d = c - a;
}
```

```
if( (a > b) && ( b > 0) )      c = c - a;
   else                        c = c- b;
```

Tests (suite)



- Enchaînement de *if* :

```
if (expression_test1)
        bloc_d_instructions_1
else if (expression_test2)
        bloc_d_instructions_2
 else if (expression_test3)
        bloc_d_instructions_3
...
else
    bloc_d_instructions_final
```

Expressions évaluées dans les tests

- if (a = b) : erreur fréquente, expression toujours vraie !
- if (a == b) : a égal à b
- if (a != b) : a différent de b
- if (a > b) : a supérieur à b
- if ((a >=b)&&(a>0)) : a supérieur ou égal à b *et* a positif
- if ((a<=b)||(a>0)) : a inférieur ou égal à b *ou* a positif
- ...

Instruction conditionnelle SWITCH … CASE

Remplace une suite de IF .. ELSE

```
switch ( expression )
{
   case valeur1 :  instruction(s) 1;     break;
   case valeur2 : instruction(s) 2;     break;
   case valeur3 : instruction(s) 3;     break;
        default : instruction(s) par défaut;
}
```

•**expression** doit être un entier ou un caractère.

•**Switch** compare cette expression aux valeurs des différents **case**.

•L ’instruction **break** permet de sortir d ’un bloc. Ici cela permet de ne pas tester les **case** suivants.

•**Default** est effectué si aucun des case n ’est effectué.

```
switch ( valeur )
{
   case 1 :  led1 = 1;   led2 = 0;    led3 = 0;     break;
   case 2 : led1 = 0;    led2 = 1;    led3 = 0;     break;
   case 7 : led3 = 1;
   case 3 : led1 = 1;    led2 = 1;     break;
 default : led1 = 0;     led2 = 0;    led3 = 0;
}
```

Répondre par
- 0
- 1
- x si pas de modification

Valeur	led1	led2	led3
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	x
4	0	0	0
5	0	0	0
6	0	0	0
7	1	1	1

Boucles et branchements

- La boucle **while** :

  while(test) { instructions;}

- exemple

```
...
int i;
    i = 0;
  while (i < 10) {
    printf(« i = %d \n »,i);
    i++; }
...
```

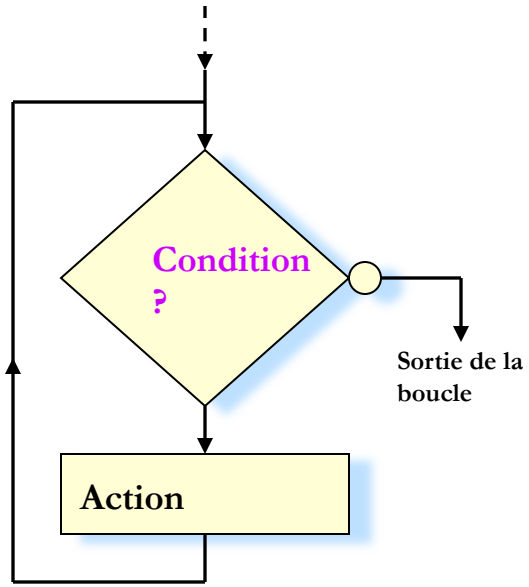
« Tant que i est inférieur à 10, écrire i à l ’écran, incrémenter i »

- La boucle do … while :

```
do {
        instructions;
} while (test);
```

- permet d ’exécuter au moins une fois les instructions avant d ’évaluer le test

Tant que condition vraie faire...



Boucle WHILE

```
while (condition)
{
 action;
  … ;
}
```

Cette boucle est identique à une boucle FOR

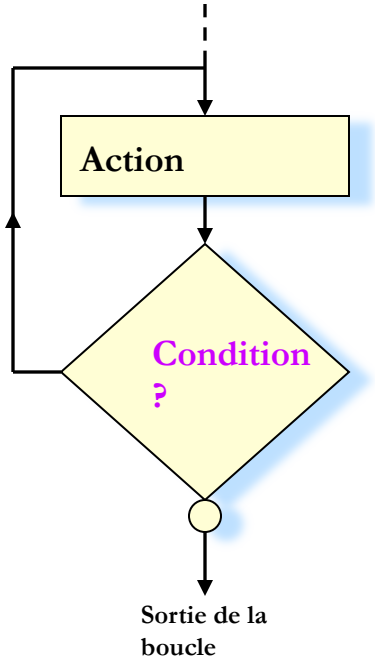
```
while ( x < 5 )
{
    action;
    x++;
}
```

```
while ( 1 )
{
    action;
    … … ;
}
```

```
x = 10;
while ( x < 5 )
{
    action;
    x++;
}
```

```
while ( bouton == 0 )
{
    action;
}
```

Boucle DO … WHILE



Faire… tant que condition vraie

```
do
{
 action;
  … ;
}
while (condition);
```

Action est toujours exécutée au moins une fois que condition soit vraie ou fausse.

```
x = 0;
total = 0;
do
{
   total = total + x;
   x++;
}
while ( x < 5 );
```

Nombre d 'exécutions de la boucle : 5

A la sortie de la boucle total = 10

Opérateur de Comparaison

Supérieur	>	if(a > b)
Supérieur ou égal	>=	if(a >= b)
Inférieur	<	if(a < b)
Inférieur ou égal	<=	if(a <= b)
Egal	==	if(a == b)
Différent	!=	if(a != b)

```
if( a = b )   ...
        else ...
```

Ne pas confondre :
- l ' affectation =
- le test de comparaison ==

Le compilateur ne détectera pas d 'erreur mais le résultat ne sera pas celui escompté. . . .

Opérateurs logique de test

ET	&&	if((a > b) && (b > 0))
OU	||	if((a > b) || (b > 0))
NON	!	if(!(a == b))

Simplification

```
if( a )      ...
    else     ...
```

ce test est :
- vrai si a > 0
- faux si a = 0

Opérateurs

Incrément ++

```
a = 55;
a++;          // a = 56
```

Décrément --

```
a = 20;
a--;          // a = 19
```

Affectation =

```
a = 55;       // a prend la valeur 55
a = a + b;    //  la somme de a et b est mise dans a
```

Séparateur ;

```
a = 55;       b = a + 1;
```

- **Bloc d 'instruction { .. .. .. }**
- **Liste de variables ou de constantes**

- Début et fin des fonctions
- Dans les tableaux pour séparer les éléments
- Utilisé dans les tests IF..THEN SWITCH.. CASE
- Utilisé dans les boucles FOR WHILE DO..WHILE
- Etc ...

Séparateur ,

```
unsigned char TAB[5] = {  2, 5, 8, 20, 5 };   // tableau de 5 valeurs
```

Exemple de fonction

```
/*Directives du preprocesseur*/
#include <math.h>
#include <stdio.h>
#define MAX(a,b) ((a) > (b) ? (a) : (b))

/* Variables externes, globales,
déclarées dans le programme principal */
extern double  a;
extern long int imax;
extern double * rho;
extern double * fphy;
extern double * fnum;

      flux_numerique          /* Interface de la fonction */
  /* Debut de bloc d 'instructions */
  long int i; /* Variable locale */

  /* Schema decentre */
  for (i = 0 ; i <= imax-1 ; i++)
  {
     fnum[i] = 0.5*(fphy[i] + fphy[i+1]) - 0.5*fabs(a)*(rho[i+1]-
rho[i]);
  }
} /* Fin de bloc */
```

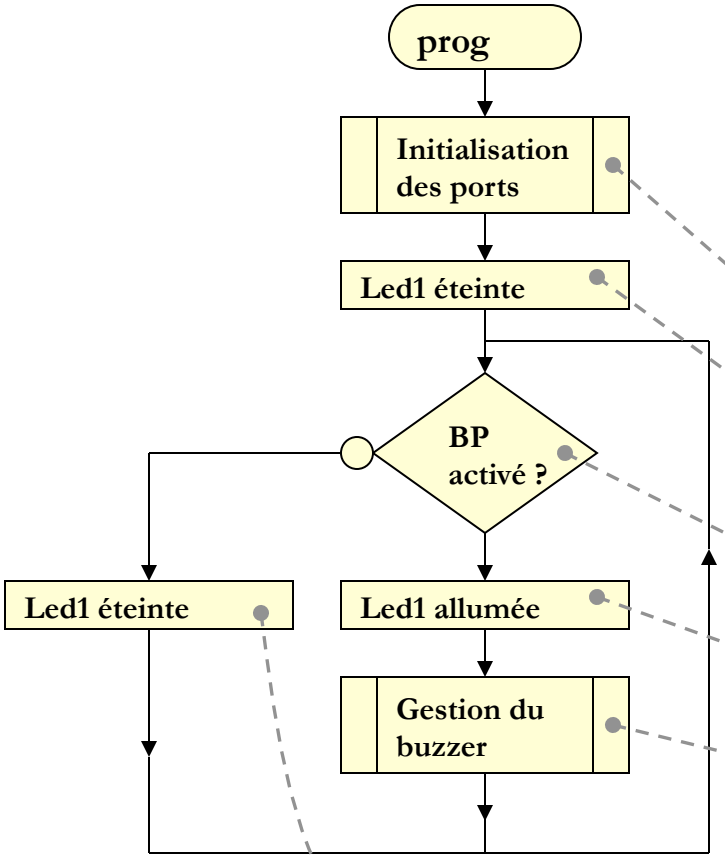
Interface d ’une fonction

- Syntaxe générale :
  type_retourné nom_de_fonction (paramètres);

  Exemples :

- **void main(void);**
- **void sauve(double *rho,long int imax,char filename,long int num);**
- **int somme(int a, int b);**
  -

Les fonctions




```
void buzzer (void)
{
        ....
        .....
}

void init_port (void)
{
        ....
        .....
}

void main (void)
{
   init_port();
   led1 = 0;
      while(1)
      {
         if(bp == 0)
         {
            led1 = 1;
            buzzer();
         }
         else  led1 = 0;
      }
}
```

Visibilité des variables

- Les variables déclarées à l ’***intérieur*** du bloc d ’une fonction sont « visibles » à l ’intérieur du bloc uniquement

- Le variables déclarées à l ’***extérieur*** du bloc d ’une fonction sont globales et accessibles à toutes les fonctions du ***fichier*** en question

- EXEMPLE : tout le texte suivant est dans le même fichier

```
int a;  /* variable globale a */

void fct1(double a) {
a = 0.5; /* Modif du paramètre local a */
}

void fct2(void) {
double a;
a = 0.5; /* Modif de la variable locale a */
}

void fct3(void) {
a = 1; /* Modif de la variable globale a */
}
```

Visibilité des variables (suite)

- Il peut être nécessaire que plusieurs fonctions accèdent aux mêmes variables. Pour cela, ces variables doivent être « visibles » des deux fonctions. On peut alors :
  - « passer » les variables en question dans les paramètres des fonctions
  - travailler sur des variables externes globales

Exemple : variables globales

```
int a,b;
int somme(void);

void main(void)
{ int c;
a = 10.0;
b = 5.0;
c = somme( );
}
```

variables globales a et b

Exemple : variables globales externes (suite)

```
  * sous programme somme.c
    dans un fichier séparé du
    programme principal */
extern int a;
extern int b;
int somme(void)
{
int c; /* variables locale c */
                c = a + b;
                return c;
}
```

Exemple : passage de paramètres

```
int somme(int a, int b);


void main(void)
{ int a,b,c;
a = 10.0;
b = 5.0;
c = somme( a,b);
}
```

Exemple : passage de paramètres (suite)

```
/* sous programme so
int somme(int a, int b)
{
int c; /* variable locale c */
            c = a + b;
            return c;
}
```

Dans ce cas, les variables a et b ne sont pas globales. On parle ici de passage d 'argument par valeur le sous programme somme effectue une copie locale des variables a et b que lui transmet le programme principal et travaille ensuite sur ces copies.
Conséquence : les variables a et b sont locales au sous programme somme.

Les structures

- Il est possible de créer des ensembles de variables regroupées sous le même nom, on parle de variables agglomérées.

  - **Exemple :**
    struct individu {
    char nom[30];
    char prenom[50];
    int age; }
  - **Utilisation :**
    struct individu eleve,professeur,directeur;
    eleve.nom = « einstein »;
    eleve.prenom = « albert »;
    eleve.age = 25;

- On peut utiliser des tableaux de structures, des pointeurs de structures …

Pointeurs
&
Allocation de mémoire

Plan

- Rappel
  - Les variables en mémoire
  - Adressage des variables
- Notion de pointeur
- Pointeurs & Tableaux
- Allocation dynamique de la mémoire

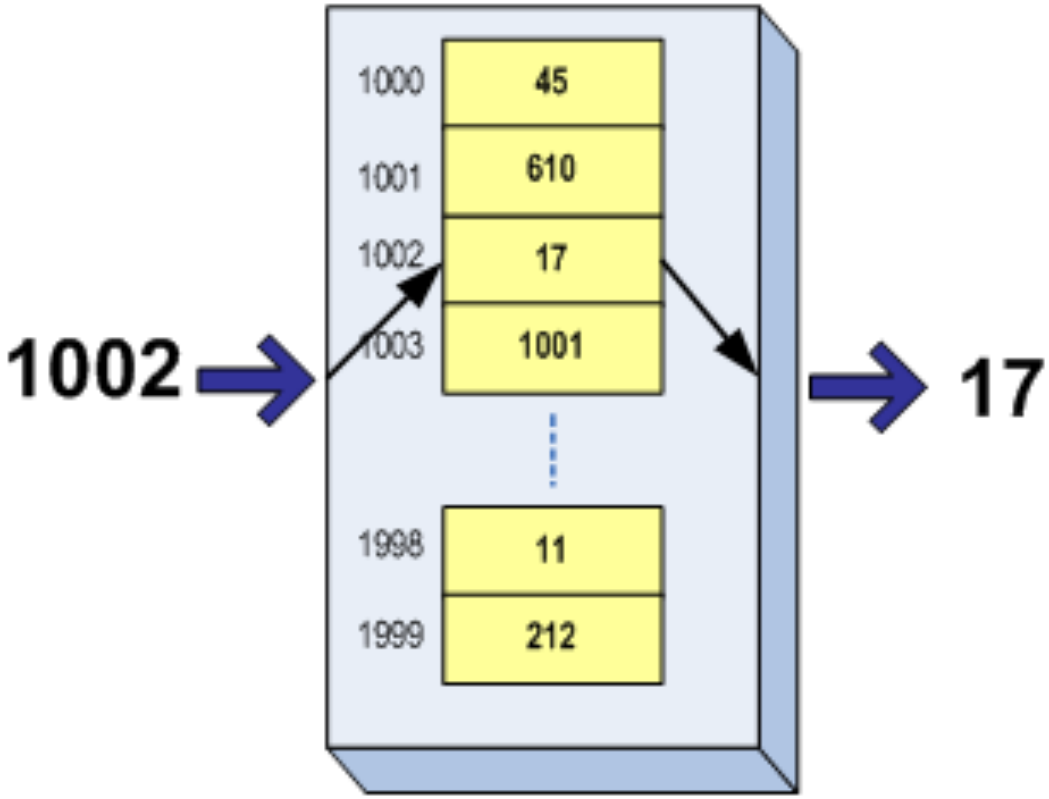
- Une variable est destinée à contenir une valeur du type avec lequel elle est déclarée.

  Physiquement cette valeur se situe en mémoire.

- Exemple : un entier nommé x

```
int x;  // Réserve un emplacement pour un entier en mémoire.
x = 10; // Écrit la valeur 10 dans l'emplacement réservé.
```

Les variables en mémoire



Adressage des variables

En C , on dispose de deux mode d'adressage :

- ***Adressage direct*** *:* **Accès au contenu d'une variable par le nom de la variable.**
- ***Adressage indirect*** *:* **Accès au contenu d'une variable par le biais de l'adresse de la variable.**

L'opérateur &

permet de récupérer l'adresse d'une variable :

&x ceci est l'adresse de x .

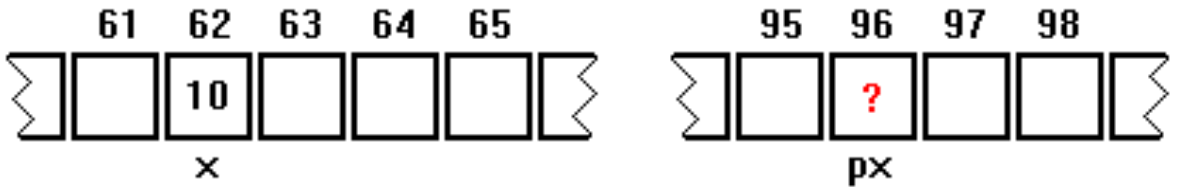
Notion de pointeur

- **Définition : "***Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable.* **"**
- **Chaque pointeur est limité à un type de données.**
- **Déclaration : <type>* nom_pointeur;**

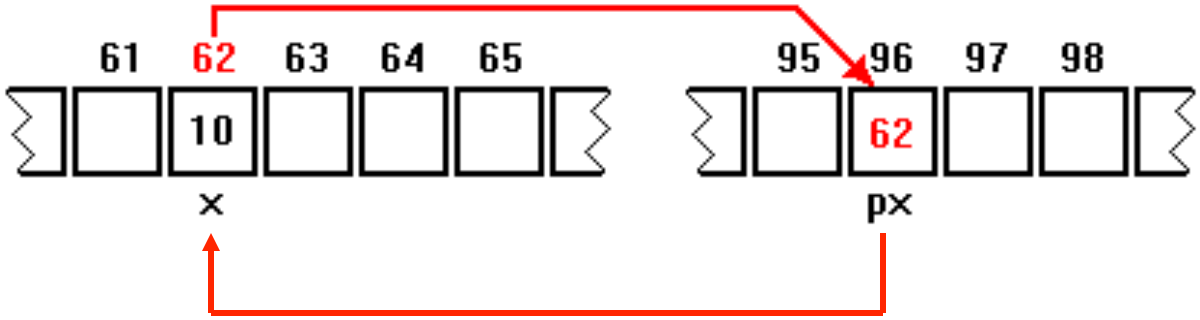
```
int *px; // Réserve un emplacement pour stocker une adresse
                  // mémoire.

px = &x; // Ecrit l'adresse de x dans cet emplacement.
```

Notion de pointeur

int *px;



px = &x;

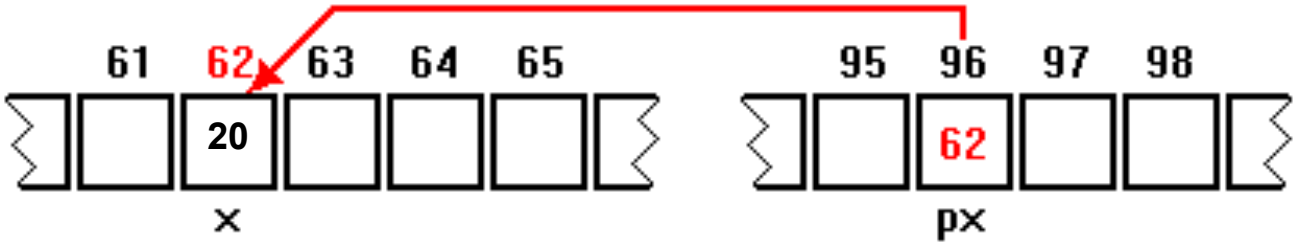


px pointe sur x

Notion de pointeur

- L'opérateur * permet d'accéder au contenu de la variable pointée :

```
*px = 20; // Maintenant x est égal à 20.
```



Notion de pointeur

Après les instructions: <type> a;

<type> *p;

p = &a;

p pointe sur a

p désigne &a (adresse de a)

*p désigne a (valeur ou contenu de a)

Allocation dynamique de la mémoire

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation.

➔ *Allocation dynamique*

La réservation de la mémoire peut donc seulement se faire *pendant l'exécution du programme***.**

Allocation dynamique de la mémoire

- *La fonction malloc* **(<*stdlib.h*>)**

  malloc(<N>) fournit l'adresse d'un bloc en mémoire de <N> octets libres ou la valeur zéro (NULL)s'il n'y a pas assez de mémoire

- *L'opérateur unaire sizeof*

  sizeof (<type>) fournit la grandeur pour un objet du type <type>

Allocation dynamique de la mémoire

```
int nbr;
int *p;
printf(" Entrez le nombre de valeurs :");
scanf("%d", &nbr);
p = (int *) malloc(nbr*sizeof(int));
```

- *Il est toujours utile de tester si l'allocation a eu lieu*

Allocation dynamique de la mémoire

- *La fonction free* **(<*stdlib.h*>)**

  free(<Pointeur>)

  libère le bloc de mémoire (déjà alloué) désigné par le <Pointeur>

- **Il est conseillé d'affecter la valeur zéro NULL au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.**

- **Si nous ne libérons pas explicitement la mémoire à l'aide free, alors elle est libérée automatiquement à la fin du programme.**

Allocation dynamique de la mémoire

```
#include <stdio.h>

#include <stdlib.h>

main()

{

  int i;

  int tab[10] ;


  for(i=0; i<10; i++)

        tab[i]=i;

  for(i=0; i<10; i++)

        printf("%d\n",tab[i]);

}
```

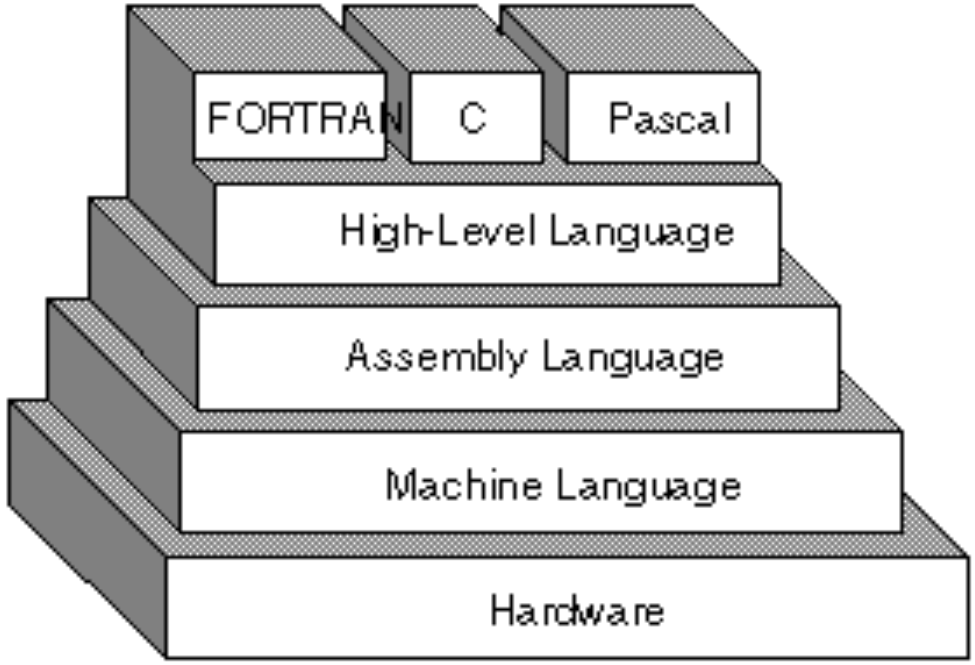
```
#include <stdio.h>
#include <stdlib.h>
main()
{
   int i, nbr;
   int* p;

   printf("entrer un nombre: ");
   scanf("%d",&nbr);

   p=(int*)malloc(nbr*sizeof(int));

   for(i=0; i<nbr; i++)
        *(p+i)=i;
   for(i=0; i<nbr; i++)
        printf("%d\n",*(p+i));

   free(p);
}
```

Double Pointeur **

- char ** tableau;

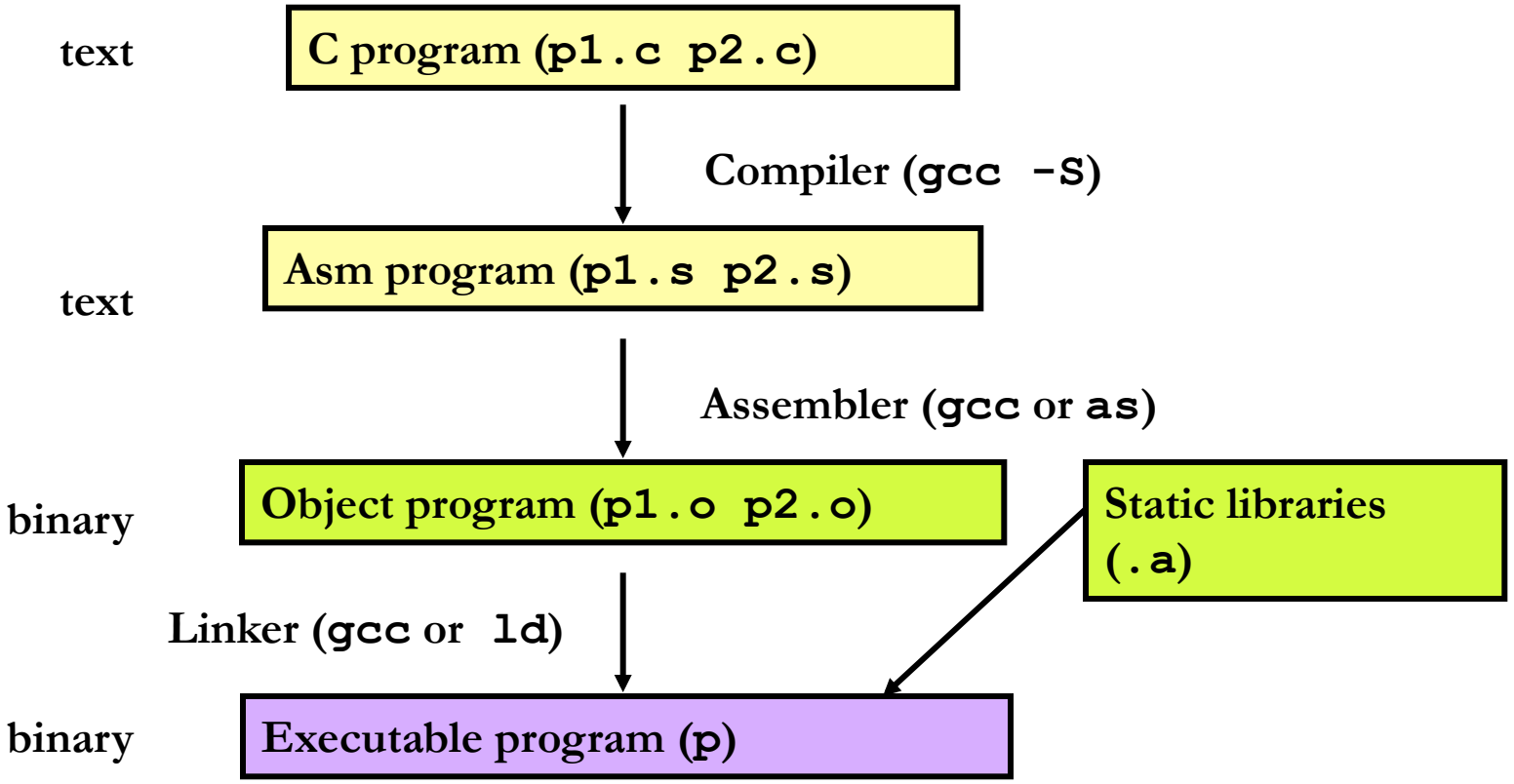
```
tableau = malloc (sizeof (char *) * nbElem1); //alloue
l'espace pour stocker les pointeurs

for (int i=0;i<nbElem1;i++))
{
tableau[i] = malloc(sizeof(char) * nbElem2);//assigne a
chaque pointeur un espace en ram
}
```

Assembleur


FORTRAN
C
Pascal
High-Level Language
Assembly Language
Machine Language
Hardware

Du fichier texte C au Code Objet

- Codage en C `p1.c p2.c`
- Compilation avec la commande : `gcc -O p1.c p2.c -o p`
  - Optimisation du code (`-O`)
  - Fichier binaire executable `p`



Compilation en Assembleur

- Code C

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Générer l'assembleur

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtenu avec la commande

```
gcc -O -S code.c
```

On obtient : `code.s`

Code

Code pour `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`

Instruction Machine

```
int t = x+y;
```

- C
  - Addition de 2 entiers

```
addl 8(%ebp),%eax
```

- Assembleur
  - Addition 2 4-byte integers
    - "Long" words en GCC
  - Opérandes:
    - `y`: Registre `%eax`
    - `x`: Mémoire `M[%ebp+8]`
    - `t`: Registre `%eax`
      - Return valeur de la fonction `%eax`

Similaire

y += x

```
0x401046:    03 45 08
```

- Code Objet
  - Instruction 3-byte
  - adresse `0x401046`

Déplacement de données

`movl` *Source*,*Dest*:

- Move 4-byte ("long") word

- Types d'opérands
  - Immediate: Constant integer data
    - Comme en C mais avec le préfixe '`$`'
    - `$0x400`, `$-533`
    - Coder sur 1, 2, ou 4 bytes
  - Registres : 8 integer
    - `%esp` et `%ebp` utilisation spéciale
  - Memory: 4 byte consecutifs
    - Différents mode d'affichage

Movl Utilisation

	Source	Destination		C Analog
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

- Il est impossible de faire un transfer de type mémoire-mémoire avec une seule instruction

Utilisation du Simple Mode d'adressage

- Normal (R) Mem[Reg[R]]
  - Registre R spécifie les adresses mémoire

```
movl  (%ecx),%eax
```

- Déplacement D(R) Mem[Reg[R]+D]
  - Registre R spécifie ou la région mémoire commence
  - Déplacement constant D spécifie par l' offset

```
movl  8(%ebp),%edx
```

Utilisation du Simple Mode d'adressage

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

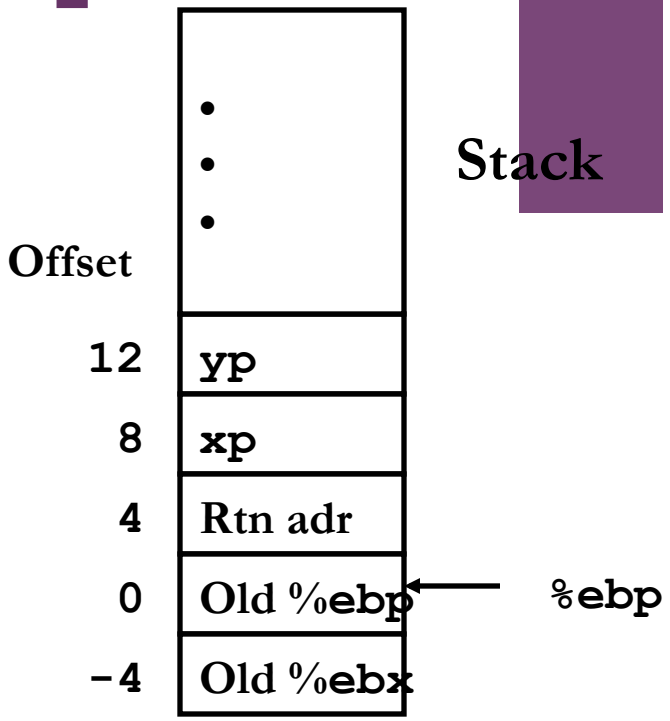
Initialisation

Corp

Fin

Comprendre le Swap

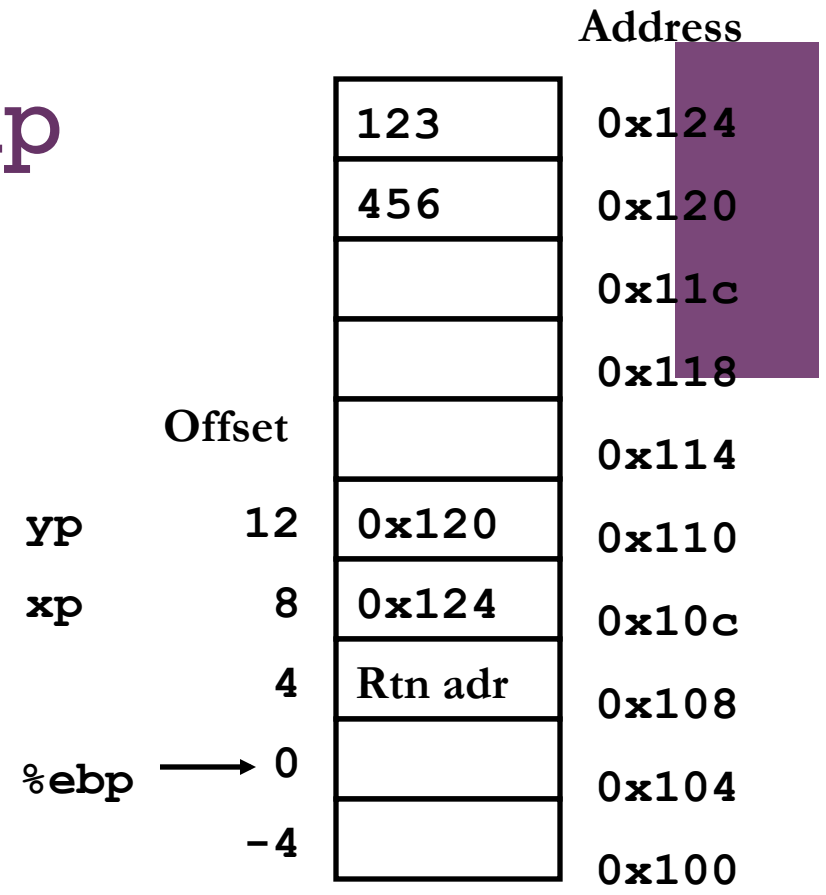
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

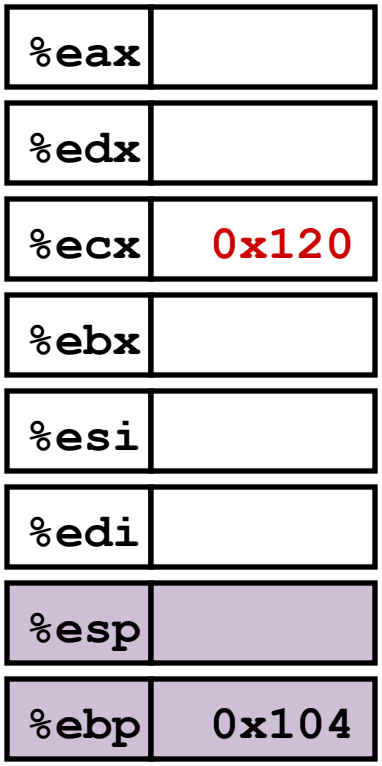
```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx   # edx = xp
movl (%ecx),%eax    # eax = *yp (t1)
movl (%edx),%ebx    # ebx = *xp (t0)
movl %eax,(%edx)    # *xp = eax
movl %ebx,(%ecx)    # *yp = ebx
```

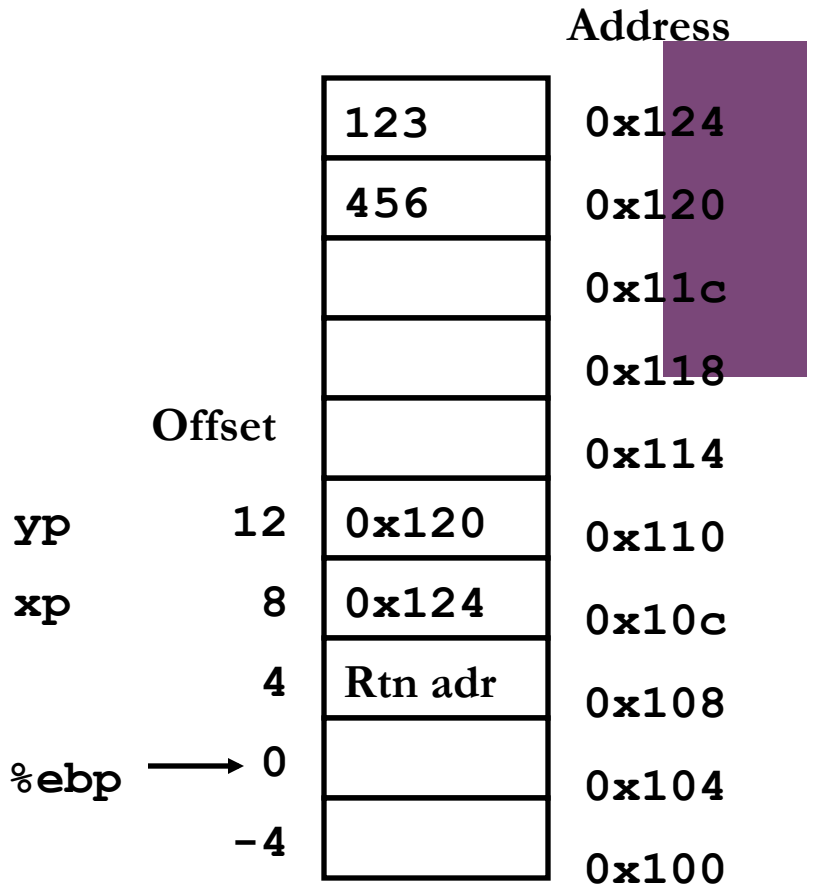
Comprendre le Swap




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Comprendre le Swap

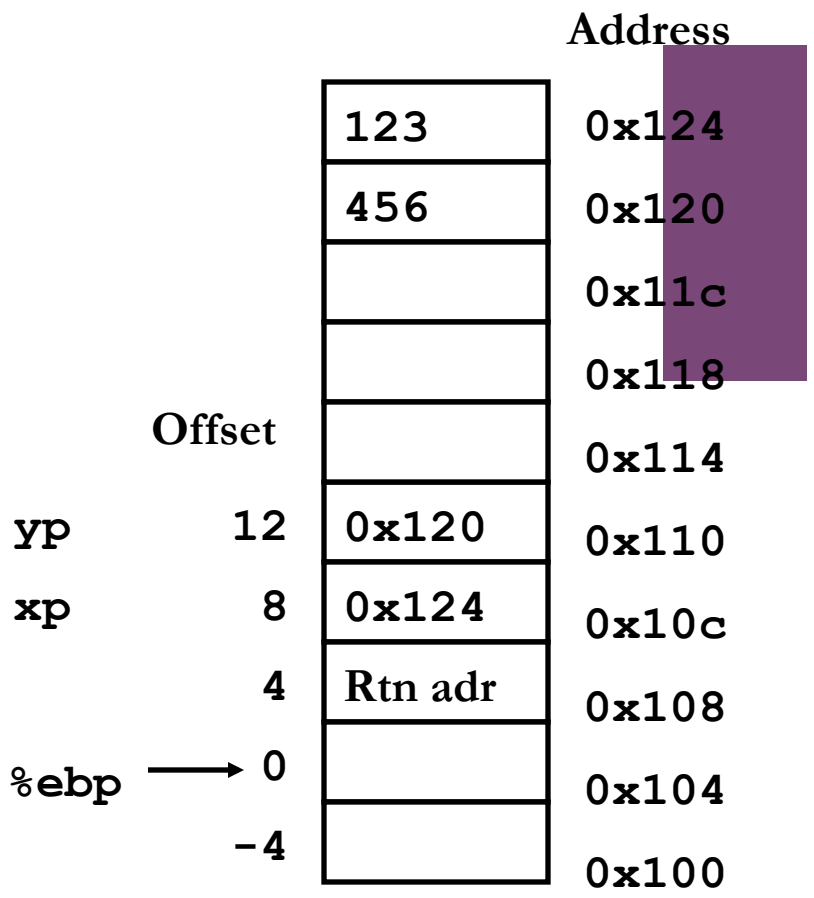





```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Comprendre Swap

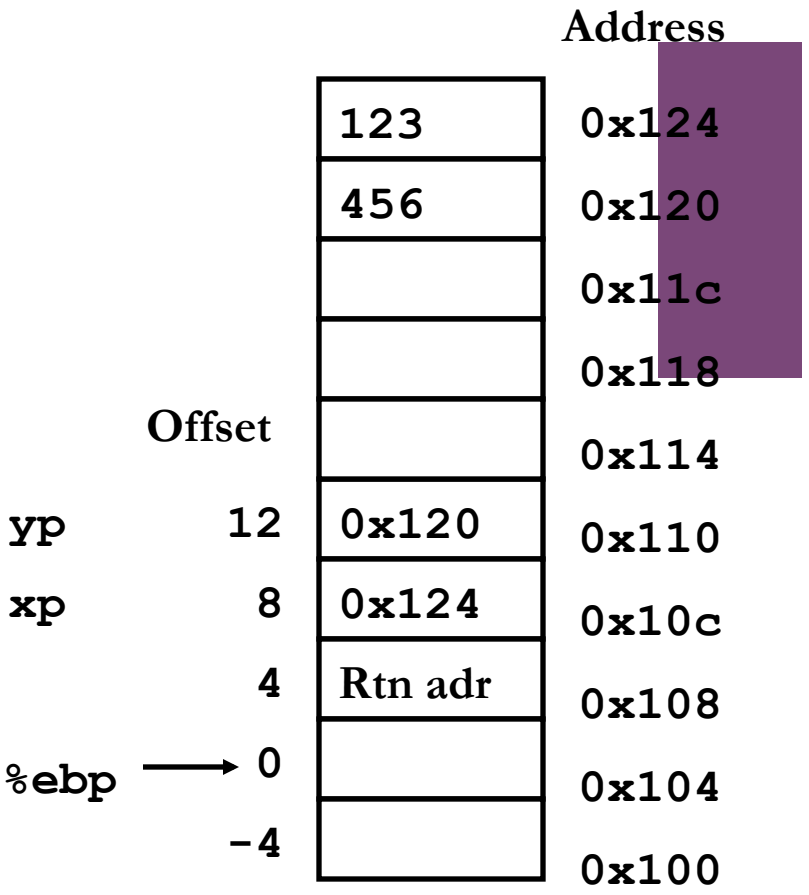
%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Comprendre le Swap

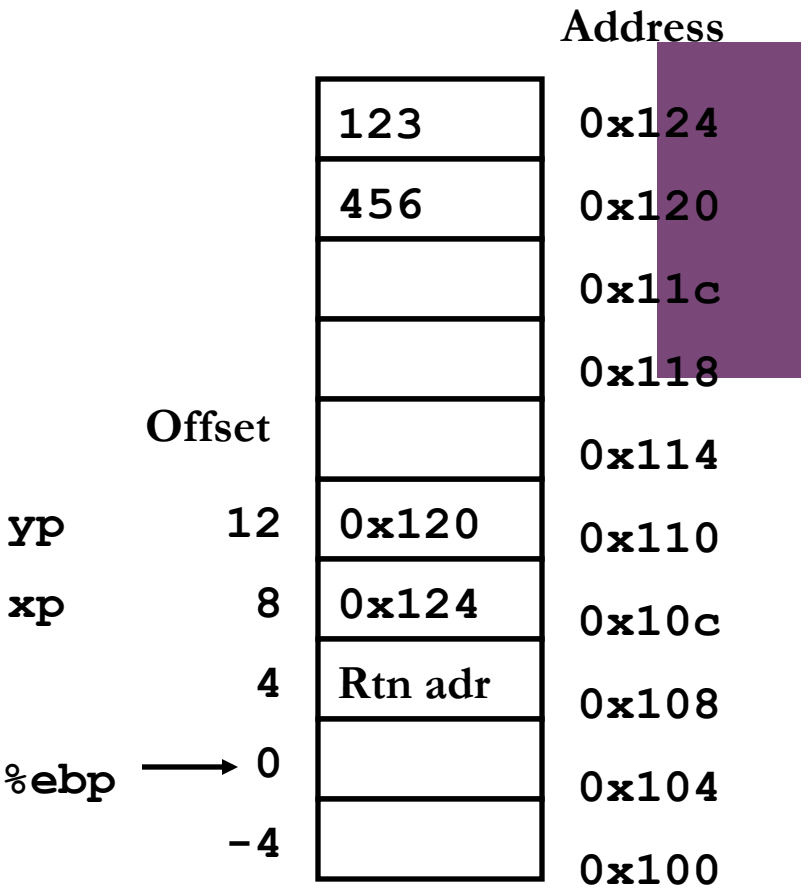
%eax	456
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Comprendre le Swap

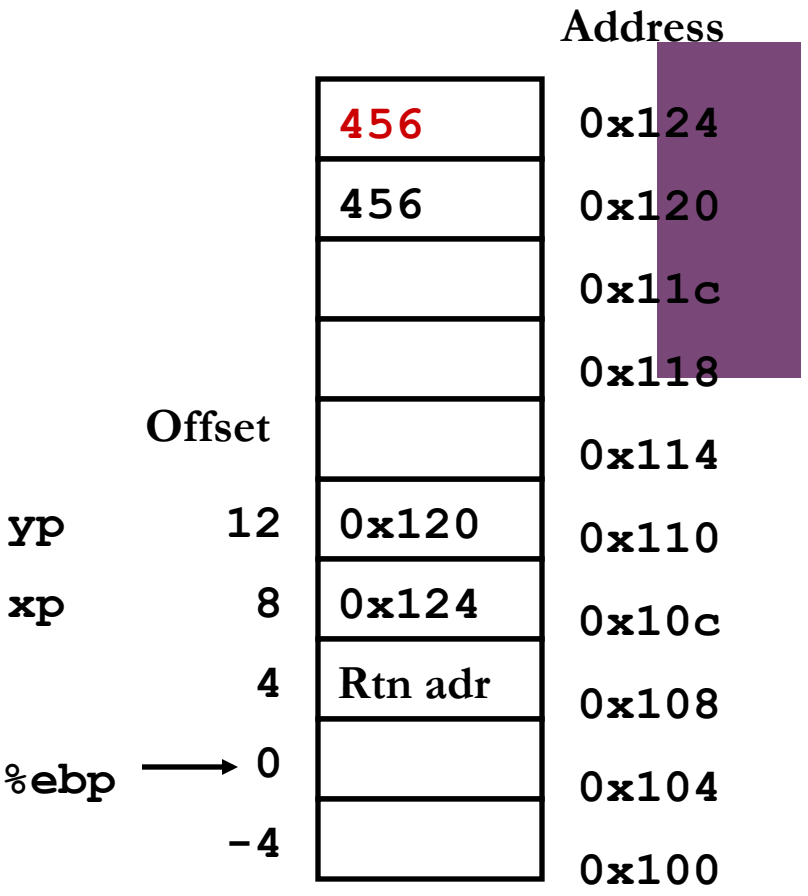
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Comprendre le Swap

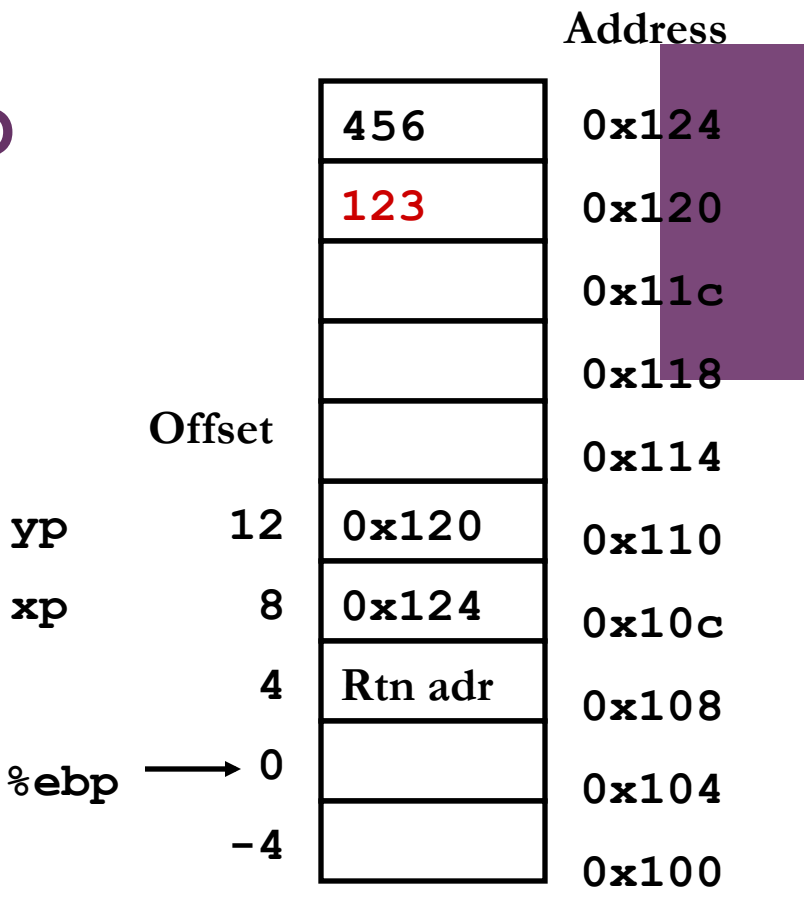
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Comprendre le Swap

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

Adressage indexé

- La forme générale
  - D(Rb,Ri,S) Mem[Reg[Rb] +S*Reg[Ri]+ D]
    - D: Constant "deplacement" 1, 2, or 4 bytes
    - Rb: Base register: Any of 8 integer registers
    - Ri: Index register: Any, exception pour `%esp`
      - Sinon`%ebp`,
    - S: Scale: 1, 2, 4, or 8
- Quelque fois :
  - (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]
- D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri] +D]
- (Rb,Ri,S) Mem[Reg[Rb] +S*Reg[Ri]]

Calcul des adresses

%edx	0xf000
%ecx	0x100

Expression	Calcul	Addresse
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

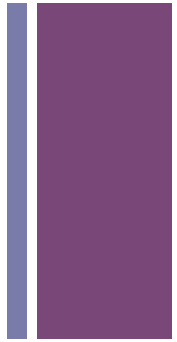
Calcul des adresses

- `leal` *Src*,*Dest*
  - *Src* adressage en mode expression
  - *Dest*
- Utilisation
  - Calcul de l'adresses simplement
  - Ex : translation de `p = &x[i];`
  - Calcul de l'adresse arithmétique x + k*y
    - k = 1, 2, 4, or 8.

Opérations Arithmétiques

Format		Actions		
■ Instruction avec 2 opérands				
`addl`	*Src*,*Dest*	*Dest* = *Dest* + *Src*		
`subl`	*Src*,*Dest*	*Dest* = *Dest* – *Src*		
`imull`	*Src*,*Dest*	*Dest* = *Dest* * *Src*		
`sall`	*k*,*Dest*	*Dest* = *Dest* << *k*	`shll`	
`sarl`	k,*Dest*	*Dest* = *Dest* >> *k*	Arithmetique	
`shrl`	*k*,*Dest*	*Dest* = *Dest* >> *k*	Logique	
`xorl`	*Src*,*Dest*	*Dest* = *Dest* ^ *Src*		
`andl`	*Src*,*Dest*	*Dest* = *Dest* & *Src*		
`orl`	*Src*,*Dest*	*Dest* = *Dest* | *Src*		

Opérations Arithmétiques



Format	Opération

- Instructions avec un seul opérand

`incl` *Dest*	*Dest* = *Dest* + 1
`decl` *Dest*	*Dest* = *Dest* - 1
`negl` *Dest*	*Dest* = -*Dest*
`notl` *Dest*	*Dest* = ~*Dest*

Utilisation de leal pour des opérations arithmétiques

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```

```
arith:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

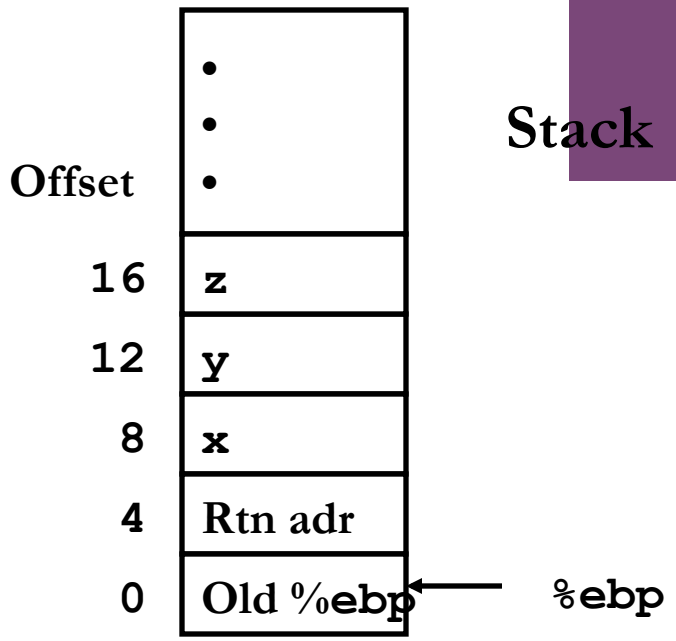
Initialisation

Corp

Fin

Comprendre arith

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```




```
movl 8(%ebp),%eax          # eax = x
movl 12(%ebp),%edx         # edx = y
leal (%edx,%eax),%ecx      # ecx = x+y   (t1)
leal (%edx,%edx,2),%edx    # edx = 3*y
sall $4,%edx               # edx = 48*y (t4)
addl 16(%ebp),%ecx         # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax     # eax = 4+t4+x (t5)
imull %ecx,%eax            # eax = t5*t2 (rval)
```

Comprendre "arith"

```
int arith
   (int x, int y, int z)
{
   int t1 = x+y;
   int t2 = z+t1;
   int t3 = x+4;
   int t4 = y * 48;
   int t5 = t3 + t4;
   int rval = t2 * t5;
   return rval;
}
```

```
# eax = x
  movl 8(%ebp),%eax
# edx = y
  movl 12(%ebp),%edx
# ecx = x+y   (t1)
  leal (%edx,%eax),%ecx
# edx = 3*y
  leal (%edx,%edx,2),%edx
# edx = 48*y (t4)
  sall $4,%edx
# ecx = z+t1 (t2)
  addl 16(%ebp),%ecx
# eax = 4+t4+x (t5)
  leal 4(%edx,%eax),%eax
# eax = t5*t2 (rval)
  imull %ecx,%eax
```

Exemple

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

$2^{13} = 8192, 2^{13} - 7 = 8185$

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%eax
    xorl 12(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    movl %ebp,%esp
    popl %ebp
    ret
```

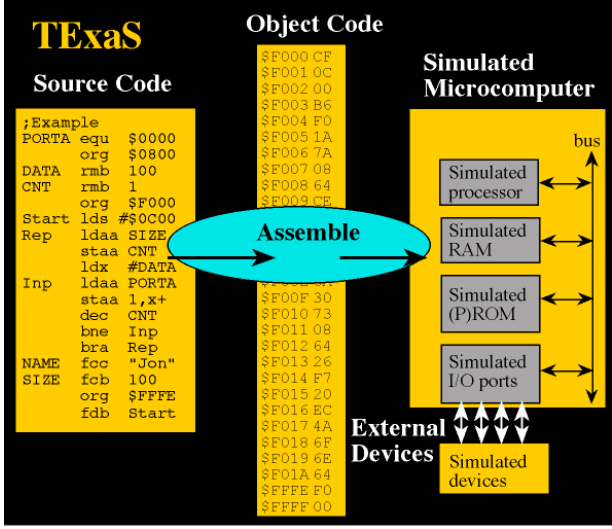
Initialisation

Corps

Fin

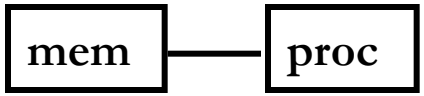
```
movl 8(%ebp),%eax      eax = x
xorl 12(%ebp),%eax     eax = x^y      (t1)
sarl $17,%eax          eax = t1>>17   (t2)
andl $8185,%eax        eax = t2 & 8185
```

Machines Abstractes

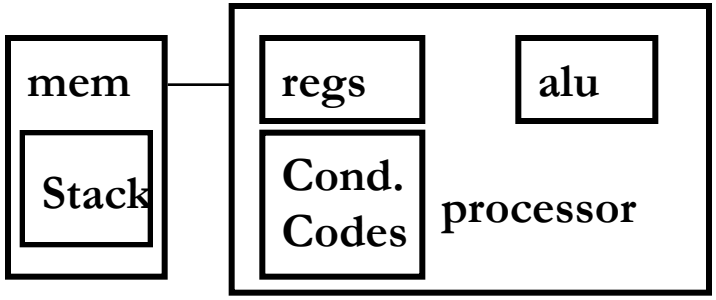


Modèle de Machine

C



Assembleur



Données

1) `char`
2) `int`, `float`
3) `double`
4) `struct`, array
5) pointer

1) byte
2) 2-byte word
3) 4-byte long word
4) contiguous byte allocation
5) address of initial byte

Control

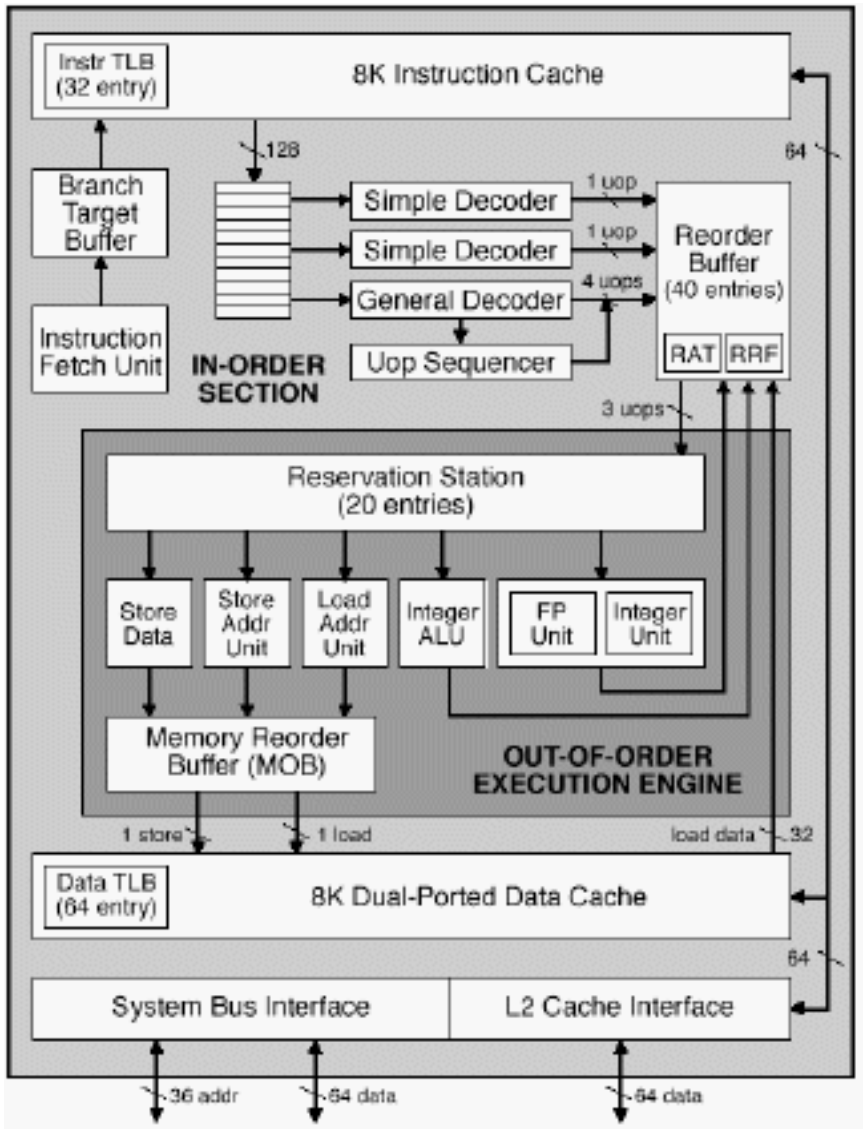
1) loops
2) conditionals
3) `switch`
4) Proc. call
5) Proc. return

3) branch/jump
4) `call`
5) `ret`

Pentium Pro (P6)

- Historique : Sortie en 95
  - Basé sur Pentium II, Pentium III, et Celeron
  - Similaire avec P4, mais différents détails
- Caractéristiques
  - Traduction des instruction dans un format plus régulier
    - Beaucoup d'instruction mais très simple
  - Execution d'opérations en paralléle
    - Jusqu'à 5 à la fois
  - Pipeline de
    - 12–18 cycles

PentiumPro : Schéma Block



Quel Assembleur?

Intel/Microsoft Format

```
lea   eax,[ecx+ecx*2]
sub   esp,8
cmp   dword ptr [ebp-8],0
mov   eax,dword ptr [eax*4+100h]
```

GAS/Gnu Format

```
leal (%ecx,%ecx,2),%eax
subl $8,%esp
cmpl $0,-8(%ebp)
movl $0x100(,%eax,4),%eax
```

- Intel/Microsoft v.s. GAS
  - Changement de l'ordre des opérands

    `mov` Dest, Src `movl` Src, Dest
  - Pour les constantes pas de '$', notation 'h' à la fin

    `100h` `$0x100`
  - La longueur des opérands est indiquée dans l'opération

    `sub` `subl`
  - L'adresse est représentée sous forme explicite

    `[eax*4+100h]` `$0x100(,%eax,4)`