

# **Programmation Assembleur X86 IA32**

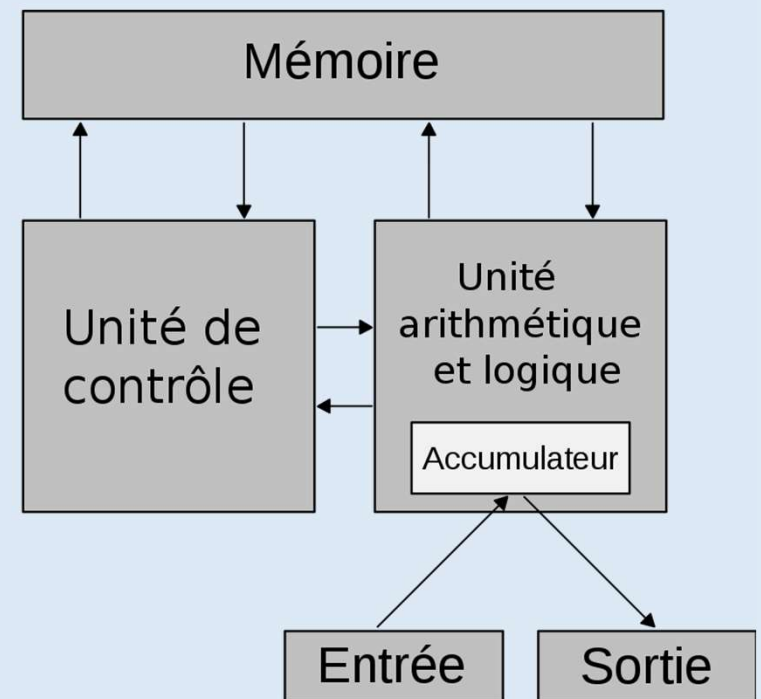
# Plan de la formation

- **Architecture des processeurs**
- **Principes de compilation**
- **Bases de l'assembleur, registres, adressage, pile**
- **Variables, tableaux, branchements, entrées/sorties**
- **Calculs numériques et logiques**
- **Calculs flottants SSE et X87**
- **Fonctions (déclaration, appels, paramètres)**
- **Programmation modulaire**

# Architecture des processeurs

# Architecture des processeurs

- La mémoire (volatile ou permanente) contient les données et les programmes à effectuer
- L'unité de contrôle (CPU ou UCT) réalise les instructions en séquence.
- L'unité de traitement (ALU ou UAL) effectue les opérations de base.
- Les dispositifs d'entrée-sortie communiquent avec le monde extérieur (disques, écran, souris, réseau, USB, etc.)



Architecture de Von Neumann

# Architecture des processeurs

## Types de mémoire

**La mémoire centrale (ou mémoire interne) permettant de mémoriser temporairement les données lors de l'exécution des programmes.**

**La mémoire centrale est réalisée à l'aide de circuits électroniques spécialisés rapides.**

**La mémoire centrale correspond à ce que l'on appelle la mémoire vive.**

# Architecture des processeurs

## Types de mémoire

**La mémoire de masse permet de stocker des informations à long terme, y compris lors de l'arrêt de l'ordinateur.**

**La mémoire de masse correspond aux dispositifs de stockage magnétiques, optiques ou électroniques tels que le disque dur, les CDROM ou DVD-ROM, disques SSD, etc.**

# Architecture des processeurs

## Caractéristiques de la mémoire

- **La capacité** : c'est le nombre total de bits que contient la mémoire. Elle est exprimée en octets.
- **Le format des données** : c'est le nombre de bits que l'on peut mémoriser par case mémoire. Aussi appelé largeur du mot mémorisable (mot de 8, 16, 32 ou 64 bits).
- **Le temps d'accès** : c'est le temps qui s'écoule entre l'instant où a été lancée une opération de lecture/écriture en mémoire et l'instant où la première information est disponible sur le bus de données.

# Architecture des processeurs

## Caractéristiques de la mémoire

- **Le temps de cycle** : il représente l'intervalle minimum qui doit séparer deux demandes successives de lecture ou d'écriture.
- **Le débit** : c'est le nombre maximum d'informations lues ou écrites par seconde.
- **La non-volatilité** caractérisant l'aptitude d'une mémoire à conserver les données lorsqu'elle n'est plus alimentée électriquement.



# Architecture des processeurs

## Hiérarchie mémoire

Dans l'idéal, on veut :

- Une mémoire extrêmement rapide
- De très grande taille

afin que la mémoire ne limite pas les performances du micro processeur.

Or, le coût d'une mémoire varie comme sa rapidité :

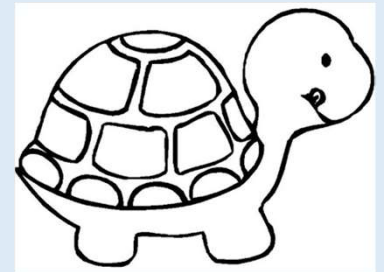
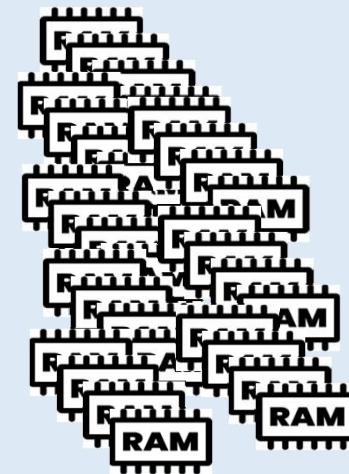
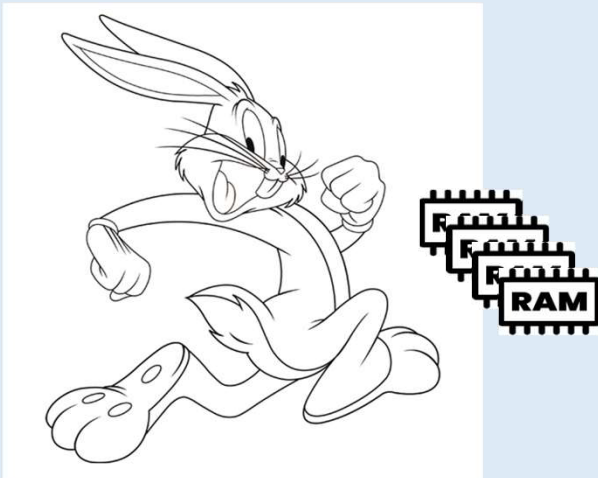
mémoire rapide = mémoire chère

# Architecture des processeurs

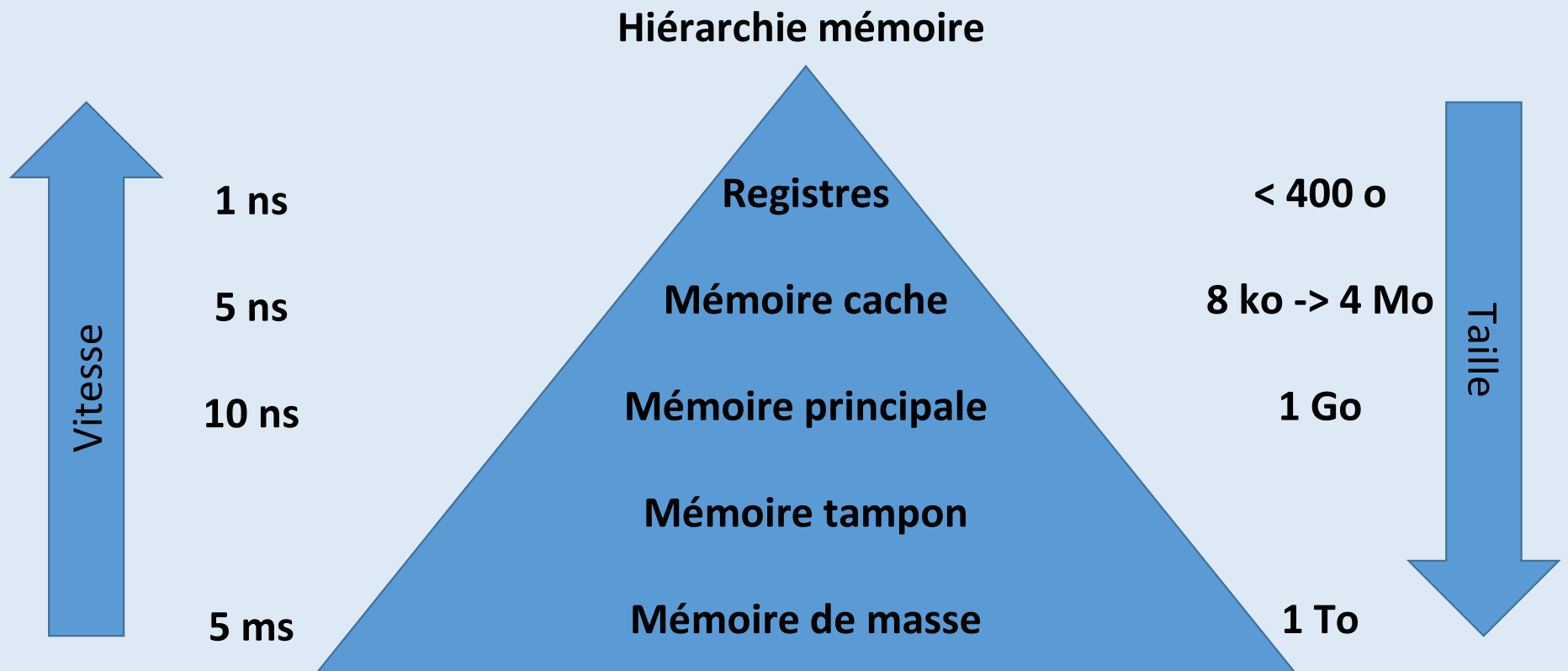
## Hiérarchie mémoire

Compromis coût / performance =

- Peu de mémoires rapides
- Beaucoup de mémoires lentes



# Architecture des processeurs



# Architecture des processeurs

## Hiérarchie mémoire

- **Les registres sont les éléments de mémoire les plus rapides. Ils sont situés au niveau du processeur et servent au stockage des opérandes et des résultats intermédiaires.**
- **La mémoire cache est une mémoire rapide de faible capacité destinée à accélérer l'accès à la mémoire centrale en stockant les données les plus utilisées.**

# Architecture des processeurs

## Hiérarchie mémoire

- La mémoire principale est l'organe principal de rangement des informations.
- La mémoire tampon sert de mémoire intermédiaire entre la mémoire centrale et les mémoires de masse. Elle joue le même rôle que la mémoire cache.
- La mémoire de masse est une mémoire périphérique de grande capacité utilisée pour le stockage permanent ou la sauvegarde des informations.

# Architecture des processeurs

## Quizz

**Dans un souci de performances, quel type de mémoire va-t-on utiliser en priorité pour les programmes écrits en assembleur ?**

# Architecture des processeurs

## Le CPU

**Le CPU (Central Processing Unit) est un circuit électronique cadencé au rythme d'une horloge interne, délivrant des « tops » à une fréquence donnée.**

**La fréquence d'horloge correspondant au nombre d'impulsions par seconde, s'exprime en Hertz (Hz).**

**La fréquence d'horloge est généralement un multiple de la fréquence du système (FSB, Front-Side Bus), c'est-à-dire un multiple de la fréquence de la carte mère**

# Architecture des processeurs

## Le CPU

A chaque top d'horloge le processeur exécute une action, correspondant à une instruction ou une partie d'instruction.

Le CPI (Cycles Par Instruction) permet de représenter le nombre moyen de cycles d'horloge nécessaire à l'exécution d'une instruction sur un microprocesseur.



La puissance est exprimée en MIPS (Millions d'Instructions Par Seconde)

$$P = \frac{\text{Fréquence(Hz)}}{\text{CPI}}$$



# Architecture des processeurs

## Fonctionnement simplifié du processeur

**Une instruction est l'opération élémentaire que le processeur peut accomplir.**

**Les instructions sont stockées dans la mémoire principale, en vue d'être traitée par le processeur.**

**Une instruction est composée de deux parties :**

- **le code opération, représentant l'action que le processeur doit accomplir ;**
- **le code opérande, définissant les paramètres de l'action.**

# Architecture des processeurs

## Fonctionnement simplifié du processeur

**Le code opérande dépend de l'opération.**

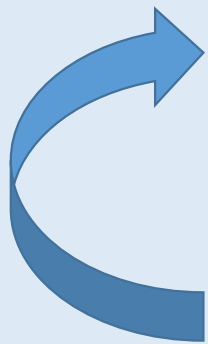
**Il peut s'agir d'une donnée, d'un registre ou bien d'une adresse mémoire.**

**Le nombre d'octets d'une instruction est variable selon le type de donnée (l'ordre de grandeur est de 1 à 4 octets).**

# Architecture des processeurs

## Fonctionnement simplifié du processeur

En l'absence d'interruption, le processeur réalise en boucle les opérations suivantes :



- Lecture de l'instruction située à l'adresse courante (contenue dans le Compteur Ordinal (CO, ou Program Counter PC)
- Exécution de l'instruction
- Augmentation du CO de la valeur correspondant à la taille de l'instruction et des ses paramètres

Ce cycle est connu sous le nom de cycle Fetch / Decode / Execute

# Architecture des processeurs

## Fonctionnement simplifié du processeur

**Soit un processeur 16 bits imaginaire.**

Il contient trois registres : **PC** (Program Counter), **IR** (Instruction Register) et **AC** un registre accumulateur

**Il sait utiliser au moins 3 instructions :**

Opcode (binaire)	Opcode (décimal)	Description
0001	1	Charge AC par un contenu mémoire
0010	2	Stocke AC dans la mémoire
0101	5	Additionne dans AC depuis la mémoire

## Le format d'une instruction 16 bits est le suivant :



# Architecture des processeurs

## Fonctionnement simplifié du processeur

Soit le programme suivant situé à l'adresse mémoire 0x300 et composé des instructions : 1940, 5941 et 2941

- 1940 : Charge AC avec le contenu de l'adresse 940
- 5941 : Additionne dans AC avec le contenu de l'adresse 941
- 2941 : Range le contenu de AC dans l'adresse 941

Adresse	Instruction
0x300	1940
0x302	5941
0x304	2941

*Mémoire programme*

Adresse	Donnée
0x940	0003
0x941	0002

*Mémoire données*

Registre	Valeur
PC	
IR	
AC	

*Registres*

# Architecture des processeurs

## Fonctionnement simplifié du processeur

### Exécution de l'instruction 1

Adresse	Instruction
0x300	1940
0x302	5941
0x304	2941

*Mémoire programme*

Adresse	Donnée
0x940	0003
0x941	0002

*Mémoire données*

Registre	Valeur
PC	302
IR	
AC	

*Etape Fetch*

*Etape Execute*

***Decode 1940 :  
Charge AC avec le  
contenu de l'adresse 940***

# Architecture des processeurs

## Fonctionnement simplifié du processeur

### Exécution de l'instruction 2

Adresse	Instruction
0x300	1940
0x302	5941
0x304	2941

*Mémoire programme*

Adresse	Donnée
0x940	0003
0x941	0002

*Mémoire données*

Registre	Valeur
PC	304
IR	5941
AC	0005

*Etape Fetch*

*Etape Execute*

***Decode 5941 :  
Additionne dans AC avec le  
contenu de l'adresse 941***

# Architecture des processeurs

## Fonctionnement simplifié du processeur

### Exécution de l'instruction 3

Adresse	Instruction
0x300	1940
0x302	5941
0x304	2941

Mémoire programme

Adresse	Donnée
0x940	0003
0x941	0005

Mémoire données

Registre	Valeur
PC	306
IR	2941
AC	0005

*Etape Fetch*

*Etape Execute*

**Decode 2941 :**  
**Range le contenu de AC dans**  
**l'adresse 941**



# Architecture des processeurs

**Le CPU est chargé de lire en séquence les instructions contenues dans la mémoire.**

**Chaque instruction est identifiée par un code, suivi par un ou plusieurs arguments.**

**Le CPU lit une instruction, l'exécute, puis passe à la suivante.**

**Ci-contre, le code, pour processeur x86 32 bits, du programme qui permet d'afficher le message « Hello, World » dans la console d'affichage.**

00000000	48656C6C6F2C20576F726C640A00
0000000E	89E5
00000010	83C4FC
00000013	6AF5
00000015	E8(00000000)
0000001A	89C3
0000001C	6A00
0000001E	8D45FC
00000021	50
00000022	6A0E
00000024	68[00000000]
00000029	53
0000002A	E8(00000000)
0000002F	83ECFC
00000032	6A00
00000034	E8(00000000)
00000039	F4

Adresses

Instructions

# Architecture des processeurs

## Quizz

Quelle est la taille de ce programme ?

00000000	48656C6C6F2C20576F726C640A00
0000000E	89E5
00000010	83C4FC
00000013	6AF5
00000015	E8(00000000)
0000001A	89C3
0000001C	6A00
0000001E	8D45FC
00000021	50
00000022	6A0E
00000024	68[00000000]
00000029	53
0000002A	E8(00000000)
0000002F	83ECFC
00000032	6A00
00000034	E8(00000000)
00000039	F4

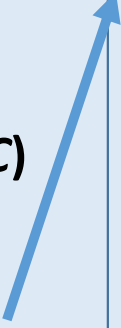
Adresses

Instructions

# Architecture des processeurs

L'instruction à exécuter est repérée par le  
Compteur Ordinal CO (ou *Program Counter : PC*)

Ici, le programme commence à l'adresse  
0000000E, c'est le point d'entrée du programme.



00000000	48656C6C6F2C20576F726C640A00
0000000E	89E5
00000010	83C4FC
00000013	6AF5
00000015	E8(00000000)
0000001A	89C3
0000001C	6A00
0000001E	8D45FC
00000021	50
00000022	6A0E
00000024	68[00000000]
00000029	53
0000002A	E8(00000000)
0000002F	83ECFC
00000032	6A00
00000034	E8(00000000)
00000039	F4

Adresses

Instructions

# Architecture des processeurs

Comme la première instruction est 89  
(« MOV » = affectation) et qu'elle accepte un  
argument (E5 ici), le compteur ordinal va  
augmenter de 2 octets après avoir exécuté  
cette instruction.

Il va alors lire, puis exécuter l'instruction située  
en 00000010.

00000000	48656C6C6F2C20576F726C640A00
0000000E	89E5
00000010	83C4FC
00000013	6AF5
00000015	E8(00000000)
0000001A	89C3
0000001C	6A00
0000001E	8D45FC
00000021	50
00000022	6A0E
00000024	68[00000000]
00000029	53
0000002A	E8(00000000)
0000002F	83ECFC
00000032	6A00
00000034	E8(00000000)
00000039	F4

AdressesInstructions

# Architecture des processeurs

Les instructions les plus courantes permettent de :

- Copier des données d'une zone de la mémoire à une autre
- Effectuer des opérations arithmétiques simples (+, -, \*, /) ou booléennes
- Evaluer des comparaisons entre opérandes
- Réaliser des déplacements dans le programme

00000000 48656C6C6E2C20576F726C640A00

0000000E 89E5

00000010 83C4FC

00000013 6AF5

00000015 E8(00000000)

0000001A 89C3

0000001C 6A00

0000001E 8D45FC

00000021 50

00000022 6A0E

00000024 68[00000000]

00000029 53

0000002A E8(00000000)

0000002F 83ECFC

00000032 6A00

00000034 E8(00000000)

00000039 F4

# Architecture des processeurs

## Exemples d'affectations

- **r1    CONST**

- Écrit CONST dans registre r1
- CONST = constante codée dans le programme

- **r3    r1 OP r2**

- Lit registres r1 et r2, exécute l'opération OP, écrit le résultat dans r3
- Opérations sur les entiers: add, sub, mul, div, and, or, xor, shift, etc...
- Opérations sur les flottants: fadd, fsub, fmul, fdiv, fsqrt, ...

- **r2    LOAD r1**

- Utilise la valeur contenue dans registre r1 comme **adresse mémoire**
- Lit la valeur stockée en mémoire à cette adresse
- Copie la valeur dans registre r2

# Architecture des processeurs

## Exemples d'instructions de contrôle

- **JNE adresse**

Branchement conditionnel (JNE = Jump if Not Equal)

Saute à l'adresse spécifiée si la dernière comparaison portait sur des valeurs différentes

- **JMP adresse**

Passe directement (Jump) à l'adresse spécifiée. Le Program Counter reçoit la valeur 'adresse'

- **CALL procédure**

Saute à l'adresse de début de la procédure spécifiée

# Architecture des processeurs

## Comparaison RISC / CISC

### CISC (Complex Instruction-Set Computer)

Taille d'instruction variable : Op code + n operandes

Exemple : Instruction (x86) MOV et RCL

Opcode	Mnemonic	Description
88 /r	MOV r/m8,r8	Move r8 to r/m8
89 /r	MOV r/m16,r16	Move r16 to r/m16
89 /r	MOV r/m32,r32	Move r32 to r/m32
8A /r	MOV r8,r/m8	Move r/m8 to r8
8B /r	MOV r16,r/m16	Move r/m16 to r16
8B /r	MOV r32,r/m32	Move r/m32 to r32
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16
8E /r	MOV Sreg,r/m16**	Move r/m16 to segment register
A0	MOV AL,moffs8*	Move byte at (seg offset) to AL
A1	MOV AX,moffs16*	Move word at (seg offset) to AX
A1	MOV EAX,moffs32*	Move doubleword at (seg offset) to EAX
A2	MOV moffs8*,AL	Move AL to (seg offset)
A3	MOV moffs16*,AX	Move AX to (seg offset)
A3	MOV moffs32*,EAX	Move EAX to (seg offset)
B0+ rb	MOV r8,imm8	Move imm8 to r8
B8+ rw	MOV r16,imm16	Move imm16 to r16
B8+ rd	MOV r32,imm32	Move imm32 to r32
C6 /0	MOV r/m8,imm8	Move imm8 to r/m8
C7 /0	MOV r/m16,imm16	Move imm16 to r/m16
C7 /0	MOV r/m32,imm32	Move imm32 to r/m32

Opcode	Mnemonic	Description
D0 /2	RCL r/m8, 1	Rotate 9 bits (CF, r/m8) left once.
D2 /2	RCL r/m8, CL	Rotate 9 bits (CF, r/m8) left CL times.
C0 /2 ib	RCL r/m8, imm8	Rotate 9 bits (CF, r/m8) left imm8 times.
D1 /2	RCL r/m16, 1	Rotate 17 bits (CF, r/m16) left once.
D3 /2	RCL r/m16, CL	Rotate 17 bits (CF, r/m16) left CL times.
C1 /2 ib	RCL r/m16, imm8	Rotate 17 bits (CF, r/m16) left imm8 times.
D1 /2	RCL r/m32, 1	Rotate 33 bits (CF, r/m32) left once.
D3 /2	RCL r/m32, CL	Rotate 33 bits (CF, r/m32) left CL times.
C1 /2 ib	RCL r/m32, imm8	Rotate 33 bits (CF, r/m32) left imm8 times.
D0 /3	RCR r/m8, 1	Rotate 9 bits (CF, r/m8) right once.
D2 /3	RCR r/m8, CL	Rotate 9 bits (CF, r/m8) right CL times.
C0 /3 ib	RCR r/m8, imm8	Rotate 9 bits (CF, r/m8) right imm8 times.
D1 /3	RCR r/m16, 1	Rotate 17 bits (CF, r/m16) right once.
D3 /3	RCR r/m16, CL	Rotate 17 bits (CF, r/m16) right CL times.
C1 /3 ib	RCR r/m16, imm8	Rotate 17 bits (CF, r/m16) right imm8 times.
D1 /3	RCR r/m32, 1	Rotate 33 bits (CF, r/m32) right once.
D3 /3	RCR r/m32, CL	Rotate 33 bits (CF, r/m32) right CL times.
C1 /3 ib	RCR r/m32, imm8	Rotate 33 bits (CF, r/m32) right imm8 times.
D0 /0 ROL r/m8, 1		Rotate 8 bits r/m8 left once.
D2 /0 ROL r/m8, CL		Rotate 8 bits r/m8 left CL times.
C0 /0 ib ROL r/m8, imm8		Rotate 8 bits r/m8 left imm8 times.
D1 /0 ROL r/m16, 1		Rotate 16 bits r/m16 left once.
D3 /0 ROL r/m16, CL		Rotate 16 bits r/m16 left CL times.
C1 /0 ib ROL r/m16, imm8		Rotate 16 bits r/m16 left imm8 times.
D1 /0 ROL r/m32, 1		Rotate 32 bits r/m32 left once.
D3 /0 ROL r/m32, CL		Rotate 32 bits r/m32 left CL times.
C1 /0 ib ROL r/m32, imm8		Rotate 32 bits r/m32 left imm8 times.
D0 /1	ROR r/m8, 1	Rotate 8 bits r/m8 right once.
D2 /1	ROR r/m8, CL	Rotate 8 bits r/m8 right CL times.
C0 /1 ib	ROR r/m8, imm8	Rotate 8 bits r/m8 right imm8 times.
D1 /1	ROR r/m16, 1	Rotate 16 bits r/m16 right once.
D3 /1	ROR r/m16, CL	Rotate 16 bits r/m16 right CL times.
C1 /1 ib	ROR r/m16, imm8	Rotate 16 bits r/m16 right imm8 times.
D1 /1	ROR r/m32, 1	Rotate 32 bits r/m32 right once.
D3 /1	ROR r/m32, CL	Rotate 32 bits r/m32 right CL times.
C1 /1 ib	ROR r/m32, imm8	Rotate 32 bits r/m32 right imm8 times.



# Architecture des processeurs

## Comparaison RISC / CISC

### CISC (Complex Instruction-Set Computer)

**Opérations complexes autorisées : traitement des chaînes de caractères (SCASB), des polynômes ou des complexes, recherche dans une table (XLAT), etc.**

**Modes d'adressage mémoire complexes autorisés**

**Accès mémoire largement utilisé**

**Exemple: IBM 360, VAX, Intel X86**

# Architecture des processeurs

## Comparaison RISC / CISC

**Des études statistiques sur des programmes ont montré les faits suivants.**

- **80 % des programmes n'utilisent que 20 % du jeu d'instructions.**
- **Les instructions les plus utilisées sont :**
  - **les instructions de chargement et de rangement,**
  - **les appels de sous-routines.**
- **Les appels de fonctions sont très gourmands en temps : sauvegarde et restitution du contexte et passage des paramètres et de la valeur de retour.**
- **80 % des variables locales sont des entiers.**
- **90 % des structures complexes sont des variables globales.**
- **La profondeur maximale d'appels imbriqués et en moyenne de 8. Une profondeur plus importante ne se rencontre que dans 1 % des cas.**

# Architecture des processeurs

## Comparaison RISC / CISC

### RISC (Reduced Instruction-Set Computer)

#### Codage uniforme des instructions

Toutes les instructions sont codées avec un même nombre de bits, généralement un mot machine. L'op-code se trouve à la même position pour toutes les instructions. Ceci facilite le décodage des instructions.

#### Registres indifférenciés et nombreux

Tous les registres peuvent être utilisés dans tous les contextes. Il n'y a par exemple pas de registre spécifique pour la pile. Les processeurs séparent cependant les registres pour les valeurs flottantes des autres registres.

# **Architecture des processeurs**

## **Comparaison RISC / CISC**

### **RISC (Reduced Instruction-Set Computer)**

#### **Limitation des accès mémoire**

**Les seules instructions ayant accès à la mémoire sont les instructions de chargement et de rangement. Toutes les autres instructions opèrent sur les registres. Il en résulte une utilisation intensive des registres.**

#### **Nombre réduit de modes d'adressage**

**Il n'y pas de mode d'adressage complexe. Les modes d'adresses possibles sont généralement immédiat, direct, indirect et relatifs.**

# Architecture des processeurs

## Comparaison RISC / CISC

### RISC (Reduced Instruction-Set Computer)

#### Nombre réduit de types de données

Types supportés : Entiers de différentes tailles (8, 16, 32 et 64 bits) et nombres flottants en simple et double précision.

Certains processeurs CISC comportent des instructions pour le traitement des chaînes de caractères, des polynômes ou des complexes

Exemple: MIPS, Alpha, PowerPC, Sparc

# Architecture des processeurs

## Jeux d'instructions RISC / CISC

- CISC

Code plus compact, prend moins de place en mémoire

Jeu d'instructions plus facile à faire évoluer

- Exemple x86: 16 $\Rightarrow$ 32  $\Rightarrow$  64 bits, MMX  $\Rightarrow$  SSE  $\Rightarrow$  SSE2  
 $\Rightarrow$  SSE3  $\Rightarrow$  etc...

- RISC

Microarchitecture plus simple

# Principes de compilation

# Principes de compilation

Pour obtenir un programme exécutable à partir d'un code source, au moins deux étapes sont nécessaires



Fichier source  
(C, ASM, etc.)



Fichier objet ou  
relogeable



Bibliothèques



Fichier  
exécutable

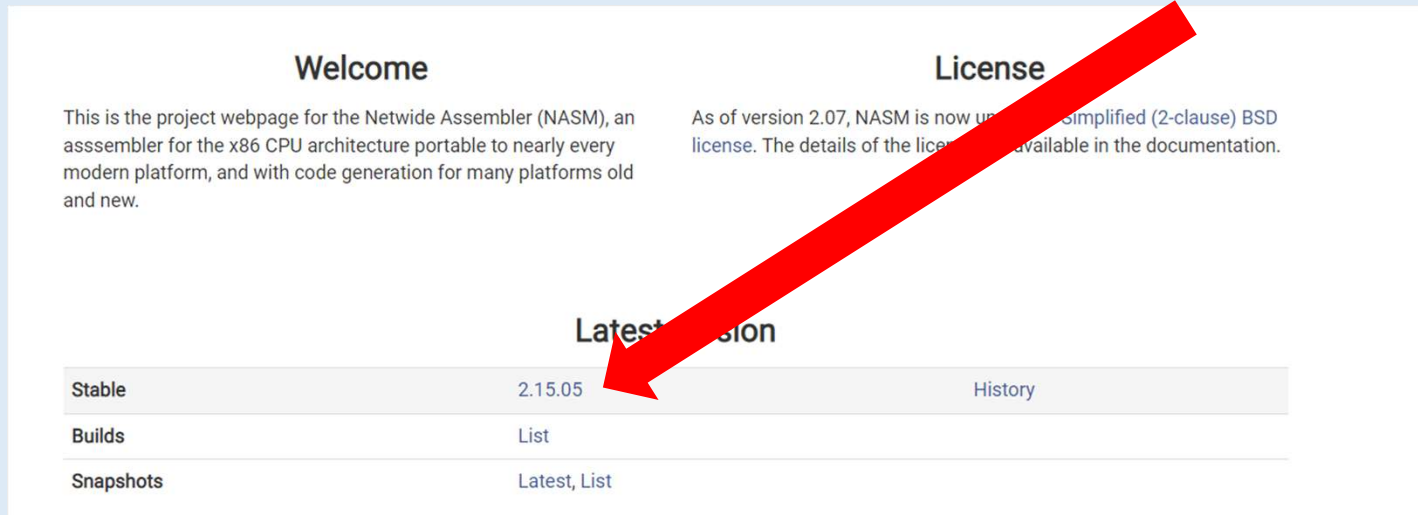


# Principes de compilation

## Installation des outils

Compilateur NASM : <http://www.nasm.us/>

Sur Windows, NASM s'installe avec les droits administrateurs.



















The screenshot shows the NASM project website. It features a 'Welcome' section on the left and a 'License' section on the right. Below these, there is a 'Latest Version' section with a table. A large red arrow points from the top right towards the 'Latest Version' section, specifically highlighting the '2.15.05' version number.

Latest Version	
Stable	2.15.05
Builds	List
Snapshots	Latest, List

# Principes de compilation

## Index of /pub/nasm/releasebuilds/2.15.05

Name	Last modified	Size	Description
 <a href="#">Parent Directory</a>		-	
 <a href="#">doc/</a>	2020-08-28 09:05	-	Documentation
 <a href="#">dos/</a>	2020-08-28 09:08	-	MS-DOS executables
 <a href="#">linux/</a>	2020-08-28 09:08	-	Linux packages
 <a href="#">macosx/</a>	2020-08-28 09:08	-	MacOS X packages
 <a href="#">win32/</a>	2020-08-28 09:08	-	Windows packages (32 bit)
 <a href="#">win64/</a>	2020-08-28 09:08	-	Windows packages (64 bit)
 <a href="#">git.id</a>	2020-08-28 09:08	41	Corresponding git revision ID
 <a href="#">nasm-2.15.05-xdoc.tar.bz2</a>	2020-08-28 09:05	900K	Downloadable documentation
 <a href="#">nasm-2.15.05-xdoc.tar.gz</a>	2020-08-28 09:05	1.0M	Downloadable documentation
 <a href="#">nasm-2.15.05-xdoc.tar.xz</a>	2020-08-28 09:05	804K	Downloadable documentation
 <a href="#">nasm-2.15.05-xdoc.zip</a>	2020-08-28 09:05	1.0M	Downloadable documentation
 <a href="#">nasm-2.15.05.tar.bz2</a>	2020-08-28 09:04	1.2M	Source code
 <a href="#">nasm-2.15.05.tar.gz</a>	2020-08-28 09:04	1.6M	Source code
 <a href="#">nasm-2.15.05.tar.xz</a>	2020-08-28 09:04	972K	Source code
 <a href="#">nasm-2.15.05.zip</a>	2020-08-28 09:05	1.8M	Source code

[Browse source code for this build](#)

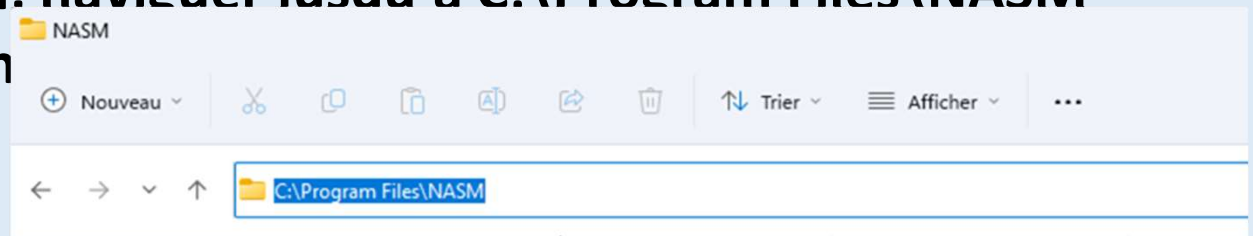
- Choisir la version win64/
- Puis l'installateur x64

## Index of /pub/nasm/releasebuilds/2.15.05/win64

Name	Last modified	Size	Description
 <a href="#">Parent Directory</a>		-	
 <a href="#">nasm-2.15.05-installer-x64.exe</a>	2020-08-28 09:08	1.0M	Installable package
 <a href="#">nasm-2.15.05-win64.zip</a>	2020-08-28 09:08	602K	Executable only

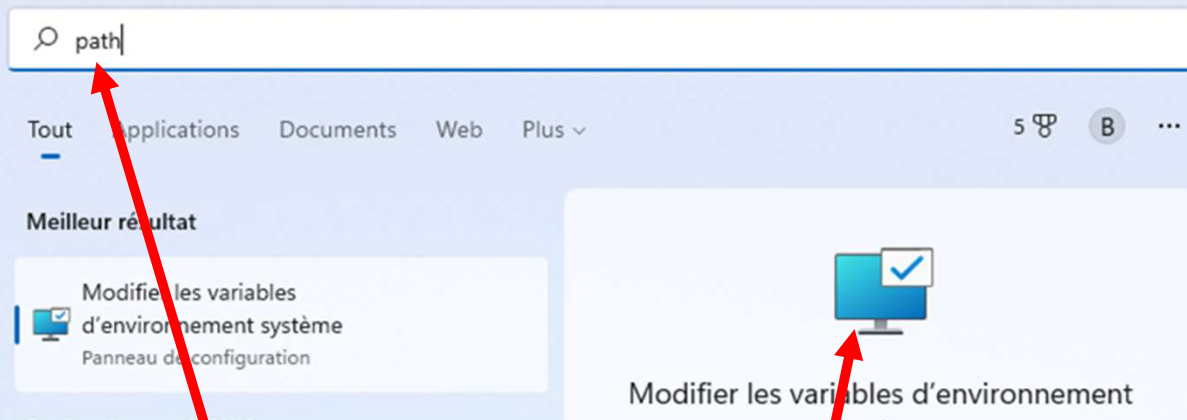
# Principes de compilation

- Il est recommandé d'installer NASM sous C:\Program Files\NASM
- Donner les droits "Contrôle total" sur ce répertoire pour tous les utilisateurs.
- Après l'installation, naviguer jusqu'à C:\Program Files\NASM  
Et copier le chemin



# Principes de compilation

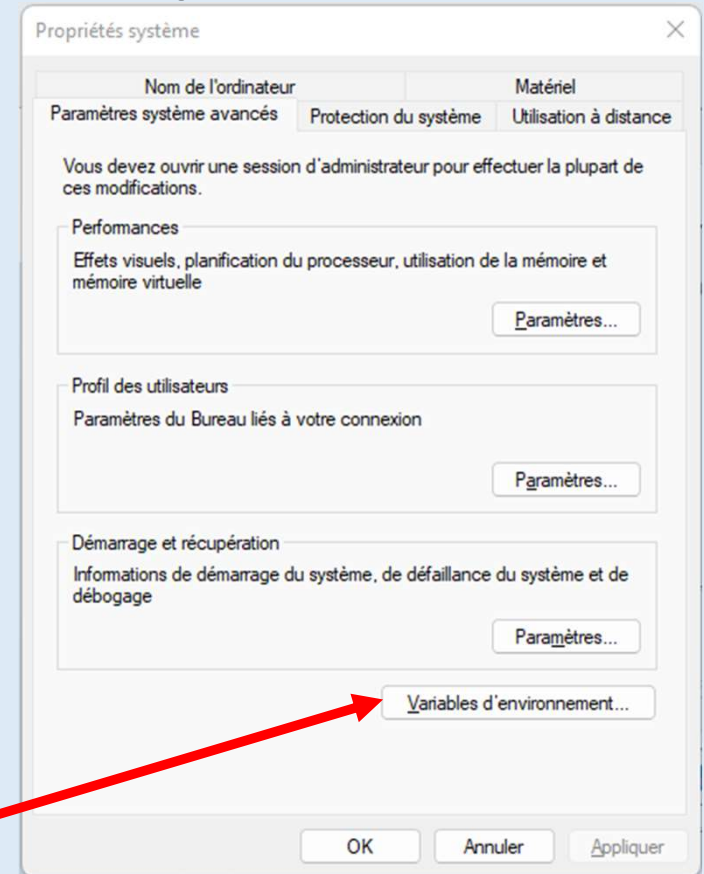
## Modifier le \$PATH windows pour rajouter le chemin copié



1 - Taper path dans la barre de recherche

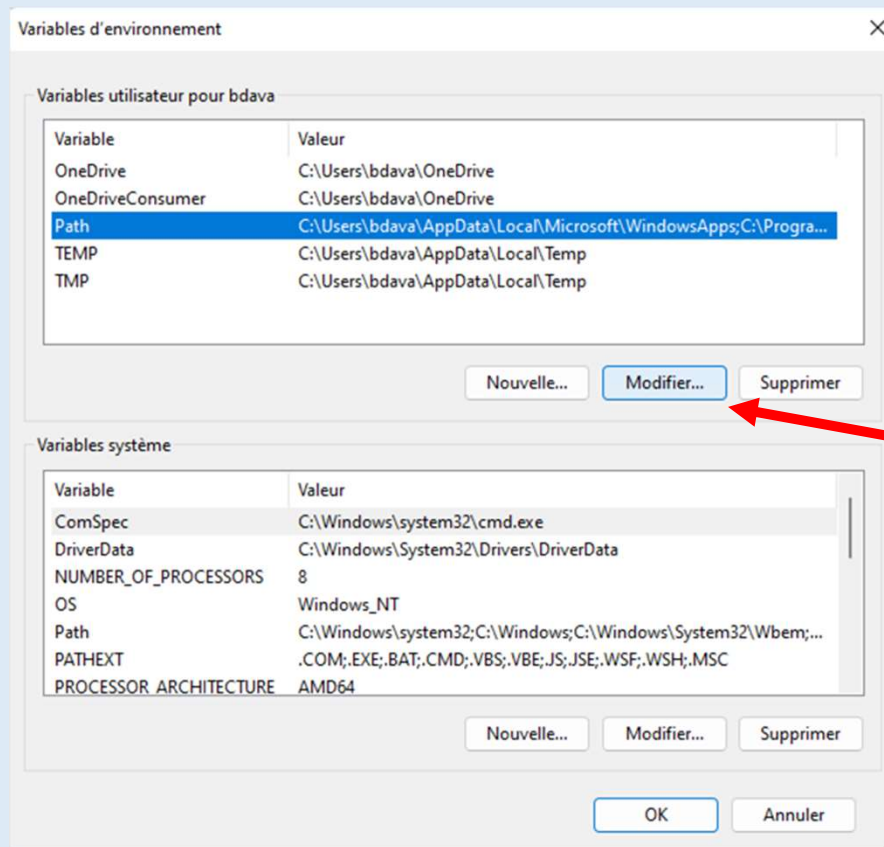
2 – Lancer le programme

3 – Choisir « Variables d'environnement »



# Principes de compilation

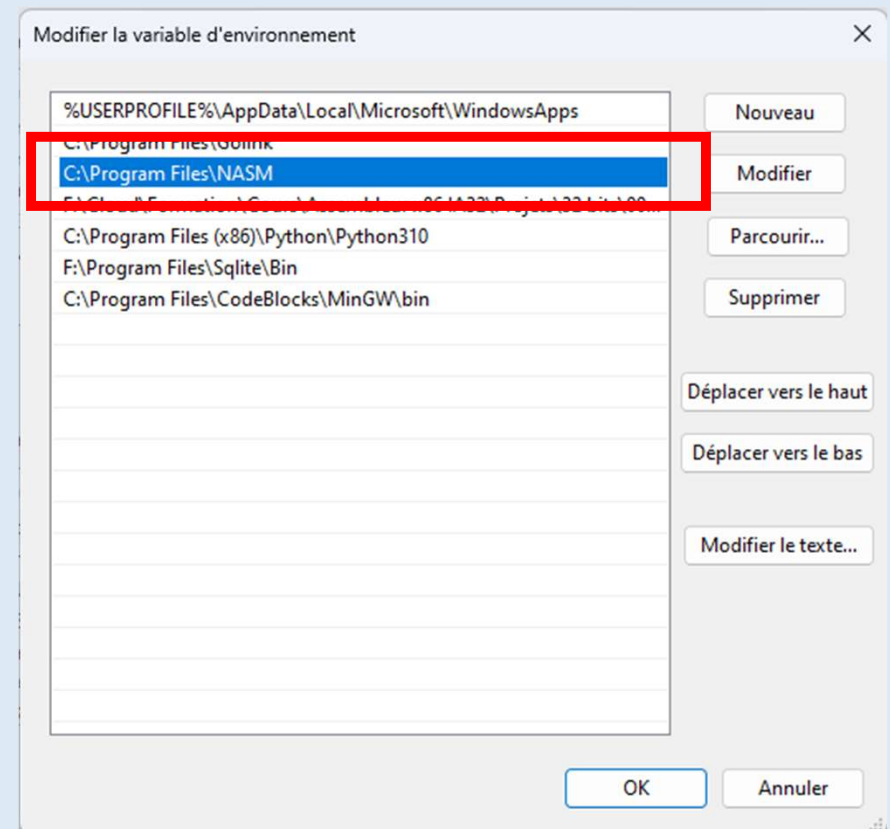
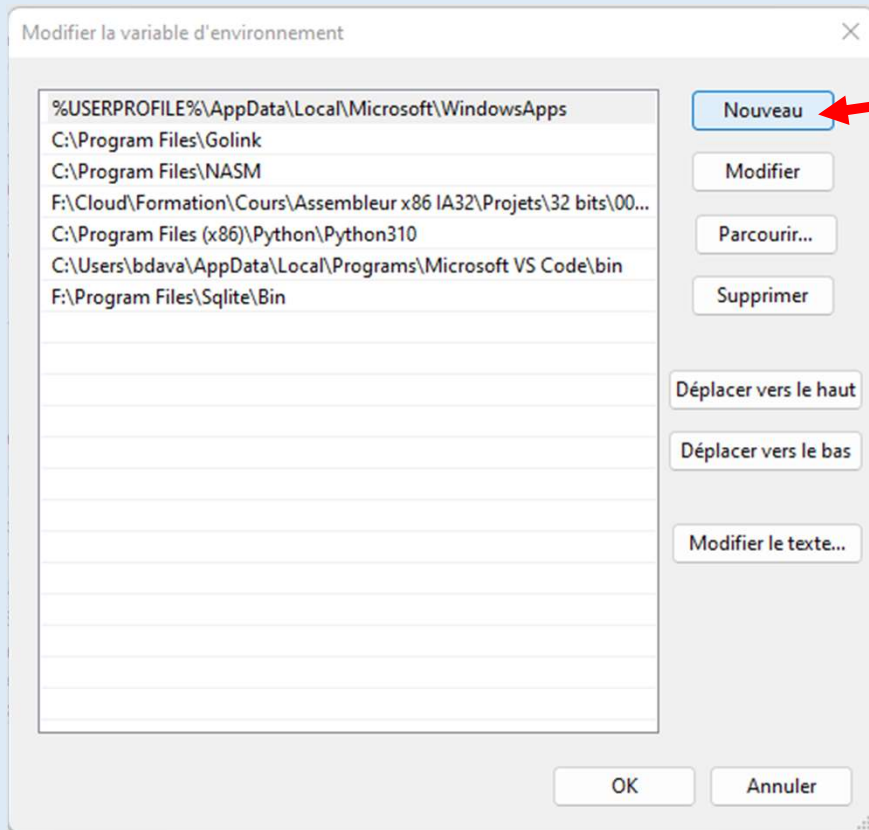
## Modifier le \$PATH windows pour rajouter le chemin copié



Choisir Path et  
« Modifier... »

# Principes de compilation

Puis « Nouveau » et  
copier le chemin



# Principes de compilation

## Installation des outils

Editeur de liens Golink : <http://www.godevtool.com/>

Aller dans la section **Linker**

Télécharger GoLink



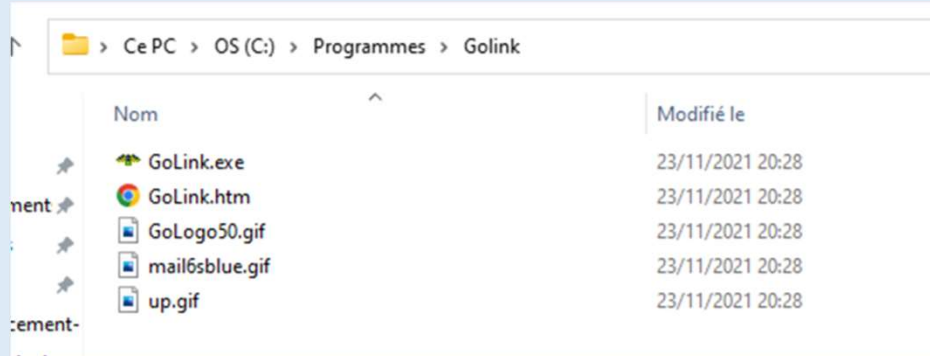
**Linker** - A free linker (GoLink.exe), which takes COFF object files and a PDB file and produces EXE or DLL executables able to run under Windows Win32 or Win64(x64) see [64-bit programming](#). This is a full featured but "reduced baggage" linker. It reduces the size of the executables to a minimum. You do not need Lib files to identify what functions reside in the DLLs. Instead GoLink looks inside the DLLs themselves. Used with GoAsm, it can also report on redundant data and code in your programs. It also allows use of Unicode filenames and labels (exports and imports).  
[View the GoLink manual](#)  
[Download GoLink version 1.0.4.2 \(with documentation 49K\)](#)  
Filename: GoLink.exe  
SHA-256: 19744d98a5c604e6bc2e146defa83f47396d14b7a188eae1df885768fcca91a2

# Principes de compilation

## Installation des outils

**Golink n'a pas de programme d'installation.**

**Il suffit de copier les fichiers téléchargés sous C:\Program Files\Golink**



**Il faut ensuite modifier le \$PATH  
Windows pour lui ajouter ce  
répertoire.**



# Principes de compilation

## Installation des outils

Syntaxe de la ligne de commande NASM (crée le fichier .obj)

***nasm.exe*** ***-f win32*** ***CodeSource.asm*** ***-l Listing.txt***

-f win32 = syntaxe  
d'entrée IA32

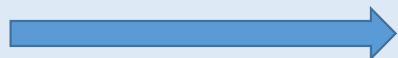
Nom du fichier  
source

Nom du fichier  
listing

Suggestion de fichier nommé ***compilation32.bat***

***nasm.exe*** ***-f win32*** ***%1.asm*** ***-l %1.txt***

Utilisation :



**C:\Users\Bda> compilation32 MonCode**

# Principes de compilation

## Installation des outils

Ligne de commande Golink

```
Golink.exe Fichier.obj /console Kernel32.dll User32.dll  
Gdi32.dll msvcrt.dll /entry:Main
```

Fichier à  
transformer en .exe

Crée un fichier à lancer depuis  
l'invite de comande (cmd)

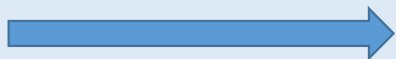
Point d'entrée du  
programme

Liste de bibliothèques à utiliser

Suggestion de fichier nommé link.bat

```
Golink.exe %1.obj /console Kernel32.dll User32.dll Gdi32.dll  
msvcrt.dll /entry:Main
```

Utilisation :

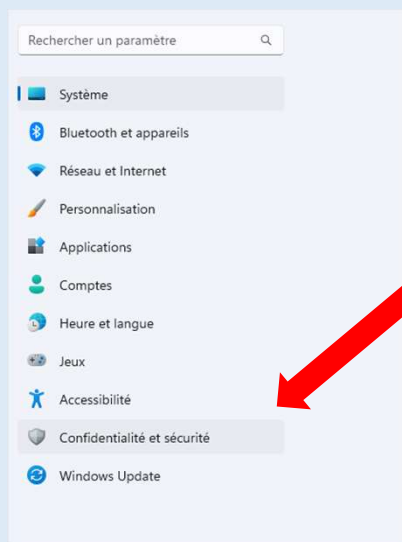


```
C:\Users\Bda> link MonCode
```

# Principes de compilation

Il peut arriver dans certains cas que Windows considère votre exécutable comme un virus.

Pour rajouter vos répertoires de travail dans les exclusions de l'antivirus, il faut aller dans le menu "Confidentialité et sécurité" des paramètres

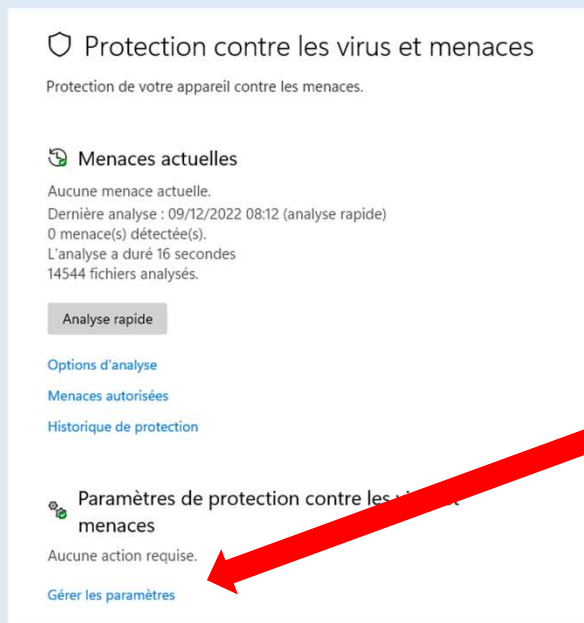


Puis choisir "Sécurité Windows"



# Principes de compilation

Sélectionner "Protection contre les virus et menaces"



Puis "Gérer les paramètres"



# Principes de compilation

Sélectionner "Ajouter ou supprimer des exclusions"

## Protection contre les falsifications

Empêche d'autres utilisateurs de falsifier des fonctionnalités de sécurité importantes.

☒ Activé

[En savoir plus](#)

## Dispositif d'accès contrôlé aux dossiers

Protégez vos fichiers, dossiers et zones de mémoire sur votre appareil contre toute modification non autorisée par des applications malveillantes.

[Gérer l'accès contrôlé aux dossiers](#)

## Exclusions

L'antivirus Microsoft Defender n'analysera pas les éléments qui ont été exclus. Les éléments exclus pourraient contenir des menaces pouvant nuire à votre appareil.

[Ajouter ou supprimer des exclusions](#)

## Exclusions

Ajoutez des éléments que vous souhaitez exclure des analyses de l'antivirus Microsoft Defender ou supprimez-en.

+ Ajouter une exclusion

F:\Cloud\Formation\Cours\Assembleur x86 IA32\Projets  
Dossier

Ajouter votre répertoire de travail  
avec "+ Ajouter une exclusion"

# Principes de compilation

Exemple de programme permettant de multiplier une valeur par 10.

valeur étant une adresse mémoire contenant la valeur à multiplier.

MOV destination, source : copie la valeur de la source dans la destination.

SHL destination, x : applique x décalages à gauche à la destination.

ADD destination, source : ajoute la valeur de la source à la destination.

```
mov eax, [valeur]    ; Copie la valeur à multiplier dans le registre eax
shl eax, 1           ; Décale 1 fois vers la gauche le registre eax
mov ebx, eax         ; Range la valeur obtenue dans le registre ebx
shl eax, 2           ; Décale 2 fois vers la gauche le registre eax
add eax, ebx         ; Ajoute la valeur d'ebx à eax
```

# **Bases de l'assembleur x86 IA 32**

# Bases de l'assembleur x86 IA32

## Données

Type de donnée	Taille	Suffixe
Octet (byte)	1 octet (char C)	b
Mot (word)	2 octets (short C)	w
Double mot (double)	4 octets (int C)	d
Quadruple mot (quad)	8 octets	q
Flottant simple précision	4 octets	d
Flottant double précision	8 octets	q
Flottant précision étendue	10 octets	t
Octoword	16 octets	o
	32 octets	y
	64 octets	z

Byte

High byte

Low byte

High word

Low word



# Rappels sur le système binaire

# Rappels sur le système binaire

## **Codages des entiers**

# Rappels sur le système binaire

Dans un système de numération de base B, il y a B symboles différents de représentation.

En base 10 (décimal), 10 symboles : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

En base 8 (octal), 8 symboles : 0, 1, 2, 3, 4, 5, 6, 7

En base 16 (hexadécimal), 16 symboles : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

En base 2 (binaire), 2 symboles : 0, 1

# Rappels sur le système binaire

Dans notre système habituel de numérotation décimale, c'est-à-dire en base 10, un nombre quelconque est la somme des unités, des dizaines, des centaines, des milliers, etc.

1235 = 5 unités + 3 dizaines + 2 centaines + 1 millier

En exprimant la base (10), cela peut s'écrire :

$$1235 = 5 \times 1 + 3 \times 10 + 2 \times 10 \times 10 + 1 \times 10 \times 10 \times 10$$

$$1235 = 5 \times 10^0 + 3 \times 10^1 + 2 \times 10^2 + 1 \times 10^3$$

C'est la somme pondérée de puissances de la base

## Rappels sur le système binaire

**Dans tout système de numération,  
n'importe quel nombre  $x$  s'exprime comme un  
polynôme de puissances croissantes de la base de numération.**

## Rappels sur le système binaire

*Dans tout système de numération, n'importe quel nombre  $x$  s'exprime comme un polynôme de puissances de la base  $b$  de numération.*

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

# Rappels sur le système binaire

Exemples : Représentation de la valeur 275

$$275 = 5 + 70 + 200$$

- En base 10 :  $5 \times 10^0 + 7 \times 10^1 + 2 \times 10^2$  275<sub>10</sub>

=1

=2

=16

- En base 2 :  $\underline{1} \times 2^0 + \underline{1} \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + \underline{1} \times 2^4 + 0 \times 2^5 + 0 \times 2^6 + 0 \times 2^7 + \underline{1} \times 2^8$  100010011<sub>2</sub>

- En base 8 :  $3 \times 8^0 + 2 \times 8^1 + 4 \times 8^2$  423<sub>8</sub>

- En base 16 :  $3 \times 16^0 + 1 \times 16^1 + 1 \times 16^2$  113<sub>16</sub>

On a aussi  $275 = 3 + 16 + 256$


$$\text{Or } 16 = 2 \times 8 = 2^4$$

$$256 = 2^8 = 4 \times 8^2 = 16^2$$

=256

# Rappels sur le système binaire

Méthode de codage en base  $b$  d'une valeur  $\underline{V}$ .

1. On commence par faire une liste de puissance croissante de la base :  $b^0, b^1, b^2, b^3$ , etc.
  2. On cherche la puissance  $\underline{P}$  immédiatement inférieure la valeur  $\underline{V}$ .
  3. On divise la valeur  $\underline{V}$  par  $\underline{P}$  donc  $V = P \times Q + R$
  4. On affecte le quotient  $\underline{Q}$  de cette division comme poids de la puissance  $P$
  5. On remplace  $\underline{V}$  par le reste  $\underline{R}$  de la division.
  6. On reprend à l'étape 2
- 



# Rappels sur le système binaire

Exemple : Codage de la valeur 431 en base 5

1. On commence par faire une liste de puissance croissante de la base

$$b^0 = 1$$

$$b^1 = 5$$

$$b^2 = 5 \times 5 = 25$$

$$b^3 = 5 \times 5 \times 5 = 125$$

$$b^4 = 5 \times 5 \times 5 \times 5 = 625$$

etc.

# Rappels sur le système binaire

Exemple : Codage de la valeur 431 en base 5

2. On cherche la puissance P immédiatement inférieure la valeur V.  
Ici  $P = 125 = 5 \times 5 \times 5 = 5^3$
3. On divise la valeur V par P.  
 $431 = 3 \times 125 + 56$ . Donc  $Q = \underline{3}$ ,  $R = \underline{56} = 2 \times 25 + 5 + 1$
4. On affecte le quotient Q de cette division comme poids de la puissance P.
5. On remplace V par le reste R de la division.
6. On reprend à l'étape 2


$$431 = \mathbf{3} \times \mathbf{5} \times \mathbf{5} \times \mathbf{5} + \mathbf{2} \times \mathbf{5} \times \mathbf{5} + \mathbf{1} \times \mathbf{5} + \mathbf{1}$$

$$\longrightarrow 431_{(10)} = 3211_{(5)}$$

# Rappels sur le système binaire

Exemple : Codage de la valeur 155 en binaire

256		0
128	$155 = 128 \times 1 + 27$	1
64		0
32		0
16	$27 = 16 \times 1 + 11$	1
8	$11 = 8 \times 1 + 3$	1
4		0
2	$3 = 2 \times 1 + 1$	1
1	$1 = 1 \times 1$	1

  $155_{(10)} = 10011011_{(2)}$

# Codage binaire

Représentez en base 2, 8 et 16 les valeurs suivantes : 314, 5, 127 et 41

Valeur	Base 2	Base 8	Base 16
314			
5			
127			
41			

# Codage binaire

Représentez en base 2, 8 et 16 les valeurs suivantes : 314, 5, 127 et 41

Valeur	Base 2	Base 8	Base 16
314	100111010	472	13A
5	101	5	5
127	1111111	177	7F
41	101001	51	29

# Codage binaire

Ça vaut le coup de mémoriser cela...

Décimal	Binaire	Hexa
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Décimal	Binaire	Hexa
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

# Addition binaire

# Addition binaire

L'addition binaire fonctionne de la même façon que l'addition classique.  
Elle est commutative et associative

$$0 + 0 = 0$$

$$0 + 1 = 1 + 0 = 1$$

$$1 + 1 = 0, \text{ avec une retenue de } 1 \text{ pour le poids fort suivant}$$

$$1 + 1 + 1 = 1, \text{ avec une retenue de } 1 \text{ pour le poids fort suivant}$$

	1	0	0	1	0	1	
+	0	0	1	1	0	1	
	<hr/>						<hr/>



# Addition binaire

L'addition binaire fonctionne de la même façon que l'addition classique.  
Elle est commutative et associative

$$0 + 0 = 0$$

$$0 + 1 = 1 + 0 = 1$$

$$1 + 1 = 0, \text{ avec une retenue de } 1 \text{ pour le poids fort suivant}$$

$$1 + 1 + 1 = 1, \text{ avec une retenue de } 1 \text{ pour le poids fort suivant}$$

			<i>1</i>	<i>1</i>		<i>1</i>	
	1	0	0	1	0	1	
+	0	0	1	1	0	1	
	<hr/>						
	1	1	0	0	1	0	

$$32+4+1=37$$

$$8+4+1=13$$


$$32+16+2=50$$

# Codages des entiers signés

# Rappels sur le système binaire

## Représentation des entiers signés

Première approche : On réserve le bit de poids fort pour le signe (1 = nombre négatif), et on conserve le même codage pour la valeur absolue

0	000		-0	100
1	001		-1	101
2	010		-2	110
3	011		-3	111

*Exemple sur 3 bits*

**Avantage : facile à mettre en œuvre**

**Inconvénients :**

- 0 a deux représentations
- La somme d'un nombre et de son opposé ne donne pas 0

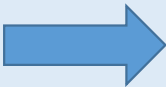
Ex :  $1 + (-1)$

$001 + 101 = 110$

# Rappels sur le système binaire

## Représentation des entiers signés

Deuxième approche : On inverse tous les bits d'une représentation pour obtenir la valeur négative

0	000		-0	111
1	001		-1	110
2	010		-2	101
3	011		-3	100

*Exemple sur 3 bits*

**Avantage : facile à mettre en œuvre**

**Inconvénients :**

- 0 a deux représentations
- La somme d'un nombre et de son opposé ne donne pas 0

Ex :  $2 + (-2)$

$010 + 101 = 111$

# Rappels sur le système binaire

## Représentation des entiers signés

Pour éviter les inconvénients des approches précédentes, on applique la méthode suivante, appelée « Complément à 2 »

On complémente (inverse) tous les bits de la représentation

Puis on ajoute 1 au résultat obtenu, sans tenir compte de la retenue finale.

Exemple, représentation de la valeur -5.

5 = 4 + 1, est codé 0101

On inverse tous les bits, son complément est 1010

En rajoutant 1, on obtient 1011

La représentation de -5 est 1011

On vérifie en  
faisant la somme :

$$\begin{array}{r} 0101 \\ + 1011 \\ \hline 0000 \end{array}$$

# Rappels sur le système binaire

## Représentation des entiers signés

Nombre positif	Représentation binaire	Complément	Ajout de 1	Nombre négatif
0	000	111	000	0
1	001	110	111	-1
2	010	101	110	-2
3	011	100	101	-3

Nombre positif	Représentation binaire	Complément	Ajout de 1	Nombre négatif
0	0000	1111	0000	0
1	0001	1110	1111	-1
2	0010	1101	1110	-2
3	0011	1100	1101	-3



**La représentation est dépendante du nombre de bits de codage !**

# Rappels sur le système binaire

## Représentation des entiers signés



Pour passer de la représentation d'une valeur positive à un valeur négative, on réalise d'abord une inversion des bits de la représentation positive, puis on rajoute 1 au résultat obtenu.



Pour passer de la représentation d'une valeur négative à un valeur positive, on fait les étapes en sens inverse.

On soustrait 1 à la représentation du nombre négatif, puis on inverse tous les bits du résultat obtenu

# Rappels sur le système binaire

## Représentation des entiers signés

Réalisez les étapes du calcul de  $73 + (-21)$  et donnez le résultat en binaire.  
Utilisez 8 bits de codage.




Diagram illustrating the conversion of 21 to -21 using 8-bit binary representation:

	128	64	32	16	8	4	2	1
73 =	0	1	0	0	1	0	0	1
21 =	0	0	0	1	0	1	0	1
Inversion	1	1	1	0	1	0	1	0
Puis on ajoute +1								
-21 =	1	1	1	0	1	0	1	1

	128	64	32	16	8	4	2	1
73 =	0	1	0	0	1	0	0	1
-21 =	1	1	1	0	1	0	1	1
<hr/>								
	0	0	1	1	0	1	0	0
	= 52							



# Rappels sur le système binaire

## Représentation des entiers signés

Réalisez les étapes du calcul de  $-92 + 37$  et donnez le résultat en binaire.  
Utilisez 8 bits de codage.

128 64 32 16 8 4 2 1

$$\begin{array}{rcl} 37 & = & 00100101 \\ 92 & = & 01011100 \\ \text{Inversion} & & 10100011 \\ \text{Puis on ajoute } +1 & & \\ -92 & = & 10100100 \end{array}$$

128 64 32 16 8 4 2 1

$$\begin{array}{rcl} 37 & = & 00100101 \\ -92 & = & 10100100 \\ \hline & & 11001001 = -55 \end{array}$$

On cherche la valeur absolue de ce nombre négatif

$$\text{On soustrait } 1 \quad 11001000 = 55$$

Puis on inverse 00110111

# Codages des décimaux

# Rappels sur le système binaire

## Représentation des nombres décimaux

Pour la partie décimale, les signes de codages représentent des puissances négatives décroissantes de la base.

Par exemple, en base 10, le nombre 5,236 est égal à :

$$5 \times 10^0 + 2 \times 10^{-1} + 3 \times 10^{-2} + 6 \times 10^{-3}$$

En base 2, les bits après la virgule  
représentent des puissances négatives de 2.

$$2^{-1} = 0,5$$

$$2^{-2} = 0,25$$

$$2^{-3} = 0,125$$

$$2^{-4} = 0,0625$$

Etc.

Ainsi 111,0011 est égal à 7,1875

# Rappels sur le système binaire

## Représentation des nombres décimaux : Virgule fixe

En base  $n$ , on a généralement besoin de  $n+1$  symboles pour coder les nombres décimaux : les  $n$  signes représentant les différents coefficients du nombre, plus le signe particulier indiquant la position du point décimal.

En binaire, ne disposant que de 2 signes (0 et 1), on doit fixer arbitrairement la place de la virgule. Il s'agit alors de nombres dits à virgule fixe.

Par exemple, sur 8 bits, on peut décider que la partie décimale sera codée sur 3 bits uniquement.

Ainsi  $5,375 = 00101011$

$$00101 = 5$$
$$011 = 0 \times 0,5 + 1 \times 0,25 + 1 \times 0,125 = 0,375$$

# Rappels sur le système binaire

## Représentation des nombres décimaux : Virgule flottante

Avec la notation en virgule flottante (en base  $b$ ), on code les nombres sous la forme :  
**Signe x Mantisse x  $b^{\text{Exposant}}$**

Ainsi, en virgule flottante, le nombre  $12,34_{(10)}$  pourrait s'écrire indifféremment :

- $0,1234 \times 10^2$
- $12,34 \times 10^0$
- $1234 \times 10^{-2}$

**Le nombre de chiffres décimaux à coder n'est plus une contrainte.**

# Rappels sur le système binaire

## Représentation des nombres décimaux : Virgule flottante

Le nombre de chiffres significatifs dépend de la précision de la mantisse.

Précision	Encodage	Signe	Exposant	Mantisse	Valeur d'un nombre	Précision	Chiffres significatifs
Simple précision	32 bits	1 bit	8 bits	23 bits	$(-1)^S \times M \times 2^{E-127}$	24 bits	Environ 7
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^S \times M \times 2^{E-1023}$	53 bits	Environ 16
Double précision étendue	80 bits	1 bit	15 bits	64 bits	$(-1)^S \times M \times 2^{E-16383}$	64 bits	Environ 19

# Rappels sur le système binaire

## Représentation des nombres décimaux : Virgule flottante

Dans n'importe quelle base, on peut écrire n'importe quel nombre sous la forme :

**Signe x Mantisse x  $b^{\text{Exposant}}$**

**Avec  $1 \leq \text{Mantisse} < \text{base}$**

**Par exemple en base 10,**

**1234 =**

**On a bien  $1 \leq 1,234 < 10$**

**0,0025 =**

**On a bien  $1 \leq 2,5 < 10$**

# Rappels sur le système binaire

Représentation des nombres décimaux : Virgule flottante

La mantisse est par convention comprise entre 1 et 2.

$$1 \leq \text{Mantisse} < 2$$

Elle est donc toujours de la forme 1.xxxx (x = 0 ou 1, puissance négatives de 2)

Le premier bit étant toujours à 1, on ne le code pas. Il est implicite.

Donc, pour le codage d'un flottant simple :

- Signe : 1 bit
- Exposant : 8 bits
- Mantisse : 23 bits



# Rappels sur le système binaire

## Représentation des nombres décimaux : Virgule flottante

Exemple d'un codage d'un flottant simple sur 32 bits :  $f = 3FA00000$

S = 0

E = 7F

M = 200000

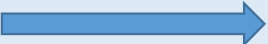
	3	F	A	0	0	0	0	0
0	011	1111	1	010	0000	0000	0000	0000
0	011	1111	1					
				010	0000	0000	0000	0000

# Rappels sur le système binaire

## Représentation des nombres décimaux : Virgule flottante

**f = 3FA00000**

- **S = 0** : Nombre positif
- **Exposant = 7F** =  $127_d$ . La mantisse sera multipliée par  $2^{E-127} = 2^0 = 1$
- **Mantisse = 200000**  
 $M = 20\ 00\ 00 = 010\ 0000\ 0000\ 0000\ 0000.. = 0 \times 2^{-1} + 1 \times 2^{-2} = 0,25$
- Rajout du 1 implicite  
 $M = 1 + 0,25 = 1,25$

  **$f = 1 \times 2^0 \times 1,25 = 1,25$**

# Rappels sur le système binaire

Représentation des nombres décimaux : Virgule flottante

À vous !

Trouvez le codage de -20,5

Pour rappel :

$$2^{-1} = 0,5$$

$$2^{-2} = 0,25$$

$$2^{-3} = 0,125$$

$$2^{-4} = 0,0625$$

$$2^{-5} = 0,03125$$

# Rappels sur le système binaire

## Représentation des nombres décimaux : Virgule flottante

À vous !

Trouvez le codage de -20,5

Pour rappel :

$$2^{-1} = 1/2 = 0,5$$

$$2^{-2} = 1/4 = 0,25$$

$$2^{-3} = 1/8 = 0,125$$

$$2^{-4} = 1/16 = 0,0625$$

$$2^{-5} = 1/32 = 0,03125$$

Nombre négatif : **S = 1**

$$20,5 = 10,25 \times 2$$

$$20,5 = 5,125 \times 4$$

$$20,5 = 2,5625 \times 8$$

$$20,5 = 1,28125 \times 16 = 1,28125 \times 2^4$$

Valeur d'exposant à coder : 131 (131-127 = 4)

E = **10000011**

$$1,28125 = 1 + 0,25 + 0,03125$$

$$1,28125 = 1 + 2^{-2} + 2^{-5}$$

Codage de la mantisse **010010000000000000000000**

**1****10000011****10**100100000000000000000000  
C 1 A 4 0 0 0 0

# Stockage interne des données

# Rappels sur le système binaire

## Stockage interne des données

Il existe plusieurs façons de stocker un mot de 32 bits (4 octets) en mémoire, dont :

- Big Endian : l'adresse du mot est l'adresse de son octet de poids fort.
- Little Endian : l'adresse du mot est l'adresse de son octet de poids faible.

Exemple : Soit la valeur 02AA5F07h, son stockage dans les 2 modes est :

	Octet 1	Octet 2	Octet 3	Octet 4
Big Endian	02	AA	5F	07
Little Endian	07	5F	AA	02

*Little Endian : Intel x86.*

*Big Endian : Motorola 68000, SPARC (Sun Microsystems), System/370 (IBM).*

# **Bases de l'assembleur x86 IA 32**

## **La syntaxe**

# Bases de l'assembleur x86 IA32

## Syntaxe Intel vs. AT&T

**Pour coder en assembleur x86, différentes syntaxes existent. Elles dépendent du compilateur utilisé.**

**Les deux principales syntaxe sont :**

- La syntaxe Intel (que nous utiliserons)**
- La syntaxe AT&T**

**Ces deux syntaxes produisent le même code machine.**

**Nous étudierons la version IA32 : Intel Assembler 32 bits.**



# Bases de l'assembleur x86 IA32

## Syntaxe Intel vs. AT&T

Les principales différences entre les syntaxes Intel et AT&T sont résumées ici :

	Intel	AT&T
Affectation	Opérateur Destination Source	Opérateur Source Destination
Commentaires	;	//
Instructions	Pas de suffixe. Ex : add	Suffixe avec la taille des opérandes. Ex : addq
Registres	eax, ebx	%eax, %ebx
Constantes	0x100	\$0x100
Contenu d'une adresse	[eax]	(%eax)
Adressage dans un tableau	[base + reg + reg * taille + déplacement]	Déplacement(reg, reg, taille)

# Bases de l'assembleur x86 IA32

Toutes les instructions, les directives et les macros utilisent le format suivant :

**[label:] mnémonique [opérandes] [; commentaire]**

Les champs entre crochets sont optionnels

- **Label:** utilisé pour représenter un identifiant ou une constante.
- **Mnémonique:** Identifie l'instruction à exécuter.  
Si la ligne ne contient qu'un label ou un commentaire, alors le mnémonique n'est pas requis.
- **Opérandes:** Spécifie les données à manipuler.
- **Commentaire:** Texte ignoré par le compilateur.

# Bases de l'assembleur x86 IA32

## Exemple

```
jmp label1      ; Ceci est un commentaire
add eax, ebx    ; Cette instruction saute au label "label1".
                ; eax <- eax+ebx. La valeur contenue dans le
                ; registre eax est augmentée de la valeur
                ; contenue dans le registre ebx
label1:         ; Définition du label "label1"
sub edx, 32     ; edx <- edx - 32. La valeur contenue dans le
                ; registre edx et diminuée de 32
```

# Bases de l'assembleur x86 IA32

## Partitionnement du programme

Un programme assembleur est en général constitué de plusieurs parties (appelées segments).

Trois segments sont considérés comme standard en assembleur X86 :

- Le segment **.data** contient les définitions des variables **initialisées** à une ou plusieurs valeurs spécifiées (instructions db, dw, dd...).
- Le segment **.bss** (Block Started by Symbol) contient les définitions des variables **non-initialisées**, c'est à dire uniquement allouées en mémoire. (instructions resb, resw, resd...).
- Le segment **.text** contient le code exécuté par le programme.

# **Bases de l'assembleur x86 IA 32**

## **Quelques instructions de base**

# Bases de l'assembleur x86 IA32

## Instructions de base

***MOV Destination, Source***

**MOV copie la valeur de Source dans Destination.**

**Destination peut être un registre ou adresse mémoire.**

**Source peut être un registre, une adresse mémoire ou une constante.**

**Destination et Source ne peuvent PAS être en même temps des adresses mémoire.**

```
mov eax, ebx      ; copie la valeur d'ebx dans eax
mov byte [var], 5  ; copie la valeur 5 dans l'octet situé
                  ; à l'adresse var
```

# Bases de l'assembleur x86 IA32

## Instructions de base

**JMP Label**

**JMP** déclenche saut inconditionnel à l'adresse Label

```
DebutBoucle :      ; Définition de l'adresse DebutBoucle
...
...                ; code à effectuer dans la boucle
...
JMP DebutBoucle   ; Saut au début de la boucle
```

# Bases de l'assembleur x86 IA32

## Instructions de base

### ADD Destination, Source

**ADD** ajoute à Destination la valeur de Source

**Destination** peut être un registre ou adresse mémoire.

**Source** peut être un registre, une adresse mémoire ou une constante.

**Destination et Source** ne peuvent **PAS** être en même temps des adresses mémoire.

```
add eax, ebx          ; Rajoute à eax la valeur d'ebx
add ecx, dword [var]  ; Ajoute à ECX la valeur contenue dans Le mot
                      ; de 32 bits à l'adresse var
add byte [var], 5      ; ajoute 5 à l'octet situé à l'adresse var
```



# Bases de l'assembleur x86 IA32

## Instructions de base

### SUB Destination, Source

Similaire à ADD, mais réalise une soustraction et place le résultat dans Destination

Destination peut être un registre ou adresse mémoire.

Source peut être un registre, une adresse mémoire ou une constante.

Destination et Source ne peuvent PAS être en même temps des adresses mémoire.

```
sub eax, ebx      ; Retire à eax la valeur d'ebx
sub ah, al        ; AH est diminué de la valeur de AL
sub byte [var], 5  ; Retire 5 à l'octet situé à l'adresse var
```

# Bases de l'assembleur x86 IA32

## Instructions de base

INC Op1  
DEC Op1

INC incrémente l'opérande.

DEC décrémente l'opérande.

L'opérande peut être un registre ou un emplacement mémoire.

```
inc ecx           ; Incrémente ecx. Equivaut à ecx++  
dec dword [var]   ; Décrémente la variable de 32 bits située  
                  ; à l'adresse var
```

# Bases de l'assembleur x86 IA32

## Instructions de base

AND Destination, Source  
OR Destination, Source  
XOR Destination, Source

Réalise l'opération logique entre Destination et Source et place le résultat dans Destination.

Destination peut être un registre ou une adresse mémoire.

Source peut être un registre, une adresse mémoire ou une constante.

Destination et Source ne peuvent PAS être en même temps des adresses mémoire.

and eax, 0Fh

????????????????????

????????????????????

# **Bases de l'assembleur x86 IA 32**

## **Les registres**

# Bases de l'assembleur x86 IA32

## Registres généraux

Il y a 8 registres généraux

Leur taille est de 32 bits.

Ils peuvent être utilisés de façon implicite pour certaines opérations

Registres de données	EAX	<u>A</u> ccumulateur
	EBX	<u>B</u> ase (adresse mémoire)
	ECX	<u>C</u> ompteur
	EDX	<u>D</u> onnée, entrées/sorties
Registres de pointeurs et d'index	ESI	<u>I</u> ndex <u>S</u> ource
	EDI	<u>I</u> ndex <u>D</u> estination
	EBP	<u>B</u> ase de la <u>P</u> ile
	ESP	<u>S</u> ommet de la <u>P</u> ile

# Bases de l'assembleur x86 IA32

## Registres de Données

Les registres en « X » peuvent être utilisés

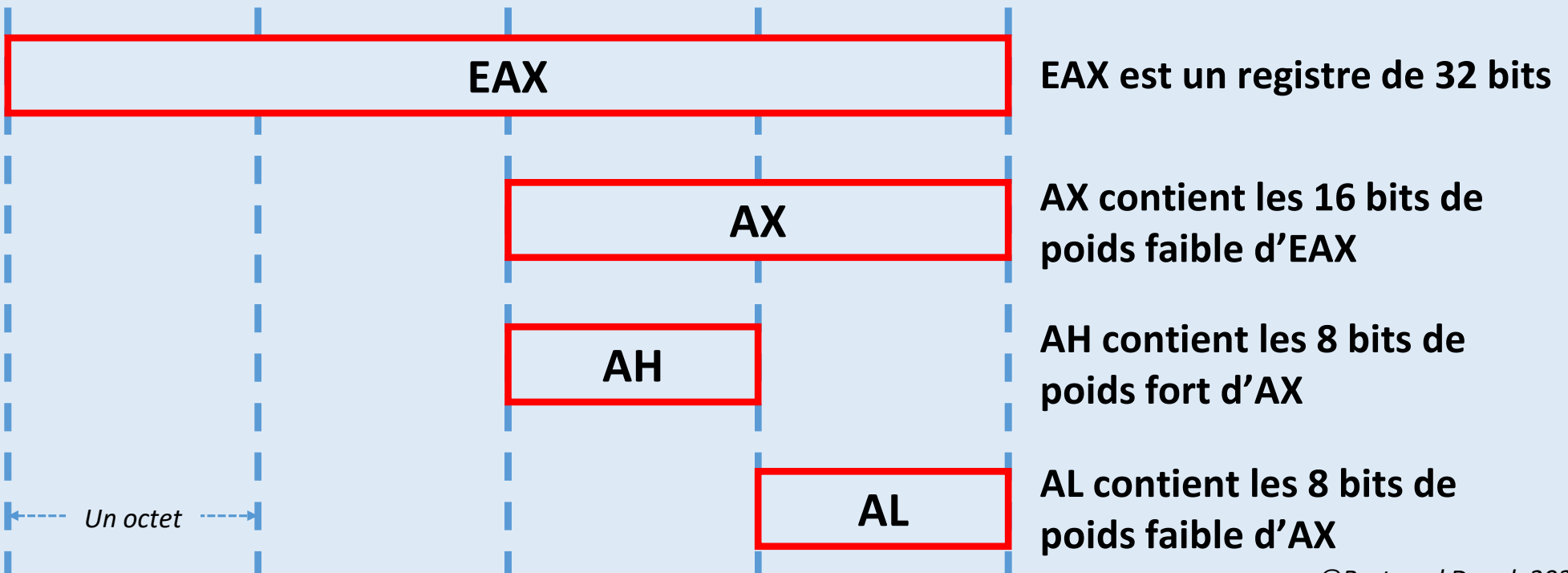
- en totalité sur 32 bits (EAX, EBX, ECX, EDX)
- Sur leurs 16 bits de poids faible (AX, BX, CX, DX)
- Sur leurs 8 bits de poids faible (AL, AH, BL, BH, CL, CH, DL, DH)

Registres 32 bits (31...0)	Bits 31...16	Bits 15...0	Bits 15...8	Bits 7...0
EAX		AX	AH	AL
EBX		BX	BH	BL
ECX		CX	CH	CL
EDX		DX	DH	DL

# Bases de l'assembleur x86 IA32

## Registres de Données

Exemple de découpage d'un registre général sur 8, 16 ou 32 bits



# Bases de l'assembleur x86 IA32

## Registres de Données

Exemple d'utilisation d'un registre sur 8, 16 ou 32 bits

```
mov  eax,12345678h    ; eax contient 12345678h
mov  ax,9988h          ; eax contient 12349988h
mov  ah,00h            ; eax contient 12340088h
```





# Bases de l'assembleur x86 IA32

## Registres de Pointeurs et d'Index

Il y a quatre registres de pointeur et d'index (ESI, EDI, ESP et EBP).

Ces registres peuvent être utilisés sur leurs 16 bits de poids faible (SI, DI, SP and EP).

**ESI** et **EDI** sont utilisés comme des registres de données, mais jouent un rôle particulier lors de l'utilisation de instructions de traitement de chaînes de caractères.

**ESP** est le pointeur de pile et **EBP** est le pointeur de base.

Registres 32 bits (31...0)	Bits 31...16	Bits 15...0	
ESI		SI	Index source
EDI		DI	Index destination
ESP		SP	Pointeur de pile
EBP		EP	Pointeur de base

# Bases de l'assembleur x86 IA32

## Registres de Contrôle

Les deux registres de contrôle les plus importants sont :

- Le registre de pointeur d'instructions : **EIP**
- Le registre d'indicateurs : **EFlags**

**EIP (Instruction Pointer Register)** pointe sur la prochaine instruction à effectuer. Il ne peut pas être modifié directement, mais seulement à l'aide d'instructions de saut (JMP ou Jx)

**EFlags** contient une série d'indicateurs qui sont mis en place lors de calculs arithmétiques et d'opérations de comparaison ou de test.

# Bases de l'assembleur x86 IA32

## Registres de Contrôle

Les indicateurs les plus utilisés de **EFlags** sont :

Zero Flag (ZF)	Positionné (mis à 1) lorsque le résultat de la dernière instruction arithmétique est égal à 0.
Carry Flag (CF)	Positionné lorsque la dernière opération sur deux <b>entiers non signés</b> est trop grande (ex : byte > 127) ou trop petite (<0).
Overflow Flag (OF)	Positionné lors d'un <b>dépassement</b> sur une opération <b>d'entiers signés</b> .
Sign Flag (SF)	Indique le <b>signe</b> du résultat d'une opération arithmétique.
Parity Flag (PF)	Indique la <b>parité</b> du nombre de 1 du résultat de 8 bits produit par une opération.

# Bases de l'assembleur x86 IA32

Zero Flag (ZF)	Utilisé pour tester les compteurs décrémenté dans les boucles	JZ : Jump if Zero JNZ : Jump if Not Zero
Carry Flag (CF)	Indique les dépassements lors d'opérations <b>non signées</b> ou de décalages. A noter : INC et DEC n'affectent pas l'indicateur CF	JC : Jump if Carry JNC : Jump if Not Carry STC : Set Carry Flag CLC : Clear Carry Flag CMC : Complement Carry Flag
Overflow Flag (OF)	Indique les dépassements sur les entiers <b>signés</b>	JO : Jump if Overflow JNO : Jump if Not Overflow
Sign Flag (SF)	Indique le signe d'une opération arithmétique	JS : Jump if Signed (négatif) JNS : Jump if Not Signed (non négatif)
Parity Flag (PF)	Indique si l'octet de poids faible du résultat contient un nombre pair de bits à 1. PF = 1 si le nombre de bits à 1 est pair.	JP : Jump if Parity (pair) JNP : Jump if Not Parity (impair)

# Bases de l'assembleur x86 IA32



**Créez en quelques instructions une boucle de type**

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

**{**

**.....**

**}**

# Bases de l'assembleur x86 IA32

**Initialiser un registre (par exemple EAX, ECX, etc.) avec la valeur 0**

**Indiquer le début de la boucle**

**Sortir de la boucle si le registre est  $\geq$  à 10**

**Exécuter le code de la boucle**

**Incrémenter le registre**

**Retourner en début de boucle**

# Bases de l'assembleur x86 IA32

## Les instructions de comparaison

Il existe deux opérations de comparaison TEST et CMP

Elles utilisent deux opérandes, Op1 et Op2

- TEST Op1, Op2  
réalise un ET logique, bit par bit entre Op1 et Op2 et positionne les flags SF (Sign Flag), ZF (Zero Flag), et PF (Parity Flag).
- CMP Op1, Op2  
réalise une soustraction entre Op1 et Op2 et met à jour EFlags en fonction du résultat.

# Bases de l'assembleur x86 IA32

## Les instructions de comparaison

### TEST Op1, Op2

réalise un ET logique, bit par bit entre Op1 et Op2 et positionne les flags

- SF (Sign Flag)
- ZF (Zero Flag)
- PF (Parity Flag)

Zero Flag (ZF)	Positionné (mis à 1) lorsque le résultat de la dernière instruction arithmétique est égal à 0.
Sign Flag (SF)	Indique le <b>signe</b> du résultat d'une opération arithmétique.
Parity Flag (PF)	Indique la <b>parité</b> du nombre de 1 du résultat de 8 bits produit par une opération.



Quelles sont les valeurs des flags après l'opération :  
TEST AX, BX

AX	BX	SF	ZF	PF
0x00	0xFF			
0x88	0x77			
0xAA	0xFF			



# Bases de l'assembleur x86 IA32

## Les instructions de comparaison

### CMP Op1, Op2

réalise une soustraction entre Op1 et Op2 et positionne les flags

- ZF (Zero Flag)
- SF (Sign Flag)
- PF (Parity Flag)
- OF (Overflow Flag)
- CF (Carry Flag)



Quelles sont les valeurs des flags après l'opération :  
CMP AL, BL

Zero Flag (ZF)	Positionné (mis à 1) lorsque le résultat de la dernière instruction arithmétique est égal à 0.
Sign Flag (SF)	Indique le <b>signe</b> du résultat d'une opération arithmétique.
Parity Flag (PF)	Indique la <b>parité</b> du nombre de 1 du résultat de 8 bits produit par une opération.
Overflow (OF)	Dépassement sur opérations <b>signées</b>
Carry Flag (CF)	Dépassement sur opérations <b>non signées</b>

AL	BL	ZF	SF	PF	OF	CF
0x00	0xFF	0	0	0	0	1
0x88	0x77	0	0	1	1	0
0xAA	0xFF	0	1	0	0	1

# Bases de l'assembleur x86 IA32

## Les instructions de saut conditionnel

Ces instructions commencent par la lettre J (Jump), suivie d'une ou plusieurs lettres spécifiant les conditions.

Ces conditions peuvent être cumulées avec des « ou ».

### *Exemples*

A	Above (NS)
E	Equal
B	Below (NS)
C	Carry
G	Greater (S)

L	Lower (S)
N	Non
O	Overflow
P	Parity
PE	Parity Even
PO	Parity Odd

JA	Jump if <u>Above</u>	Saut si plus grand que
JNE	Jump if <u>Not Equal</u>	Saut si différent
JBE	<u>Jump if Below or Equal</u>	Saut si <=

# Bases de l'assembleur x86 IA32

## Les instructions de saut conditionnel

Instruction	Description	Flags
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if sign	SF = 1
JNS	Jump if not sign	SF = 0
JE	Jump if equal	ZF = 1
JZ	Jump if zero	
JNE	Jump if not equal	ZF = 0
JNZ	Jump if not zero	

# Bases de l'assembleur x86 IA32

## Les instructions de saut conditionnel

Instruction	Description	Flags
JB JNAE JC	Jump if below Jump if not above or equal Jump if carry	CF = 1
JNB JAE JNC	Jump if not below Jump if above or equal Jump if not carry	CF = 0
JBE JNA	Jump if below or equal Jump if not above	CF = 1 ou ZF = 1
JA JNBE	Jump if above Jump if not below or equal	CF = 0 and ZF = 0

# Bases de l'assembleur x86 IA32

## Les instructions de saut conditionnel

Instruction	Description	Flags
JL JNGE	Jump if less Jump if not greater or equal	SF <> OF
JGE JNL	Jump if greater or equal Jump if not less	SF = OF
JLE JNG	Jump if less or equal Jump if not greater	ZF = 1 ou SF <> OF
JG JNLE	Jump if greater Jump if not less or equal	ZF = 0 ou SF = OF
JP JPE	Jump if parity Jump if parity even	PF = 1
JNP JPO	Jump if not parity Jump if parity odd	PF = 0

# Bases de l'assembleur x86 IA32

## Les instructions de saut conditionnel

Instruction	Description	Flags
JCXZ	Jump si CX = 0	CX = 0
JECXZ	Jump si ECX = 0	ECX = 0

Le registre CX étant souvent utilisé comme compteur de boucle, il existe une instruction spéciale pour tester directement sa valeur, sans passer au préalable par une instruction de comparaison (TEST ou CMP)

# Bases de l'assembleur x86 IA32

## Les instructions de saut conditionnel

Dans le programme ci-dessous, pour quelles valeurs d'ebx la 3<sup>ème</sup> instruction provoque-t-elle un saut à l'adresse « label1 » ?



```
mov eax, -1    ; eax := -1  
cmp eax, ebx   ; Comparaison eax et ebx  
jb label1      ; Jump if Below (Comparaison non signée) à l'adresse « location »
```

# Bases de l'assembleur x86 IA32



**Créez en quelques instructions une boucle de type**

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

**{**

**.....**

**}**



# Bases de l'assembleur x86 IA32

```
xor ecx, ecx          ; Initialise ecx à 0
DebutBoucle :         ; définition du début de la boucle
...
...
inc ecx               ; ecx++
cmp ecx, 10           ; comparaison d'ecx avec la valeur 10
jb DebutBoucle        ; On boucle si ecx < 10 (Jump if below)
FinBoucle :           ; Label de sortie de boucle
```

# Bases de l'assembleur x86 IA32

Ecrire le code assembleur équivalent à :

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

    return 0;
}
```

# Bases de l'assembleur x86 IA32

Ouvrir un éditeur de texte et mettre au début du fichier :

```
extern PrintNum
extern ExitProcess

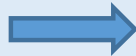
global main

section .data
format:    db 'Cpt = %d',10,0

section .text
main:
```

# Bases de l'assembleur x86 IA32

*Ecrire le code  
assembleur  
équivalent à :*



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

Pour imprimer la valeur du compteur,  
mettre dans le corps de la boucle :

```
push <la valeur du compteur>  
push format  
call PrintNum
```

Après la boucle mettre ces deux lignes :

```
push 0  
call ExitProcess
```

# Bases de l'assembleur x86 IA32

```
extern PrintNum
extern ExitProcess

global main

section .data
format: db 'Cpt = %d',10,0

section .text
main:
    xor ecx, ecx                ; Mise à 0 d'ecx, qui sert de compteur
Start :
    push ecx                    ; Empilement des parametres
    push format                  ; de PrintNum (format et valeur)
    call PrintNum                ; Appel de PrintNum

    inc ecx                      ; Increment du compteur
    cmp ecx, 10                  ; Comparaison avec la valeur max
    jb Start                     ; Si < 10, saut vers l'étiquette Start

    push 0                       ; ExitProcess(0)
    call ExitProcess
```

# Les fonctions d'entrée/sortie

# Bases de l'assembleur x86 IA32

## Les fonctions d'entrée sortie

**On utilise deux fonctions de bas niveau, disponibles dans l'API Windows pour écrire sur la console et récupérer des entrées clavier :**

- **WriteConsoleA**
- **ReadConsoleA**

# Bases de l'assembleur x86 IA32

## WriteConsoleA

WriteConsoleA est la version ANSI de WriteConsole.

Son interface d'appel est la suivante :

```
BOOL WINAPI
WriteConsole(
    _In_      HANDLE hConsoleOutput,          ; Handle de la console de sortie
    _In_      const VOID *lpBuffer,          ; Buffer contenant la chaîne à
                                                ; afficher
    _In_      DWORD nNumberOfCharsToWrite,    ; Nombre de caractères à écrire
    _Out_opt_ LPDWORD lpNumberOfCharsWritten, ; Nombre de caractères effectivement
                                                ; écrits
    _Reserved_ LPVOID lpReserved );          ; Réservé
```



# Bases de l'assembleur x86 IA32

## ReadConsoleA

**ReadConsoleA est la version ANSI de WriteConsole.**

**Son interface d'appel est la suivante :**

```
BOOL WINAPI
ReadConsole(
    _In_      HANDLE hConsoleInput,      ; Handle de la console de sortie
    _Out_     LPVOID lpBuffer,           ; Buffer recevant la chaîne à lire
    _In_      DWORD nNumberOfCharsToRead, ; Nombre max de caractères à lire
    _Out_     LPDWORD lpNumberOfCharsRead, ; Nombre de caractères effectivement lus
    _In_opt_  LPVOID pInputControl );    ; Réservé
```

# Bases de l'assembleur x86 IA32

## Les fonctions d'entrée sortie

**WriteConsoleA et ReadConsoleA** doivent connaître l'identifiant (handle) de l'entrée et de la sortie standard.

On utilise la fonction **GetStdHandle**

```
HANDLE WINAPI  
GetStdHandle(  
_In_          DWORD nStdHandle); ; Constante identifiant le canal d'entrée/sortie :  
                                     ; Entrée standard : STD_INPUT_HANDLE ((DWORD)-10)  
                                     ; Sortie standard : STD_OUTPUT_HANDLE ((DWORD)-11)  
                                     ; Sortie d'erreur : STD_ERROR_HANDLE ((DWORD)-12)
```

# Utilisation de WriteConsoleA

# Bases de l'assembleur x86 IA32

```
extern GetStdHandle, WriteConsoleA
Extern ExitProcess

STD_OUTPUT_HANDLE equ -11

global main

section .data
chaine db 'Chaine a afficher',10,0
longueur equ $-chaine-1

section .bss
buffer resb 1

section .text
main:
; Appel de GetStdHandle avec le paramètre -11
; Le handle de sortie est dans eax

push    dword STD_OUTPUT_HANDLE
call    GetStdHandle
```

```
; Appel de WriteConsole avec les paramètres :
; _In_ HANDLE hConsoleOutput,
; _In_ const VOID *lpBuffer,
; _In_ DWORD nNumberOfCharsToWrite,
; _Out_opt_ LPDWORD lpNumberOfCharsWritten,
; _Reserved_ LPVOID lpReserved

push    dword 0 ; On passe les paramètres en sens
                ; inverse
push    buffer ; Pointeur sur le nb de caractères
                ; écrits

push    dword longueur ; longueur de la chaîne
push    chaine ; Adresse de la chaîne
push    eax ; Handle de sortie
call    WriteConsoleA ; Appel de l'écriture sur le
                    ; canal de sortie

push    0 ; Sortie du programme (return 0;)
call    ExitProcess
```

# **Affichage des caractères imprimables ASCII**

# Bases de l'assembleur x86 IA32

## Affichage des caractères imprimables ASCII

**Ecrire un programme affichant les caractères ASCII imprimables, séparés par des virgules, en commençant par le caractère ' ' et en finissant par '~'.**

```
 , ! , " , # , $ , % , & , ' , ( , ) , * , + , , - , . , / , 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , : , ; , < , = , > , ? ,  
 @ , A , B , C , D , E , F , G , H , I , J , K , L , M , N , O , P , Q , R , S , T , U , V , W , X , Y , Z , [ , \ , ] , ^ , _ ,  
 ` , a , b , c , d , e , f , g , h , i , j , k , l , m , n , o , p , q , r , s , t , u , v , w , x , y , z , { , | , } , ~ ,
```

# Bases de l'assembleur x86 IA32

```
extern GetStdHandle, WriteConsoleA, ExitProcess
global main

chardebut      equ ' '
charfin        equ '~'

section .data
message db " ,"

section .bss
consolehandle  resd 1
buffer         resb 1
```

```
section .text
main:
    push dword -11          ;Récupération handle
    call GetStdHandle
    mov [consolehandle], eax ; Mémorisation
    mov ebx, chardebut      ; Initialisation ebx
boucle :
    mov [message], bl
    push 0
    push buffer
    push 2
    push message
    push dword [consolehandle]
    call WriteConsoleA
    inc ebx                 ; Incrémentation ebx
    cmp ebx, charfin ; Test boucle
    jna boucle

    push 0
    call ExitProcess
```

# **Utilisation des fonctions PrintNum, PrintStr et ReadInt**



# Bases de l'assembleur x86 IA32

## Utilisation de PrintNum et ReadInt

**PrintNum, PrintStr et ReadInt sont fournies dans la bibliothèque libESGI.obj.**

**PrintNum accepte 2 arguments, une chaîne de caractères S et un nombre D et fait l'équivalent de printf(S,D)**

```
valeur    dd 7
chaine1   db 'Resultat %d = ',10,0

push dword [valeur]
push chaine1
call PrintNum
```

# Bases de l'assembleur x86 IA32

## Utilisation de PrintNum et ReadInt

**ReadInt permet la saisie d'un entier positif au clavier et renvoie sa valeur dans eax.**

**edx contient le code de retour :**

- **0 = OK,**
- **-1 = des caractères non numériques ont été saisis**
- **-2 = Overflow**

# Bases de l'assembleur x86 IA32

## Utilisation de PrintNum et ReadInt

**PrintStr** accepte comme argument une chaîne de caractères **S** et fait l'équivalent de **printf(S)**

```
chaine1  db 'Affichage du resultat : ',0  
  
push chaine1  
call PrintStr
```

# Bases de l'assembleur x86 IA32

## Utilisation de PrintNum et ReadInt

**Ecrire un programme utilisant ReadInt et PrintNum permettant de saisir un nombre et de l'afficher.**

# Bases de l'assembleur x86 IA32

## Suite de Syracuse

**On appelle suite de Syracuse une suite d'entiers naturels définie de la manière suivante :**

- **On part d'un nombre entier strictement positif ;**
- **s'il est pair, on le divise par 2 ;**
- **s'il est impair, on le multiplie par 3 et l'on ajoute 1.**

**En répétant l'opération, on obtient une suite d'entiers strictement positifs dont chacun ne dépend que de son prédécesseur.**

# Bases de l'assembleur x86 IA32

## Suite de Syracuse

```
extern PrintNum, PrintStr, ReadInt, ExitProcess
global main

Section .data
prompt: db 'Valeur de depart (entier positif): ',0
format: db '%d,',0

section .text
main:
    push prompt
    call PrintStr
    call ReadInt
    cmp edx, 0
    jne main

    mov ebx, eax
    jmp Print
```

```
Debut :
    test ebx, 1
    jnz Impair
    shr ebx, 1
    jmp Print

Impair :
    mov eax, ebx
    shl ebx, 1
    add ebx, eax
    inc ebx

Print :
    push ebx
    push format
    call PrintNum
    cmp ebx, 1
    jne Debut

    push 0
    call ExitProcess
```

# Bases de l'assembleur x86 IA32

## Suite de Syracuse

Valeur de depart (entier positif): 555555

555555,1666666,833333,2500000,1250000,625000,312500,156250,78125,234376,117188,58594,29297,8789  
2,43946,21973,65920,32960,16480,8240,4120,2060,1030,515,1546,773,2320,1160,580,290,145,436,218,  
109,328,164,82,41,124,62,31,94,47,142,71,214,107,322,161,484,242,121,364,182,91,274,137,412,206  
,103,310,155,466,233,700,350,175,526,263,790,395,1186,593,1780,890,445,1336,668,334,167,502,251  
,754,377,1132,566,283,850,425,1276,638,319,958,479,1438,719,2158,1079,3238,1619,4858,2429,7288,  
3644,1822,911,2734,1367,4102,2051,6154,3077,9232,4616,2308,1154,577,1732,866,433,1300,650,325,9  
76,488,244,122,61,184,92,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1,

# **Bases de l'assembleur x86 IA 32**

## **La pile**



# Bases de l'assembleur x86 IA32

## La pile

La pile est un espace mémoire destiné à stocker temporairement des informations, qui seront récupérées plus tard.

### **PUSH**

- Met une valeur sur le haut de la pile

### **POP**

- Récupère la valeur située en haut de la pile

La pile est une structure LIFO : **Last In, First Out**

# Bases de l'assembleur x86 IA32

## La pile

Programme :

```
PUSH eax    ; empile la valeur du
             ; registre eax

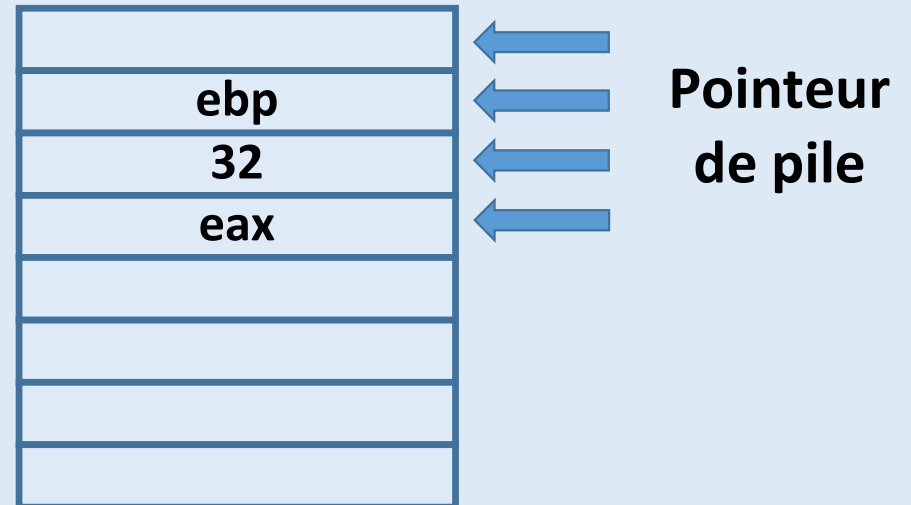
PUSH 32     ; empile la valeur 32

PUSH ebp    ; empile la valeur du
             ; registre ebp
             .....

POP ebx     ; charge ebx avec la
            ; valeur du haut de la pile

POP ebx     ; charge ebx avec la
            ; valeur du haut de la pile

POP eax     ; charge eax avec la
            ; valeur du haut de la pile
```



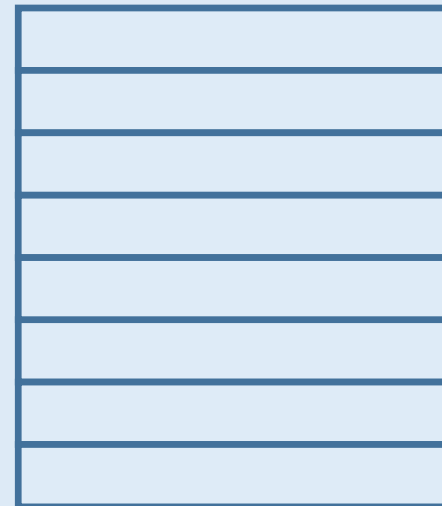
# Bases de l'assembleur x86 IA32

## La pile



En x86, le bas de la pile correspond à l'adresse la plus haute.  
Le haut de la pile correspond à l'adresse la plus basse

Empile  
PUSH  
Décrémente SP



00000000h



Dépile  
POP  
Incrémente SP

FFFFFFFFh

Empiler un mot de 32 bits revient à décrémenter le pointeur de pile de 4  
Dépiler revient à incrémenter le pointeur de pile de 4

# Bases de l'assembleur x86 IA32

## La pile

Pour empiler et dépiler l'ensemble des registres en une seule instruction, on utilise

- **pushad**
- **popad**

Pour empiler et dépiler le **registre des flags (EFlags)**, on utilise

- **pushfd**
- **popfd**

# Bases de l'assembleur x86 IA32

## La pile

En utilisant la pile, proposez une suite d'instructions permettant d'échanger les valeurs des registres `ebx` et `ecx`

```
push ebx ; On empile la valeur d'EBX
push ecx ; On empile la valeur d'ecx
pop ebx  ; On met la dernière valeur empilée dans ebx
pop ecx  ; On met l'avant-dernière valeur empilée dans ecx
```

# **Bases de l'assembleur x86 IA 32**

## **Partitionnement du programme**

# Partitionnement du programme

Trois segments standard en assembleur X86 : **.data**, **.bss** et **.text**.

- Le segment **.data** contient les définitions des variables initialisées à une ou plusieurs valeurs spécifiées (instructions db, dw, dd...)

**D = Définition**

- Le segment **.bss** contient les définitions des variables non-initialisées, c'est à dire uniquement allouées en mémoire. (instructions resb, resw, resd...).

**RES = Réservation**

- Le segment **.text** contient le code exécuté par le programme.

# Partitionnement du programme

```
global main                ; déclaration de Main en global
                           ; => export du point d'entrée pour créer
                           ; le programme

segment .data              ; création de variables initialisées
Message: db 'Bonjour ',0  ; Initialisation de la chaîne 'Bonjour \0'
                           ; à l'adresse Message

Segment .bss              ; création de variables non-initialisées
VarA      resb 1           ; VarA : 1 octet réservé
Facteur   resd 1           ; Facteur : 1 double (32 bits) réservé

segment .text              ; création des procédures/fonctions ainsi
                           ; que du point d'entrée

main:                     ; Point d'entrée du programme
```



# **Bases de l'assembleur x86 IA 32**

## **Déclaration de variables**

# Déclaration de variables

## Initialisation de variables

Utilisation de l'instruction d (définition), avec le suffixe indiquant la taille.

**IMPORTANT**

Les variables initialisées se déclarent dans la section .data

# Déclaration de variables

## Initialisation de variables

Utilisation de l'instruction **d** (définition), avec le suffixe indiquant la taille.

```
var1 db 123      ; b = Byte. 8 bits, initialisés à la valeur  
                  ; 123 = 7Bh  
var2 dw 456      ; w = Word. 16 bits, initialisés à la valeur  
                  ; 456 = 01C8h  
var3 dd 789      ; d = Double. 32 bits, initialisés à  
                  ; 789 = 00000315h  
var4 dq 123      ; q = Quad. 64 bits, initialisés à la valeur  
                  ; 123 = 0000000000000007Bh  
var5 dt 1.23     ; t = exTended word. Var4, 80 bits initialisés  
                  ; à la valeur 1.23
```

# Déclaration de variables

## Initialisation de tableau

Pour initialiser un tableau, on liste les valeurs, séparées par des virgules.

Pour répéter une initialisation, on utilise « times »

```
var1 db 123,12,3      ; Tableau d'octets (byte)
                        ; [var1] = 123, [var1+1] = 12, [var1+2] = 3
var2 db 'Hi',0xa      ; Tableau d'octets (byte)
                        ; [var2] = 'H', [var2+1] = 'i',
                        ; [var2+2] = '\n' = 0xa = LF
var3 times 4 db '*'    ; Tableau d'octets = chaîne de caractères
                        ; var3 = "****"
```

# Déclaration de variables

## Initialisation de tableau

**IMPORTANT**

Attention à la déclaration du type de données.  
Celui-ci définit la taille de la réservation

```
var3 dd 255,6554 ; Tableau de double  
                ; (32 bits)  
                ; [var3] = 255  
                ; [var3+4] = 6554
```

Label	Valeur	Adresse physique
var3	FF	\$008AFE30
	00	\$008AFE31
	00	\$008AFE32
	00	\$008AFE33
var3 + 4	9A	\$008AFE34
	19	\$008AFE35
	00	\$008AFE36
	00	\$008AFE37

# Déclaration de variables

## Déclaration de variables non initialisées

Utilisation de res (réservation) + suffixe indiquant la taille.

**IMPORTANT**

Les variables non initialisées se déclarent dans la section `.bss`

```
var1 resb 1024 ; Réserve 1024 Bytes. Var 1 est l'adresse  
                ; de début d'un tableau de 1024 octets  
var2 resw 1     ; Réserve 1 Word  
var3 resd 12    ; Réserve 12 Double (12 * 32 bits)  
var4 resq 24    ; Réserve 24 Quad (24 * 64 bits)  
var5 rest 2     ; Réserve 2 exTended word (2 * 80 bits)
```

# Pseudo instructions

## Instruction EQU (Equal)

Permet de définir des constantes. Il n'y a pas de réservation mémoire, c'est simplement l'équivalence entre une valeur et un identifiant.

Semblable à la directive `#define` en C

```
Trois equ 3 ; Définit la chaîne 3
```

Lorsque le pré-processeur rencontrera la chaîne de caractères `Trois`, il la remplacera par la valeur `3`.

# Pseudo instructions

## Instruction \$

**Renvoie la valeur courante du compteur d'adresse. Utilisé pour calculer la longueur d'une chaîne de caractères**

```
Message db 'Hello World !', 0 ; Définit la chaîne de caractères  
                                ; Message
```

```
MsgLen equ $-Message          ; Met dans MsgLen la longueur de la  
                                ; chaîne Message
```



# **Appel de procédures**

# Bases de l'assembleur x86 IA32

## Appel de procédures : Principe

### Appelant

...

Place les arguments de la procédure

Appelle la procédure

Récupère les valeurs de retour

### Appelé

Initialise des variables locales

...

Renseigne les valeurs de retour

Libère les variables locales

Rend la main à l'appelant

# Bases de l'assembleur x86 IA32

## Appel de procédures : Principe

L'appelant doit savoir où placer les arguments d'appel

L'appelé doit savoir comment récupérer les arguments d'appel

L'appelé doit connaître l'adresse de retour

L'appelant doit savoir où trouver les valeurs de retour

L'appelant et l'appelé étant sur le même CPU, ils partagent les mêmes registres

- L'appelant peut vouloir sauvegarder des registres susceptibles d'être utilisés par l'appelé
- L'appelé peut vouloir restaurer des registres pour l'appelant

# Bases de l'assembleur x86 IA32

## Appel de procédures : Mécanisme d'appel

Côté appelant, on utilise l'instruction **call** pour appeler une procédure

Côté appelé, on utilise l'instruction **ret** en fin de procédure

### Appelant

```
...  
call MyProcedure  
push eax  
...
```

L'instruction **call** va:

1. Empiler l'adresse de l'instruction qui suit le call
2. Faire un jump à l'adresse MyProcedure

```
804854e e8 3d 06 00 00 call 8048b90  
8048553 50 push eax
```

### MyProcedure

```
...  
...  
ret
```

L'instruction **ret** va:


1. Dépiler l'adresse de retour
2. Faire un Jump à l'adresse cette adresse

Au moment de cet appel, c'est la valeur **0x8048553** qui est empilée.  
C'est l'adresse de retour

# Bases de l'assembleur x86 IA32

## Appel de procédures : Mécanisme d'appel

804854e	<b>e8</b>	3d 06 00 00	call 8048b90
8048553	50		push eax



Au moment où l'instruction **e8** est lue, le compteur de programme s'incrémente de 5 octets et passe à  $0x804854e + 5 =$  **0x8048553**

Cette valeur est empilée, c'est l'adresse de retour.

Comme l'instruction est un **call**, elle est suivie d'une adresse **relative** sur 32 bits. La valeur ici est **3d060000** en **little endian**, soit **63d**.

Le compteur de programme est augmenté de cette valeur :  
 $0x8048553 + 63d = 0x8048b90$ , qui est l'adresse de **début de la procédure appelée**.

# Bases de l'assembleur x86 IA32

## Appel de procédures : Valeurs de retour

Par convention :

- La valeur de retour est placé dans le registre **eax**
- L'appelant doit s'assurer de **sauvegarder eax** avant de réaliser un appel
- L'appelé place la valeur de retour dans **eax**. Si cette valeur a une taille supérieure à 4 octets, alors il faut utiliser un pointeur
- L'appelant trouve la valeur de retour dans **eax**

# Bases de l'assembleur x86 IA32

## Appel de procédures : Valeurs de retour

Par convention :

- **L'appelant** est en charge de sauvegarder les registres : eax, edx, ecx  
L'appelant sauvegarde ces registres avant l'appel et les restaure ensuite.
- **L'appelé** est en charge de garder intacts les registres : ebp, ebx, esi, edi  
L'appelé sauvegarde ces registres au début de la procédure et les restaure à la fin.

# Bases de l'assembleur x86 IA32

## Appel de procédures : Utilisation de la pile

Lorsque la procédure utilise des paramètres, l'appelant va placer ceux-ci dans la pile juste avant l'appel.

Exemple :

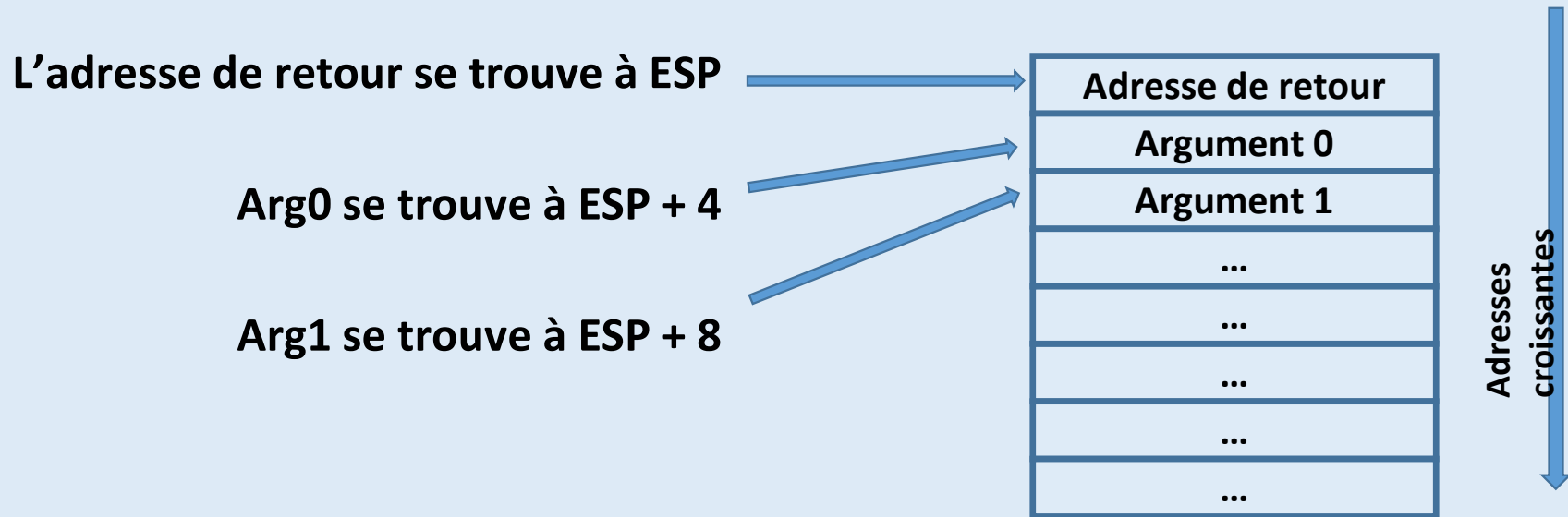
```
push ebx          ;Empilement de Arg[1]  
push eax          ;Empilement de Arg[0]  
call MaProcedure  ;Appel de La procédure
```



# Bases de l'assembleur x86 IA32

## Appel de procédures : Utilisation de la pile

À l'arrivée dans la procédure, appelée avec 2 paramètres, la pile est dans l'état suivant :



# Bases de l'assembleur x86 IA32

## Appel de procédures : Utilisation de la pile

### Bonnes pratiques

Au début d'une procédure :

On sauvegarde la valeur de EBP, qui représente l'état de référence de la pile pour l'appelant : **push ebp**

On met à jour ebp avec la valeur courante d'esp. Ainsi, la nouvelle référence devient l'adresse du haut de la pile : **mov ebp, esp**

On accède à la variable n à l'adresse : **ebp + 8 + 4 \* n**

```
mov eax, [ebp + 8] ; Met Arg[0] dans eax
```

```
mov ebx, [ebp + 12] ; Met Arg[1] dans ebx
```

# Bases de l'assembleur x86 IA32

## Appel de procédures : Utilisation de la pile

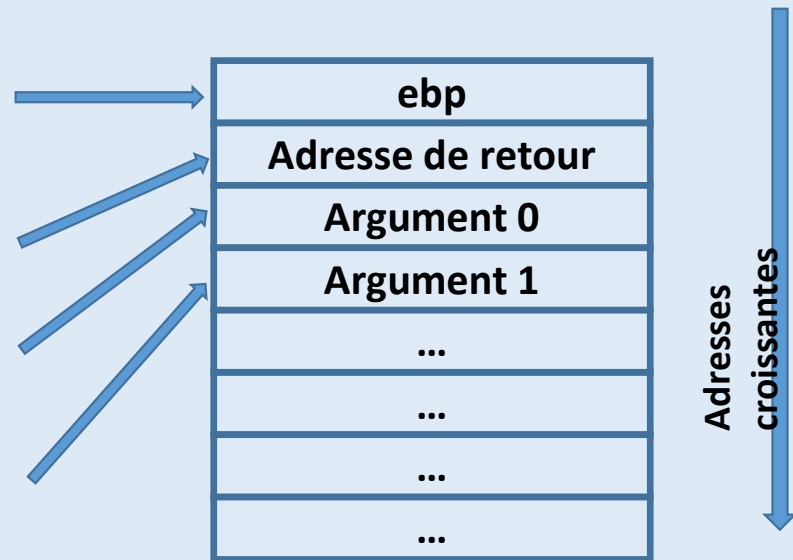
À l'arrivée dans la procédure, appelée avec 2 paramètres, la pile est dans l'état suivant :

Ancienne valeur sauvegardée d'ebp.  
esp et ebp pointent sur cet emplacement

L'adresse de retour se trouve à  $ebp + 4$

Arg0 se trouve à  $ebp + 8$

Arg1 se trouve à  $ebp + 12$



# Bases de l'assembleur x86 IA32

## Prologue et Epilogue

myFunc:

; Prologue

push ebp ; Sauvegarde de l'ancienne base

mov ebp, esp ; La nouvelle base pointe sur le sommet de la pile

...

... ; Corps de la procédure

...

; Epilogue

mov esp, ebp ; Restauration de l'ancienne valeur d'esp

pop ebp ; restauration de l'ancienne base

ret ; Fin de la procédure