# Virologie
# 1 - Prérequis : x86

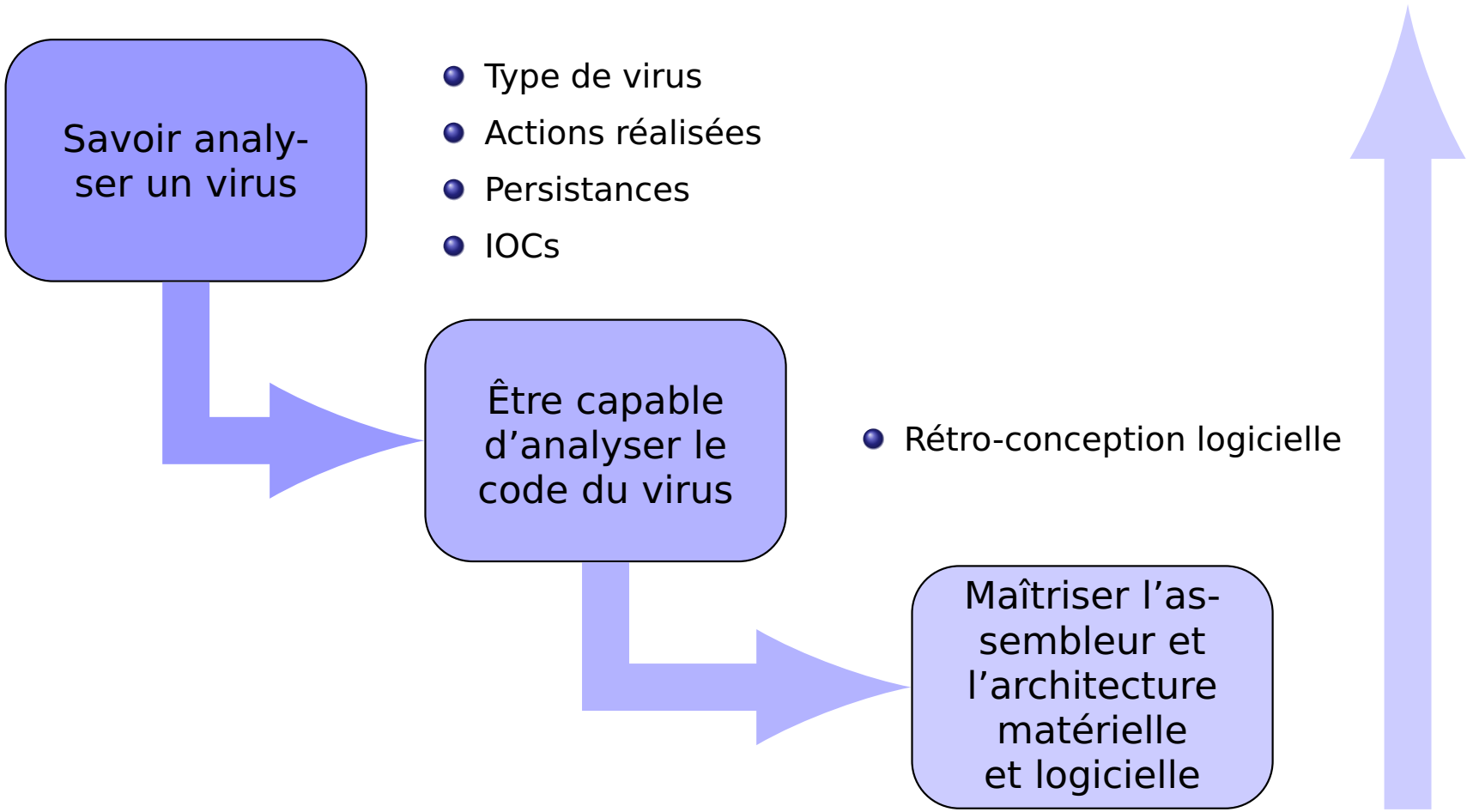
Jérémy Rubert

rubert.jeremy@gmail.com

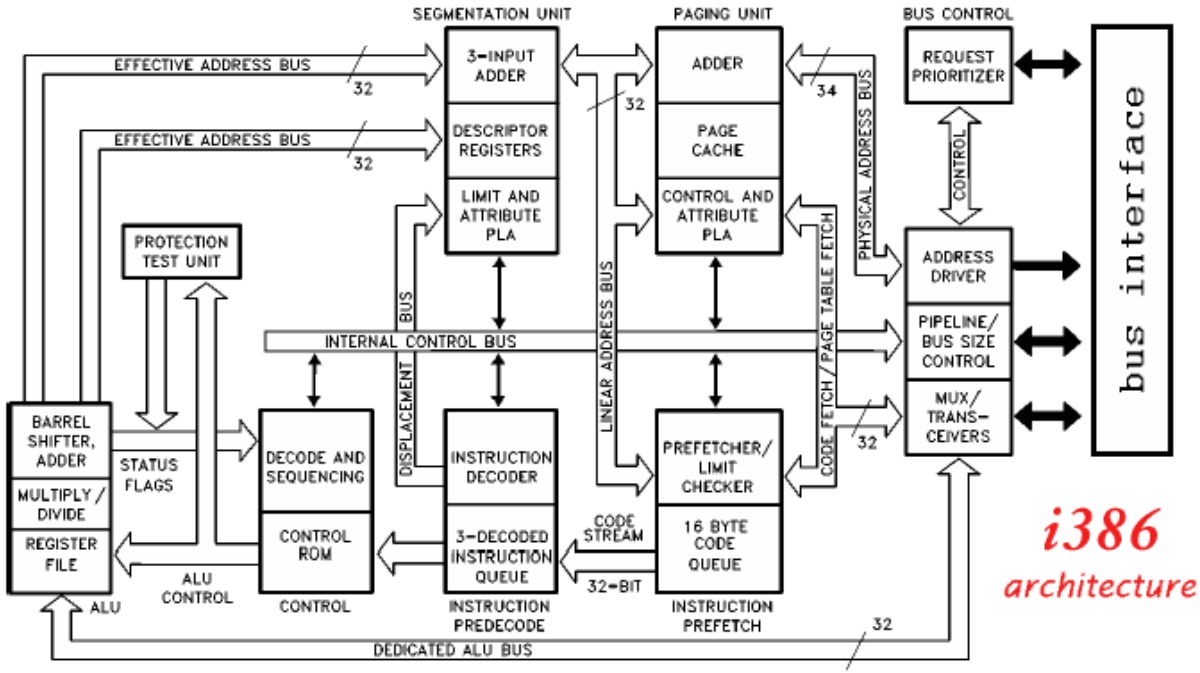
20 octobre 2025

# Plan

# Introduction



# Architecture x86



# Plan

# Présentation des Processeurs

Tout processeur (**CPU**[1]) comporte au moins les éléments suivants :

- une **unité arithmétique et logique** (**ALU**[2]) ;
- une **unité de contrôle**, appelée aussi séquenceur qui gère notamment les interruptions ;
- une **horloge** permettant de synchroniser les actions du CPU ;
- des **registres** permettant à l'ALU de manipuler leur contenu à chaque cycle d'horloge
- une **unité d'E/S**[3]

Un processeur possède 3 types de bus :

- un bus d'**adresses** qui fournit l'adresse visée par le CPU (unidirectionnel, du CPU vers le périphérique) ;
- un bus de **contrôle** qui spécifie le sens du transfert sur le bus de données (lecture/écriture).
- un bus de **données** sur lequel transite les données entre le CPU et ses périphériques (bidirectionnel) ;

1. **C**entral **P**rocessor **U**nit
2. **A**rithmetic and **L**ogical **U**nit
3. **E**ntrée/**S**orties

# Types de processeurs

Les CPUs se classent en deux catégories :

- les **RISC**[4] à taille d'encodage fixe (généralement 2 ou 4 octets)

**Exemple de RISC : "Hello World" en ARM**

```
10 40 2D E9 STMFD SP!, {R4,LR}
1E 0E 8F E2 ADR   R0, aHelloWorld ; "hello, world"
15 19 00 EB BL    __2printf
00 00 A0 E3 MOV   R0, #0
10 80 BD E8 LDMFD SP!, {R4,PC}
```

- les **CISC**[5] à taille d'encodage variable (de 1 à 15 octets pour le x86)

**Exemple de CISC : "Hello World" en x86 64-bits**

```
48 83 EC 08    sub  rsp, 8
BF E8 05 40 00 mov  edi, offset format ; "hello, world"
31 C0          xor  eax, eax
E8 D8 FE FF FF call _printf
31 C0          xor  eax, eax
48 83 C4 08    add  rsp, 8
C3             retn
```

4. **R**educed **I**nstruction **S**et **C**omputing
5. **C**omplex **I**nstruction **S**et **C**omputing

# Types de processeurs

- **CISC** : principalement x86 d'Intel et 68000 de Motorola
- **RISC** : ARM, MISC, SPARC, PowerPC

## Avantages du CISC

- empreinte mémoire du code beaucoup plus dense, permet de minimiser la taille du cache instruction
- possibilité de microprogrammation -> correction du jeu d'instructions
- instructions complexes très rapides (ex : instructions **SIMD**[a])
- polyvalent

a. **S**ingle **I**nstructions **M**ultiple **D**ata

## Inconvénients du CISC

- processeur compliqué à accélérer
- processeur plus complexe qu'un RISC
- difficile d'optimiser les instructions pour les compilateurs (emploi des instructions complexes)
- plus gourmand en énergie

# Les registres en x86

- 8 registres **GPR**[6] de 32 bits : EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP ;
- 8 registres **FPU**[7] de 80 bits : ST(0) à ST(7) ;
- 8 registres **MMX**[8] de 64 bits : MM0 à MM7 ;
- 8 registres **XMM** de 128 bits (la technologie **SSE**[9]) : XMM0 à XMM7 ;
- 4 registres de **contrôle** de 32 bits : CR0, CR2, CR3, CR4 ;
- 6 registres de **debug** de 32 bits : DR0, DR1, DR2, DR3, DR6 et DR7 ;
- 6 registres de **segment** de 16 bits : CS, DS, ES, SS, FS et GS ;
- le registre **EFLAGS** de 32 bits décrit plus loin ;
- le registre **EIP**[10] de 32 bits qui contient l'adresse de la prochaine instruction à exécuter.

6. **G**eneral **P**urpose **R**egister
7. **F**loating **P**oint **U**nit
8. **M**ulti**M**edia e**X**tensions
9. **S**treaming **S**IMD **E**xtensions
10. **I**nstruction **P**ointer

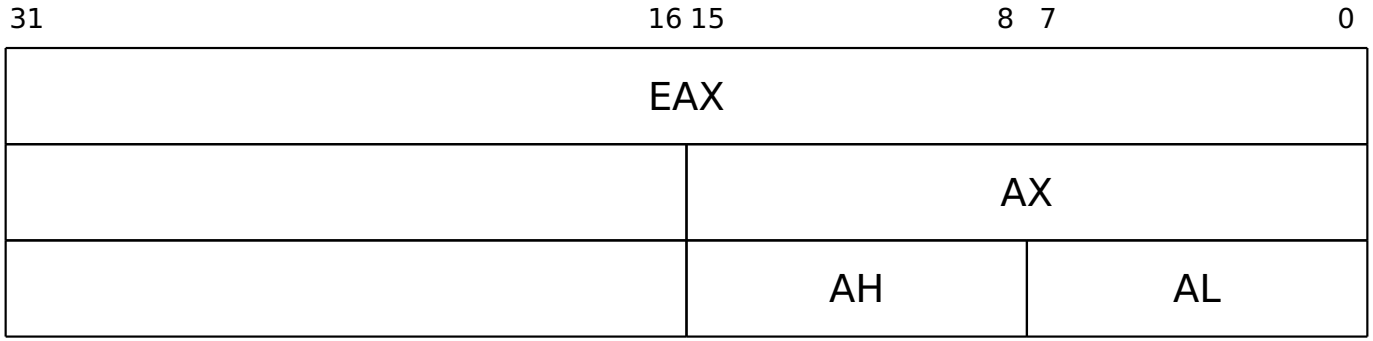
# Les registres généraux

La moitié des **GPR**s de 32-bits se décompose en sous-registres de 16 et 8-bits :

- EAX, AX, AH et AL : *Accumulator* ;
- EBX, BX, BH et BL : *Base* ;
- ECX, CX, CH et CL : *Counter* ;
- EDX, DX, DH et DL : *Data*.

L'autre moitié ne comporte qu'une sous partie de 16-bits :

- ESI et SI : *Source* ;
- EDI et DI : *Destination* ;
- ESP et SP : *Stack Pointer* ;
- EBP et BP : *Base Pointer*.



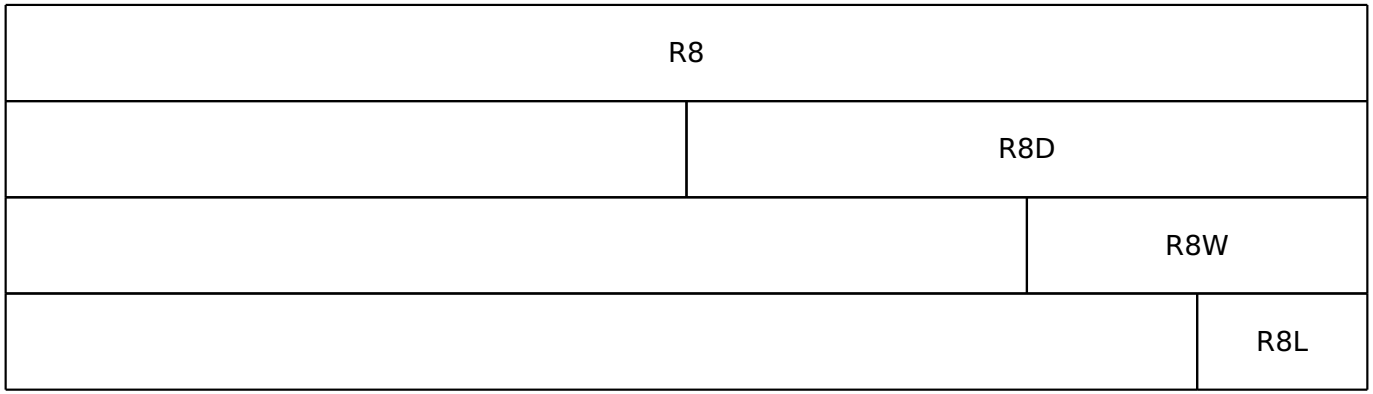
# Les registres généraux en x86-64

Les registres généraux sont étendus de 32 bits à 64 (préfixe R), exemple :

- RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP et RIP

8 nouveaux registres sont ajoutés de R8 à R15. Ces registres se décomposent en sous-registres de 32, 16 et 8-bits, par exemple :

- 64 bits : R8
- 32 bits : R8D
- 16 bits : R8W
- 8 bits : R8L




Remarque : les registres AH, BH, CH et DH n'ont pas d'équivalents en 64 bits.

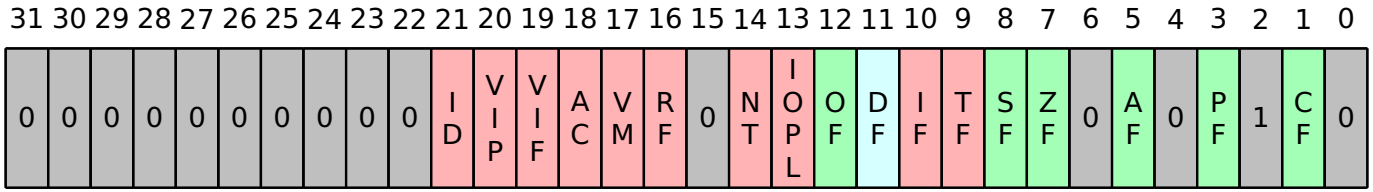
# Les registres généraux en x86-64

| 64-bits | 32-bits | 16-bits | 8 upper bits | 8 lower-bits |
|---|---|---|---|---|
| rax | eax | ax | ah | al |
| rbx | ebx | bx | bh | bl |
| rcx | ecx | cx | ch | cl |
| rdx | edx | dx | dh | dl |
| rsi | esi | si | | sil |
| rdi | edi | di | | dil |
| rbp | ebp | bp | | bpl |
| rsp | esp | sp | | spl |
| r8 | r8d | r8w | | r8l |
| r9 | r9d | r9w | | r9l |
| r10 | r10d | r10w | | r10l |
| r11 | r11d | r11w | | r11l |
| r12 | r12d | r12w | | r12l |
| r13 | r13d | r13w | | r13l |
| r14 | r14d | r14w | | r14l |
| r15 | r15d | r15w | | r15l |

Remarque : les registres en rouge ne sont accessibles qu'en 64-bits.

# Le registre EFLAGS

Ce registre de 32-bits comporte des drapeaux (*flags*) de status , de contrôle et du système .




Les principaux flags à connaître sont :

- le *Direction Flag* (**DF**), permet d'orienter les opérations sur les chaînes de caractères ;
- le *Sign Flag* (**SF**), correspond au bit de poids fort (MSB[11]) du résultat ;
- le *Zero Flag* (**ZF**), positionné si le résultat est nul ;
- le *Carry Flag* (**CF**), positionné si le résultat génère une retenue ;
- l'*OverFlow flag* (**OF**), positionné si le résultat est de signe différent des deux autres valeurs.

Remarque : en 64-bits, le registre eflags est étendu à 64-bits (registre RFLAGS) mais les bits de poids forts sont réservés.

11. **M**ost **S**ignificant **B**it

# Le registre EFLAGS

## Différence entre CF et OF

1. Le *Carry Flag* (CF) est positionné si une opération sur de 2 nombres provoque un retenue au-delà du bit de poids fort et n'est pas positionné sinon ;

   exemple :

   ```
   1111 + 0001 = 0000 (CF=1)
   0000 - 0001 = 1111 (CF=1)
   ```

   En **arithétique non-signée, CF permet de détecter des erreurs**.

2. L'*Overflow Flag* (OF) est positionné si une opération sur 2 nombres de même signe (bit de poids fort) donne un résultat de signe contraire et n'est pas positionné sinon ;

   exemple :

   ```
   0100 + 0100 = 1000 (OF=1)
   1000 + 1000 = 0000 (OF=1)
   ```

   En **arithmétique signée, OF permet de détecter des erreurs**.

# Types de bases

Les principaux types de base gérés par le x86 sont :

- le *Byte* (**DB**), 8 bits soit 1 octet, c'est la taille de cellule de base du x86. Il s'agit du type le plus utilisé en informatique ;
- le *Word* (**DW**), 16 bits soit 2 octets (ne pas confondre avec Double Word qui suit) ;
- le *Double Word* (**DD**), 32 bits soit 4 octets ;
- le *Quad Word* (**DQ**), 64 bits soit 8 octets.

# Représentations de la mémoire

Il existe 2 manières de représenter un groupe d'octets en mémoire, ce qu'on appelle l'«**endianness**» :

- la représentation dite «**Big Endian**», où l'écriture en mémoire commence par l'octet de poids fort ;
- la représentation dite «**Little Endian**» , où l'écriture en mémoire commence par l'octet de poids faible.

**Exemple**

- Considérons la valeur de 32 bits suivante : 0x08022012. La façon la plus naturelle de la représenter est la suivante :

| *adresses* | 0 | 1 | 2 | 3 | ... |
|---|---|---|---|---|---|
| *valeurs* | 08 | 02 | 20 | 12 | ... |

(«**Big Endian**»)

- Une autre façon de la représenter en mémoire :

| *adresses* | 0 | 1 | 2 | 3 | ... |
|---|---|---|---|---|---|
| *valeurs* | 12 | 20 | 02 | 08 | ... |

(«**Little Endian**»)

# Représentations de la mémoire

Avantages du **Big Endian**

- plus facile à lire pour la majorité des êtres humains ;
- facilité de test de signe ;
- facilité de comparaison d'entiers (à partir du MSB) ;
- pas besoin de conversion pour les données issues du réseau .

Avantages du **Little Endian**

- plus naturel pour les calculs (LSB to MSB) ;
- plus facile lors de la lecture avec registre à taille variable ;
- facilité pour "**caster**" des valeurs.

# Gestion de la mémoire

**Unité de gestion de mémoire (MMU)**

*La (**MMU**[a]) est un composant permettant de contrôler les accès qu'un processeur fait à la mémoire.*

a. **M**emory **M**anagement **U**nit

- Le CPU manipule des adresses **virtuelles** (ou encore **logiques**) de la forme `segment:offset;`
- Dans la mémoire, les adresse sont dites **physiques**.
- La MMU assure la translation d'adresses (virtuelles → physiques).

# Gestion de la mémoire : MMU en mode réel

**Définition**

*Le mode réel est le mode historique du x86. Il se caractérise par un espace d'adressage de 20 bits, ce qui permet d'adresser 1MB. Chaque segment correspond à une zone mémoire contigüe de 64K. Les offsets sont limités à 16 bits.*

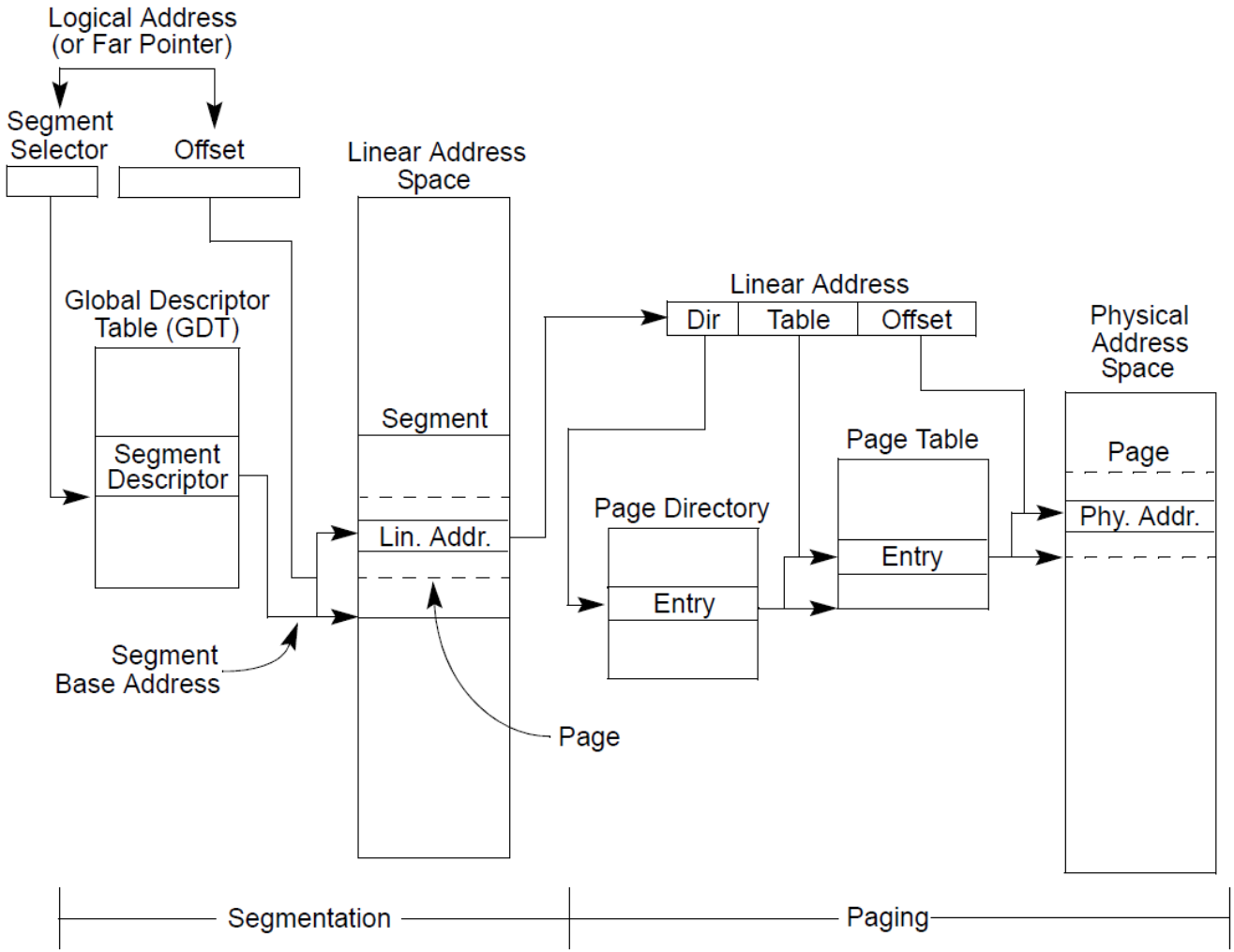
Le fonctionnement de la MMU en mode réel est très simple :

**Adresse Physique= Segment x 10h + Offset**

**Exemples d'adressage en mode réel**

*(Virtuelle)* 07C0 :0000 → 07C00 *(Physique)*
*(Virtuelle)* 02A0 :00C4 → 02AC4 *(Physique)*
*(Virtuelle)* 0000 :2AC4 → 02AC4 *(Physique)*

# Gestion de la mémoire : MMU en mode protégé



# La pile

**Définition**

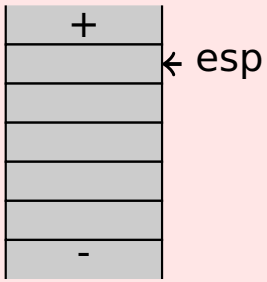
*La pile est une structure de donnée de type **LIFO**[a] permettant de stocker temporairement la valeur des registres (ou autres).*

a. **L**ast **I**n **F**irst **O**ut

- Le haut de la pile est toujours pointé par **ESP** (**RSP** en 64-bits) ;
- Nous verrons plus loin que des instructions dédiées servent à empiler (**PUSH**) et à dépiler (**POP**) des éléments ;
- La pile permet les appels récursifs de fonctions.

**Représentation**

Attention, il faut toujours orienter la pile lorsqu'on la représente.



# Assembleur x86

# Plan

# L'assembleur

**Définition**

*L'assembleur est la représentation humaine du langage machine.*

Les 3 lignes suivantes désignent la même instruction en x86 :

- 1000 1011 0100 0100 0010 0100 0011 0000 (*binaire*)
- 8B 44 24 30 (*hexadécimal*)
- **MOV EAX, [ESP+30h]** (*assembleur*)

**Exemple : ARM**

```
STMFD SP!, {R4,LR}
ADR   R0, aHelloWorld
BL    __2printf
MOV   R0, #0
LDMFD SP!, {R4,PC}
```

**Exemple : x86-64**

```
sub  rsp, 40
lea  rcx,aHelloWorld
call printf
xor  eax, eax
add  rsp, 40
ret  0
```

**Exemple : MIPS**

```
lui   gp, 0x42
addiu sp, sp, -32
addiu gp, gp, -30624
sw    ra, 28(sp)
sw    gp, 16(sp)
lw    t9, -32716(gp)
lui   a0, 0x40
jalr  t9
addiu a0, a0, 2080
lw    ra, 28(sp)
move  v0, zero
jr    ra
addiu sp, sp, 32
```

# Instruction ASM x86

Le format générique d'une instruction est le suivant :

**Instruction x86**

```
(prefixes) MNEMONIC (Operand1, Operand2, Operand3)
```

- Le nombre de préfixes n'est pas limité mais peut être nul ;
- Le mnémonique désigne le nom d'instruction (seul élément requis)
- Le nombre d'opérandes est limité à 3 et peut être nul en fonction du mnémonique.

**Remarque**

La limite d'encodage d'une instruction en x86 est de 15 octets. Au delà, une exception de type «UnDefined Opcode» #UD est levée.

# Instruction ASM : les préfixes

Les préfixes d'instructions se subdivisent comme suit :

- **lock** (F0h), force l'accès exclusif a une mémoire partagée en environnement multiprocesseur ;
- **repne/repe** (F2h/F3h), permet de répéter une instruction sur chaque élément d'une chaîne de caractère ;
- **segment override** : CS (2Eh), SS (36h), DS (3Eh), ES (26h), FS (64h) et GS (65h) ;
- **operand-size override** (66h), permet de permuter la taille de l'opérande (16 ou 32-bits) ;
- **address-size override** (67h), permet de permuter le mode d'adressage (16 ou 32-bits) ;
- **REX**, utilisé pour accéder aux extension 64-bits (GPRs, taille d'opérande 64-bits et CRs).

# Instruction ASM : les opérandes

Les opérandes peuvent être de 3 types :

- **registre** ;
- **valeur immédiate** (entier naturel ou flottant) ;
- **référence mémoire**.

## Format des références mémoire

seg :[reg1 + reg2 * scale + immediate]

- *seg* désigne un registre de segment ;
- *reg1* et *reg2* désignent des registres ;
- *scale* désigne un facteur d'échelle pouvant être 0,1,2,4 ou 8 ;
- *immediate* désigne une valeur de 8, 16 ou 32 bits.

## Exemples

```
mov eax, es:[esi*4+405000h]
mov edx, fs:[30h]
```

# Instructions de base

Le x86 est riche en instructions. Nous allons voir les instructions les plus utilisées, regroupées par catégorie :

- **affectation**
- **arithmétiques**
- **logique**
- **transfert**
- **diverses**

**Remarque**

Les compilateurs utilisent un jeu restreint d'instructions ;-)

# Instructions d'affectation

**MOV - Move**

*Copie l'opérande source (SrcOperand) vers l'opérande de destination (DestOperand)*

**MOV *DestOperand*, *SrcOperand***

*Pseudo-code :* DestOperand ← SrcOperand

*Flags Affectés :* aucun

**Exemples**

```
mov esi, 401000h
mov eax, [ebx+esi*4]
mov edx, eax
```

# Instructions d'affectation

**LEA - Load Effective Address**

*Calcule l'adresse effective du second opérande (SrcOperand) et le stocke dans l'opérande de destination (DestOperand)*

**LEA *DestOperand*, *SrcOperand***

*Pseudo-code :* DestOperand ← EffectiveAddress(SrcOperand)

*Flags Affectés :* aucun

**Exemples**

```
lea esi, [eax]
lea eax, [ebx+4*esi]
lea edx, [edx+2*edx]
```

# Instructions arithmétiques

**INC - Increment by 1**

*Incrémente de 1 l'opérande de destination tout en préservant le Carry Flag (CF)*

**INC *DestOperand***

*Pseudo-code :* DestOperand ← DestOperand + 1

*Flags Affectés :* OF, SF, ZF, AF et PF

**Exemples**

```
inc eax
inc byte ptr [edx]
lock inc word ptr [ebx]
```

# Instructions arithmétiques

**DEC - Decrement by 1**

*Décrémente de 1 l'opérande de destination tout en préservant le Carry Flag (CF)*

**DEC *DestOperand***

*Pseudo-code :* DestOperand ← DestOperand - 1

*Flags Affectés :* OF, SF, ZF, AF et PF

**Exemples**

```
dec eax
dec dword ptr [edx]
lock dec byte ptr [ebx]
```

# Instructions arithmétiques

**ADD - Add**

*Ajoute l'opérande source à l'opérande de destination et stocke le résultat dans l'opérande de destination*

**ADD *DestOperand*, *SrcOperand***

*Pseudo-code :* DestOperand ← DestOperand + SrcOperand

*Flags Affectés :* OF, SF, ZF, AF, CF et PF

**Variante : ADC - Add with Carry**

Même opération que l'instruction **ADD** en ajoutant en plus le Carry Flag (**CF**).

*Pseudo-code :* DestOperand ← DestOperand + SrcOperand + CF

# Instructions arithmétiques

**SUB - Substract**

*Soustrait l'opérande source à l'opérande de destination et stocke le résultat dans l'opérande de destination*

**SUB *DestOperand*, *SrcOperand***

*Pseudo-code :* DestOperand ← DestOperand - SrcOperand

*Flags Affectés :* OF, SF, ZF, AF, CF et PF

**Variante : SBB - Substract with Carry**

Même opération que l'instruction **SUB** en soustrayant en plus le Carry Flag (**CF**).

*Pseudo-code :* DestOperand ← DestOperand - (SrcOperand + CF)

# Instructions arithmétiques

**MUL - Unsigned Multiply**

*Effectue une multiplication non-signée entre le registre EAX et l'opérande spécifié. Le résultat est stocké dans EDX :EAX*

**MUL *Operand***

*Pseudo-code :* EDX :EAX ← EAX * Operand

*Flags Affectés :* OF et CF

**Variante : IMUL - Signed Multiply**

Même opération que l'instruction **MUL** mais dans le cas signé.

# Instructions arithmétiques

**DIV - Unsigned Divide**

*Effectue une division non-signée entre EDX :EAX et l'opérande spécifié. Le quotient est stocké EAX, le reste dans EDX*

**DIV *Operand***

*Pseudo-code :*
EAX ← EDX :EAX / Operand
EDX ← EDX :EAX % Operand

*Flags Affectés :* OF et CF

**Variante : IDIV - Signed Divide**

Même opération que l'instruction **DIV** mais dans le cas signé.

# Instructions arithmétiques

**SHR - Shift Right**

*Effectue un décalage à droite de l'opérande destination du nombre de bits spécifié*

**SHR *DestOperand*, Value**

*Pseudo-code :*
DestOperand ← DestOperand » Value

*Flags Affectés :* OF, SF, ZF et PF

# Instructions arithmétiques

**SHL - Shift Left**

*Effectue un décalage à gauche de l'opérande destination du nombre de bits spécifié*

**SHL *DestOperand*, Value**

*Pseudo-code :*
DestOperand ← DestOperand « Value

*Flags Affectés :* OF, SF, ZF et PF

# Instructions logiques

**CMP - Compare Two Operands**

*Compare le premier opérande avec le second et positionne les flags en fonction.*

**CMP *Operand1, Operand2***

*Pseudo-code :* temp ← Operand1 - SignExtend(Operand2)

*Flags Affectés :* CF, OF, SF, ZF AF et PF

# Instructions logiques

**TEST - Logical Compare**

*Calcule le ET logique entre les deux opérandes et positionne les flags en fonction.*

**TEST *Operand1*, *Operand2***

*Pseudo-code :* temp ← Operand1 & Operand2

*Flags Affectés :* CF et OF sont à 0, SF, ZF et PF

**Examples**

```
test eax, eax    ; ZF set if eax is null
jz   Success
```

# Instructions logiques

**XOR - Logical Exclusive OR**

*Calcule le OU exclusif entre les deux opérandes et positionne les flags en fonction du résultat.*

**XOR *DestOperand*, *SrcOperand***

*Pseudo-code :* DestOperand ← DestOperand ⊕ SrcOperand

*Flags Affectés :* CF et OF sont à 0, SF, ZF et PF

**Examples**

```
xor ebx, ebx    ; reset ebx
xor esi, ecx    ; used in crypto operations
```

# Instructions logiques

**OR - Logical Inclusive OR**

*Calcule le OU logique entre les deux opérandes et positionne les flags en fonction du résultat.*

**OR *DestOperand*, *SrcOperand***

*Pseudo-code :* DestOperand ← DestOperand | SrcOperand2

*Flags Affectés :* CF et OF sont à 0, SF, ZF et PF

# Instructions logiques

## AND - Logical AND

*Calcule le ET logique entre les deux opérandes et positionne les flags en fonction du résultat.*

**AND *DestOperand*, *SrcOperand***

*Pseudo-code :* DestOperand ← DestOperand & SrcOperand

*Flags Affectés :* CF et OF sont à 0, SF, ZF et PF

## Examples

```
and esi, 0FFh          ; only the 8 lowest bits of esi are kept
```

# Instructions transfert inconditionnelles

**JMP - Jump**

*transfère le contrôle du programme au point désigné.*

**JMP *DestOperand***

*Pseudo-code :* EIP ← DestOperand

*Flags Affectés :* aucun

**Examples**

```
jmp 401000h
jmp eax
jmp dword ptr [500000h+esi*4]
```

# Instructions transfert inconditionnelles

**CALL - Call Procedure**

*transfère le contrôle du programme à la procédure désignée.*

**CALL *DestOperand***

*Pseudo-code :*
PUSH @Return
JMP DestOperand

*Flags Affectés :* aucun

**Examples**

```
call 401000h
call eax
call dword ptr [500000h+esi*4]
```

# Instructions transfert inconditionelles

**RET - Return from Procedure**

*Transfert le contrôle du programme à une adresse de retour stockée sur la pile.*

**RET *Immediate16***

*Pseudo-code :*
POP Tmp
ADD ESP, Immediate16
JMP Tmp

*Flags Affectés :* aucun

**Examples**

```
ret
ret 0Ch
```

# Instructions transfert conditionelles

**Jcc—Jump if Condition Is Met**

*Vérifie l'état du registre EFLAGS et, le cas échéant, saut à la destination spécifiée.*

**Jcc *Relative***

*Pseudo-code :* IF condition THEN EIP ← EIP + SignExtend(Relative)

*Flags Affectés :* aucun

# Instructions transfert conditionelles

| Unsigned Jcc | Condition | Description |
|---|---|---|
| JA/JNBE | (CF \| ZF) = 0 | Above/not below or equal |
| JAE/JNB | CF = 0 | Above or equal/not below |
| JB/JNAE | CF = 1 | Below/not above or equal |
| JBE/JNA | (CF \| ZF) = 1 | Below or equal/not above |
| JC | CF = 1 | Carry |
| JE/JZ | ZF = 1 | Equal/zero |
| JNC | CF = 0 | Not carry |
| JNE/JNZ | ZF = 0 | Not equal/not zero |
| JNP/JPO | PF = 0 | Not parity/parity odd |
| JP/JPE | PF = 1 | Parity/parity even |
| JCXZ | CX = 0 | Register CX is zero |
| JECXZ | ECX = 0 | Register ECX is zero |

| Signed Jcc | Condition | Description |
|---|---|---|
| JG/JNLE | ((SF $\oplus$ OF) \| ZF) = 0 | Greater/not less or equal |
| JGE/JNL | (SF $\oplus$ OF) = 0 | Greater or equal/not less |
| JL/JNGE | (SF $\oplus$ OF) = 1 | Less/not greater or equal |
| JLE/JNG | ((SF $\oplus$ OF) \| ZF) = 1 | Less or equal/not greater |
| JNO | OF = 0 | Not overflow |
| JNS | SF = 0 | Not sign (non-negative) |
| JO | OF = 1 | Overflow |
| JS | SF = 1 | Sign (negative) |

# Instructions diverses

**PUSH - Push Onto the Stack**

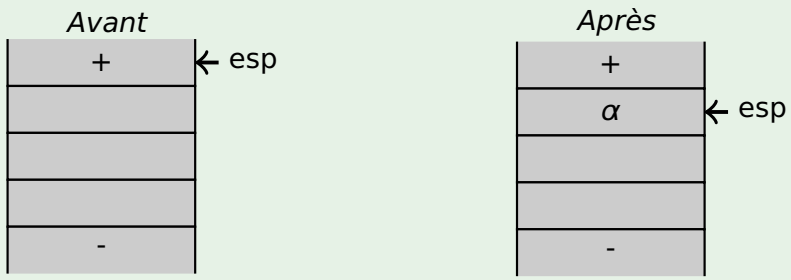
*Décrémente le pointeur de pile et stocke l'opérande sur le haut de la pile.*

**PUSH *Operand***

*Pseudo-code :*
ESP ← ESP - 4

*Flags Affectés :* aucun

**Example : PUSH $\alpha$**



# Instructions diverses

**POP - Pop a Value from the Stack**

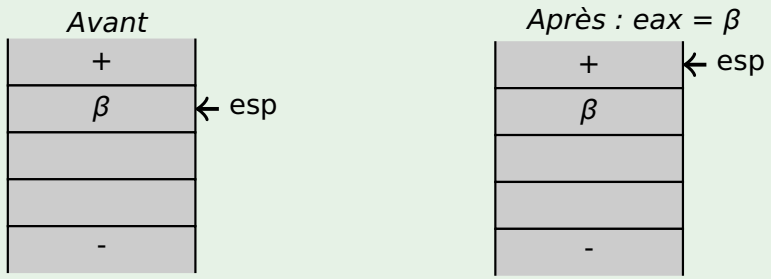
*Charge dans l'opérande spécifié la valeur contenue sur le haut de la pile puis incrémente le pointeur de pile.*

## POP *Operand*

*Pseudo-code :*
Operand ← ESP ← ESP + 4

*Flags Affectés :* aucun

**Example : POP eax**



# Les procédures

**Définition**

*Une procédure (ou routine) est un sous-programme réalisant une action. Il s'agit d'une boîte noire pour les programmes qui l'emploient.*

Chaque procédure possède sa signature propre appelée un prototype composé de :

- un type de retour (`void` pour les procédure et tout autre type pour les fonctions) ;
- une convention d'appel ;
- un nom de symbole (nom explicite en cas d'appel) ;
- d'éventuels arguments.

**Exemple en langage C**

```
int __fastcall function1(long value1, int value2);
```

# Conventions d'appel

**Définition**

*Une convention d'appel définit un contrat d'invocation d'une fonction, c'est-à-dire comment une fonction (dite appelante) passe les informations d'appel à une fonction(dite appelée) et comment elle les récupère.*

Une convention d'appel définit :

- Comment sont passés les paramètres (registre/pile) ;
- Ordre de passage des paramètres sur la pile ;
- Comment sont récupérées les valeurs de retour ;
- Registres sauvegardés et utilisés ;
- Correction de pile en cas de passage de paramètres par la pile.

# Conventions d'appel

## Exemples : MSVC Win32

```
int XXXXX funtion(int A, int B, int C, int D);
```

| | C (*__cdecl*) | standard (*__stdcall*) | fast (*__fastcall*) |
|---|---|---|---|
| correction de la pile | appelant | appelé | |
| passage des paramètres | pile en ordre inverse | | ecx, edx, et inverse |
| valeur de retour | edx :eax | | |
| registres sauvegardés | ebp, ebx, esi, edi | | |
| code appelant | push D<br>push C<br>push B<br>push A<br>call function1<br>add esp, 10h<br>... | push D<br>push C<br>push B<br>push A<br>call function 1<br>... | mov ecx, A<br>mov edx, B<br>push D<br>push C<br>call function1<br>... |
| code appelé | ...<br>ret | ...<br>ret 10h | ...<br>ret 8 |

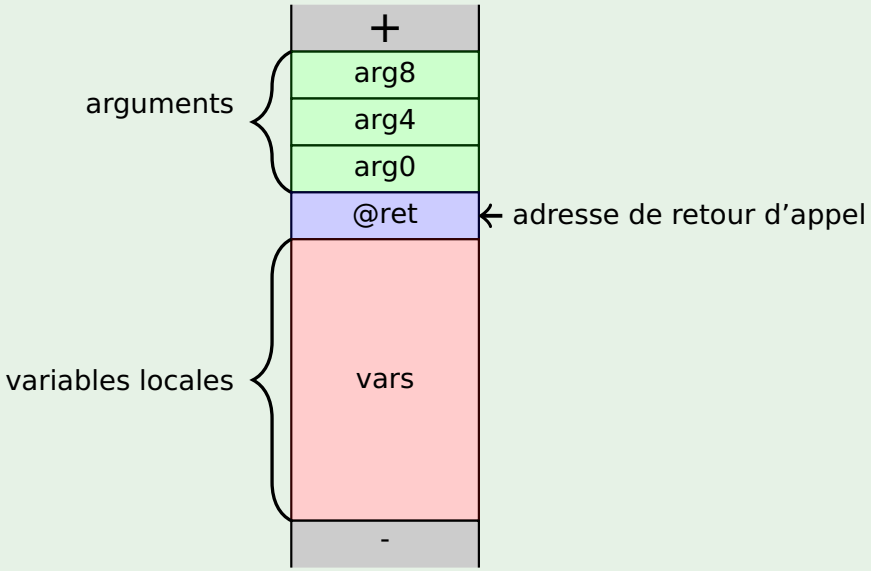
# Le cadre de pile

## Définition

*Le cadre de pile d'une fonction (stack frame) correspond à sa vue locale, composée :*

- des arguments (passés par la pile) ;
- de l'adresse de retour (adresse juste après l'appel de fonction) ;
- des variables locales.

## Représentation



# Prologue et épilogue de fonctions

Un registre particulier peut être utilisé comme base de pile en entrée de fonction : le frame pointer. Il permet de faciliter l'accès aux paramètres et variables locales. Par convention le registre utilisé est **ebp**.

## Prologue

```
push ebp
mov  ebp,esp
```

ou encore,

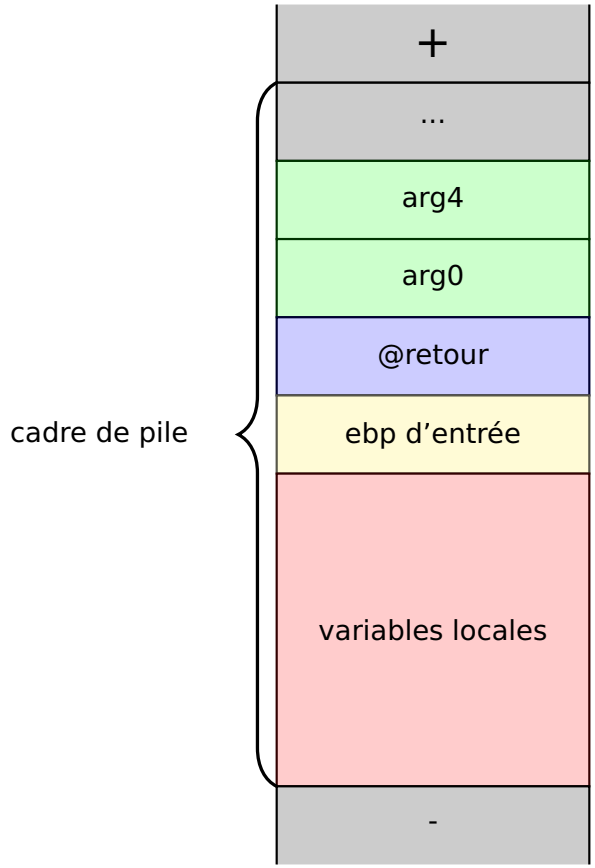
```
enter
```

## Épilogue

```
mov esp,ebp
pop ebp
```

ou encore,

```
leave
```



# Application

## Exemple : calcul de factorielle(4)

```
 main:
  ...
  push  4
  call  factorielle
print:
  push  eax
  push  offset Format
  call  printf
  ...
```

```
 factorielle:
  push  ebp
  mov   ebp, esp
  xor   eax, eax
  push  esi
  mov   esi, [ebp+8]
  inc   eax
  cmp   esi, eax
  jbe   short end
  lea   eax, [esi-1]
  push  eax
  call  factorielle
mul:
  imul  eax, esi
end:
  pop  esi
  pop  ebp
  retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

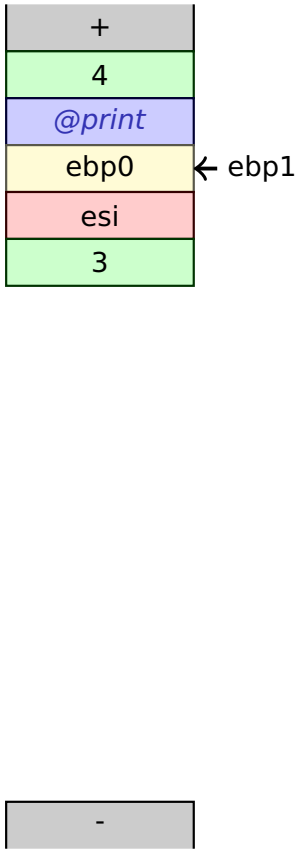
```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

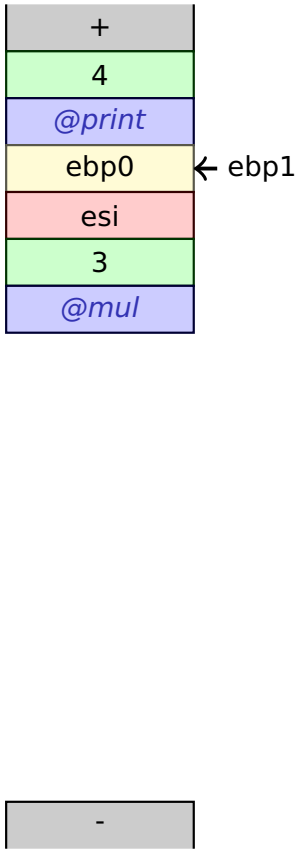


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

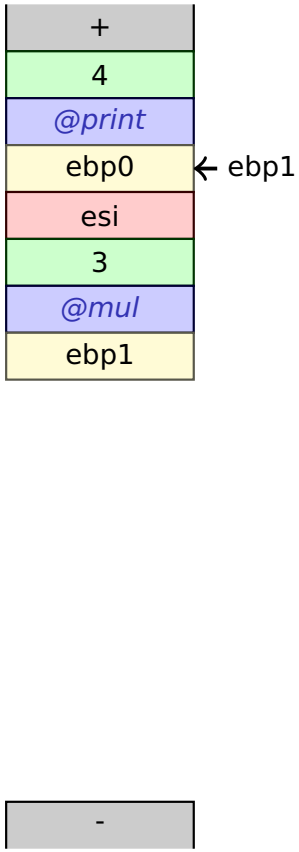


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

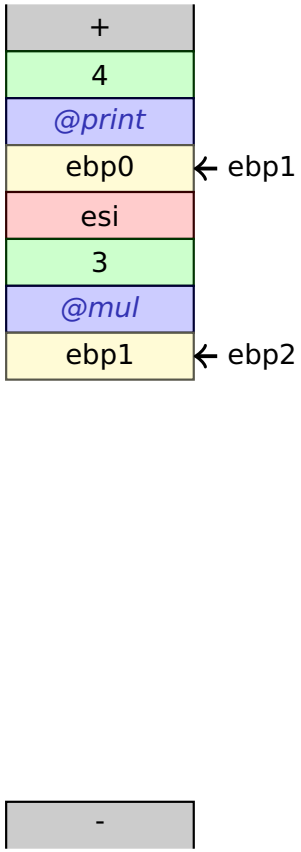


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

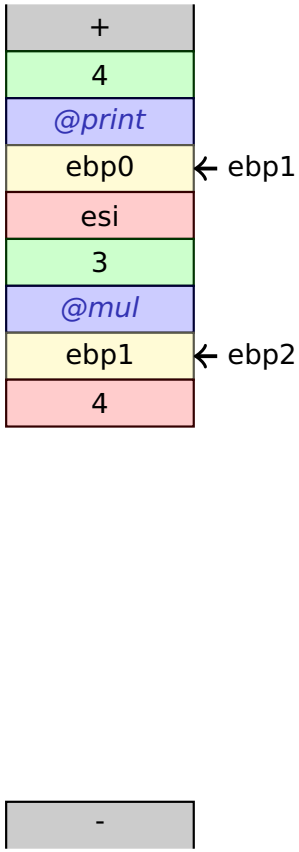


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

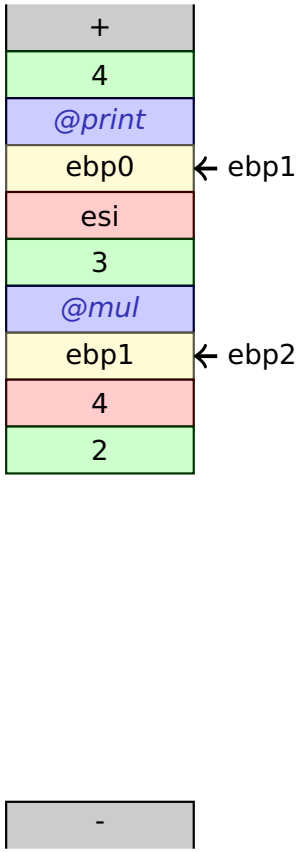


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

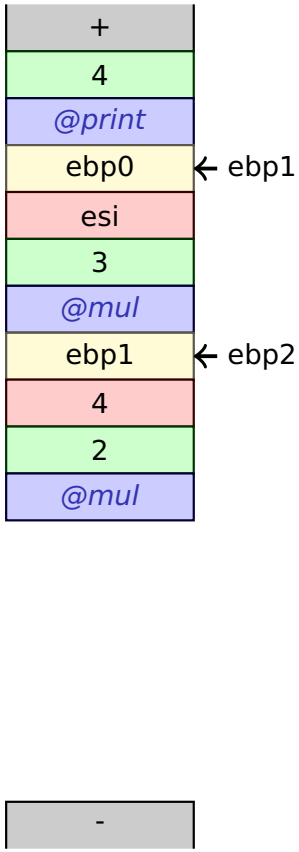


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

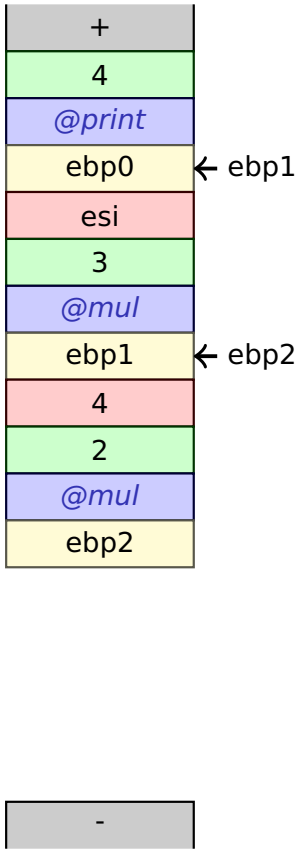


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

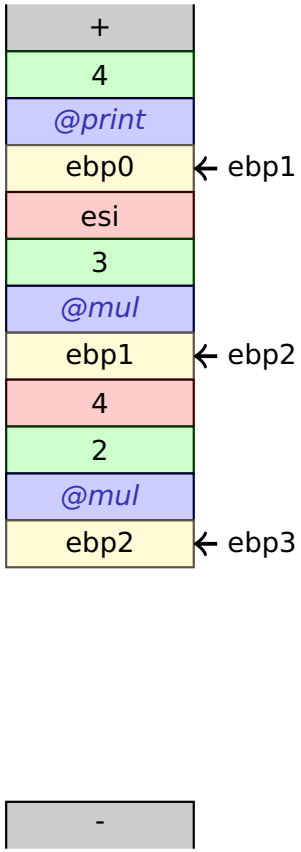


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

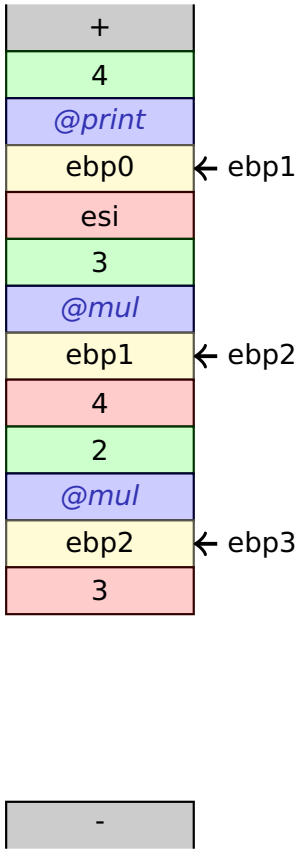


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

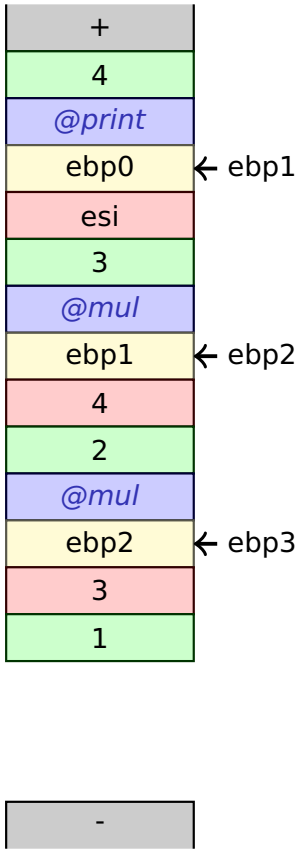


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

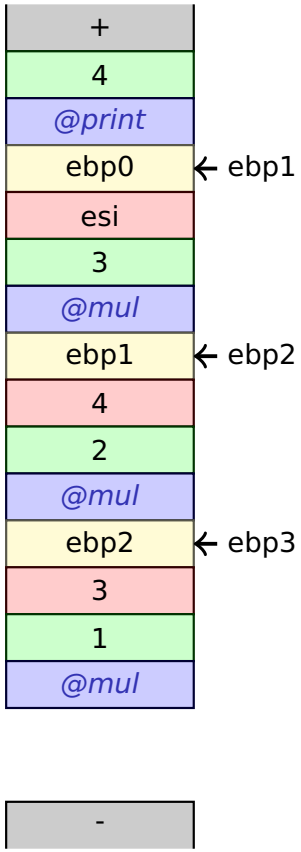


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

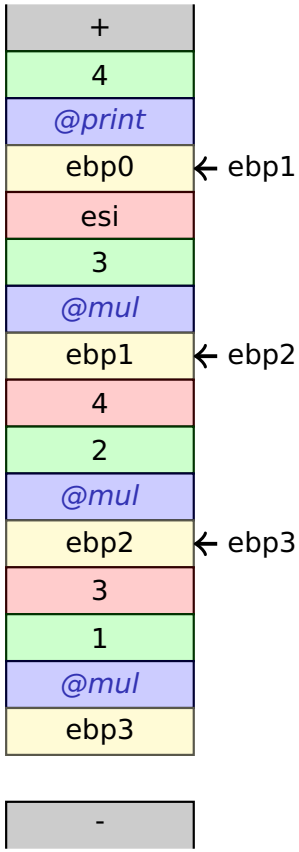


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

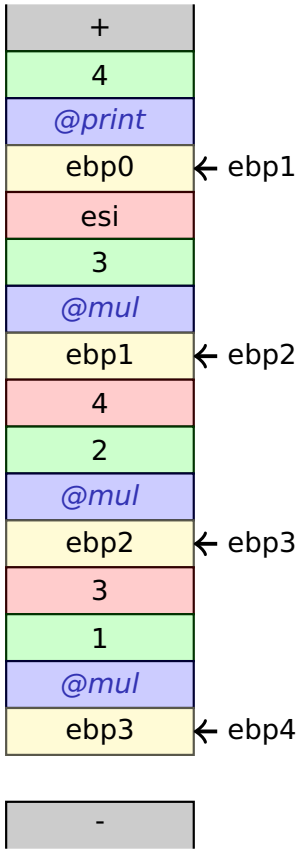


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

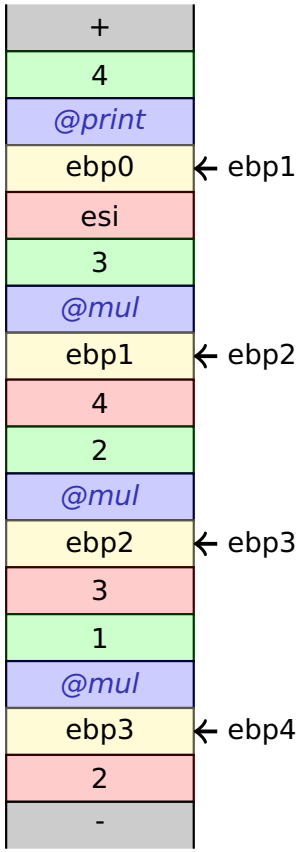


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

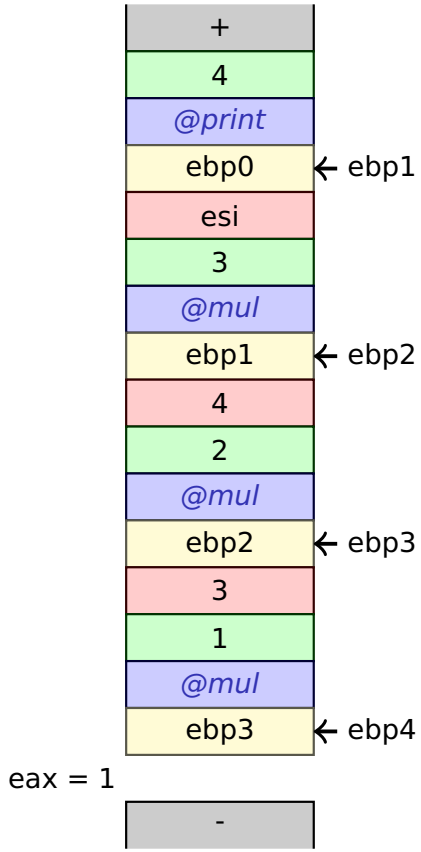


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop   esi
   pop   ebp
   retn 4
```

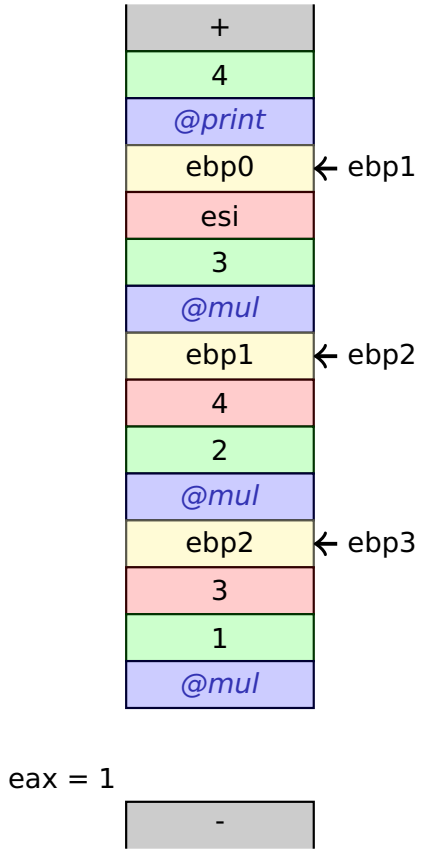


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

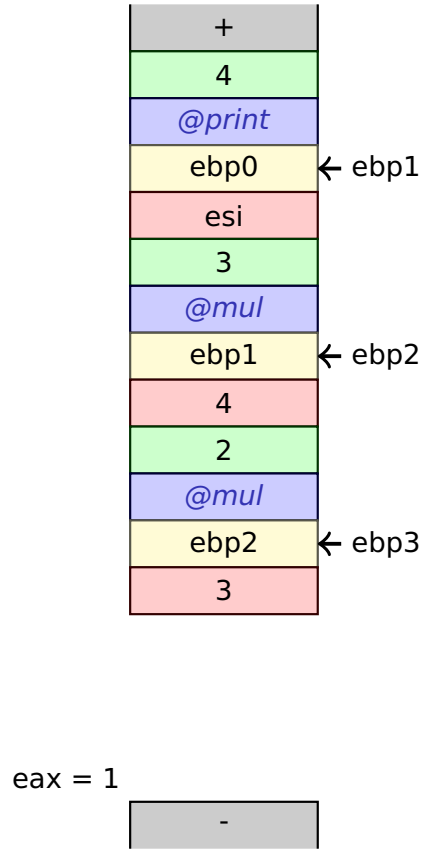


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

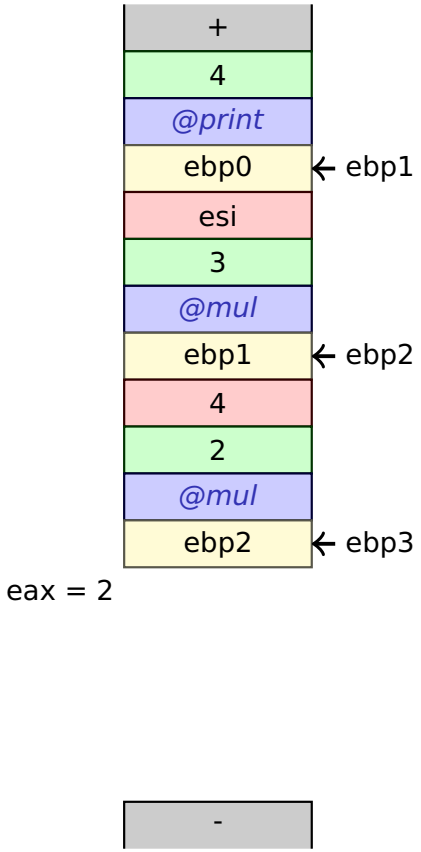


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop   esi
   pop   ebp
   retn 4
```

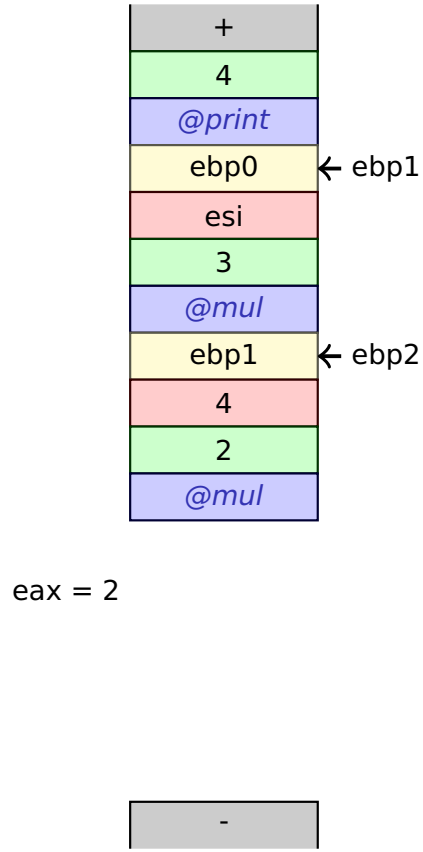


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

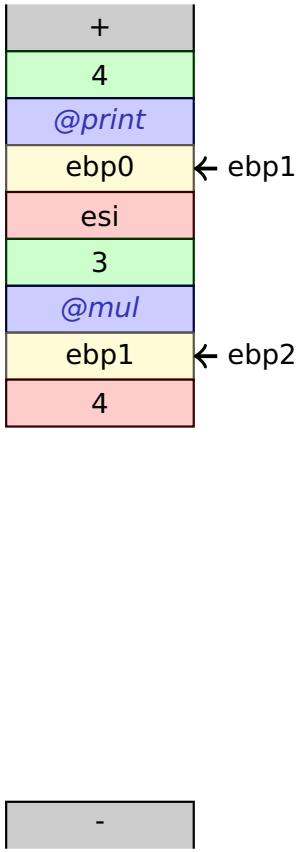


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

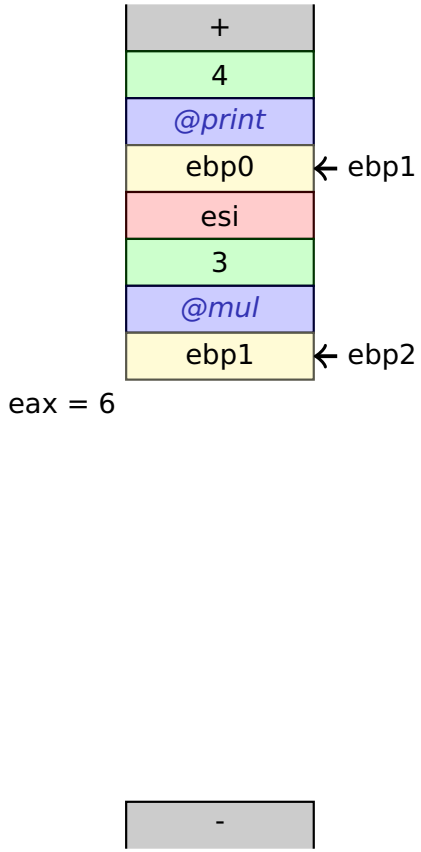


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

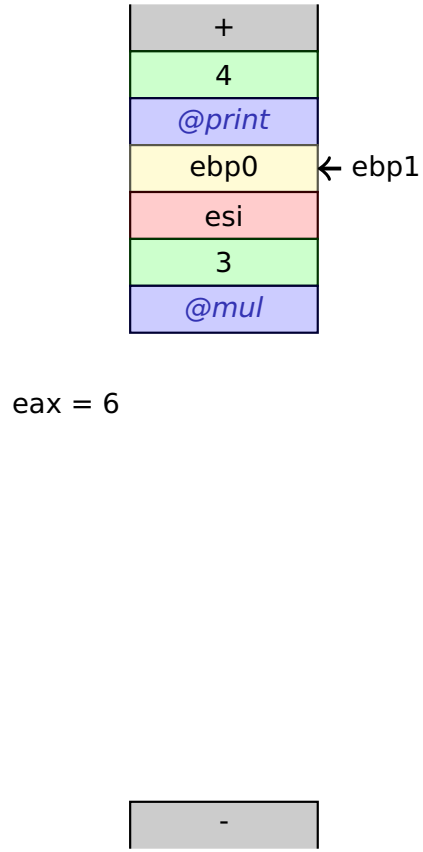


# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```



# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

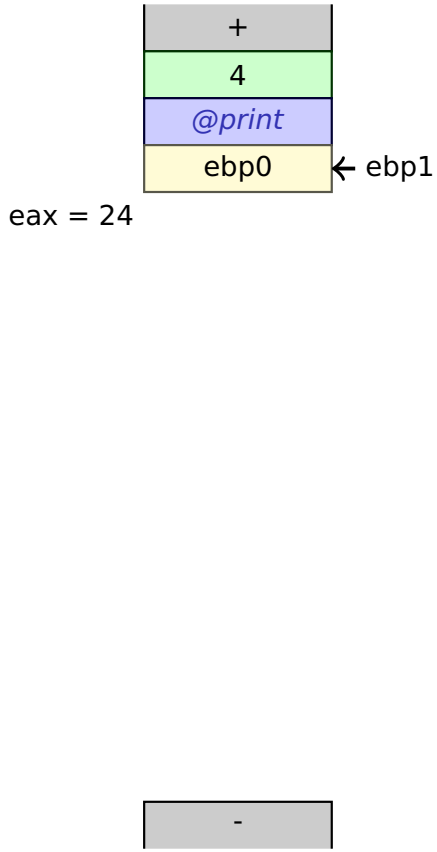
```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop   esi
   pop   ebp
   retn 4
```



# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
  ...
  push  4
  call  factorielle
print:
  push  eax
  push  offset Format
  call  printf
  ...
```

```
 factorielle:
  push  ebp
  mov   ebp, esp
  xor   eax, eax
  push  esi
  mov   esi, [ebp+8]
  inc   eax
  cmp   esi, eax
  jbe   short end
  lea   eax, [esi-1]
  push  eax
  call  factorielle
mul:
  imul  eax, esi
end:
  pop  esi
  pop  ebp
  retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

# Application

## Exemple : calcul de factorielle(4)

```
 main:
   ...
   push  4
   call  factorielle
print:
   push  eax
   push  offset Format
   call  printf
   ...
```

```
 factorielle:
   push  ebp
   mov   ebp, esp
   xor   eax, eax
   push  esi
   mov   esi, [ebp+8]
   inc   eax
   cmp   esi, eax
   jbe   short end
   lea   eax, [esi-1]
   push  eax
   call  factorielle
mul:
   imul  eax, esi
end:
   pop  esi
   pop  ebp
   retn 4
```

# Que fait cette fonction ?

```
sub_411E60 proc near

var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 54h
push    ebx
push    esi
push    edi
mov     [ebp+var_4], ecx
mov     eax, [ebp+var_4]
push    eax             ; Str
call    strlen
add     esp, 4
mov     [ebp+var_8], eax
mov     [ebp+var_10], 0
mov     eax, [ebp+var_8]
sub     eax, 1
mov     [ebp+var_14], eax
jmp     short loc_411E9F
```