

# Reverse Engineering

Jean-Philippe Luyten - [jp@r-3-t.org](mailto:jp@r-3-t.org)

# Contents

---

- [www.r-3-t.org/msis/](http://www.r-3-t.org/msis/)

# Plans

---

- Introduction
- Systèmes numériques
  - ▲ Décimal
  - ▲ Hexadécimal
  - ▲ Binaire
  - ▲ Conventions
- Organisation de l' ordinateur
  - ▲ Le CPU
- La mémoire
  - ▲ Définitions
  - ▲ La pile
  - ▲ Gestion de la mémoire
  - ▲ Les registres
  - ▲ Les modes d' adressage

# Plans

---

- Les bases de l' ASM
  - ▲ Opérations de bases sur les registres
  - ▲ Opérations de base sur la pile
  - ▲ Opérations arithmétiques et logiques
  - ▲ Opérations de contrôles
  - ▲ Les boucles
  - ▲ Les procédures

# Introduction

---

- Langage de très bas niveau
  - C'est le seul langage que comprend réellement le processeur de l'ordinateur.
  - Il y'a une correspondance entre mnemonic et opcode
    - `8B 44 24 28`
    - `mov eax, [esp+28h]`
  - Ces 2 lignes représente la même chose...
  - Tous les langages compilés ont pour résultat un fichier exécutable (format PE sous Windows, ELF sous Linux par exemple) qui sera une suite d'instructions en langage machine. La compilation transforme un langage de haut niveau en un langage que peut comprendre le processeur.

# Système Numérique

---

- Représentation des nombres entiers:
  - Complément à 2
  - Décimal
  - Hexadécimal
  - Binaire
- Représentation des nombres réels

# Représentation des entiers relatifs

---

- Le complément à deux est une représentation binaire des entiers relatifs qui permet d'effectuer les opérations arithmétiques usuelles naturellement.
- $00000010 = +2$  en décimal
- $11111110 = -2$  en décimal
- Les nombres positifs sont représentés comme attendu
- Les nombres négatifs sont obtenus de la manière suivante :
  - $\sim n + 1$

# Système Numérique

Décimal

- C'est un système en base 10.
  - Les dix nombres de 0 à 9 sont utilisés pour faire tous les autres.
- Exemples:

	Centaines	Dizaines	Unités
Digits	1	2	5
Signification	$1 \cdot 10^2$	$2 \cdot 10^1$	$5 \cdot 10^0$
Valeur	100	20	5

# Système Numérique

## Binaire

- C'est un système en base 2.
  - Les nombres 0 et 1 servent à représenter tous les autres.
  - Il semble plus difficile que le langage décimal, mais ce n'est pas le cas.
  - Les nombres s'écrivent comme la décomposition de puissance de 2.
- Exemple:
  - $10110 = 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$   
= **22** en base 10
  - On va écrire:  $10110_b = 22_d$ .

# Système Numérique

## Binaire

- ▲ Pour la conversion décimal vers binaire on procède de la façon suivante:

$22 / 2$  reste 0

$11 / 2$  reste 1

$5 / 2$  reste 1

$2 / 2$  reste 0

$1 / 2$  reste 1

Donc  $22d = 10110b$

- ▲ Il suffit de diviser le nombre original par 2. Si le résultat est entier, alors le reste est 0, sinon il est de 1. On procède ainsi de suite jusqu' à 0 en arrondissant le nombre obtenu par la division par deux à la valeur inférieure.

# Système Numérique

## Hexadecimal

- C'est un système en base 16.
  - ▶ Les nombres de 0 à 9 et les lettres de A à F servent à représenter tous les autres.
- Table de conversion:

Hexadécimal	Décimal	Binaire	Hexadécimal	Décimal	Binaire
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

# Système Numérique

## Hexadecimal

- Exemples:

- $\blacktriangleright 2A4F = F \cdot 16^0 + 4 \cdot 16^1 + A \cdot 16^2 + 2 \cdot 16^3$   
 $= 15 \cdot 1 + 4 \cdot 16 + 10 \cdot 256 + 2 \cdot 4096$   
 $= \textcolor{red}{57005}$  en base 10

- $\blacktriangleright$  On va écrire:  $2A4Fh = 57005d$ .

- $\blacktriangleright$  Pour la conversion décimal vers hexadécimal on procède de la façon suivante:

$$\begin{aligned}1324 / 16 &= 82.75 ; 82 * 16 = 1312; 1324 - 1312 = 12 = \textcolor{red}{C} \\82 / 16 &= 5.125 ; 5 * 16 = 80; 82 - 80 = 2 = \textcolor{red}{2} \\5 / 16 &= 0.3125; 0 * 16 = 0; 5 - 0 = 5 = \textcolor{red}{5}\end{aligned}$$

Donc  $1324d = \textcolor{red}{52Ch}$

# Système numérique

- Les nombres à virgule flottante sont des approximations de nombres réels.
- Ils possèdent un signe, une mantisse et un exposant

	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	7
Double précision	64 bits	1 bit	11 bits	52 bits	$(-1)^S \times M \times 2^{(E-1023)}$	53 bits	16

source : wikipedia

# Architecture des ordinateurs

---

- CPU / Mémoire
- Registre
- La pile

# La mémoire

---

## **bit, nibble, octet, word, dword, qword**

- Un **bit** est la plus petite unité de données d'un ordinateur. Sa valeur est de 0 ou 1.
- Un **nibble** est un ensemble de 4 bits.
- Un **octet** est un ensemble de 2 nibbles (donc de 8 bits). C'est la grandeur la plus utilisées en informatique.
- Un **word** est ensemble de 2 octets (16 bits).
- Un double word (**dword**) est ensemble de 2 words (32 bits).
- Un quad word (**qword**) est ensemble de 4 words (64 bits).

# Endianness

---

- Il existe 2 façons de représenter les nombres entiers (codés sur plusieurs octets). L'ordre dans lequel ces octets sont organisés est appelé endianness
- Les processeurs x86 sont ***Little-Endian***
  - mov [BaseAddress], 0xDEADBEEF

Memory Location	Big Endian	Little Endian
Base Address + 0	DE	EF
Base Address + 1	AD	BE
Base Address + 2	BE	AD
Base Address + 3	EF	DE

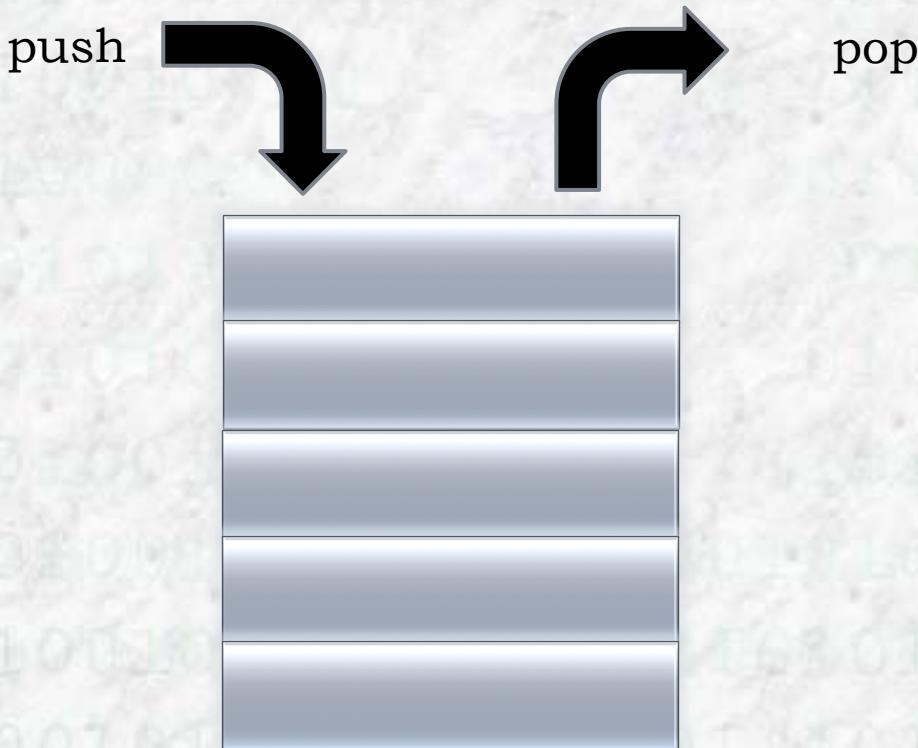
# La Pile (Stack)

---

- La pile est une structure de données. C'est une zone de mémoire dans laquelle on peut stocker temporairement la valeur des registres (ou autres). Elle est, par exemple, très utile lors du passage de paramètres à une fonction.
- Afin de mieux comprendre son fonctionnement, nous pouvons l'assimiler à une pile de livre. Le dernier élément posé sur le haut de la pile sera aussi le premier à être retiré. Last In First Out (LIFO).
- Les instructions push et pop permettent d'empiler et de dépiler la pile.
- Nous verrons par la suite que certains registres servent à gérer la pile.

# La pile (stack)

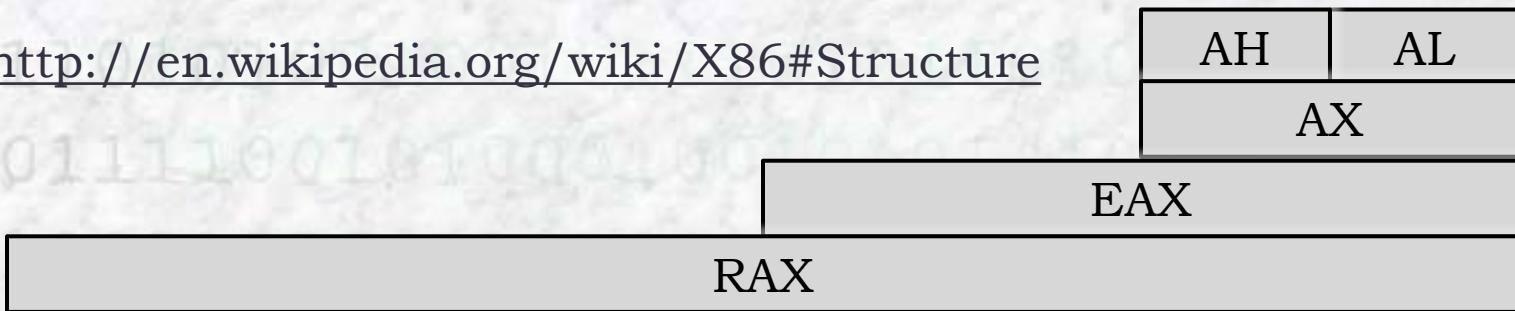
- LIFO (Last In First Out)



- push / pop / call / ret / enter / leave

# Les Registres

- Ce sont des petites zones de stockage (max 128 bits) pouvant contenir des données.
- Le nom des registres n'a pas évolué depuis l'époque 16 bits, seule la présence d'un E devant le nom a été ajouté pour signifier une capacité de 32 bits ou un R pour signifier une capacité de 64 bits.
- La plupart des registres de 16 bits sont composés d'une partie haute et d'une partie basse.
- <http://en.wikipedia.org/wiki/X86#Structure>



# Les Registres

---

- AX/EAX/RAX: Accumulator
- BX/EBX/RBX: Base index (for use with arrays)
- CX/ECX/RCX: Counter
- DX/EDX/RDX: Data/general
- SI/ESI/RSI: *Source index* for string operations.
- DI/EDI/RDI: *Destination index* for string operations.
- SP/ESP/RSP: Stack pointer for top address of the stack.
- BP/EBP/RBP: Stack base pointer for holding the address of the current stack frame.
- IP/EIP/RIP: Instruction pointer. Holds the program counter, the current instruction address.

# Les Registres

---

- En 64 bits, 8 nouveaux registres : r8-r15
- r8b-r15b : 8-bits de poids faible
- r8w-r15w : 16-bits de poids faible
- r8d-r15d : 32-bits de poids faible

# Les Registres

---

- Les registres de segments:
  - Ils définissent le segment de mémoire qui est utilisée. Ils ne sont pas directement utile lorsque l'on fait de la programmation 32 bits car nous pouvons adresser 4Go de mémoire (flat model). Par contre en programmation DOS, nous ne pouvons adresser que des blocs de 64ko, donc, pour définir une adresse mémoire nous devons spécifier un segment et un offset. Ce sont des registres de 16 bits.
- **CS** : Segment de code.
- **DS** : Segment de données.
- **ES** : Extra segment.
- **FS, GS**.
- **SS** : Segment de pile, registre utilisé par le processeur pour stocker l'adresse de retour d'une procédure.

A4	MOVSB	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R E)SI to (R E)DI.
A5	MOVSW	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R E)SI to (R E)DI.
A5	MOVSD	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R E)SI to (R E)DI.

A4	MOVSB	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R E)SI to (R E)DI.
A5	MOVSW	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R E)SI to (R E)DI.
A5	MOVSD	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R E)SI to (R E)DI.

# Les registres

---

- Le registre de flags :
  - ▲ C'est un registre de 32 bits dont certains bits correspondent à des flags permettant de connaître l'état du processeur (nous verrons leurs utilités lors des sauts conditionnels). Nous ne détaillerons pas tous les flags existants, juste les plus utilisés.
- **ZF**: (zero flag) vaut 1 lorsque le résultat d'un calcul est nul.
- **SF**: (sign flag) vaut 1 lorsque le résultat d'un calcul est négatif.
- **CF**: (carry flag) contient la retenue lorsque le résultat d'un calcul ne tient pas dans le registre
- **OF**: (overflow flag) vaut 1 si le résultat d'un calcul signé ne tient pas dans le registre destination
- **PF**: (parity flag) vaut 1 si le nombre de bit d'un opérande est pair.
- **AF**: (auxilliary flag) s'apparente au CF.
- **IF**: (interrupt flag) enlève la possibilité au processeur de contrôler les interruptions si sa valeur vaut 0.

# Les Registres

---

- Les registres spécialisés :
  - Ce sont des registres faiblement documentés qui existent depuis le 386. Ils servent entre autres pour la gestion du mode protégé et la mise au point des programmes.
- **CR0 à CR4**: registres de contrôles.
- **DR0 à DR7**: registres de Debug.
- **TR3 à TR7**: registres de test.
- **GDTR**: Global Descriptor Table Register.
- **IDTR**: Interrupt Descriptor Table Register.
- **LDTR**: Local Descriptor Table Register.
- **TR**: Task Register.
- **TSC**: Time Stamp Counter.
- Les registres **MMX** et **SIMD**

# Les modes d'adressages

---

- Les opérandes d'une instruction peuvent être exprimées de plusieurs façons :
  - Directe (valeur immédiate)
  - Indirecte (valeur dans une case mémoire)
  - Registre (valeur dans un registre)
  - Basée (valeur indirecte via registre)
  - Indexée (Basée + Registre {+ Echelle})
  - Basée/Indexée + décalage

# Les modes d'adressage

---

- Adressage immédiat
  - ▲ opérande contient une donnée
  - ▲ mov eax,1234h
- Adressage relatif
  - ▲ Ex: EB FE
- Adressage indirect
  - ▲ L'opérande contient une adresse mémoire (ex : variable globale)
  - ▲ mov eax, dword ptr [0x401000]
  - ▲ Attention écrire mov [0x401000], 0 est ambigu, il faut préciser la taille de la destination (byte, word, dword, qword)

# Les modes d'adressage

---

- Adressage basé

- ▲ Notion de pointeur : un registre pointe sur une donnée
- ▲ `mov eax, [ebx]`
- ▲ `eax` a donc le contenu de la zone mémoire de l'adresse contenu dans `ebx`

- Adressage basé avec déplacement

- ▲ Adressage basé + valeur contenu dans l'instruction
- ▲ Notion de tableau, structure, ...
- ▲ `mov eax, [ebp + 8]`

# Les Modes d'adressage

---

- Adressage indexé + scale contenue dans le code opératoire
  - ▲ scale = 1, 2, 4, ou 8 (en 32 bits)
  - ▲ registre1 + scale \* registre2
- Adressage indexé + scale + décalage :
  - ▲ Adressage indexé avec échelle et décalage fixes contenus dans le code opératoire
  - ▲ registre1 + scale \* registre2 + offset
  - ▲ `mov eax, dword ptr [edx+eax*8+401000]`

# Le jeux d'instructions

---

- Mouvement de données
- Flux du programme
- Manipulation de la pile
- Les flags
- Opérations sur les bits
- Chaînes et tableaux

# Les Instructions d'affectation

---

- L' instruction MOV

- ▲ Syntaxe:

**MOV destination, source**

- ▲ Cette instruction est utilisée pour copier une valeur d'un endroit à un autre. Cet endroit peut être un registre, une adresse mémoire ou une valeur immédiate.
  - ▲ La taille de source et de destination doit être la même.

# Les Instructions d'affectation

---

## • L'instruction MOV: exemples

▲ mov edx, ecx

- *Copie le contenu du registre ECX dans le registre EDX.*

▲ mov eax, 56h

- *Le registre EAX va prendre la valeur hexadécimale 56.*

▲ mov eax, dword ptr [0000003Ah]

- *Copie la valeur qui à la taille d'un dword (32 bits) se trouvant à l'emplacement mémoire 3A dans le registre EAX.*

▲ mov eax, [0000003Ah]

- *Donne le même résultat que précédemment.*

▲ Les [] sont utilisés pour récupérer la valeur stockée à l'emplacement mémoire se trouvant entre les crochets.

# Les Instructions d'affectation

---

- L' instruction XCHG

- Syntaxe:

```
XCHG registre1, registre2
```

- Cette instruction est utilisée pour échanger la valeur contenue dans 2 registres (ou emplacement mémoire) entre eux.
  - La taille de la source et de la destination doit être la même.

# Les Instructions d'affectation

---

- L' instruction XCHG: exemples

- mov eax, 56h

- mov ebx, 57h

- xchg eax, ebx

- *Le registre EAX a pour valeur 57 et EBX prend la valeur 56.*

# Les Opérations sur la pile

---

- Les instructions PUSH, POP

- ▲ Syntaxe:

**PUSH, POP registre ou valeur immédiate**

- ▲ Push place la valeur de registre en haut de la pile et pop place la valeur en haut de la pile dans le registre.
  - ▲ La valeur du registre ESP (pointeur de pile) est décrémenté de la taille du registre (32 bits) lorsque l'on push une valeur sur la pile et est incrémentée de la taille du registre lorsque l'on pop.

# Les Opérations sur la pile

- Les instructions PUSH, POP: exemples

- mov eax,100h  
mov ebx,200h  
push eax  
push ebx  
pop eax  
pop ebx

- *Le registre EAX a pour valeur 100h et EBX 200h.*
    - *On place le contenu de EAX sur le haut de la pile (100h).*
    - *On place le contenu de EBX sur le haut de la pile (200h).*
    - *On place la valeur au sommet de la pile dans EAX*  
EAX a pour valeur 200h
    - *On place la valeur au sommet de la pile dans EBX*  
EBX a pour valeur 100h

# Les Opérations sur la pile

---

- Les instructions **PUSHA, POPA**

- ▲ Syntaxe:

**PUSHA, POPA**

- ▲ pusha place la valeur de tous les registres généraux en haut de la pile et popa récupère les valeurs en haut de la pile pour les mettre dans les registres généraux.

# Les Opérations Arithmétiques

---

- Les instructions ADD, SUB

- ▲ Syntaxe:

- ADD, SUB destination, source**

- ▲ Ces opérations ajoutent ou soustraient la valeur source à la valeur destination. Le résultat est stocké dans destination.

# Les Opérations Arithmétiques

---

- Les instructions INC, DEC: exemples

- ▲ mov eax, 56h

- inc eax

- *Le registre EAX a pour valeur  $56h + 1h = 57h$ .*

- ▲ mov eax, 56h

- dec eax

- *Le registre EAX a pour valeur  $56h - 1h = 55h$ .*

# Les Opérations Arithmétiques

---

- Les instructions ADD, SUB: exemples

- ▲ mov eax, 200h

- add eax, 100h

- Le registre EAX a pour valeur final 300h.*

- ▲ mov eax, 200h

- sub eax, 100h

- Le registre EAX a pour valeur final 100h.*

# Les Opérations Arithmétiques

---

- Les instructions IMUL, IDIV

- Syntaxe:

**IMUL, IDIV destination, source**

- Ces opérations multiplie ou divise la valeur destination par la valeur source. Le résultat est stocké dans destination.
  - Si on ne place qu'un seul opérande, on suppose que l'opérande destination est EAX par défaut (registre implicite).

# Les Opérations Arithmétiques

---

- Les instructions IMUL, IDIV: exemples

- ▲ mov eax, 2Fh  
imul eax, Ah
  - *Le registre EAX a pour valeur final 1D6h.*
- ▲ mov eax, 2Fh  
idiv eax, Ah
  - *Le registre EAX a pour valeur final 4h.*

# LEA (Load Effective Address)

---

- `lea eax, [ebp + var_3]`
- `lea eax, [base + index * {0,1,2,4,8} + offset]`
- `lea eax,[eax+eax*4] ;MUL eax,5`

# Les instructions logiques

- AND destination, source
- OR destination, source
- XOR destination, source
- NOT destination

Instructions	AND				OR				XOR				NOT	
Source	0	0	1	1	0	0	1	1	0	0	1	1	0	1
Destination	0	1	0	1	0	1	0	1	0	1	0	1	x	x
Résultat	0	0	0	1	0	1	1	1	0	1	1	0	1	0

# Les instructions de comparaisons

---

- **CMP**

- **CMP**
- ▲ Syntaxe cmp oprande1, oprande2
- ▲ Cette instruction soustrait la valeur de operand2 à la valeur de operand1 et ajuste les flags en fonction du résultat.

- **TEST**

- **TEST**
- ▲ Syntaxe : test operand1, operand2
- ▲ Cette instruction réalise l'opération de ET logique entre la valeur de operand2 et la valeur de operand1 et ajuste les flags en fonction du résultat.

# Instructions de branchement

---

- Inconditionnel

- Inconditionnel
  - JMP
  - Cette instruction transfert l'exécution du programme à un autre endroit du code. L'opérande peut être un label ou une valeur.

- Conditionnel

- Conditionnel
  - jcc
    - JA : jump if above ( $CF=0$  et  $ZF=0$ )
    - JAE : jump if above or equal ( $CF=0$ )
    - JB : jump if below ( $CF=1$ )
    - JBE : jump if below or equal ( $CF=1$  ou  $ZF=1$ )
    - JNE : jump if not equal ( $ZF=0$ )
    - JZ : jump if zero ( $ZF=1$ )
    - JNZ : jump if not zero ( $ZF=0$ )

# carry flag / overflow flag

A				B				A - B				Flags				
h	ud	d		h	ud	d		h	ud	d			OF	SF	ZF	CF
FF	255	-1		FE	254	-2		1	1	1			0	0	0	0
7E	126	126		FF	255	-1		7F	127	127			0	0	0	1
FF	255	-1		FF	255	-1		0	0	0			0	0	1	0
FF	255	-1		7F	127	127		80	128	-128			0	1	0	0
FE	254	-2		FF	255	-1		FF	255	-1			0	1	0	1
FE	254	-2		7F	127	127		7F	127	127			1	0	0	0
7F	127	127		FF	255	-1		80	128	-128			1	1	0	1

# Carry flag / overflow flag

```
mov al, 0x7F  
mov bl, 0xFF  
cmp al, bl
```

# Carry flag / overflow flag

A				B				A - B				Flags												
h	ud	d	h	ud	d	h	ud	d	h	ud	d	OF	SF	ZF	CF									
7F		127		127		FF		255		-1		80		128		-128		1		1		0		1

```
mov al, 0x7F  
mov bl, 0xFF  
cmp al, bl
```

# Carry flag / overflow flag

A				B				A - B				Flags												
h	ud	d	h	ud	d	h	ud	d	h	ud	d	OF	SF	ZF	CF									
7F		127		127		FF		255		-1		80		128		-128		1		1		0		1

```
mov al, 0x7F  
mov bl, 0xFF  
cmp al, bl
```

jb (jump below CF=1)

jl (jump less SF != OF)

jg (jump greater ZF=0 and SF=OF)

ja (jump above CF=0 and ZF=0)

# Les instructions de manipulation de chaînes

---

- scas {b,w,d} : Scan String
- movs {b, w, d} : Move Data from String to String (esi to edi)
- stos {b, w, d} : Store String
- cmps {b, w, d} : Compare String Operands
- lod\$ {b, w, d} : Load String
- rep : Repeat String Operation Prefix
- exemple : strlen

# Les instructions de manipulation de chaînes

---

xor ecx, ecx

not ecx

xor eax, eax

cld

mov edi, [str]

repne scasb

; ecx = -strlen - 2

not ecx

dec ecx

# Appel de fonctions

---

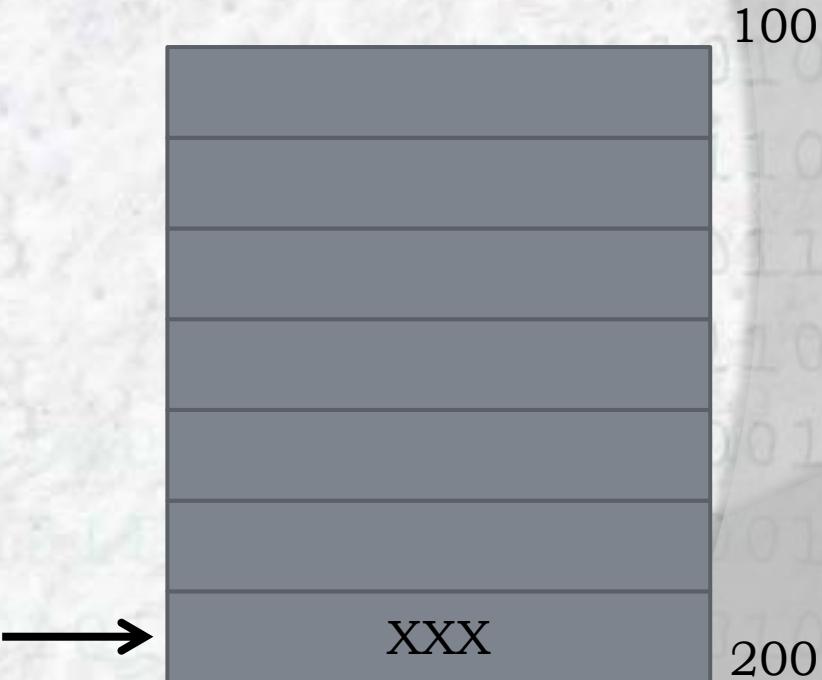
- call / ret
- call 0baadf00dh
  - ▲ push @return\_address
  - ▲ jmp 0baadf00dh
- ret
  - ▲ pop @return\_address
  - ▲ jmp @return\_address
- Etablissement d'un cadre de pile

# Exemple

→ @0: push 10  
@1: push 20  
@2: call add  
@3: ....

add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret

eip : 0  
esp : 200

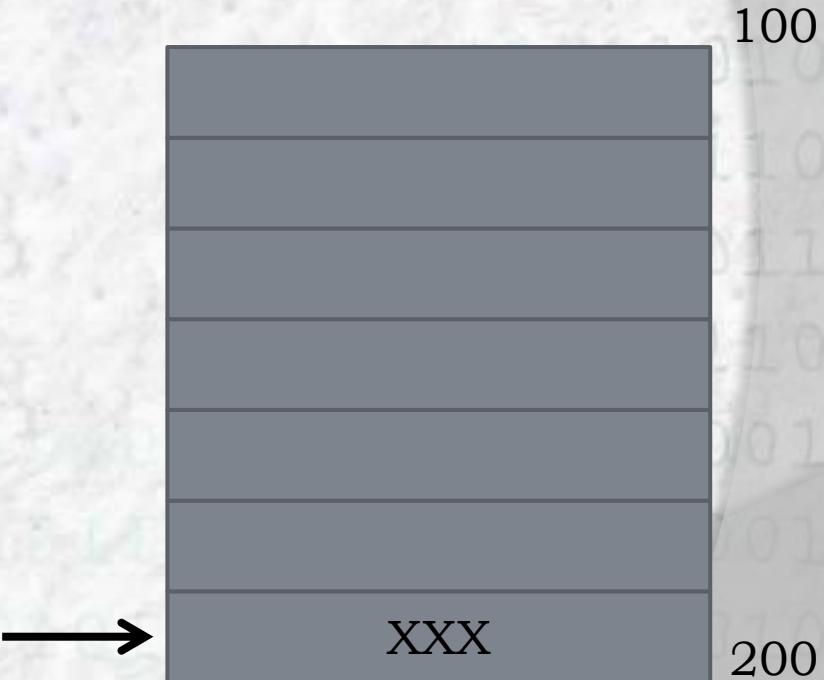


# Exemple

→ @0: push 10  
@1: push 20  
@2: call add  
@3: ....

add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret

eip : 0  
esp : 200

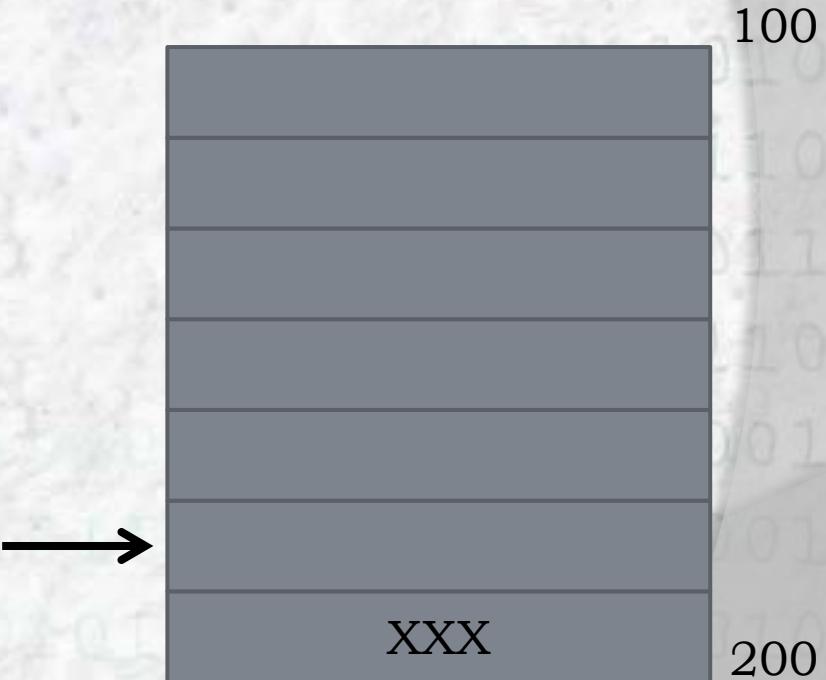


# Exemple

→ @0: push 10  
@1: push 20  
@2: call add  
@3: ....

add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret

eip : 0  
esp : 196

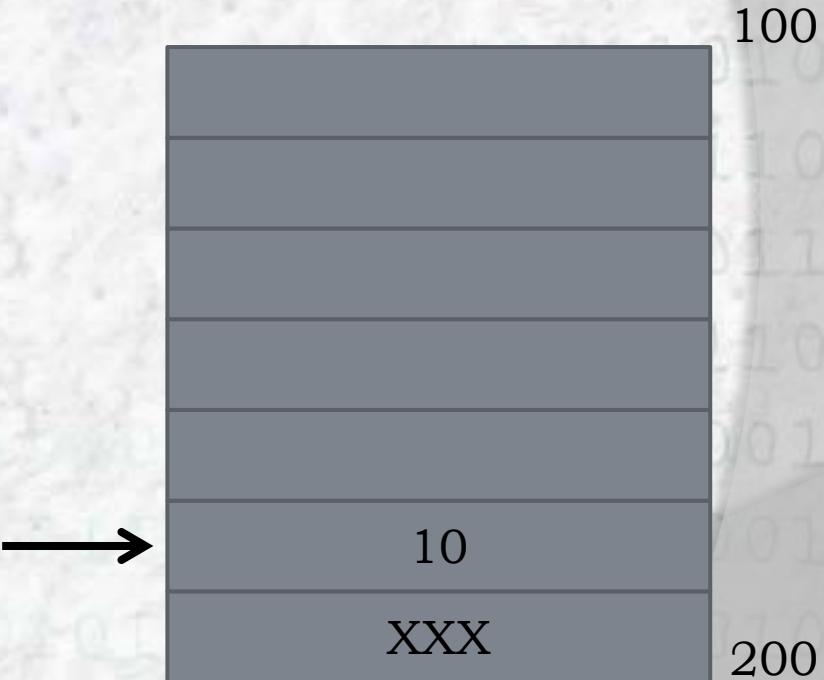


# Exemple

→ @0: push 10  
@1: push 20  
@2: call add  
@3: ....

add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret

eip : 0  
esp : 196

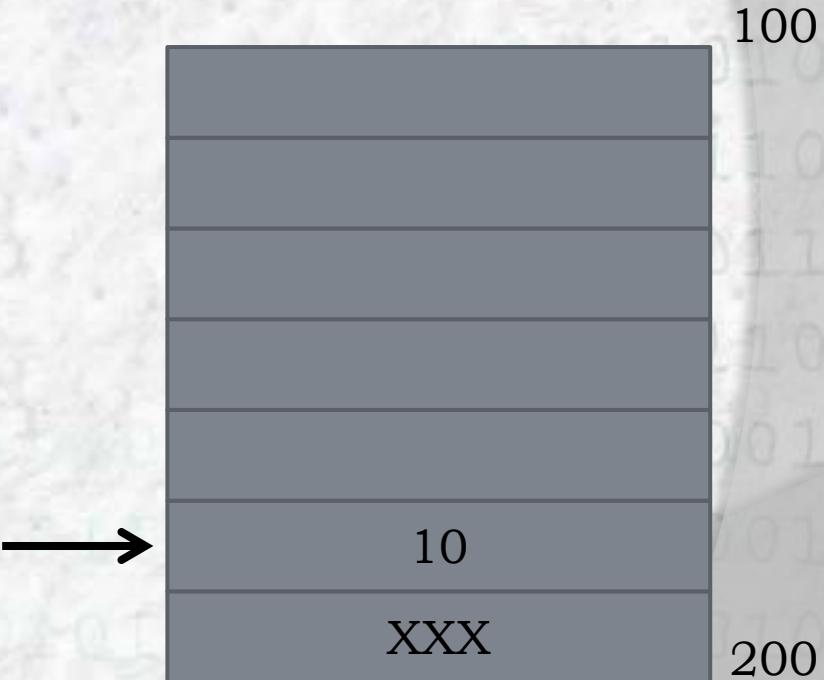


# Exemple

→  
@0: push 10  
@1: push 20  
@2: call add  
@3: ....

add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret

eip : 1  
esp : 196

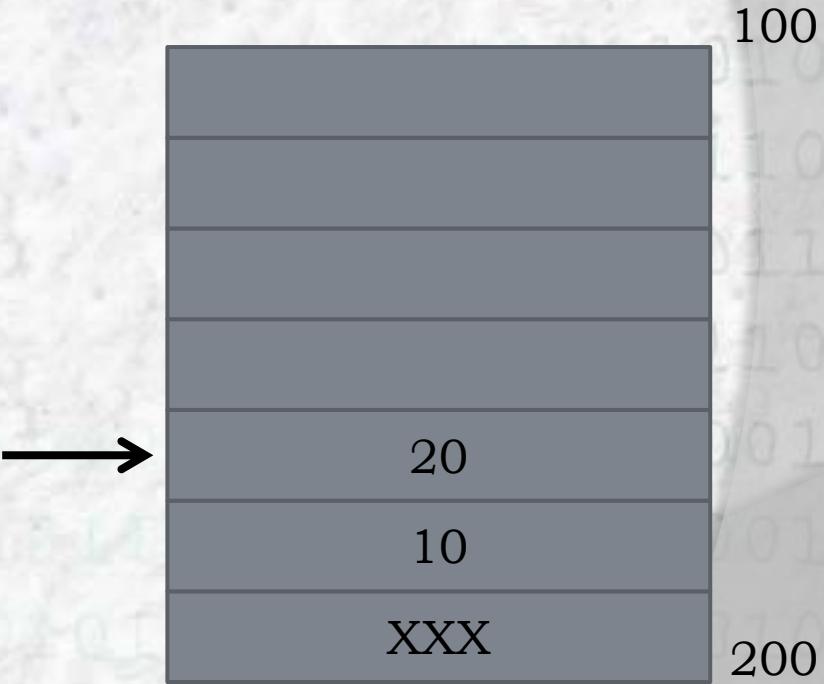


# Exemple

→  
@0: push 10  
@1: push 20  
@2: call add  
@3: ....

add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret

eip : 2  
esp : 192

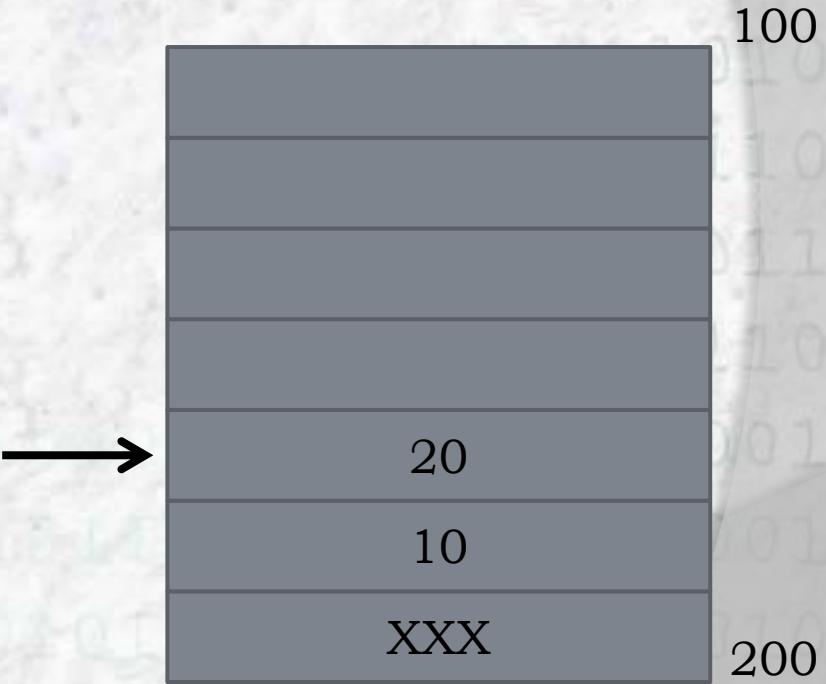


# Exemple

→  
@0: push 10  
@1: push 20  
@2: call add  
@3: ....

add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret

eip : 2  
esp : 192



# Exemple

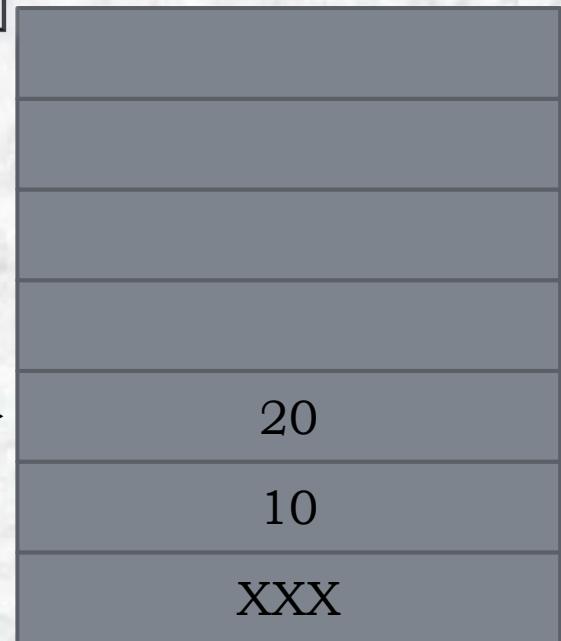
```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

```
push @ret_addr (@3)  
jmp add
```

eip : 2  
esp : 192

100

```
add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret
```



# Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

```
push @ret_addr (@3)  
jmp add
```

eip : 2  
esp : 192

100

```
add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret
```



@3

20

10

XXX

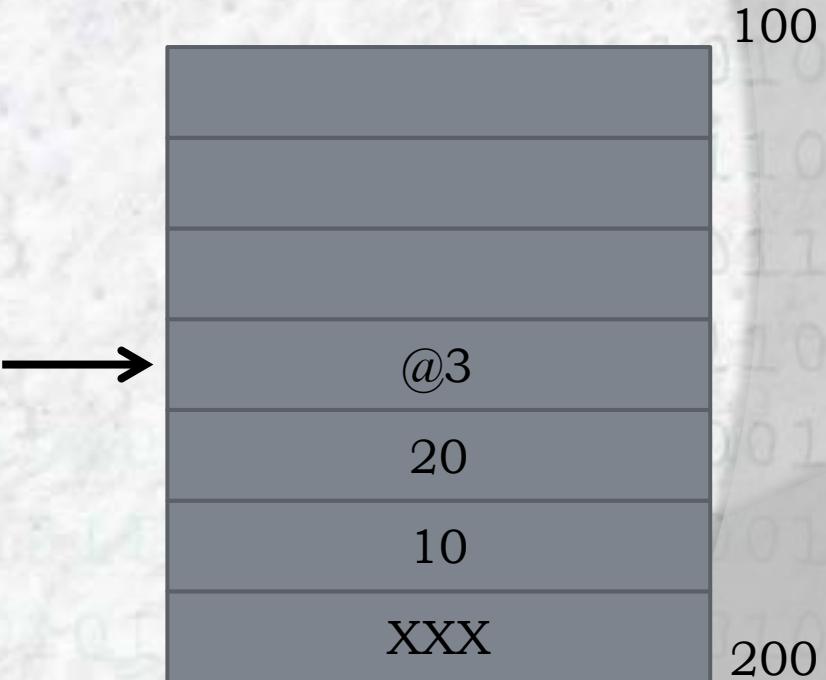
200

## Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

```
add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret
```

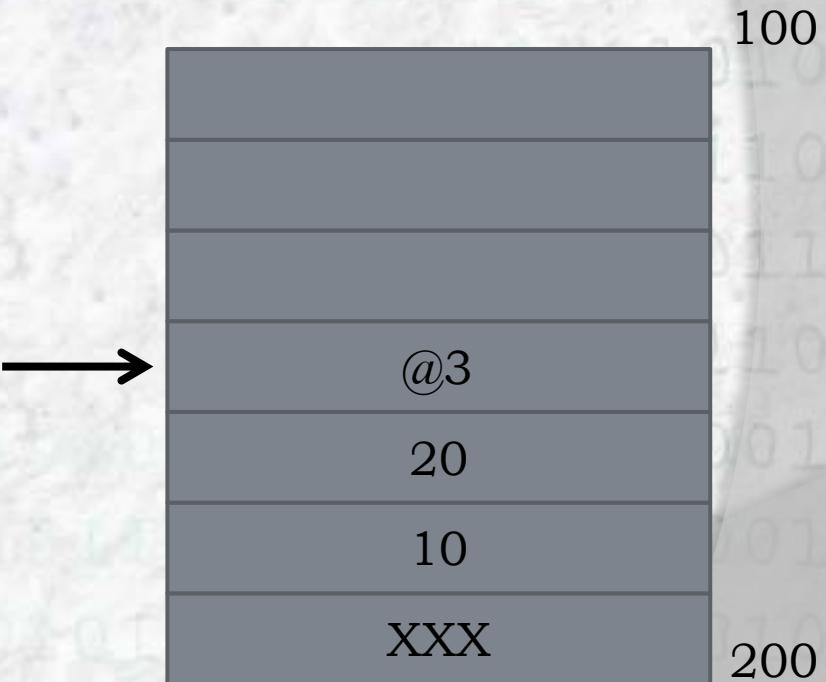
eip : 20  
esp : 192



## Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

eip : 20  
esp : 192  
eax : xxx



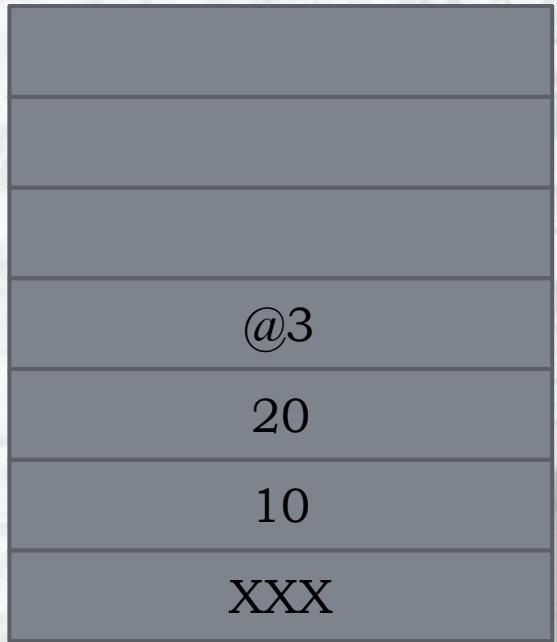
```
add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret
```

# Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

add:  
→ @20: mov eax, [esp+4]  
→ @21: add eax, [esp+8]  
@22: ret

eip : 21  
esp : 192  
eax : 20

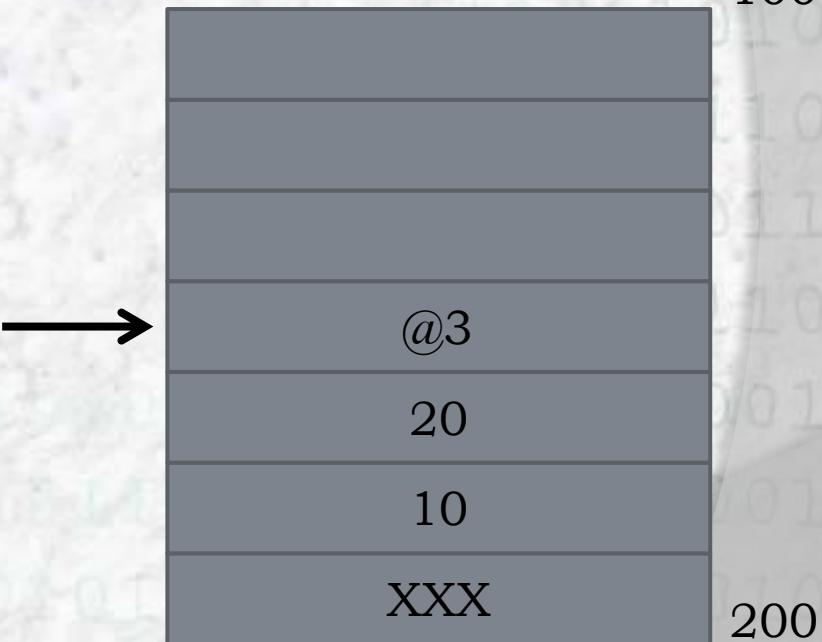


## Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

```
add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret
```

eip : 22  
esp : 192  
eax : 30

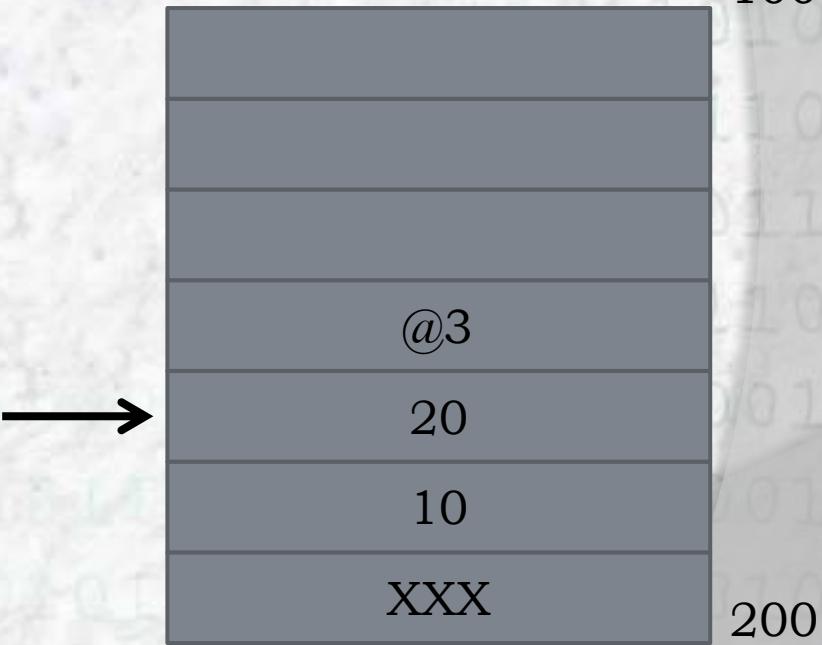


# Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
→ @3: ....
```

add:  
@20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret

eip : 3  
esp : 196  
eax : 30

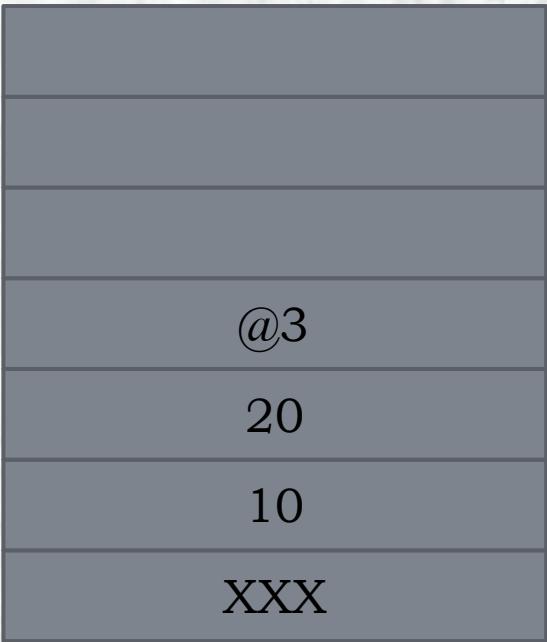


# Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

add:  
→ @20: mov eax, [esp+4]  
@21: add eax, [esp+8]  
@22: ret

eip : 20  
esp : 192  
eax : XXX



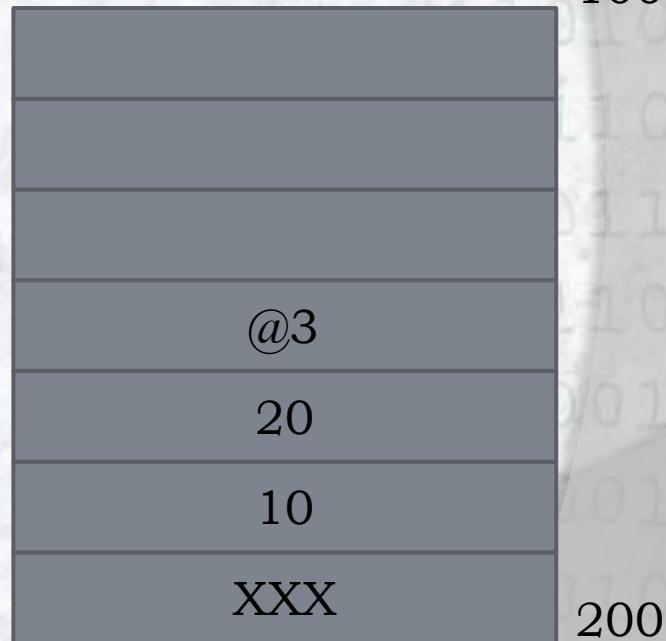
# Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

add:

```
→ @20: push ecx  
@21: mov eax, [esp+8]  
@22: add eax, [esp+12]  
@23: pop ecx  
@24: ret
```

eip : 20  
esp : 192  
eax : XXX



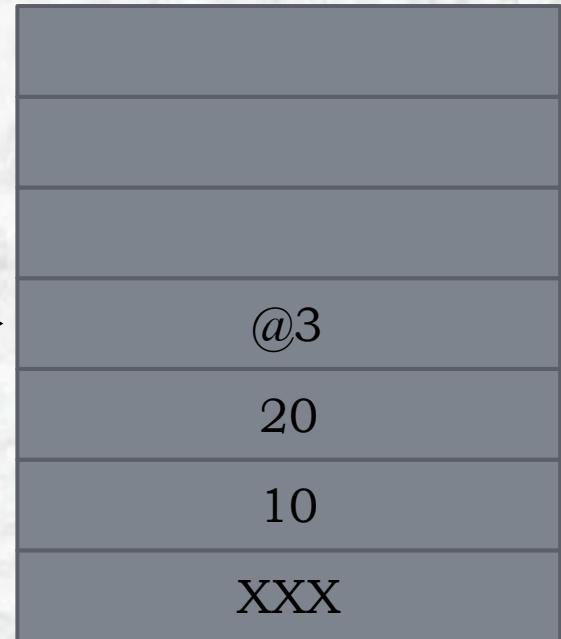
# Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

add:

→ @20: push ebp  
@21: mov ebp, esp  
@22: sub esp, 8  
@23: mov eax, [ebp+8]  
@24: add eax, [ebp+12]  
@25: mov esp, ebp  
@26: pop ebp  
@27: ret

eip : 20  
esp : 192  
eax : XXX



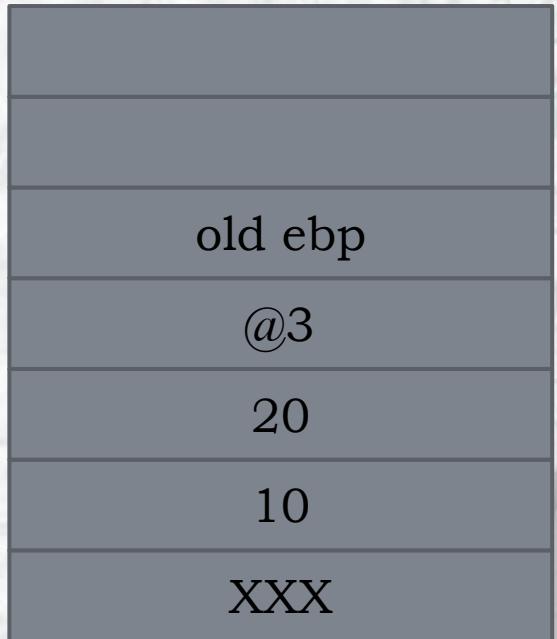
# Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

add:

```
→ @20: push ebp  
@21: mov ebp, esp  
@22: sub esp, 8  
@23: mov eax, [ebp+8]  
@24: add eax, [ebp+12]  
@25: mov esp, ebp  
@26: pop ebp  
@27: ret
```

eip : 21  
esp : 188  
eax : XXX



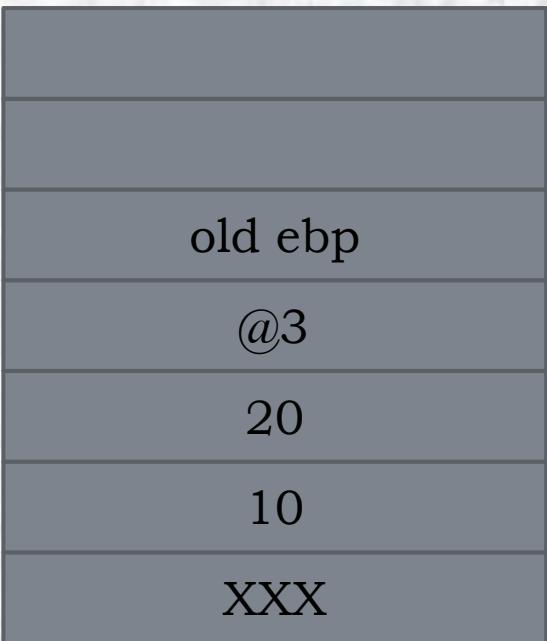
# Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

ebp →

eip : 22  
esp : 188  
eax : XXX

add:  
→ @20: push ebp  
@21: mov ebp, esp  
@22: sub esp, 8  
@23: mov eax, [ebp+8]  
@24: add eax, [ebp+12]  
@25: mov esp, ebp  
@26: pop ebp  
@27: ret

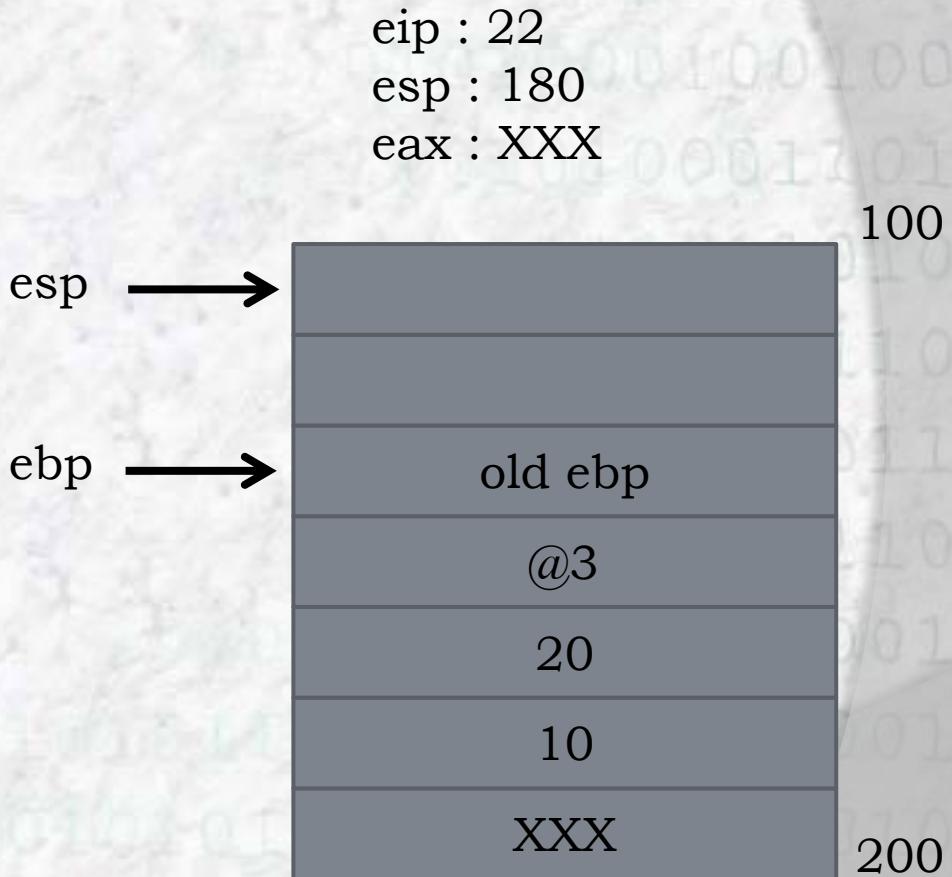


# Exemple

```
@0: push 10  
@1: push 20  
@2: call add  
@3: ....
```

add:

```
@20: push ebp  
@21: mov ebp, esp  
@22: sub esp, 8  
→ @23: mov eax, [ebp+8]  
@24: add eax, [ebp+12]  
@25: mov esp, ebp  
@26: pop ebp  
@27: ret
```



```
str_debug db 'trace : my_add ok', 0
```

```
→ push 10  
push 20  
call my_add
```

```
my_add:
```

```
arg1 equ [ebp+8]  
Arg2 equ [ebp+12]  
result equ [ebp-4]  
push ebp  
mov ebp, esp
```

```
mov eax, arg1  
add eax, arg2  
mov result, eax
```

```
push 3  
push offset str_debug  
call log_level
```

```
esp →
```

```
mov eax, result  
mov esp, ebp  
pop ebp  
ret 8
```



```
...
```

```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

→ push 20

```
call my_add
```

```
my_add:
```

```
arg1 equ [ebp+8]
```

```
Arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

```
push 3
```

```
push offset str_debug esp →
```

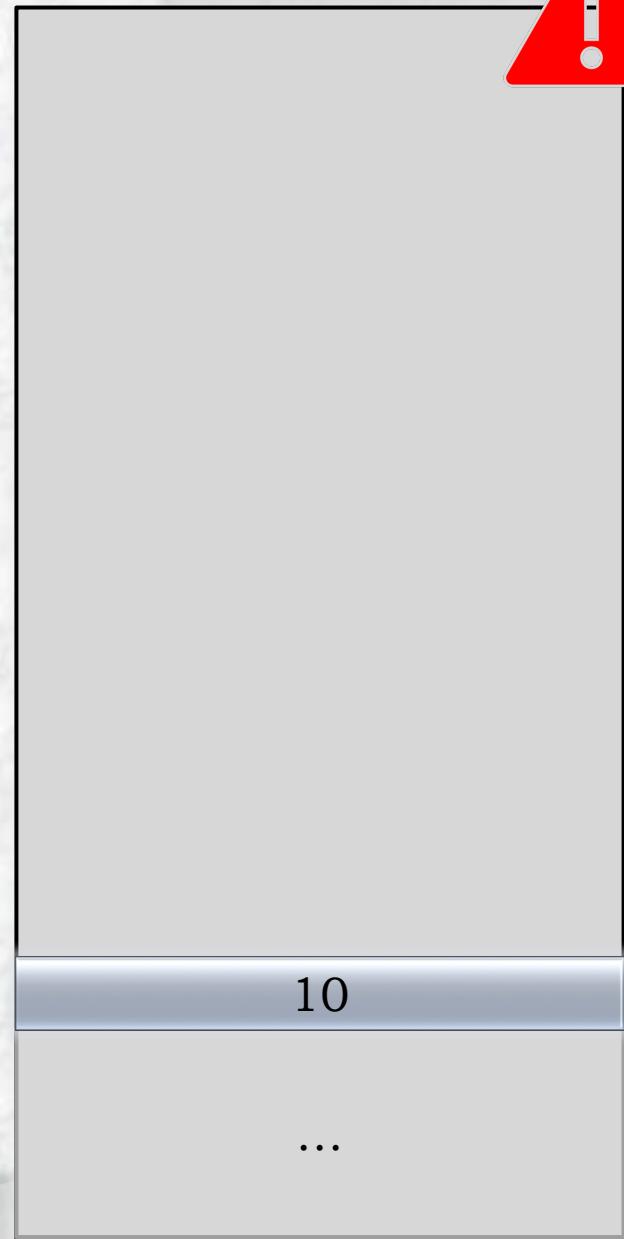
```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```



```
str_debug db 'trace : my_add ok', 0  
push 10  
push 20  
→ call my_add
```



```
my_add:  
arg1 equ [ebp+8]  
arg2 equ [ebp+12]  
result equ [ebp-4]  
push ebp  
mov ebp, esp
```

```
mov eax, arg1  
add eax, arg2  
mov result, eax
```

```
push 3  
push offset str_debug  
call log_level
```

```
mov eax, result  
mov esp, ebp  
pop ebp  
ret 8
```

esp →



```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

my\_add:

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

esp →

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```



@retour my\_add

20

10

...

```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

my\_add:

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

esp →

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```



saved ebp

@retour my\_add

20

10

...

```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

```
my_add:
```

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

→  

```
mov eax, arg1  
add eax, arg2  
mov result, eax
```

esp  
ebp →

```
push 3  
push offset str_debug  
call log_level
```

```
mov eax, result  
mov esp, ebp  
pop ebp  
ret 8
```



saved ebp

@retour my\_add

20

10

...

```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

```
my_add:
```

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
→ mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

esp  
ebp



saved ebp

@retour my\_add

20

10

...

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```



```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

```
my_add:
```

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

esp  
ebp



saved ebp

@retour my\_add

20

10

...

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```



```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

```
my_add:
```

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

esp  
ebp



```
push 3
```

```
push offset str_debug
```

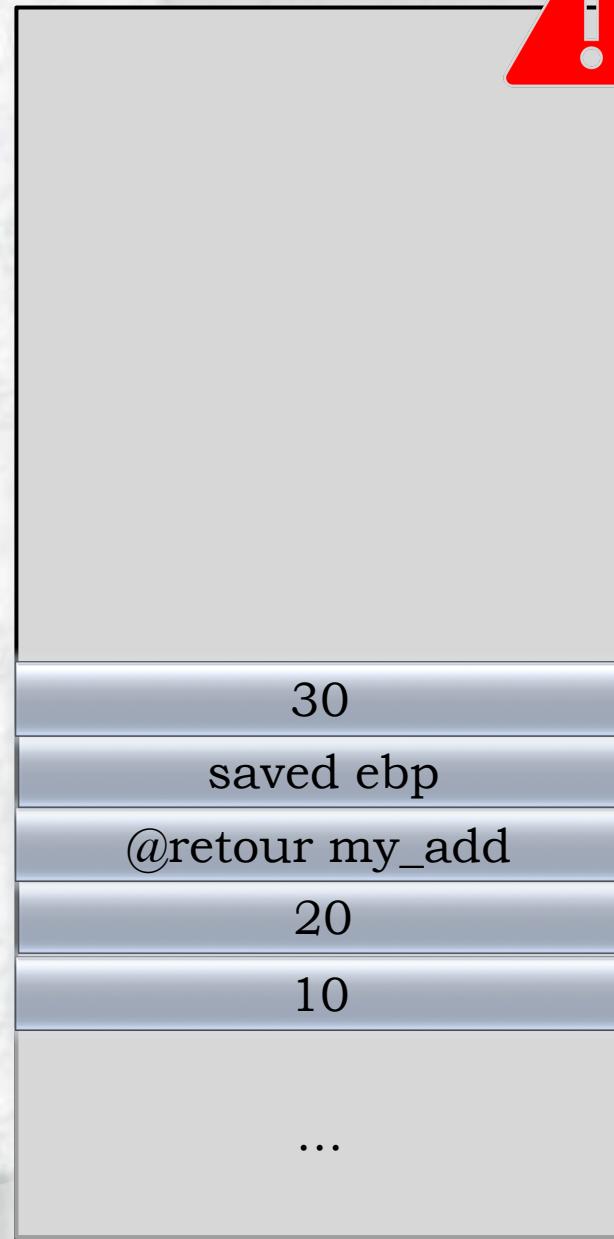
```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```



```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

```
my_add:
```

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```

esp →

ebp →

3

saved ebp

@retour my\_add

20

10

...



```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

```
my_add:
```

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```

esp →

ebp →



@str_debug
3
saved ebp
@retour my_add
20
10
...

```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

my\_add:

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
sub esp, 4
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

esp →

@retour my\_add

20

10

...

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```

```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

my\_add:

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
sub esp, 4
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

esp →

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```

saved ebp

@retour my\_add

20

10

...

```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

my\_add:

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
sub esp, 4
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

esp  
ebp

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```



saved ebp

@retour my\_add

20

10

...

```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

my\_add:

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
sub esp, 4
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```

esp →

ebp →

saved ebp

@retour my\_add

20

10

...

```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

→  
my\_add:

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
sub esp, 4
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```

esp →

ebp →

saved ebp

@retour my\_add

20

10

...

```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

my\_add:

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
sub esp, 4
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

→ esp →

→ ebp →

```
push 3
```

```
push offset str_debug
```

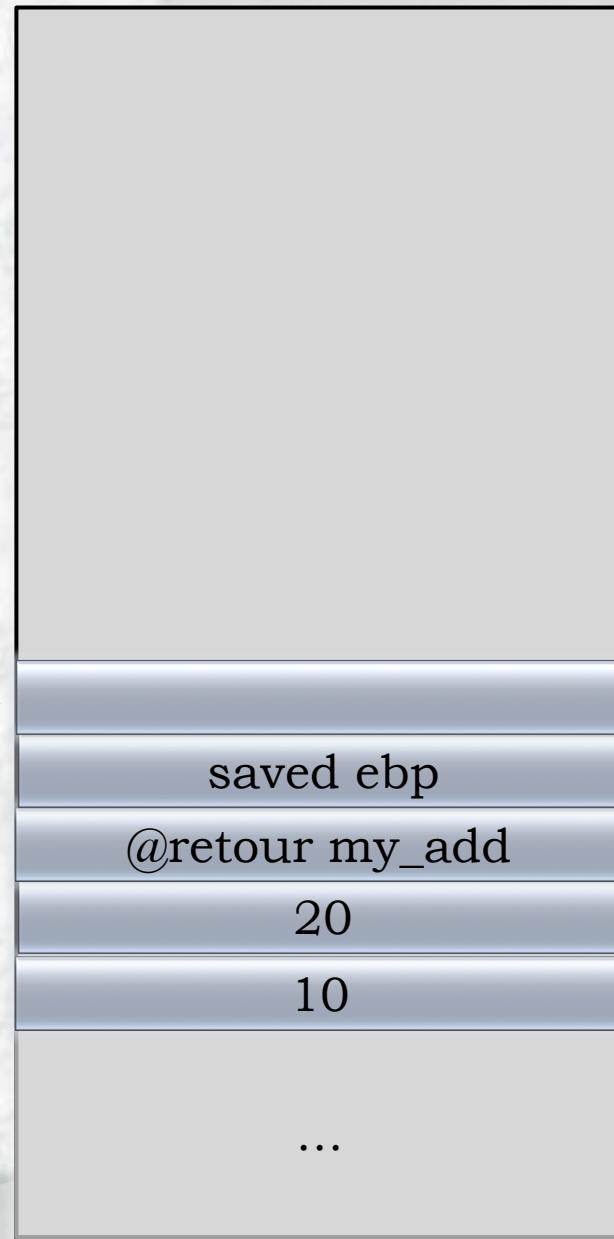
```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```



```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

```
my_add:
```

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
sub esp, 4
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

esp →

ebp →

```
push 3
```

```
push offset str_debug
```

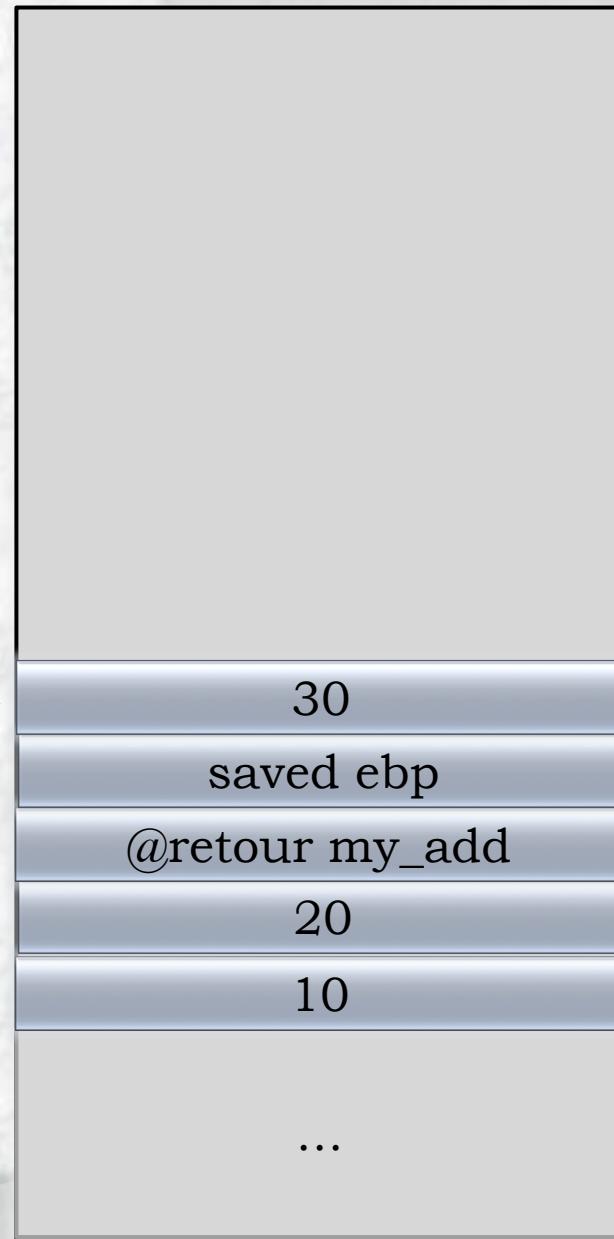
```
call log_level
```

```
mov eax, result
```

```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```



```
str_debug db 'trace : my_add ok', 0
```

```
push 10
```

```
push 20
```

```
call my_add
```

```
my_add:
```

```
arg1 equ [ebp+8]
```

```
arg2 equ [ebp+12]
```

```
result equ [ebp-4]
```

```
push ebp
```

```
mov ebp, esp
```

```
sub esp, 4
```

```
mov eax, arg1
```

```
add eax, arg2
```

```
mov result, eax
```

```
push 3
```

```
push offset str_debug
```

```
call log_level
```

```
mov eax, result
```

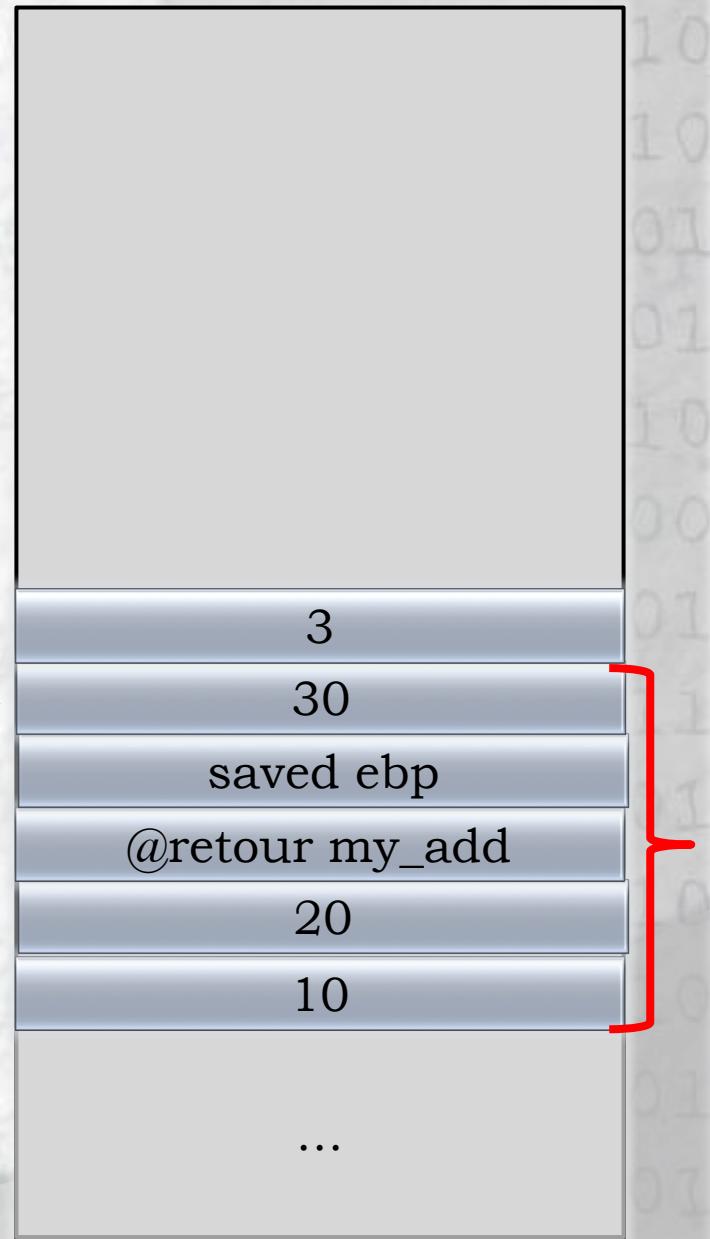
```
mov esp, ebp
```

```
pop ebp
```

```
ret 8
```

esp →

ebp →



**My\_func:**

```
arg1    equ [ebp+8]  
arg2    equ [ebp+12]
```

esp →

```
push ebp  
mov ebp, esp  
sub esp, 264
```

...

...

ebp →

saved ebp

@retour my\_addr

20

10

...

My\_func:

```
arg1    equ [ebp+8]
arg2    equ [ebp+12]
path    equ [ebp-264]
handle  equ [ebp-4]
```

```
push  ebp
mov   ebp, esp
sub   esp, 264
```

...

esp →

path

ebp →

handle

saved ebp

@retour my\_addr

20

10

...

My\_func:

```
arg1    equ [ebp+8]
arg2    equ [ebp+12]
path    equ [ebp-260]
handle  equ [ebp-264]
```

```
push  ebp
mov   ebp, esp
sub   esp, 264
```

...

esp →

handle

path

ebp →

saved ebp

@retour my\_addr

20

10

...

# Les conventions d'appels

- cdecl

- ▲ Les arguments sont passés de la droite vers la gauche
- ▲ L'appelant nettoie la pile
- ▲ Valeur de retour dans eax (x86)

```
int __cdecl add(int a, int b)
{
    return a + b;
}
```

```
x = add(2, 3);
```

```
add:
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret
```

```
push 3
push 2
call add
add esp, 8
```

# Les conventions d'appels

- stdcall (WINAPI)

- ▲ Les arguments sont passés de la droite vers la gauche
- ▲ L'appelé nettoie la pile
- ▲ Valeur de retour dans eax (x86)

```
int __stdcall add(int a, int b)
{
    return a + b;
}
```

```
x = add(2, 3);
```

```
add:
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret 8
```

```
push 3
push 2
call add
```

# Les conventions d'appels

## • Fastcall

- ▲ Les deux premiers arguments (de 32 bits max) sont passés dans ecx et edx
- ▲ Les autres arguments sont passés de la droite vers la gauche
- ▲ L'appelé nettoie la pile
- ▲ Valeur de retour dans eax (x86)

```
int __fastcall add(int a, int b)
{
    return a + b;
}
```

```
x = add(2, 3);
```

```
add:
push ebp
mov ebp, esp
mov eax, edx
add eax, ecx
pop ebp
ret
```

BADGE

```
mov edx, 3
mov ecx, 2
call add
```

# Link register

---

- Registre qui contient l'adresse de retour lors d'un appel.
- Utilisé sur ARM / MIPS / PPC ...

# PPC

---

Register	Classification	Notes
r0	local	commonly used to hold the old link register when building the stack frame
r1	dedicated	stack pointer
r2	dedicated	table of contents pointer
r3	local	commonly used as the return value of a function, and also the first argument in
r4-r10	local	commonly used to send in arguments 2 through 8 into a function
r11-r12	local	
r13-r31	global	
lr	dedicated	link register; cannot be used as a general register.
cr	dedicated	condition register

# PPC

---

Register	Classification	Notes
r0	local	commonly used to hold the old link register when building the stack frame
r1	dedicated	stack pointer
r2	dedicated	table of contents pointer
r3	local	commonly used as the return value of a function, and also the first argument in
r4-r10	local	commonly used to send in arguments 2 through 8 into a function
r11-r12	local	
r13-r31	global	
lr	dedicated	link register; cannot be used as a general register.
cr	dedicated	condition register

# PPC

---

bar:

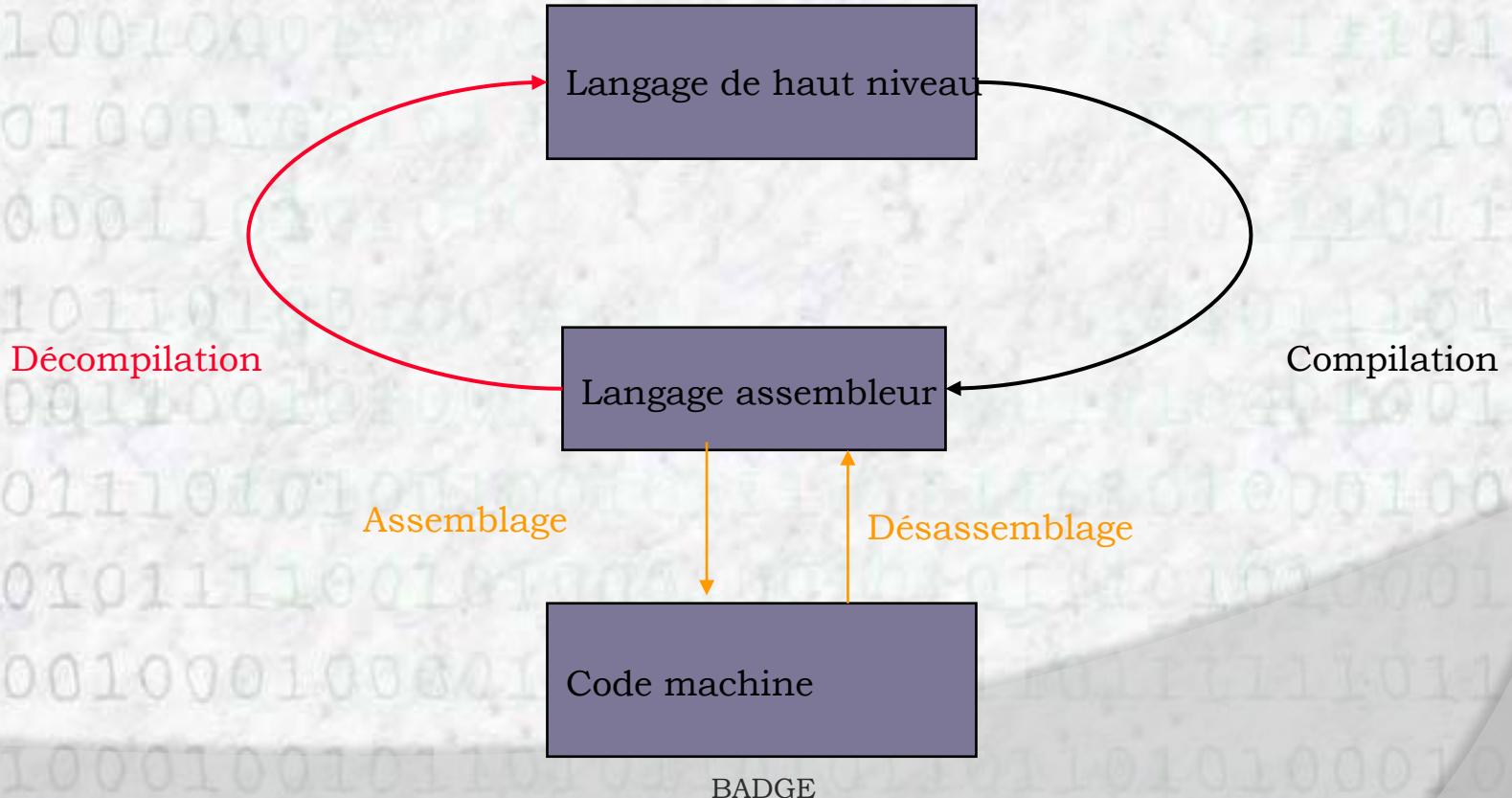
```
mflr r0          // set up the stack frame
stw r0, 8(r1)
stwu r1, -16(r1)
addi r3, r3, 3   // add 3 to the argument and return it
addi r1, r1, 16  // destroy the stack frame
lwz r0, 8(r1)
mtlr r0
blr             // return
```

.globl \_main  
\_main:

```
mflr r0          // set up the stack frame
stw r0, 8(r1)
stwu r1, -16(r1)
lis r3, hi16(847318093) // load big number into r3
ori r3, r3, lo16(847318092)
bl bar           // call stuff
addi r1, r1, 16  // destroy the stack frame
lwz r0, 8(r1)
mtlr r0
blr             // return
```

# La Compilation

- Compilation
- Edition de lien



# Structure d'un executable

```
int gGlobalFactor = 10;  
  
int gPreviousResult;  
  
int __stdcall Compute(int a, int b)  
{  
    int intermediate = a + b;  
    gPreviousResult = intermediate;  
    return intermediate * gGlobalFactor;  
}
```

# Structure d'un executable

```
int gGlobalFactor = 10; // global value initialized  
  
int gPreviousResult; // global value non-initialized  
  
int __stdcall Compute(int a, int b) // code  
{  
    int intermediate = a + b; // value stored in stack  
    gPreviousResult = intermediate;  
    return intermediate * gGlobalFactor;  
}
```

# Structure d'un executable

Nom	Description
<b>.text</b>	<b>Le instructions du programme</b>
<b>.bss</b>	<b>Les données non-initialisées</b>
<b>.data</b>	<b>Les données initialisées</b>
<b>.reloc</b>	La table des relocalisations
<b>.rsrc</b>	Les ressources du fichier (Curseurs, Sons, Menus...)
<b>.rdata</b>	Les données en lectures seule
<b>.idata</b>	La table d'import (les fonctions chargées dynamiquement)

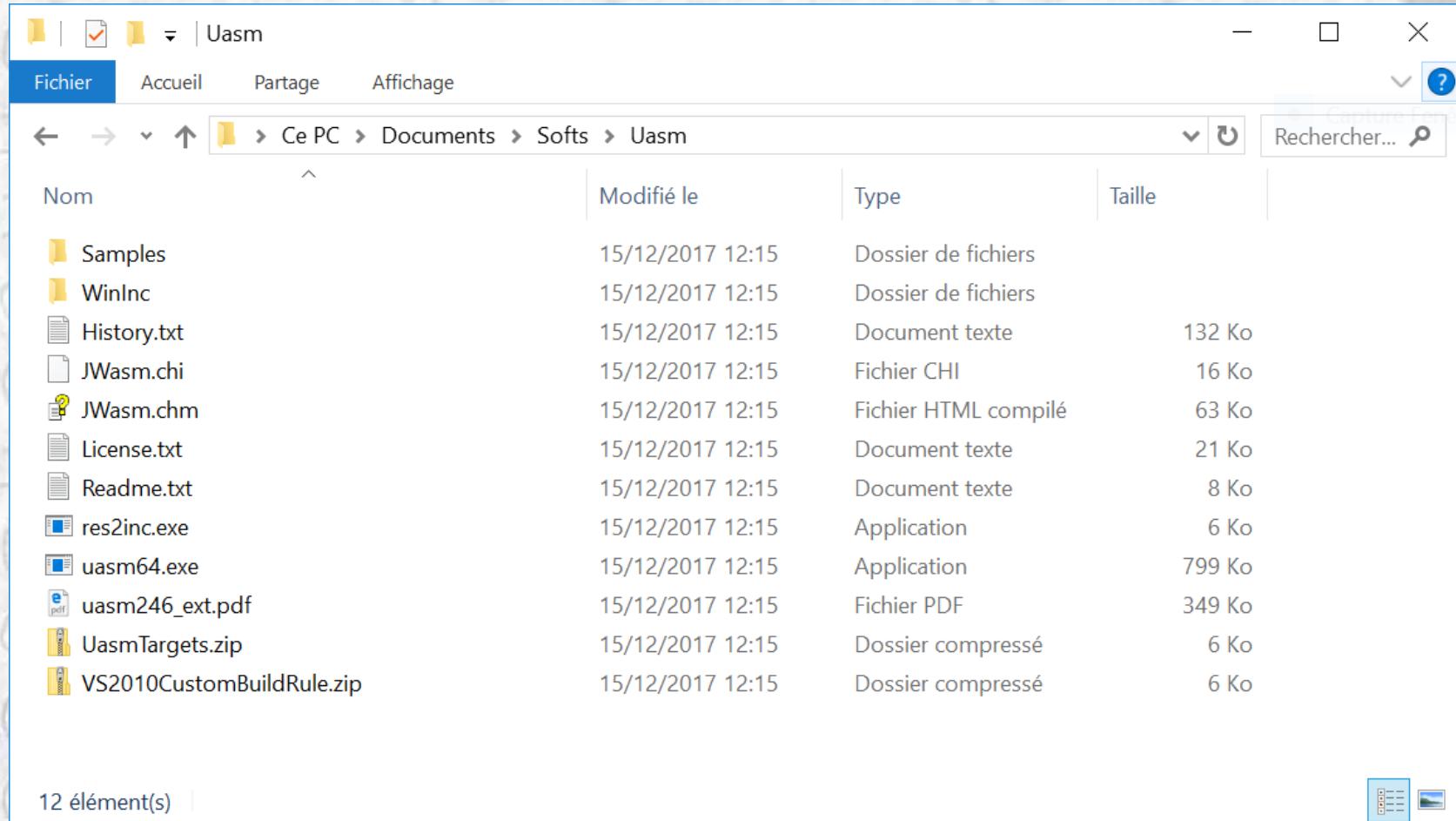
# Installation assembler / linker

---

- Trois composants :
  - UASM : assembler au format MASM
  - WinInc : fichier include et lib (à construire)
  - PellesC : linker

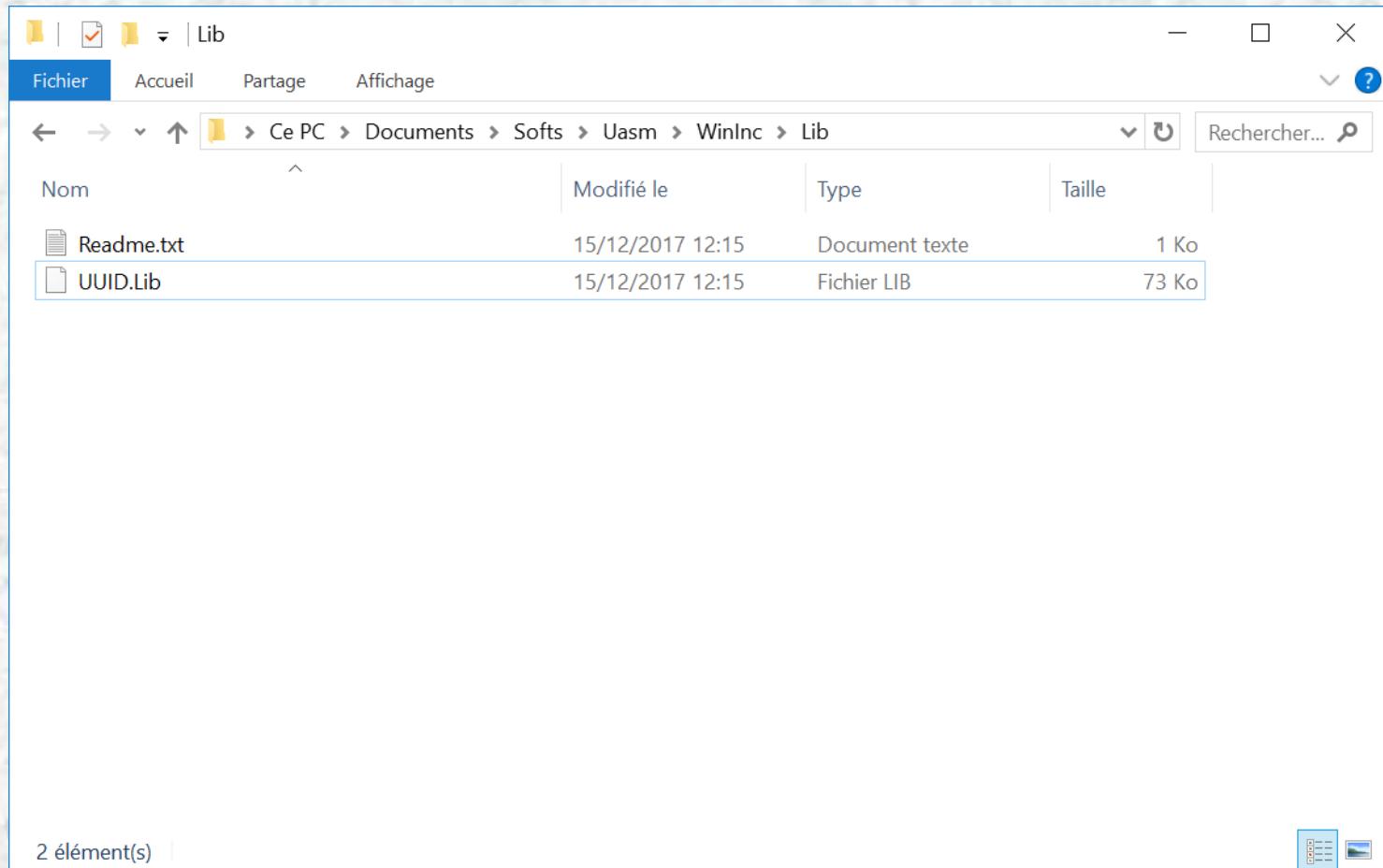
# Installation assembler / linker

**ATTENTION : PAS D'ESPACE DANS LE CHEMIN !!!**



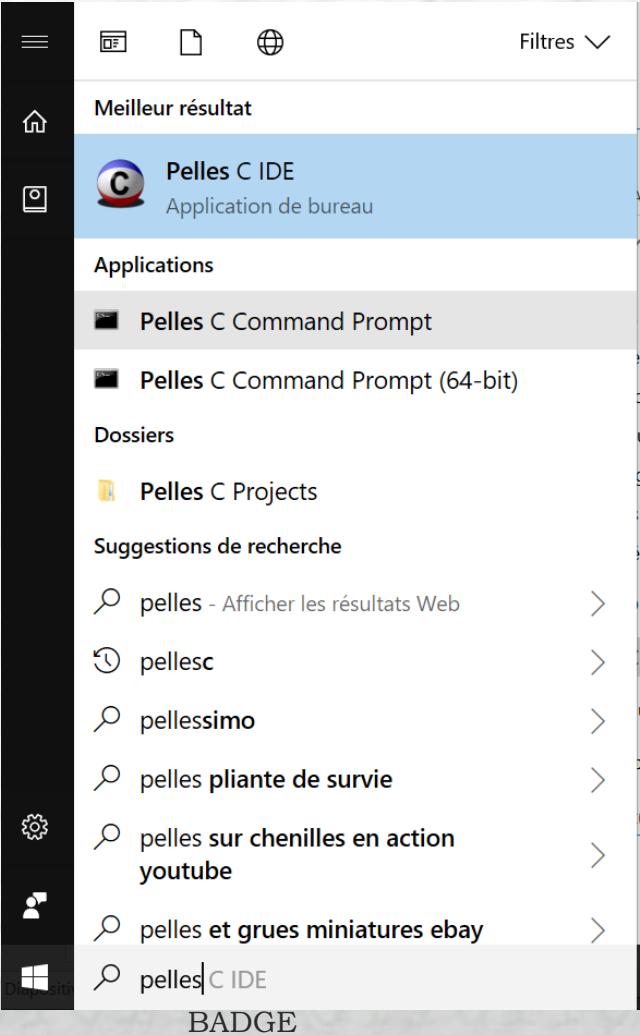
# Installation assembler / linker

- Créer les Lib (pour le moment répertoire vide)



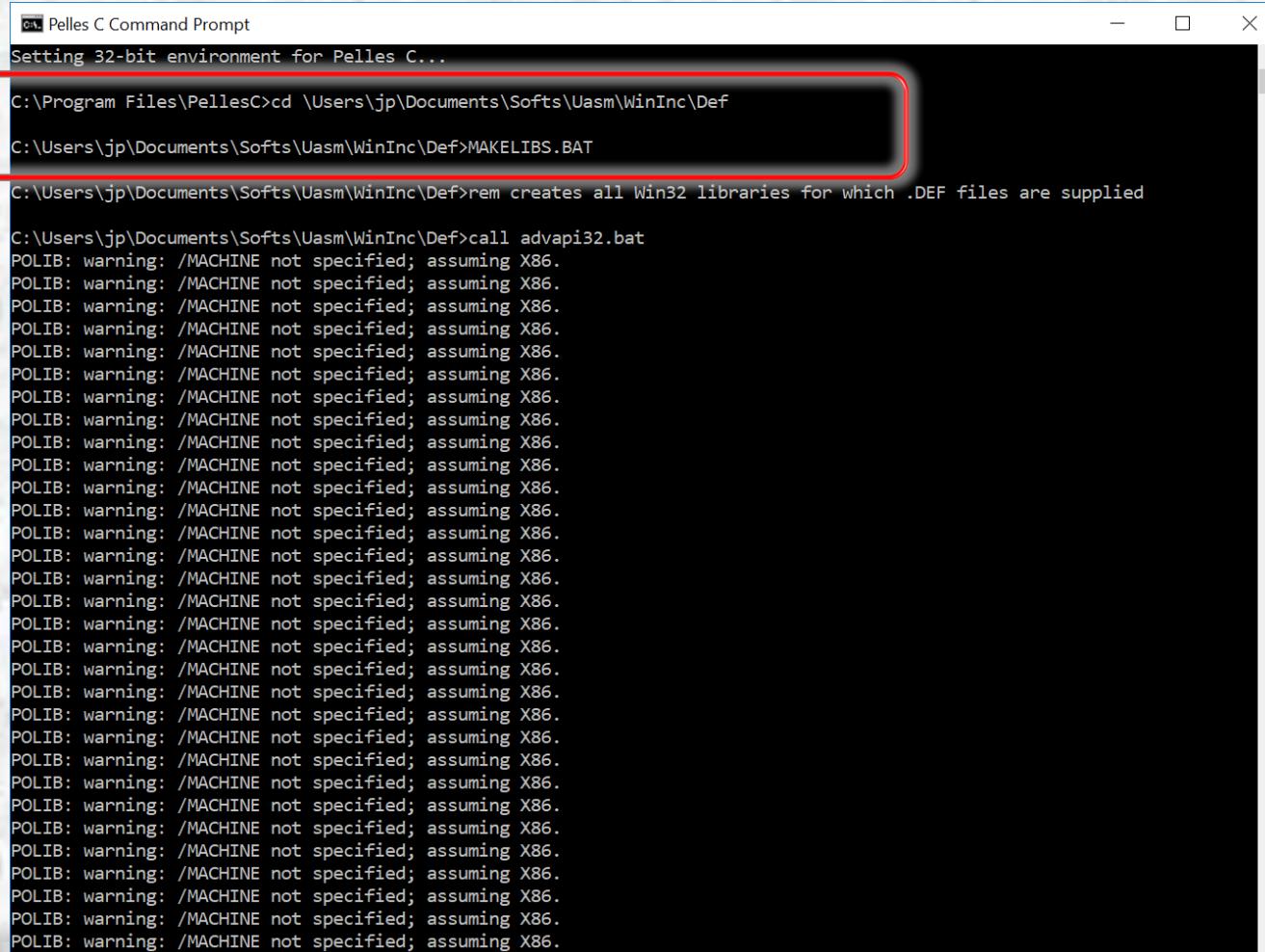
# Installation assembler / linker

- Lancer Pelles C Command Prompt



# Installation assembler / linker

- Lancer MAKELIBS.BAT



```
Pelles C Command Prompt
Setting 32-bit environment for Pelles C...
C:\Program Files\PellesC>cd \Users\jp\Documents\Softs\Uasm\WinInc\Def
C:\Users\jp\Documents\Softs\Uasm\WinInc\Def>MAKELIBS.BAT

C:\Users\jp\Documents\Softs\Uasm\WinInc\Def>rem creates all Win32 libraries for which .DEF files are supplied

C:\Users\jp\Documents\Softs\Uasm\WinInc\Def>call advapi32.bat
POLIB: warning: /MACHINE not specified; assuming X86.
```

BADGE

# Installation assembler / linker

- Créer les Lib

Nom	Modifié le	Type	Taille
ADVAPI32.LIB	15/12/2017 13:45	Fichier LIB	74 Ko
COMCTL32.LIB	15/12/2017 13:45	Fichier LIB	22 Ko
COMDLG32.LIB	15/12/2017 13:45	Fichier LIB	7 Ko
CRTDLL.LIB	15/12/2017 13:45	Fichier LIB	81 Ko
DBGENG.LIB	15/12/2017 13:45	Fichier LIB	2 Ko
DBGHELP.LIB	15/12/2017 13:45	Fichier LIB	23 Ko
DDRAW.LIB	15/12/2017 13:45	Fichier LIB	3 Ko
DSOUND.LIB	15/12/2017 13:45	Fichier LIB	4 Ko
GDI32.LIB	15/12/2017 13:45	Fichier LIB	73 Ko
GDIPLUS.LIB	15/12/2017 13:45	Fichier LIB	157 Ko
GLU32.LIB	15/12/2017 13:45	Fichier LIB	13 Ko
IMAGEHLP.LIB	15/12/2017 13:45	Fichier LIB	26 Ko
KERNEL32.LIB	15/12/2017 13:45	Fichier LIB	179 Ko
MPR.LIB	15/12/2017 13:45	Fichier LIB	12 Ko
MSVCRT.LIB	15/12/2017 13:45	Fichier LIB	123 Ko
MSWSOCK.LIB	15/12/2017 13:45	Fichier LIB	7 Ko

# Exemple 1

The image shows two windows side-by-side. On the left is the Pelles C Command Prompt window, and on the right is the Uasm editor window.

**Pelles C Command Prompt (Left Window):**

```
C:\Users\jp\Documents\MSIS>compile.bat empty.asm
C:\Users\jp\Documents\MSIS>"C:\Users\jp\Documents\Softs\Uasm\uasm64.exe" -I"C:\Users\jp\Documents\Softs\Uasm\WinInc\Include" -coff empty.asm
UASM v2.46, Dec 14 2017, Masm-compatible assembler.
Portions Copyright (c) 1992-2002 Sybase, Inc. All Rights Reserved.

Source code is available under the Sybase Open Watcom Public License.

empty.asm: 15 lines, 2 passes, 1 ms, 0 warnings, 0 errors

C:\Users\jp\Documents\MSIS>POLINK /ENTRY:start@0 /SUBSYSTEM:CONSOLE /LIBPATH:"C:\Users\jp\Documents\Softs\Uasm\WinInc\lib" empty.obj msvcrt.lib kernel32.lib -OUT:empty.exe

C:\Users\jp\Documents\MSIS>empty.exe
ExitCode: 42

C:\Users\jp\Documents\MSIS>
```

**Uasm Editor (Right Window):**

```
empty.asm
1 .386
2 .model FLAT, stdcall
3 option casemap:none
4
5
6 .code
7
8 start proc
9
10    mov eax, 42
11    ret
12
13 start endp
14
15 end
16
```

The assembly code in the editor includes a `mov eax, 42` instruction, which corresponds to the exit code 42 mentioned in the command prompt output.

# Exemple 2

```
C:\Users\jp\Documents\MSIS>compile.bat hello_world.asm
C:\Users\jp\Documents\MSIS>"C:\Users\jp\Documents\Softs\Uasm\Uasm64.exe" -I"C:\Users\jp\Documents\Softs\Uasm\WinInc\Include" -coff hello_world.asm
UASM v2.46, Dec 14 2017, Masm-compatible assembler.
Portions Copyright (c) 1992-2002 Sybase, Inc. All Rights Reserved.

Source code is available under the Sybase Open Watcom Public License.

hello_world.asm: 30 lines, 2 passes, 32 ms, 0 warnings, 0 errors

C:\Users\jp\Documents\MSIS>POLINK /ENTRY:start@0 /SUBSYSTEM:CONSOLE /LIBPATH:"C:\Users\jp\Documents\Softs\Uasm\WinInc\lib" hello_world.obj msvcrt.lib kernel32.lib -OUT:hello_world.exe

C:\Users\jp\Documents\MSIS>hello_world.exe
Hello World !
ExitCode: 42

C:\Users\jp\Documents\MSIS>
```

# TP 1

---

- Ecrire la programme factorial.asm en appelant une fonction recursive.

# TP 2

---

- Ecrire un programme qui parcours et affiche les fichiers d'un dossier recursivement (tree)

```
HANDLE WINAPI FindFirstFile(
    _In_     LPCTSTR             lpFileName,
    _Out_    LPWIN32_FIND_DATA  lpFindFileData ) ;

BOOL WINAPI FindNextFile(
    _In_     HANDLE              hFindFile,
    _Out_    LPWIN32_FIND_DATA  lpFindFileData
) ;

BOOL WINAPI FindClose(
    _Inout_   HANDLE             hFindFile
) ;
```

# TP 2

[https://msdn.microsoft.com/en-us/library/windows/desktop/aa365200\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365200(v=vs.85).aspx)

```
// Find the first file in the directory.

hFind = FindFirstFile(szDir, &ffd);

if (INVALID_HANDLE_VALUE == hFind)
{
    DisplayErrorBox(TEXT("FindFirstFile"));
    return dwError;
}

// List all the files in the directory with some info about them.

do
{
    if (ffd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
    {
        _tprintf(TEXT(" %s <DIR>\n"), ffd.cFileName);
    }
    else
    {
        filesize.LowPart = ffd.nFileSizeLow;
        filesize.HighPart = ffd.nFileSizeHigh;
        _tprintf(TEXT(" %s %ld bytes\n"), ffd.cFileName, filesize.QuadPart);
    }
} while (FindNextFile(hFind, &ffd) != 0);

dwError = GetLastError();
if (dwError != ERROR_NO_MORE_FILES)
{
    DisplayErrorBox(TEXT("FindFirstFile"));
}

FindClose(hFind);
```