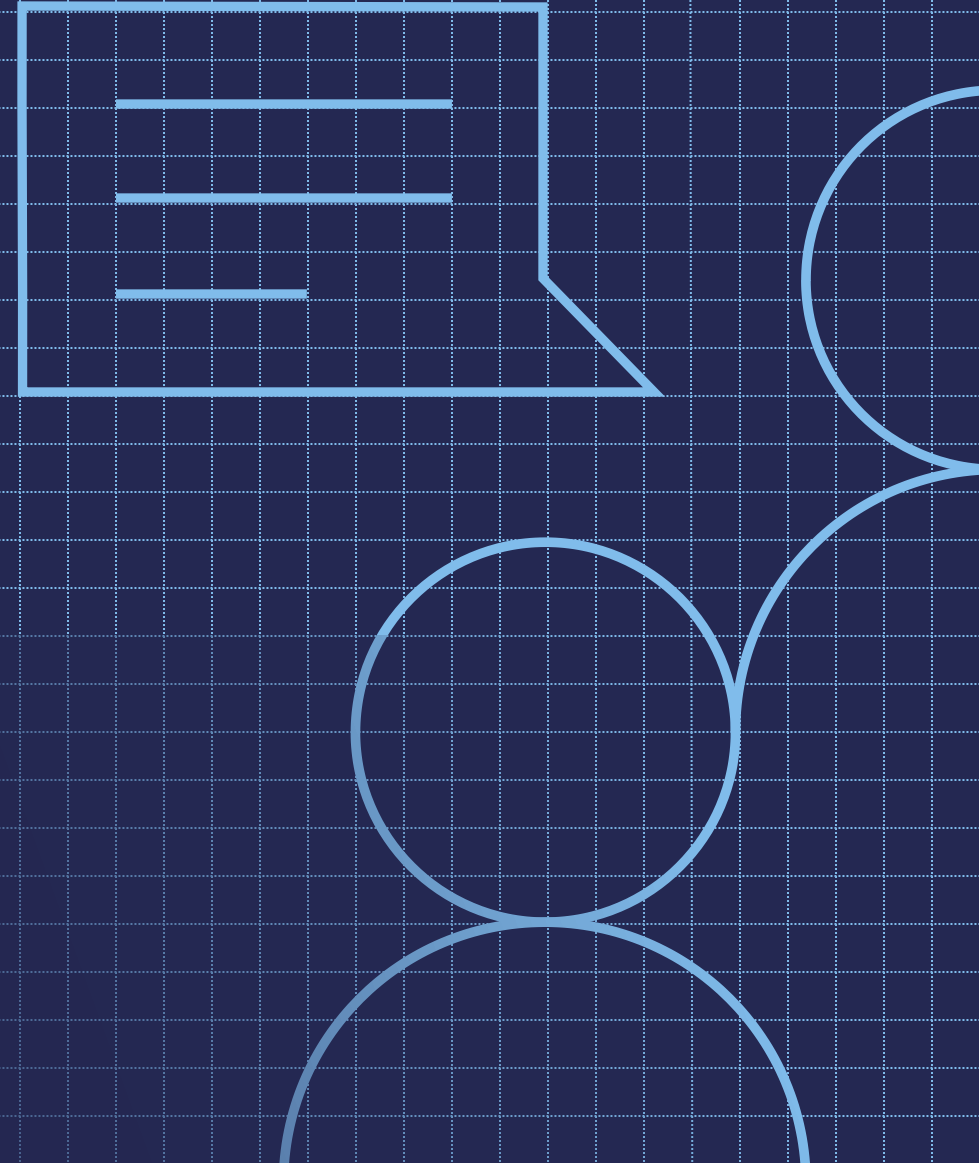


Analyse statique avancée - Assembleur

EFREI-PARIS

B. AIT SALEM



Plan

Les bases

- Chaîne de compilation – Chargement d'un programme en mémoire
- Désassembleur – Introduction au langage assembleur –
- TD

Types et catégories de malwares

Techniques utilisées

Approches et techniques d'analyse de malwares

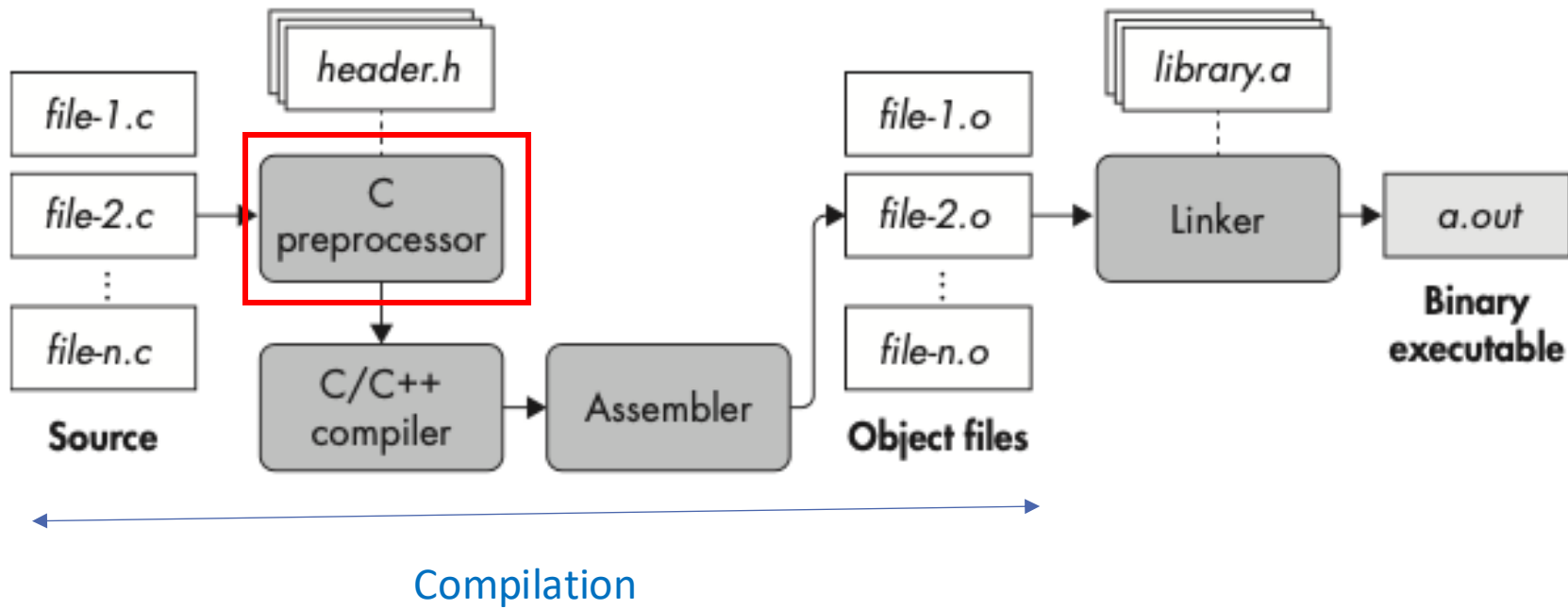
Mapping avec le Framework MITRE ATT&CK

Chaîne de compilation

```
#include <stdio.h>
#define FORMAT_STRING "%s"
#define MESSAGE "Hello, world!\n"

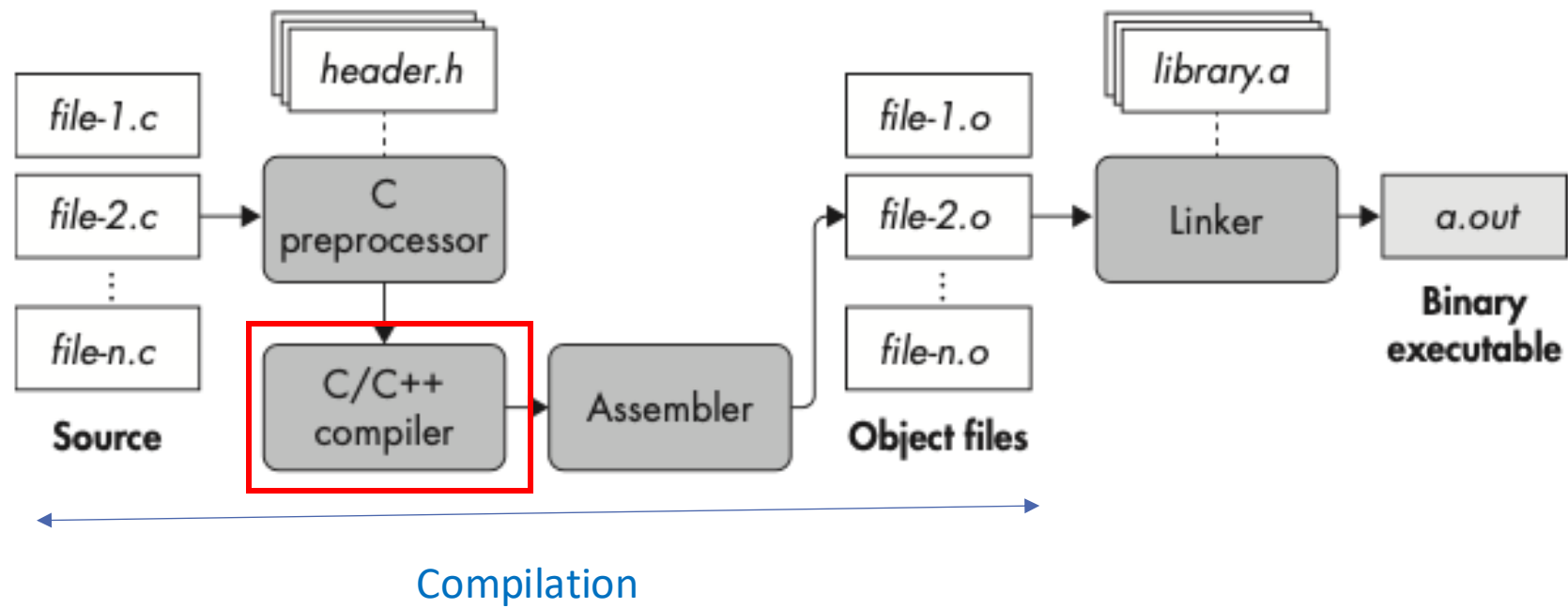
int main(int argc, char *argv[]) {
    printf(FORMAT_STRING, MESSAGE);
    return 0;
}
```

gcc -E -P file-2.c



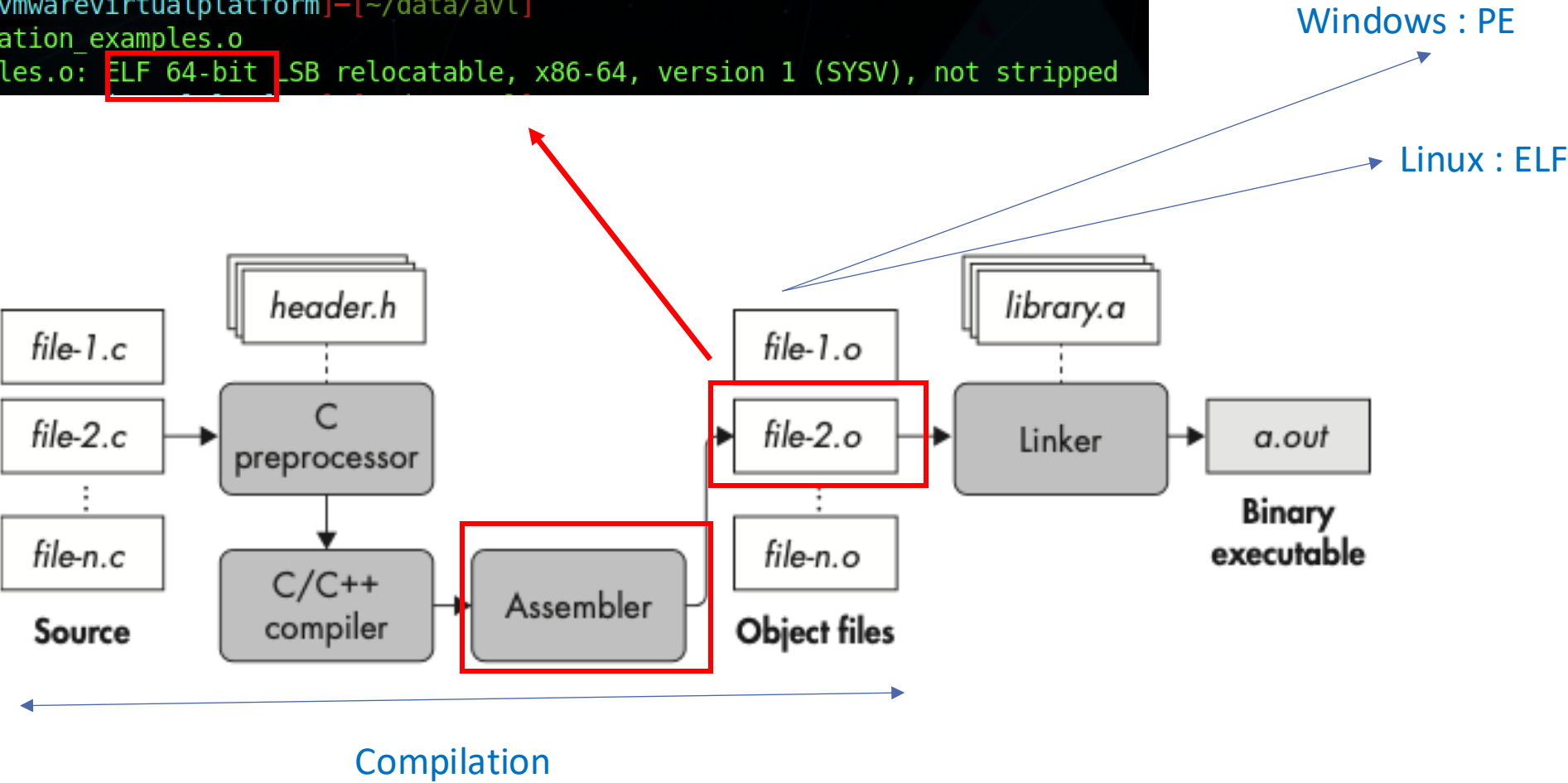
Chaîne de compilation

```
gcc -S -masm=intel file-2.c
```



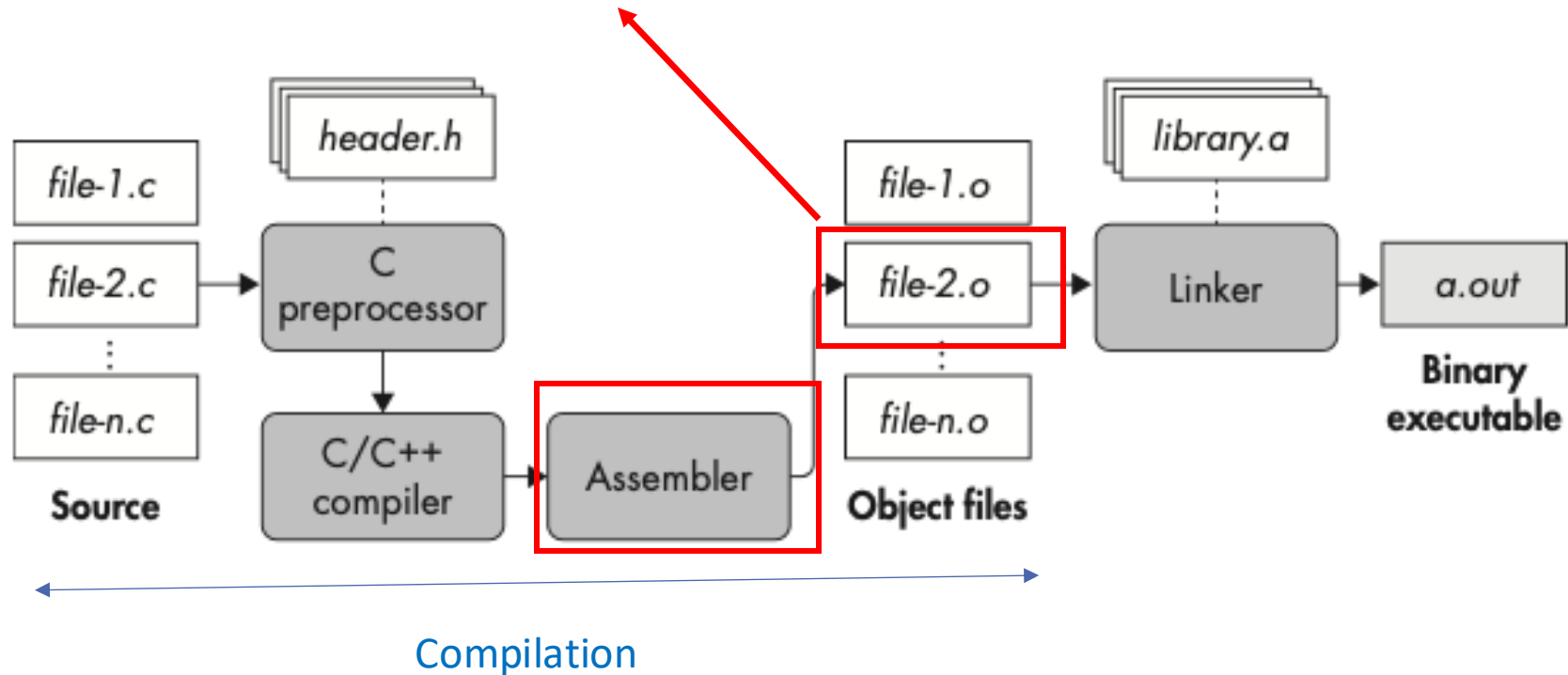
Chaîne de compilation

```
[parrot@parrot-vmwarevirtualplatform]--[~/data/avl]
$gcc -c compilation_examples.c
[parrot@parrot-vmwarevirtualplatform]--[~/data/avl]
$file compilation_examples.o
compilation_examples.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```



Chaîne de compilation

```
[parrot@parrot-vmwarevirtualplatform]--[~/data/avl]
→ $gcc -c compilation_examples.c
[parrot@parrot-vmwarevirtualplatform]--[~/data/avl]
→ $file compilation_examples.o
compilation_examples.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```



Chaîne de compilation – Relocation de symboles

- Symboles : nom des variables et de fonctions
- Table des symboles: Correspondance symbole <-> adresse dans le fichier objet

```
#include <stdio.h>
#define FORMAT_STRING "%s"
#define MESSAGE "Hello, world!\n"

int main(int argc, char *argv[]) {
    printf(FORMAT_STRING, MESSAGE);
    return 0;
}
```

```
[parrot@parrot-vmwarevirtualplatform] ~/data/avl
$readelf --relocs compilation_examples.o

Section de réadressage '.rel.text' à l'adresse de décalage 0x228 contient 2 entrées:
Décalage      Info      Type      Val.-symboles Noms-symb.+ Addenda
000000000012  000500000002 R_X86_64_PC32  0000000000000000 .rodata - 4
000000000017  000b00000004 R_X86_64_PLT32  0000000000000000 puts - 4

$objdump -sj .rodata compilation_examples.o
README:ncurses
compilation_examples.o:      format de fichier elf64-x86-64

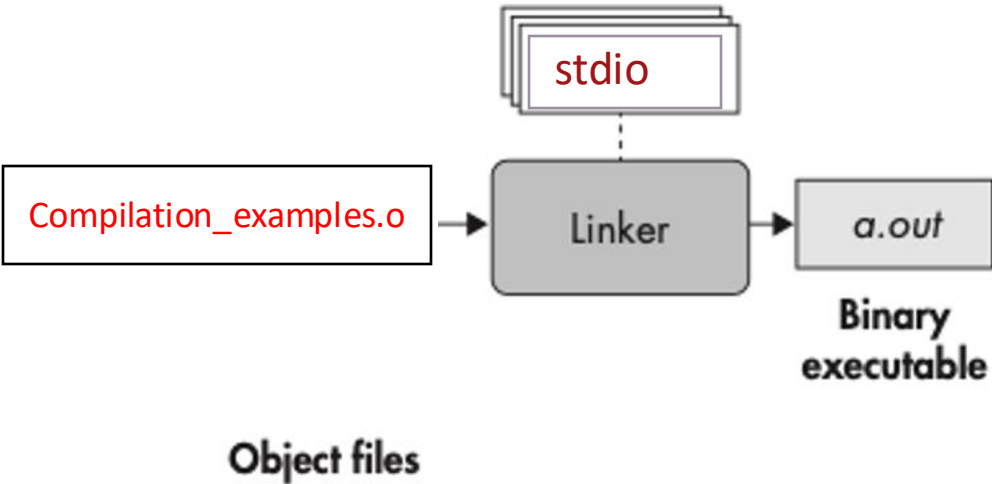
Contenu de la section .rodata :
0000 48656c6c 6f2c2077 6f726c64 2100      Hello, world!.
[parrot@parrot-vmwarevirtualplatform] ~/data/avl
$objdump -M intel -d compilation_examples.o

compilation_examples.o:      format de fichier elf64-x86-64

Déassemblage de la section .text :
0000000000000000 <main>:
0: 55                push    rbp
1: 48 89 e5          mov     rbp, rsp
4: 48 83 ec 10       sub     rsp, 0x10
8: 89 7d fc          mov     DWORD PTR [rbp-0x4], edi
b: 48 89 75 f0       mov     QWORD PTR [rbp-0x10], rsi
f: 48 8d 3d 00 00 00 00 lea     rdi, [rip+0x0]          # 16 <main+0x16>
16: e8 00 00 00 00    call   1b <main+0x1b>
1b: b8 00 00 00 00    mov     eax, 0x0
20: c9               leave   %edi
21: c3               ret
```

Chaîne de compilation

Edition de lien



```
[parrot@parrot-vmwarevirtualplatform] [~/data/avl]
$readelf --relocs compilation_examples.o

Section de réadressage '.rela.text' à l'adresse de décalage 0x228 contient 2 entrées:
Décalage      Info      Type      Val.-symboles Noms-symb.+ Addenda
000000000012  000500000002 R_X86_64_PC32  0000000000000000 .rodata - 4
000000000017  000b00000004 R_X86_64_PLT32  0000000000000000 puts - 4

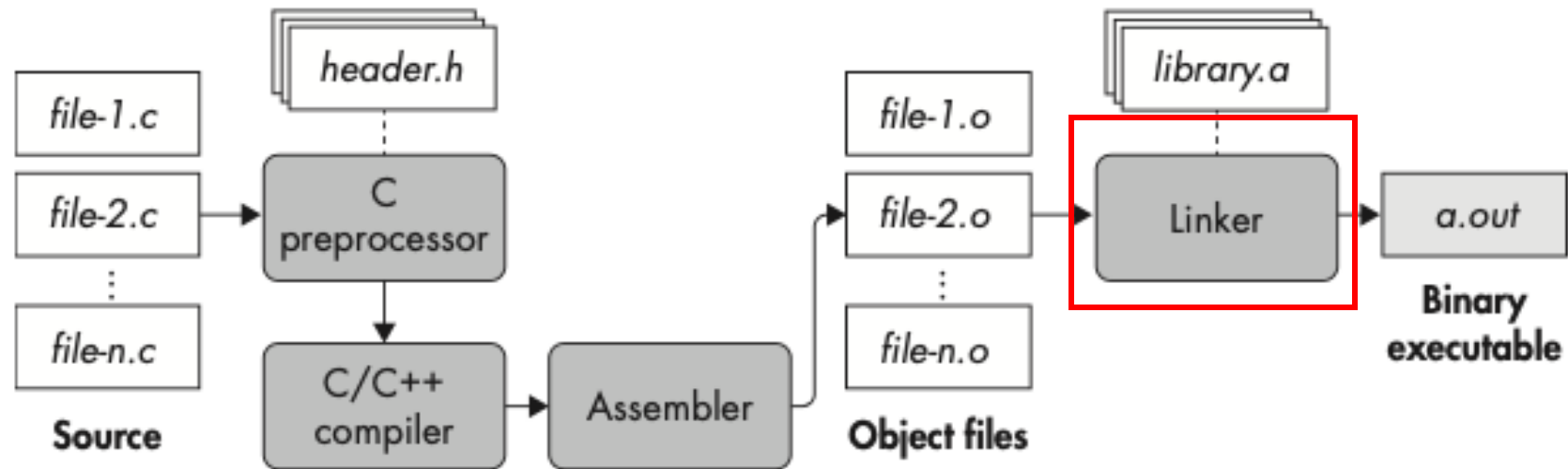
$objdump -sj .rodata compilation_examples.o
compilation_examples.o:  format de fichier elf64-x86-64

Contenu de la section .rodata :
0000 48656c6c 6f2c2077 6f726c64 2100      Hello, world!.
[parrot@parrot-vmwarevirtualplatform] [~/data/avl]
$objdump -M intel -d compilation_examples.o
compilation_examples.o:  format de fichier elf64-x86-64

Désassemblage de la section .text :
0000000000000000 <main>:
0: 55          push    rbp
1: 48 89 e5    mov     rbp, rsp
4: 48 83 ec 10 sub     rsp, 0x10
8: 89 7d fc    mov     DWORD PTR [rbp-0x4], edi
b: 48 89 75 f0 mov     QWORD PTR [rbp-0x10], rsi
f: 48 8d 3d 00 00 00 00 lea     rdi, [rip+0x0]          # 16 <main+0x16>
16: e8 00 00 00 00 call    1b <main+0x1b>
1b: b8 00 00 00 00 mov     eax, 0x0
20: c9          leave   %edi
21: c3          ret
```


Chaîne de compilation

Edition de lien



Chaîne de compilation

Edition de lien → Statique

```
gcc -static compilation_example.c -o  
compilation_example.o
```

Il combine tous les fichiers objets et toutes les bibliothèques requises en un seul exécutable :

- Le linker résout les **références de symboles** (noms de fonctions et de variables) qui sont utilisées dans un fichier objet mais définies dans un autre.
- Le linker intègre tout le code nécessaire des bibliothèques externes directement dans le fichier exécutable.

Utilisation de la fonction printf définie dans la librairie STDIO

```
000000000401c8d <main>:$  
401c8d:> 55 > push rbp$  
401c8e:> 48 89 e5 > mov rbp, rsp$  
401c91:> 48 83 ec 10 > sub rsp, 0x10$  
401c95:> 89 7d fc > mov DWORD PTR [rbp-0x4], edi$  
401c98:> 48 89 75 f0 > mov QWORD PTR [rbp-0x10], rsi$  
401c9c:> 48 8d 3d 61 f3 07 00 > lea rdi, [rip+0x7f361] # 481004 < IO stdin used+0x4>$  
401ca3:> e8 38 7a 00 00 > call 4096e0 < IO puts>$  
401ca8:> b8 00 00 00 00 > mov eax, 0x0$  
401cad:> c9 > leave++$  
401cae:> c3 > ret+++$  
401caf:> 90 > nop$  
  
0000000004096d0 < IO_new_fopen>:$  
4096d0:> ba 01 00 00 00 > mov edx, 0x1$  
4096d5:> e9 06 ff ff ff > jmp 4095e0 < _fopen_internal>$  
4096da:> 66 0f 1f 44 00 00 > nop WORD PTR [rax+rax*1+0x0]$  
  
0000000004096e0 < IO_puts>:$  
4096e0:> 41 55 > push r13$  
4096e2:> 41 54 > push r12$  
4096e4:> 49 89 fc > mov r12, rdi$  
4096e7:> 55 > push rbp$  
4096e8:> 53 > push rbx$  
4096e9:> 48 83 ec 08 > sub rsp, 0x8$  
4096ed:> e8 d6 79 ff ff > call 4010c8 < .plt+0xb0>$  
4096f2:> 48 8b 2d d7 2f 0a 00 > mov rbp, QWORD PTR [rip+0xa2fd7] # 4ac6d0 < stdout>$  
4096f9:> 48 89 c3 > mov rbx, rax$  
4096fc:> 8b 55 00 > mov edx, DWORD PTR [rbp+0x0]$  
4096ff:> 81 e2 00 80 00 00 > and edx, 0x8000$  
409705:> 75 69 > jne 409770 < IO_puts+0x90>$  
409707:> 4c 8b 85 88 00 00 00 > mov r8, QWORD PTR [rbp+0x88]$  
40970e:> 64 4c 8b 2c 25 10 00 > mov r13, QWORD PTR fs:0x10$
```

Chaîne de compilation

- Edition de lien → dynamique

GOT (Global Offset Table)

C'est une structure de données utilisée dans les programmes utilisant l'édition de lien dynamique. Elle stocke les adresses des variables globales et des fonctions qui sont importées à partir de bibliothèques partagées. Lorsqu'un programme est compilé, les adresses réelles de ces fonctions et variables ne sont pas connues, car elles seront résolues au moment de l'exécution.

PLT (Procedure Linkage Table)

C'est un mécanisme qui fonctionne aux côtés de la GOT pour faciliter l'appel de fonctions dans des bibliothèques partagées. La PLT contient des stubs pour chaque fonction qui est appelée à partir d'une bibliothèque partagée.

```
> gdb -q example_compilation
pwndbg: loaded 199 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from example_compilation...(no debugging symbols found)...done.
pwndbg> disassemble main
Dump of assembler code for function main:
0x08048426 <+0>: lea ecx,[esp+0x4]
0x0804842a <+4>: and esp,0xffffffff
0x0804842d <+7>: push DWORD PTR [ecx-0x4]
0x08048430 <+10>: push ebp
0x08048431 <+11>: mov ebp,esp
0x08048433 <+13>: push ebx
0x08048434 <+14>: push ecx
0x08048435 <+15>: call 0x8048462 <__x86.get_pc_thunk.ax>
>
0x0804843a <+20>: add eax,0x1bc6
0x0804843f <+25>: sub esp,0xc
0x08048442 <+28>: lea edx,[eax-0x1b10]
0x08048448 <+34>: push edx
0x08048449 <+35>: mov ebx,eax
0x0804844b <+37>: call 0x80482e0 <puts@plt>
0x08048450 <+42>: add esp,0x10
0x08048453 <+45>: mov eax,0x0
0x08048458 <+50>: lea esp,[ebp-0x8]
0x0804845b <+53>: pop ecx
0x0804845c <+54>: pop ebx
0x0804845d <+55>: pop ebp
0x0804845e <+56>: lea esp,[ecx-0x4]
0x08048461 <+59>: ret
End of assembler dump.
pwndbg> break *0x0804844b
Breakpoint 1 at 0x0804844b
pwndbg> x/3i 0x80482e0
0x80482e0 <puts@plt>: jmp DWORD PTR ds:0x804a00c
0x80482e6 <puts@plt+6>: push 0x0
0x80482eb <puts@plt+11>: jmp 0x80482d0
pwndbg> x/wx 0x804a00c
0x804a00c: 0x080482e6
pwndbg>
> objdump -d -j .text ./example_compilation | grep puts
804844b: e8 90 fe ff ff call 80482e0 <puts@plt>
> objdump -R example_compilation
example_compilation: file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049ffc R_386_GLOB_DAT __gmon_start__
0804a00c R_386_JUMP_SLOT puts@GLIBC_2.0
0804a010 R_386_JUMP_SLOT __libc_start_main@GLIBC_2.0
```

Chaîne de compilation

- Edition de lien → dynamique

GOT (Global Offset Table)

C'est une structure de données utilisée dans les programmes utilisant l'édition de lien dynamique. Elle stocke les adresses des variables globales et des fonctions qui sont importées à partir de bibliothèques partagées. Lorsqu'un programme est compilé, les adresses réelles de ces fonctions et variables ne sont pas connues, car elles seront résolues au moment de l'exécution.

PLT (Procedure Linkage Table)

C'est un mécanisme qui fonctionne aux côtés de la GOT pour faciliter l'appel de fonctions dans des bibliothèques partagées. La PLT contient des stubs pour chaque fonction qui est appelée à partir d'une bibliothèque partagée.

```
> gdb -q example_compilation
pwndbg: loaded 199 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from example_compilation...(no debugging symbols found)...done.
pwndbg> disassemble main
Dump of assembler code for function main:
0x08048426 <+0>: lea ecx,[esp+0x4]
0x0804842a <+4>: and esp,0xffffffff
0x0804842d <+7>: push DWORD PTR [ecx-0x4]
0x08048430 <+10>: push ebp
0x08048431 <+11>: mov ebp,esp
0x08048433 <+13>: push ebx
0x08048434 <+14>: push ecx
0x08048435 <+15>: call 0x8048462 <__x86.get_pc_thunk.ax>
>
0x0804843a <+20>: add eax,0x1bc6
0x0804843f <+25>: sub esp,0xc
0x08048442 <+28>: lea edx,[eax-0x1b10]
0x08048448 <+34>: push edx
0x08048449 <+35>: mov ebx,eax
0x0804844b <+37>: call 0x80482e0 <puts@plt>
0x08048450 <+42>: add esp,0x10
0x08048453 <+45>: mov eax,0x0
0x08048458 <+50>: lea esp,[ebp-0x8]
0x0804845b <+53>: pop ecx
0x0804845c <+54>: pop ebx
0x0804845d <+55>: pop ebp
0x0804845e <+56>: lea esp,[ecx-0x4]
0x08048461 <+59>: ret
End of assembler dump.
pwndbg> break *0x0804844b
Breakpoint 1 at 0x0804844b
pwndbg> x/3i 0x80482e0
0x80482e0 <puts@plt>: jmp DWORD PTR ds:0x804a00c
0x80482e6 <puts@plt+6>: push 0x0
0x80482eb <puts@plt+11>: jmp 0x80482d0
pwndbg> x/wx 0x804a00c
0x804a00c: 0x080482e6
pwndbg>
> objdump -d -j .text ./example_compilation | grep puts
804844b: e8 90 fe ff ff call 80482e0 <puts@plt>
> objdump -R example_compilation
example_compilation: file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049ffc R_386_GLOB_DAT __gmon_start__
0804a00c R_386_JUMP_SLOT puts@GLIBC_2.0
0804a010 R_386_JUMP_SLOT __libc_start_main@GLIBC_2.0
```

Chaîne de compilation

- Edition de lien → dynamique

GOT (Global Offset Table)

C'est une structure de données utilisée dans les programmes utilisant l'édition de lien dynamique. Elle stocke les adresses des variables globales et des fonctions qui sont importées à partir de bibliothèques partagées. Lorsqu'un programme est compilé, les adresses réelles de ces fonctions et variables ne sont pas connues, car elles seront résolues au moment de l'exécution.

PLT (Procedure Linkage Table)

C'est un mécanisme qui fonctionne aux côtés de la GOT pour faciliter l'appel de fonctions dans des bibliothèques partagées. La PLT contient des stubs pour chaque fonction qui est appelée à partir d'une bibliothèque partagée.

```
> gdb -q example_compilation
pwndbg: loaded 199 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from example_compilation...(no debugging symbols found)...done.
pwndbg> disassemble main
Dump of assembler code for function main:
0x08048426 <+0>: lea ecx,[esp+0x4]
0x0804842a <+4>: and esp,0xffffffff
0x0804842d <+7>: push DWORD PTR [ecx-0x4]
0x08048430 <+10>: push ebp
0x08048431 <+11>: mov ebp,esp
0x08048433 <+13>: push ebx
0x08048434 <+14>: push ecx
0x08048435 <+15>: call 0x08048462 <__x86.get_pc_thunk.ax>
>
0x0804843a <+20>: add eax,0x1bc6
0x0804843f <+25>: sub esp,0xc
0x08048442 <+28>: lea edx,[eax-0x1b10]
0x08048448 <+34>: push edx
0x08048449 <+35>: mov ebx,eax
0x0804844b <+37>: call 0x080482e0 <puts@plt>
0x08048450 <+42>: add esp,0x10
0x08048453 <+45>: mov eax,0x0
0x08048458 <+50>: lea esp,[ebp-0x8]
0x0804845b <+53>: pop ecx
0x0804845c <+54>: pop ebx
0x0804845d <+55>: pop ebp
0x0804845e <+56>: lea esp,[ecx-0x4]
0x08048461 <+59>: ret
End of assembler dump.
pwndbg> break *0x0804844b
Breakpoint 1 at 0x0804844b
pwndbg> x/3i 0x080482e0
0x080482e0 <puts@plt>: jmp DWORD PTR ds:0x0804a00c
0x080482e6 <puts@plt+6>: push 0x0
0x080482eb <puts@plt+11>: jmp 0x080482d0
pwndbg> x/wx 0x0804a00c
0x0804a00c: 0x080482e6
pwndbg>
```

```
> objdump -d -j .text ./example_compilation | grep puts
804844b: e8 90 fe ff ff call 80482e0 <puts@plt>
> objdump -R example_compilation
example_compilation: file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049ffc R_386_GLOB_DAT __gmon_start__
0804a00c R_386_JUMP_SLOT puts@GLIBC_2.0
0804a010 R_386_JUMP_SLOT __libc_start_main@GLIBC_2.0
```

GOT

- Find id of puts
- Update GOT

Chaîne de compilation

- **Edition de lien → dynamique**

GOT (Global Offset Table)

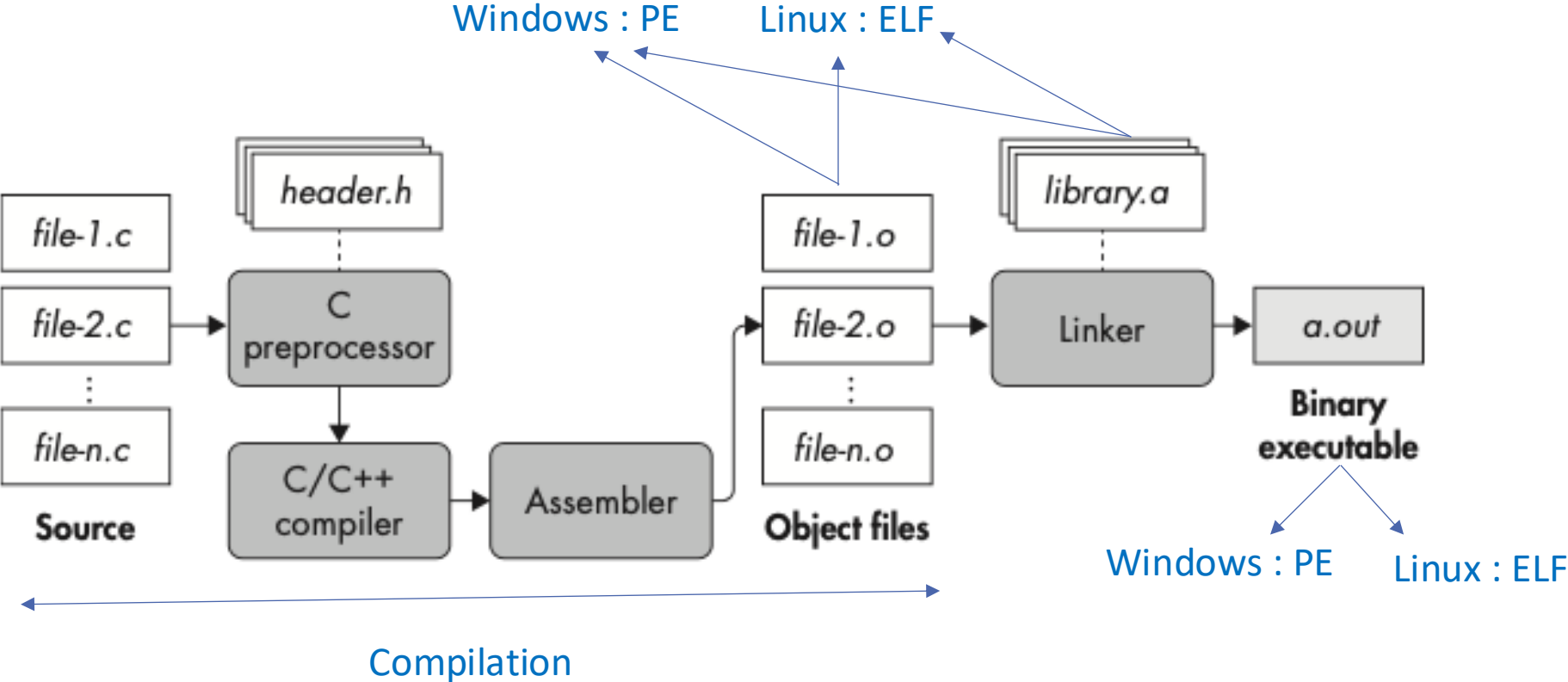
C'est une structure de données utilisée dans les programmes utilisant l'édition de lien dynamique. Elle stocke les adresses des variables globales et des fonctions qui sont importées à partir de bibliothèques partagées. Lorsqu'un programme est compilé, les adresses réelles de ces fonctions et variables ne sont pas connues, car elles seront résolues au moment de l'exécution.

PLT (Procedure Linkage Table)

C'est un mécanisme qui fonctionne aux côtés de la GOT pour faciliter l'appel de fonctions dans des bibliothèques partagées. La PLT contient des stubs pour chaque fonction qui est appelée à partir d'une bibliothèque partagée.

```
pwndbg> x/3i 0x80482e0
0x80482e0 <puts@plt>:      jmp     DWORD PTR ds:0x804a00c
0x80482e6 <puts@plt+6>:    push    0x0
0x80482eb <puts@plt+11>:   jmp     0x80482d0
pwndbg> x/wx 0x804a00c
0x804a00c: 0xf7e4fca0
pwndbg> x/5i 0xf7e4fca0
0xf7e4fca0 <_IO_puts>:    push    ebp
0xf7e4fca1 <_IO_puts+1>:    mov     ebp,esp
0xf7e4fca3 <_IO_puts+3>:    push    edi
0xf7e4fca4 <_IO_puts+4>:    push    esi
0xf7e4fca5 <_IO_puts+5>:    push    ebx
pwndbg> █
```

Chaîne de compilation



Format ELF

Un programme ELF est composé :

- D'un entête général
- D'entêtes de sections
- De sections:

.init contient du **code exécutable** réalisant certaines initialisations et s'exécutant avant tout autre code

.fini est analogue à .init mais s'exécute après le main

.text section de **code** où le main et les fonctions du programme résident

.rodata section de **données** qui est en lecture seule et destinée au stockage des constantes du programme

.data section de données en écriture et contient les variables initialisées

.bss section de données en écriture et contient les variables non initialisées

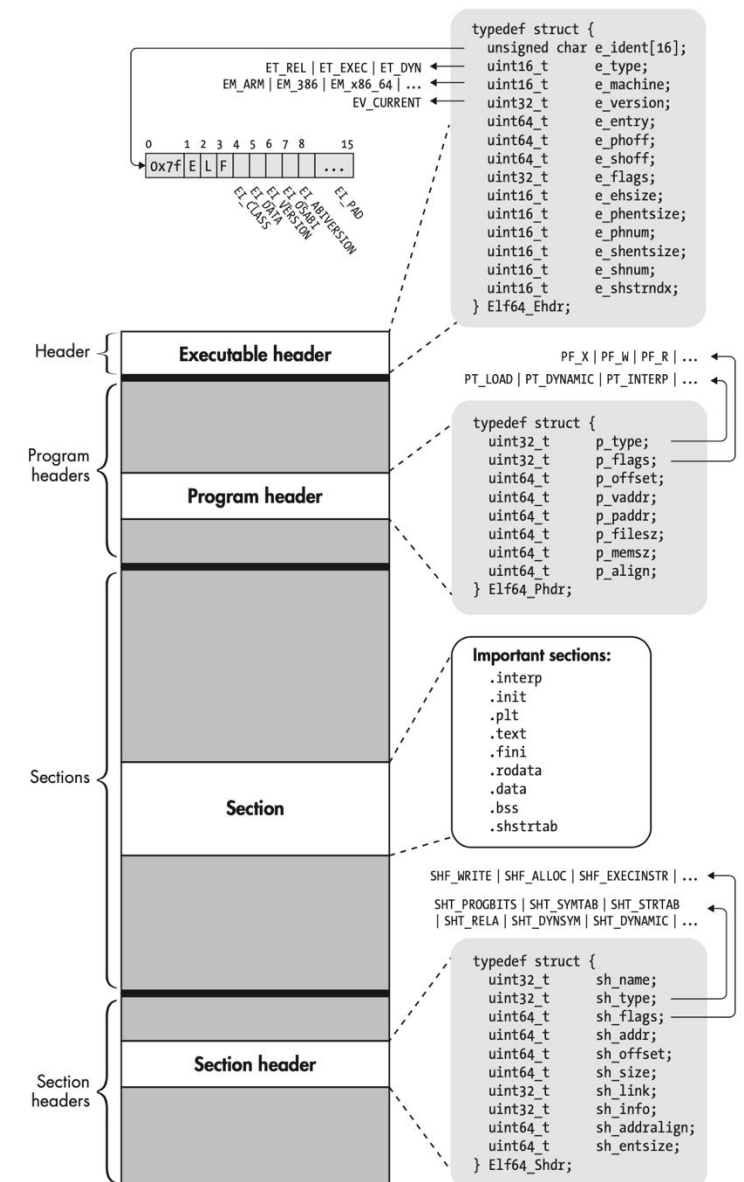
.interp : Contient le chemin vers l'éditeur de lien

rel.* et rela.* : tables de **relocation** de **symboles**

.got et .got.plt : On reparlera dans la section librairies partagées

.plt : On reparlera dans la section librairies partagées

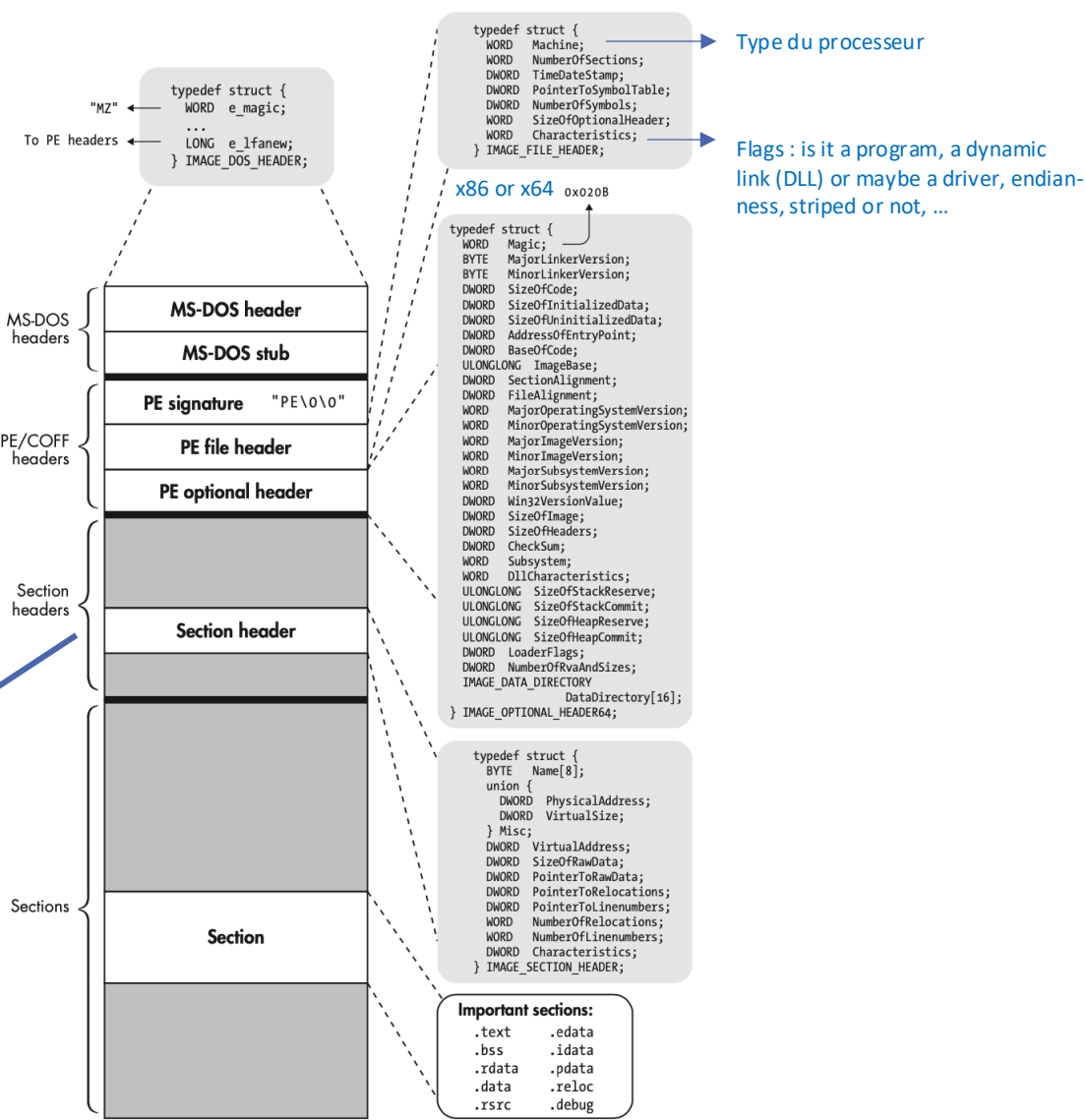
- D'entêtes de programmes



PE (Portable Executable) - Format

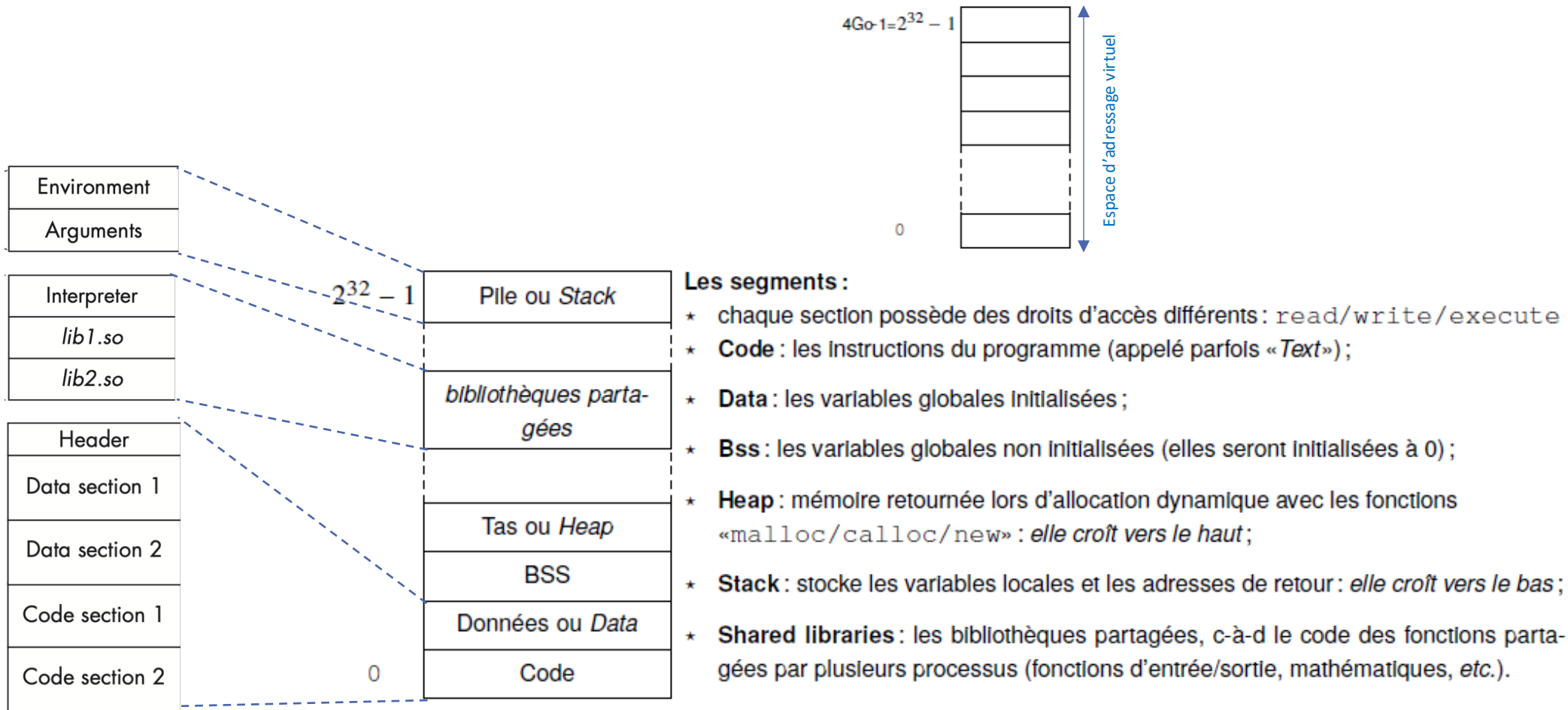
- Format Windows pour les fichiers exécutables, codes objets, DLLs, les dumps du noyau Windows
- C'est une structure de données contenant les informations nécessaires au chargeur et à l'éditeur de liens Windows.

Sections table					
Name	VirtualSize	VirtualAddress	SizeOfRawData	PointerToRawData	Characteristics
.text	0x1000	0x1000	0x200	0x200	CODE EXECUTE READ
.rdata	0x1000	0x2000	0x200	0x400	INITIALIZED READ
.data	0x1000	0x3000	0x200	0x600	DATA READ WRITE



Chargement du programme en mémoire

Un programme sur un adressage sur 32 bits voit la mémoire de l'adresse 0 à $2^{32} - 1$, c-à-d 4Go:



Exo

- Trouvez la section de chaque variable

```
int index = 5;
```

```
char * str;
```

```
int nothing;
```

```
void funct1(int c){
```

```
    int i=c;
```

```
    str = (char*) malloc (10 * sizeof (char));
```

```
    strncpy(str, "abcde", 5);
```

```
}
```

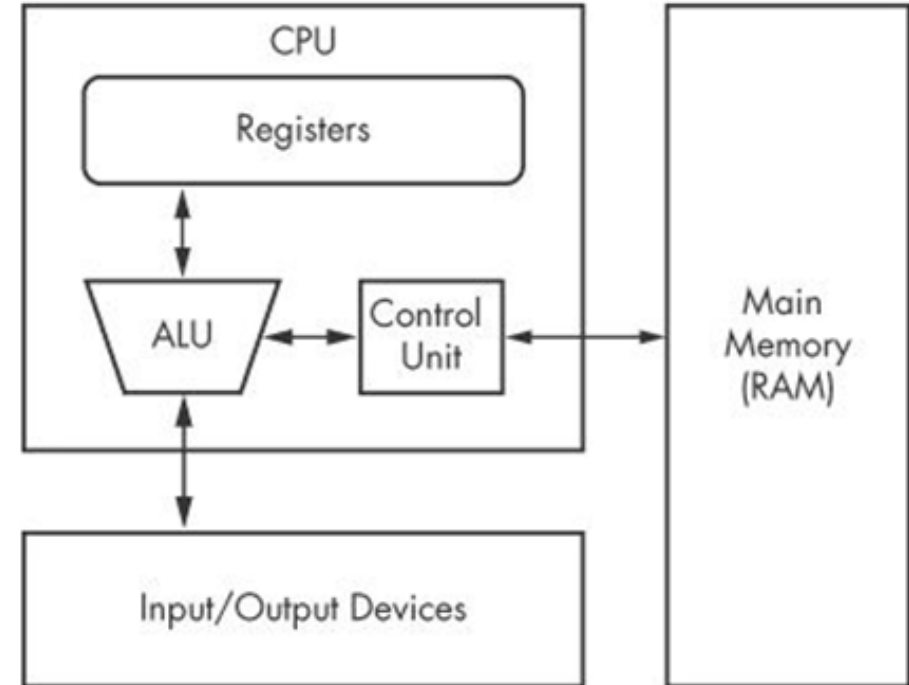
```
void main (){
```

```
    funct1(1);
```

```
}
```

Architecture machine de base

- Unité de contrôle (Control unit)
 - Recherche une instruction dont l'adresse se trouve dans le **compteur ordinal (program counter)** à partir de la RAM en utilisant un registre appelé **instruction pointer** (le registre **EIP** dans les architecture intel), la decode puis l'envoie à l'UAL pour l'exécuter.
- Registres
 - Mémoire au niveau de la CPU
 - Plus petite, mais plus rapide que la RAM
- UAL (ALU (Arithmetic Logic Unit))
 - Exécute une opération arithmétique et logique et place le résultat dans un registre ou dans la RAM



Architecture de Von Neumann

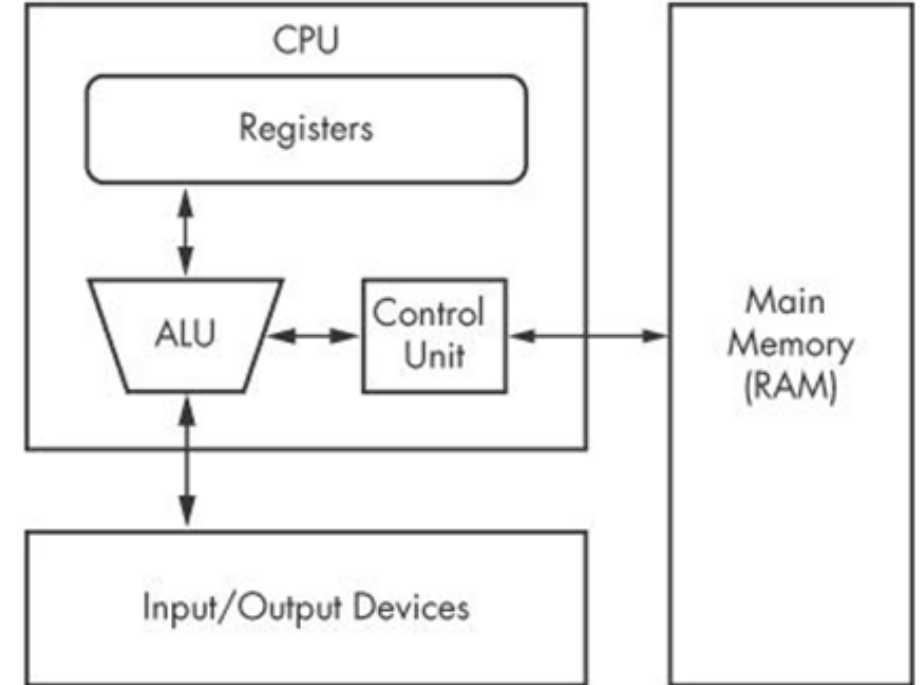
- CPU (**Central Processing Unit**) exécute des instructions machines
- RAM sauvegarde les données et le code
- I/O interface avec les périphériques

Instructions machines

- **Mnemonic** suivi d'un ou plusieurs opérandes "**operands**"

Exemple: **mov ecx 0x42**

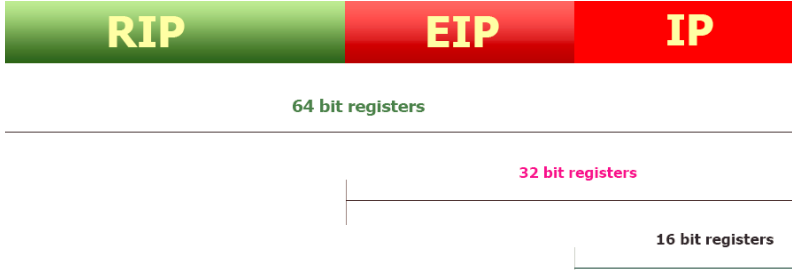
- Charger dans le registre **ecx** la valeur **42** (hex)
- En langage binaire cette instruction devient:
 - **0xB942000000**
 - **mov ecx** est la représentation de **0xB9** (en hexadecimal)



Architecture de Von Neumann

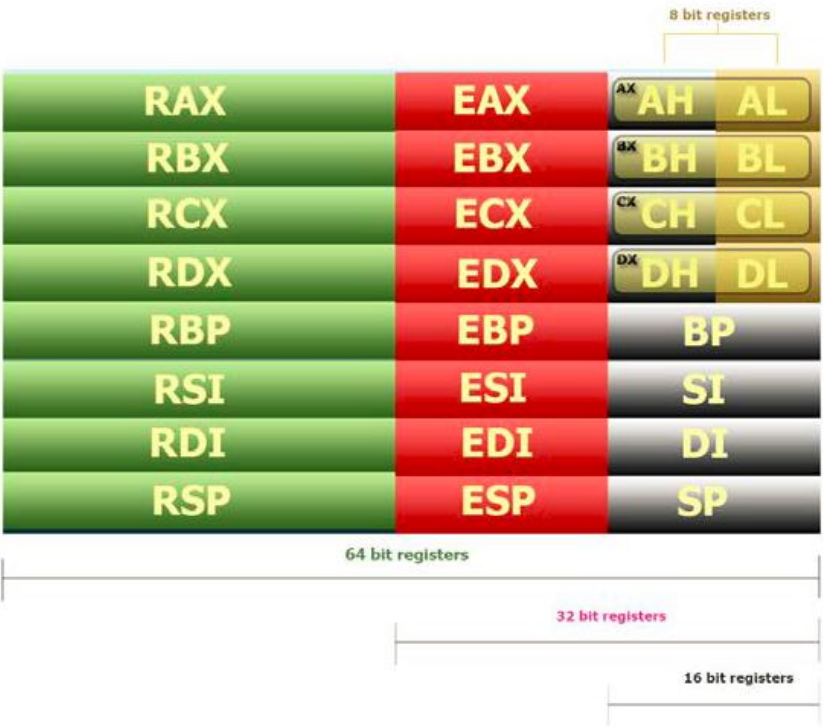
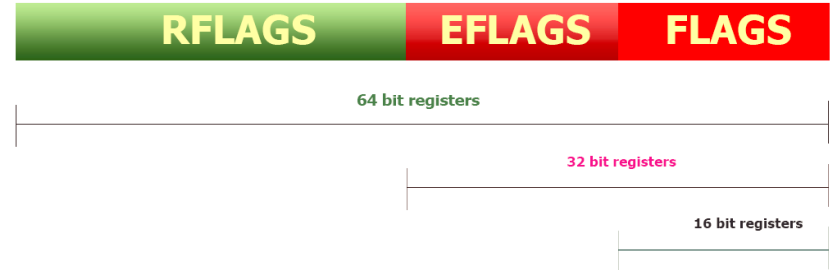
Registres

REGISTER	PURPOSE
ECX	Counter in loops
ESI	Source in string/memory operations
EDI	Destination in string/memory operations
EBP	Base frame pointer
ESP	Stack pointer



EAX contient souvent la valeur de retour lors d'un appel de fonction

La Multiplication et la division utilisent EAX et EDX



Bytes—8 bits. AL, BL, CL, ...

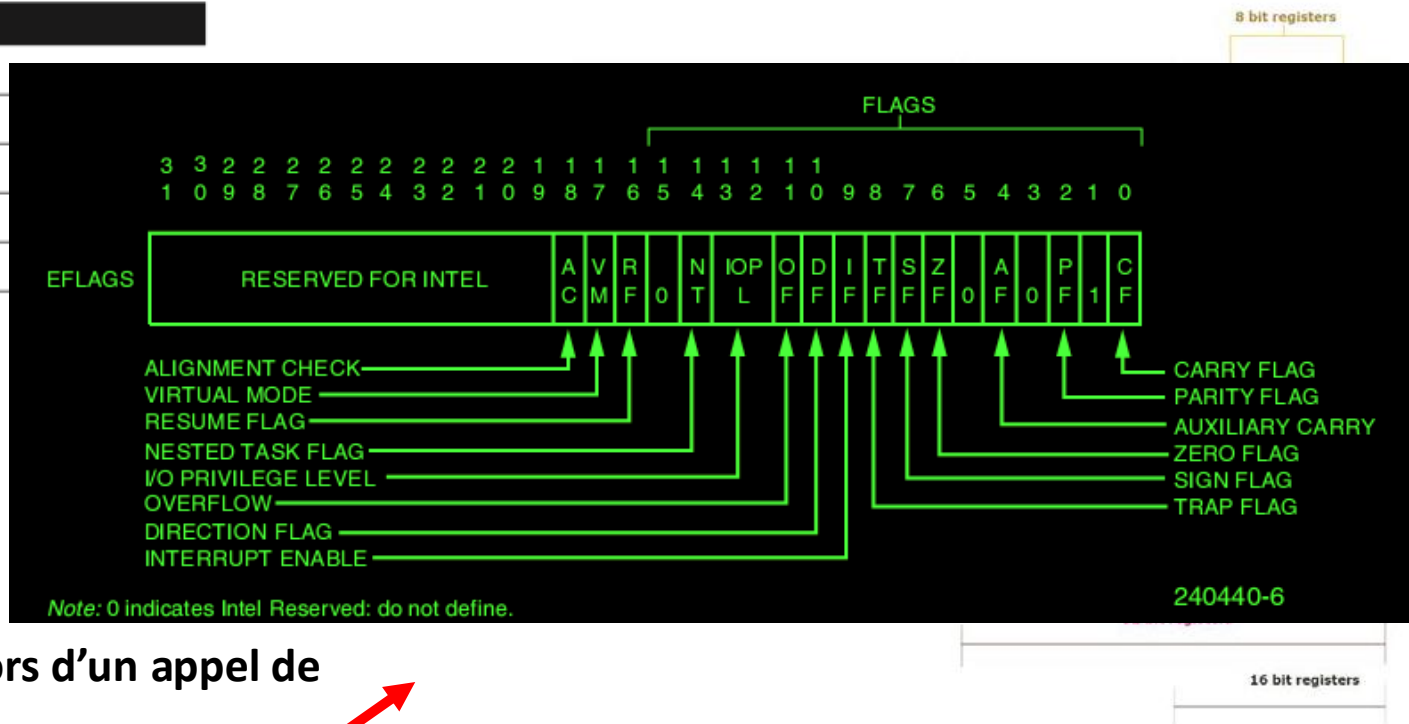
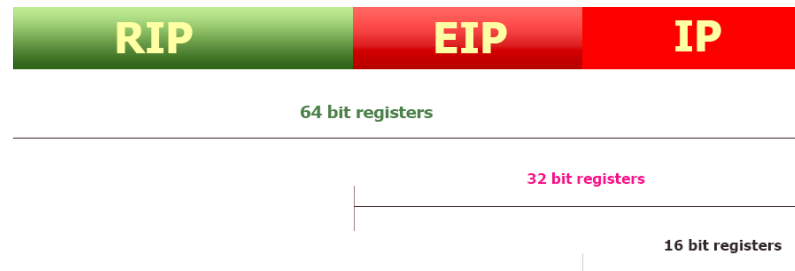
Word—16 bits. AX, BX, CX, ...

Double word—32 bits. EAX, EBX, ECX, ...

Quad word—64 bits. RAX, RBX, RCX, ...

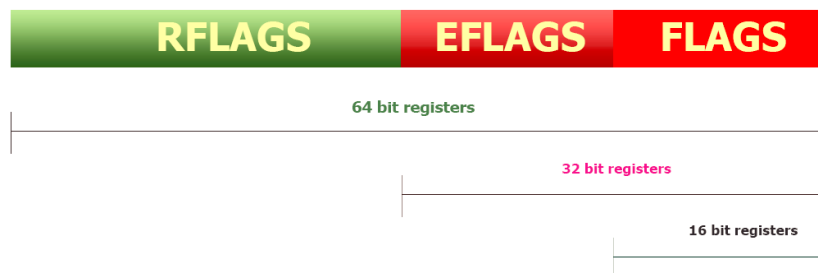
Registres

REGISTER	PURPOSE
ECX	Counter in loops
ESI	Source in string/memory operations
EDI	Destination in string/memory operations
EBP	Base frame pointer
ESP	Stack pointer



EAX contient souvent la valeur de retour lors d'un appel de fonction

La Multiplication et la division utilisent EAX et EDX



Bytes—8 bits. AL, BL, CL, ...

Word—16 bits. AX, BX, CX, ...

Double word—32 bits. EAX, EBX, ECX, ...

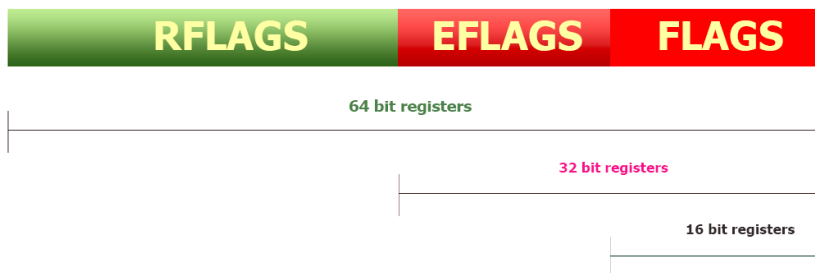
Quad word—64 bits. RAX, RBX, RCX, ...

Registres

- **ZF** Zero flag
 - Activé lorsque le résultat d'une opération est égal à zéro
- **CF** Carry flag
 - Activé lorsque le résultat d'une opération est plus grand ou plus petit que la destination
- **SF** Sign Flag
 - Activé lorsque le résultat est négatif ou lorsque le bit de poids fort est mis à 1 après une opération arithmétique
- **TF** Trap Flag
 - Utilisé pour le débogage. Lorsqu'il est activé, le processeur exécute une seule instruction à la fois.
- **DF** Direction Flag
 - A voir ci-dessous avec l'instruction **rep**

EAX contient souvent la valeur de retour lors d'un appel de fonction

La Multiplication et la division utilisent EAX et EDX



Bytes—8 bits. AL, BL, CL, ...

Word—16 bits. AX, BX, CX, ...

Double word—32 bits. EAX, EBX, ECX, ...

Quad word—64 bits. RAX, RBX, RCX, ...

Mouvement de données

Addressing Mode	Description	NASM Examples
Register	Registers hold the data to be manipulated. No memory interaction. Both registers must be the same size.	<code>mov ebx, edx</code> <code>add al, ch</code>
Immediate	The source operand is a numerical value. Decimal is assumed; use h for hex.	<code>mov eax, 1234h</code> <code>mov dx, 301</code>
Direct	The first operand is the address of memory to manipulate. It's marked with brackets.	<code>mov bh, 100</code> <code>mov[4321h], bh</code>
Register Indirect	The first operand is a register in brackets that holds the address to be manipulated.	<code>mov [di], ecx</code>
Based Relative	The effective address to be manipulated is calculated by using ebx or ebp plus an offset value.	<code>mov edx, 20[ebx]</code>
Indexed Relative	Same as Based Relative, but edi and esi are used to hold the offset.	<code>mov ecx, 20[esi]</code>
Based Indexed-Relative	The effective address is found by combining Based and Indexed Relative modes.	<code>mov ax, [bx][si]+1</code>

Endianness

- **Quelle est la plus petite unité de données adressable ?**
- **Big-Endian**
 - Placer les octets de poids forts en premier
 - 0x42 comme valeur de 64-bits sera sauvegardée en mémoire ainsi **0x00000042**
- **Little-Endian**
 - Placer les octets de poids faibles en premier
 - 0x42 comme valeur de 64-bit sera sauvegardée en mémoire ainsi **0x42000000**
- Les protocoles réseaux utilisent le format big-endian
- Les programmes x86 utilisent le format little-endian

Endianness Exemple d'une adresse IP

- 127.0.0.1 qui vaut **7F 00 00 01** en hex
- Elle est envoyée sur le réseau comme **0x7F000001**
- Elle sera par contre sauvegardée comme **0x0100007F** en mémoire centrale

Mov vs lea (Load Effective Address)

- **lea destination, source**
- `lea eax, [ebx+8]`
 - mettre `ebx + 8` dans `eax`
- **Vs mov**
 - `mov eax, [ebx+8]`
 - Copie la donnée se trouvant à l'adresse `ebx+8` dans `eax`

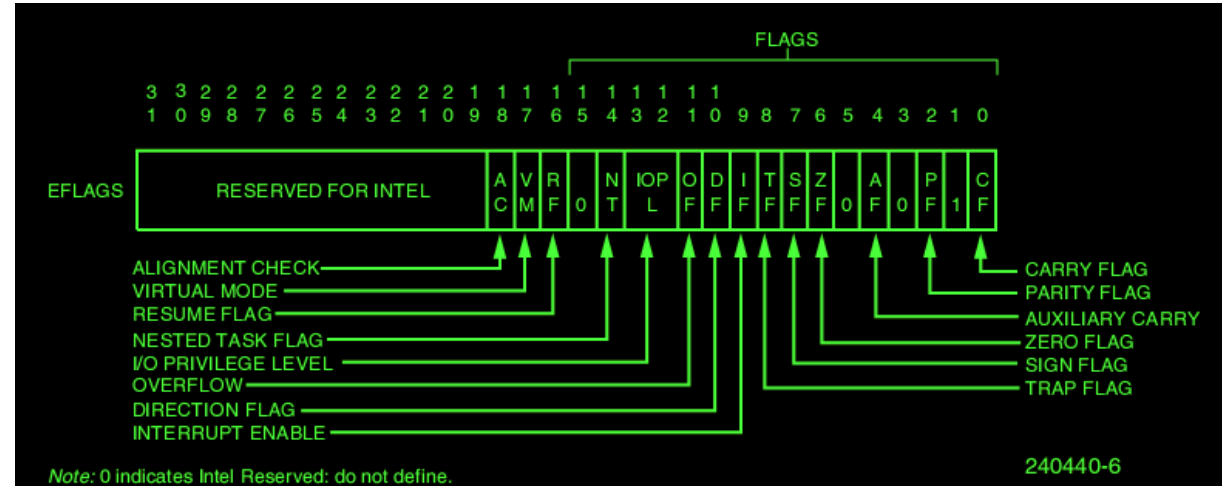


`mov eax, [ebx+8]` → place **0x20** dans `eax`

`Lea eax, [ebx + 8]` → place la valeur **0xB30048** dans `eax`

Instructions Arithmétiques

- **sub** Subtracts
- **add** Adds
- **inc** Increments
- **dec** Decrements
- **mul** Multiplies
- **div** Divides
- **Shr**



→ Ces opérations modifient le registre d'état

→ Le résultat de l'opération est sauvegardé dans le premier opérande

Exemple:

Après une instruction **sub**, le flag **ZF** est mis à **1** si le résultat est égal à zéro et le flag **CF** est positionné à **1** si la destination est plus petite que la valeur soustraite.

NOP

- Ne fait rien et EIP est incrémenté (la CPU passe à l'instruction suivante)
 - Son opcode = **0x90**
- Permet à un attaquant d'exécuter un code malveillant (shellcode) même en étant imprécis dans l'adresse du saut vers ce code.

Instructions de comparaison

- **test**
 - Compare deux valeurs en réalisant un **AND** des deux valeurs, mais ne modifie pas les deux valeurs
 - **test eax, eax**
 - Active le drapeau Zéro (**ZF**) si **eax** est égale à zéro
- **cmp eax, ebx**
 - Ressemble à l'instruction sub, mais ne modifie pas la valeur des deux opérandes
 - Le drapeau **ZF** est mis à **1** si les deux opérandes sont égaux

cmp dst, src	ZF	CF
dst = src	1	0
dst < src	0	1
dst > src	0	0

Branchements conditionnels

- Elle suivent toujours une instruction arithmétique ou souvent une instruction de comparaison

- **jz loc**

- Se brancher à l'adresse **loc** si le drapeau Zéro **ZF** est activé (**1**)

- **jnz loc**

- Se brancher à l'adresse **loc** si le drapeau Zéro **ZF** est désactivé (**0**)

jz loc	Jump to specified location if ZF = 1.
jnz loc	Jump to specified location if ZF = 0.
je loc	Same as jz, but commonly used after a cmp instruction. Jump will occur if the destination operand equals the source operand.
jne loc	Same as jnz, but commonly used after a cmp. Jump will occur if the destination operand is not equal to the source operand.
jg loc	Performs signed comparison jump after a cmp if the destination operand is greater than the source operand.
jge loc	Performs signed comparison jump after a cmp if the destination operand is greater than or equal to the source operand.
ja loc	Same as jg, but an unsigned comparison is performed.
jae loc	Same as jge, but an unsigned comparison is performed.
j1 loc	Performs signed comparison jump after a cmp if the destination operand is less than the source operand.
jle loc	Performs signed comparison jump after a cmp if the destination operand is less than or equal to the source operand.
jb loc	Same as j1, but an unsigned comparison is performed.
jbe loc	Same as jle, but an unsigned comparison is performed.
jo loc	Jump if the previous instruction set the overflow flag (OF = 1).
js loc	Jump if the sign flag is set (SF = 1).
jecz loc	Jump to location if ECX = 0.

Quelques opérateurs

- L'opérateur **offset** : nous donne le déplacement d'une variable ou un label par rapport au début de son segment
 - **Mov BX, OFFSET VAR1**
- L'opérateur **PTR**: peut être utilisé pour forcer la taille d'un opérande
- **DB, DW, DD**, servent à définir resp. un byte, un word et double word dans le segment data

```
ByteVal  DB  05h, 06h, 08h, 09h, 0Ah, 0Dh, '$'  
WordVal  DW  5510h, 6620h, 0A0Dh, '$'
```

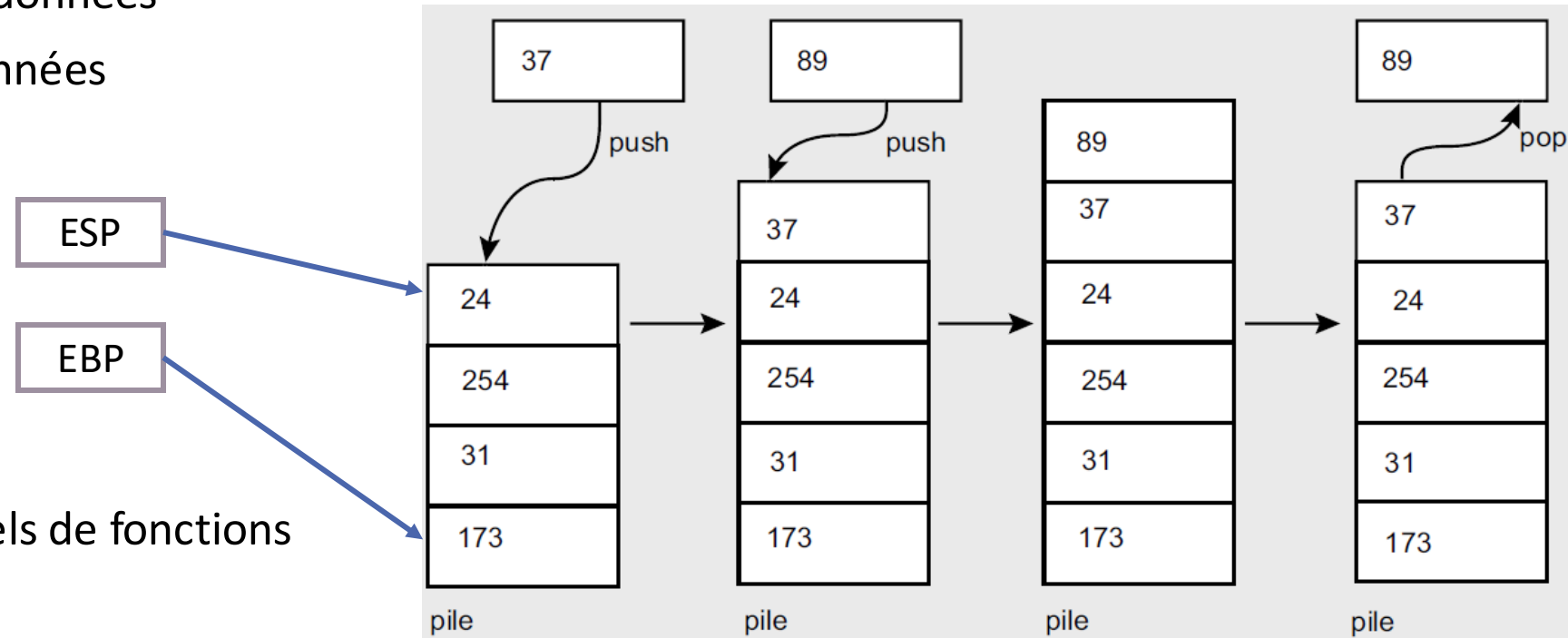
```
MOV  AX, WORD PTR ByteVal + 2    ; AX = 0908h  
MOV  BL, BYTE PTR WordVal + 4    ; BL = 0Dh
```

```
MOV  BX, OFFSET WordVal          ; BX pointe vers WordVal  
INC  [BX]                        ; ambigu – la taille n'est pas spécifiée  
INC  BYTE PTR [BX]                ; modifie la valeur en mémoire en 11h  
INC  WORD PTR [BX]                ; modifie la valeur en mémoire en 5511h
```

La pile

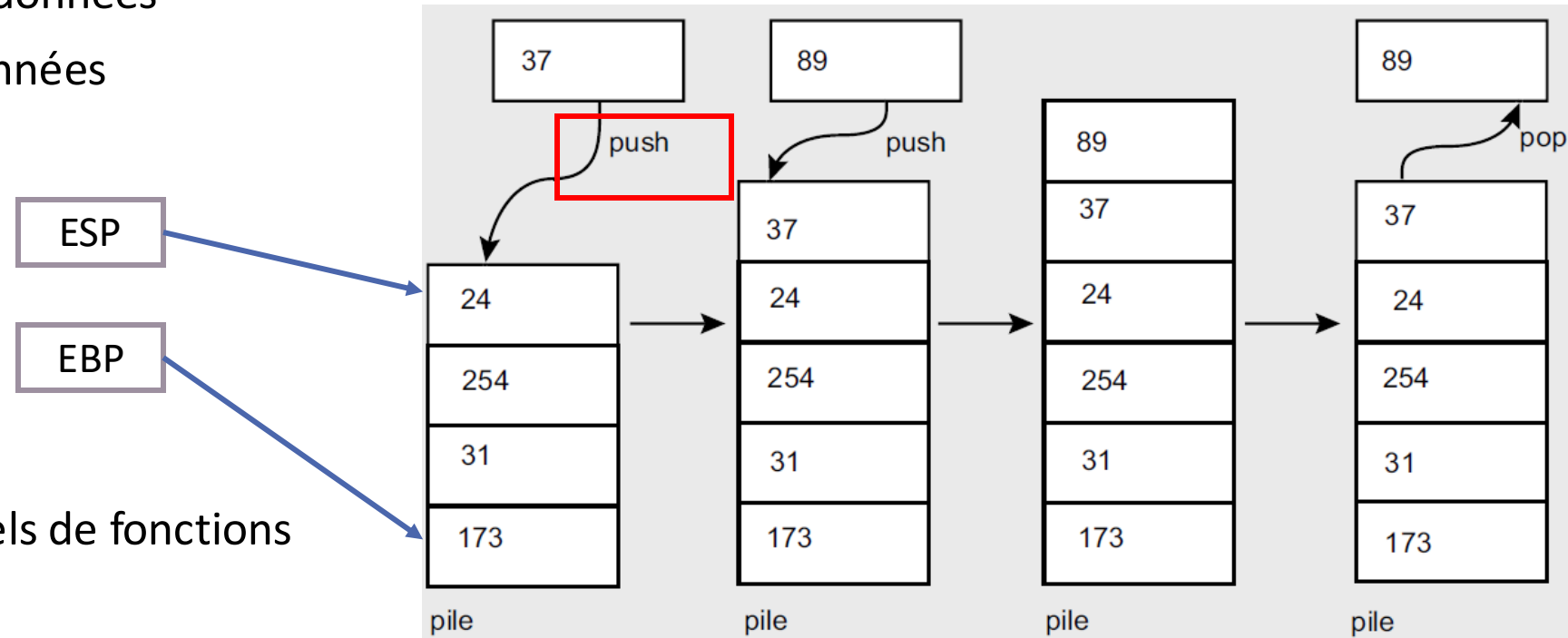
Pile – Élément clés

- Last in, First out
 - La pile d'un programme en exécution est partagée par toutes les fonctions du programme
 - **ESP** (Extended Stack Pointer) – sommet de la pile
 - **EBP** (Extended Base Pointer) – Botton de la pile (base de la pile)
 - **Instruction PUSH** empiler des données
 - **Instruction POP** dépiler des données
-
- Elle est utilisée dans lors d'appels de fonctions avec l'instruction **call**



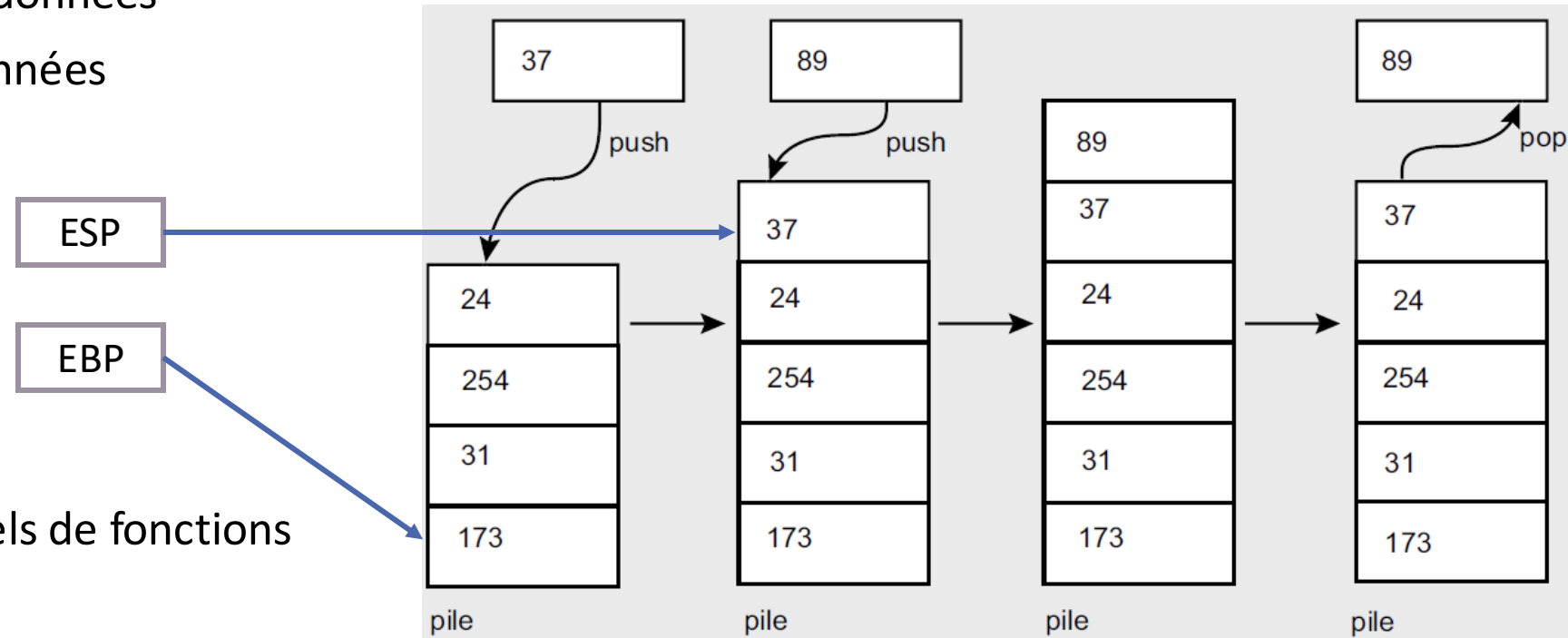
Pile – Élément clés

- Last in, First out
 - La pile d'un programme en exécution est partagée par toutes les fonctions du programme
 - **ESP** (Extended Stack Pointer) – sommet de la pile
 - **EBP** (Extended Base Pointer) – Botton de la pile (base de la pile)
 - **Instruction PUSH** empiler des données
 - **Instruction POP** dépiler des données
-
- Elle est utilisée dans lors d'appels de fonctions avec l'instruction **call**



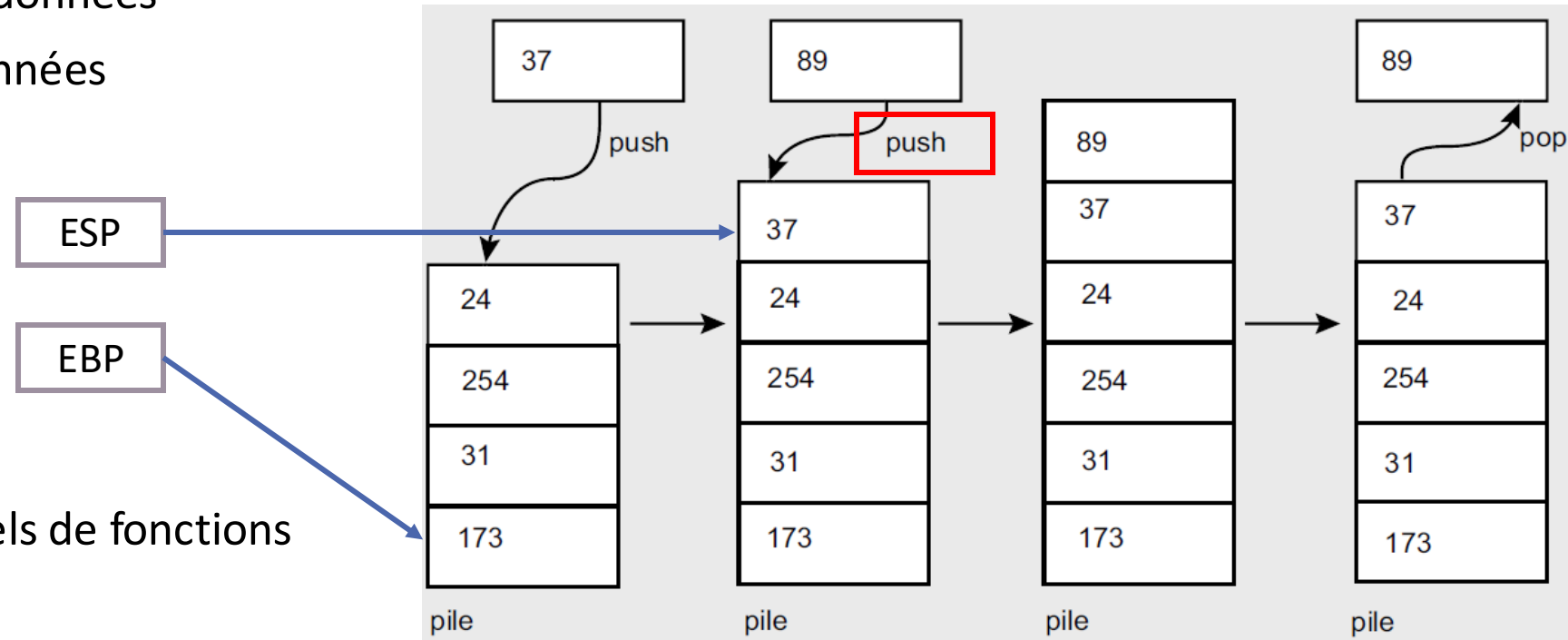
Pile – Élément clés

- Last in, First out
 - La pile d'un programme en exécution est partagée par toutes les fonctions du programme
 - **ESP** (Extended Stack Pointer) – sommet de la pile
 - **EBP** (Extended Base Pointer) – Botton de la pile (base de la pile)
 - **Instruction PUSH** empiler des données
 - **Instruction POP** dépiler des données
-
- Elle est utilisée dans lors d'appels de fonctions avec l'instruction **call**



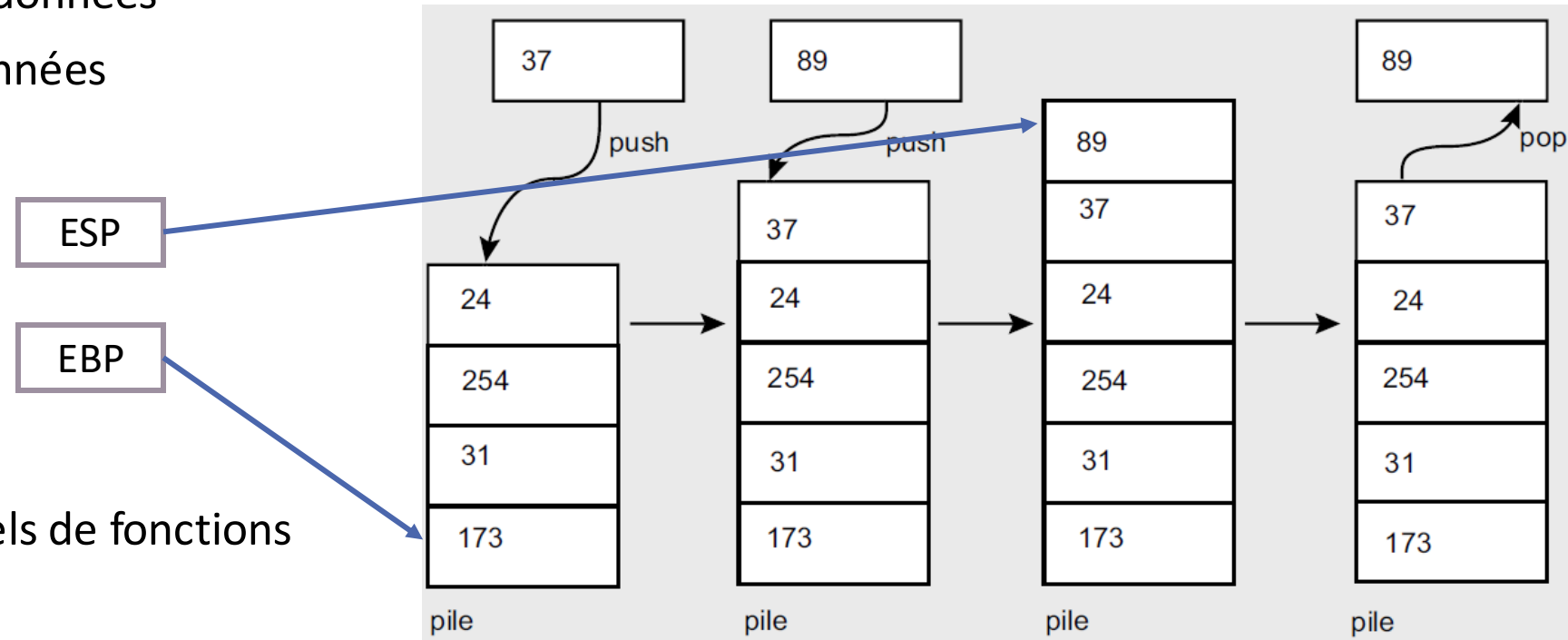
Pile – Élément clés

- Last in, First out
 - La pile d'un programme en exécution est partagée par toutes les fonctions du programme
 - **ESP** (Extended Stack Pointer) – sommet de la pile
 - **EBP** (Extended Base Pointer) – Botton de la pile (base de la pile)
 - **Instruction PUSH** empiler des données
 - **Instruction POP** dépiler des données
-
- Elle est utilisée dans lors d'appels de fonctions avec l'instruction **call**



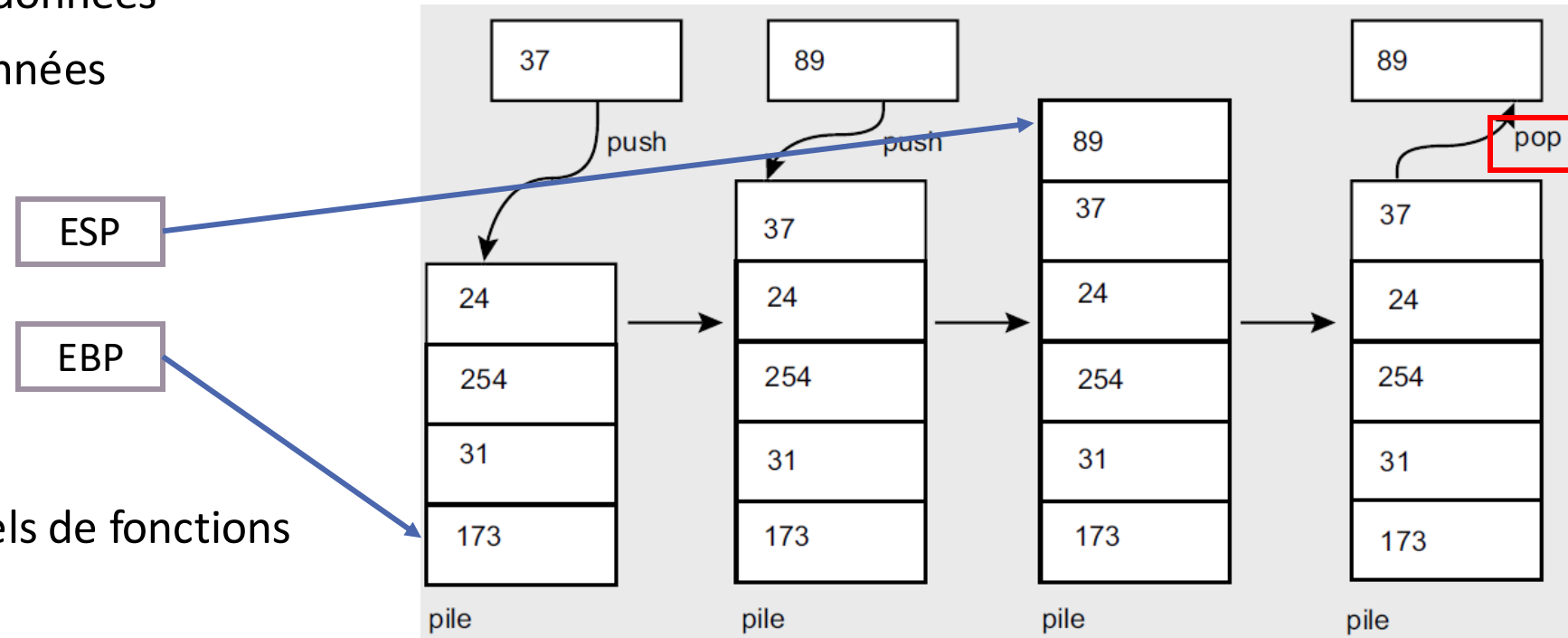
Pile – Élément clés

- Last in, First out
 - La pile d'un programme en exécution est partagée par toutes les fonctions du programme
 - **ESP** (Extended Stack Pointer) – sommet de la pile
 - **EBP** (Extended Base Pointer) – Botton de la pile (base de la pile)
 - **Instruction PUSH** empiler des données
 - **Instruction POP** dépiler des données
-
- Elle est utilisée dans lors d'appels de fonctions avec l'instruction **call**



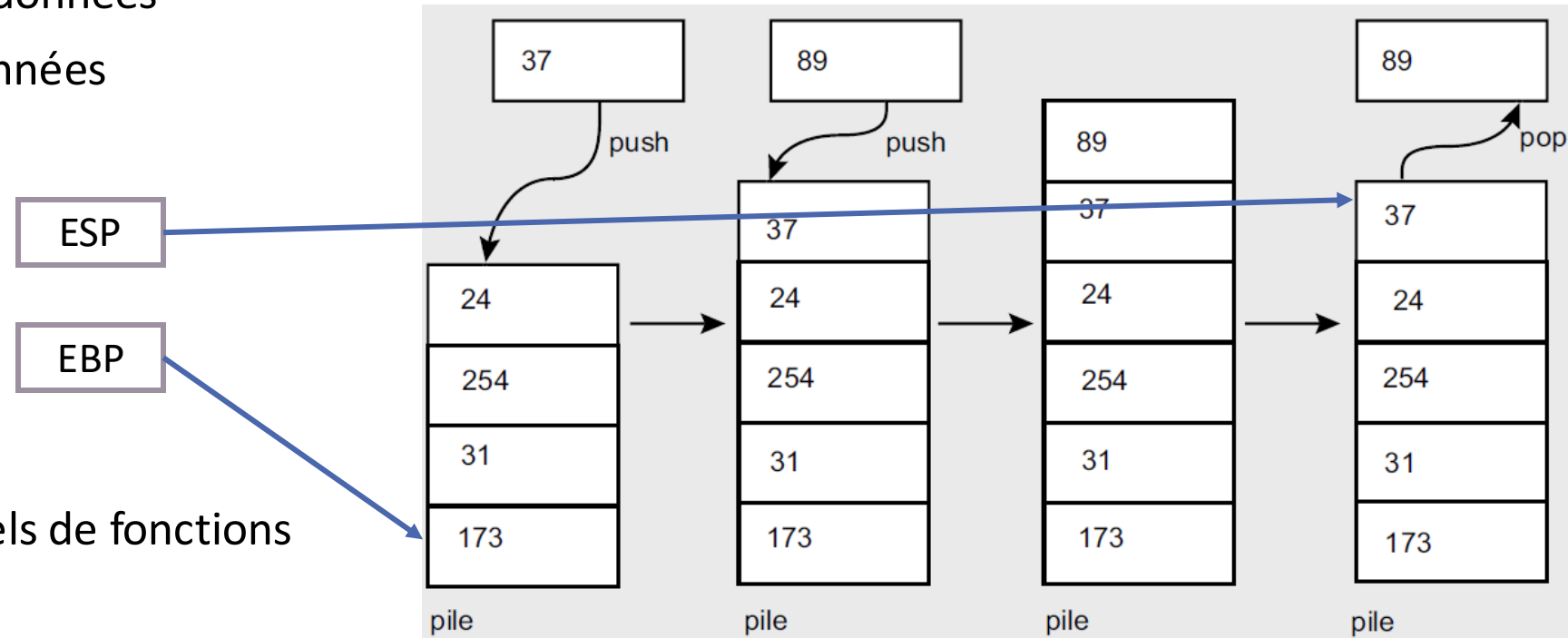
Pile – Élément clés

- Last in, First out
 - La pile d'un programme en exécution est partagée par toutes les fonctions du programme
 - **ESP** (Extended Stack Pointer) – sommet de la pile
 - **EBP** (Extended Base Pointer) – Botton de la pile (base de la pile)
 - **Instruction PUSH** empiler des données
 - **Instruction POP** dépiler des données
-
- Elle est utilisée dans lors d'appels de fonctions avec l'instruction **call**



Pile – Élément clés

- Last in, First out
 - La pile d'un programme en exécution est partagée par toutes les fonctions du programme
 - **ESP** (Extended Stack Pointer) – sommet de la pile
 - **EBP** (Extended Base Pointer) – Botton de la pile (base de la pile)
 - **Instruction PUSH** empiler des données
 - **Instruction POP** dépiler des données
-
- Elle est utilisée dans lors d'appels de fonctions avec l'instruction **call**



Pile – Appel de fonction

A suivre ici

Le résumé Vidéo

Vidéo à regarder

Convention d'appel de fonction

- **cdecl** : les paramètres sont mis dans la pile de la droite vers la gauche et c'est à la fonction appelante de nettoyer la pile à la fin de la fonction appelée. La valeur de retour est sauvegardée dans le registre EAX
- **stdcall**: Même fonctionnement que **cdecl** sauf qu'ici c'est à la fonction appelée de nettoyer la pile à la fin de son exécution
- C'est ce qui est utilisé par défaut par l'API Windows
- **fastcall**: Les premiers paramètres (souvent deux) sont passés à la fonction appelée via les registres (EDX et ECX dans Windows) et le reste obéit aux mêmes règles que la convention cdecl.

```
int test(int x, int y, int z);  
int a, b, c, ret;
```

```
ret = test(a, b, c);
```

```
push c  
push b  
push a  
call test  
add esp, 12  
mov ret, eax
```

```
int test(int x, int y, int z);  
int a, b, c, ret;
```

```
ret = test(a, b, c);
```

```
push c  
push b  
push a  
call test  
mov ret, eax
```
