Assembleur x86 (Nasm)

Introduction

Ce cours est une introduction à l'assembleur X86.

Il vous permettra, en autres, de pouvoir faire de la rétroconception.

Ce cours s'appuie sur les cours suivants :

- https://cs.lmu.edu/~ray/notes/nasmtutorial/
- https://www.lacl.fr/tan/asm
- https://www.tutorialspoint.com/assembly_programming/assembly_system_calls.htm

Rappel sur la génération de code

Architecture X86

rax	registre général, accumulateur, contient la valeur de retour des fonctions
rbx	registre général
rcx	registre général, compteur de boucle
rdx	registre général, partie haute d'une valeur 128 bits
rsi	registre général, adresse source pour déplacement ou comparaison
rdi	registre général, adresse destination pour déplacement ou comparaison
rsp	registre général, pointeur de pile (stack pointer)
rbp	registre général, pointeur de base (base pointer)
r8	registre général
r9	registre général
⋮	
r15	registre général
rip	compteur de programme (instruction pointer)

Dans le cadre notre TP, les seules registres à connaitre sont :

- Le rbp qui contient l'adresse de la frame de la stack courante.
- Le rsp qui contient l'adresse de la pile. C'est-à-dire la première adresse mémoire disponible en RAM pour y stocker des données temporaires.
- Le rip qui contient l'adresse de l'instruction à exécuter.

Architecture

Mémoire volatile :

- RAM - Mémoire rapide

Processeur :

- Registres – Mémoire très rapide

Mémoire Non Volatile :

- Disque Dur – Mémoire lente
- Mémoire Flash – Mémoire lente

Code assembleur x86

Exemple de lecture sur le clavier

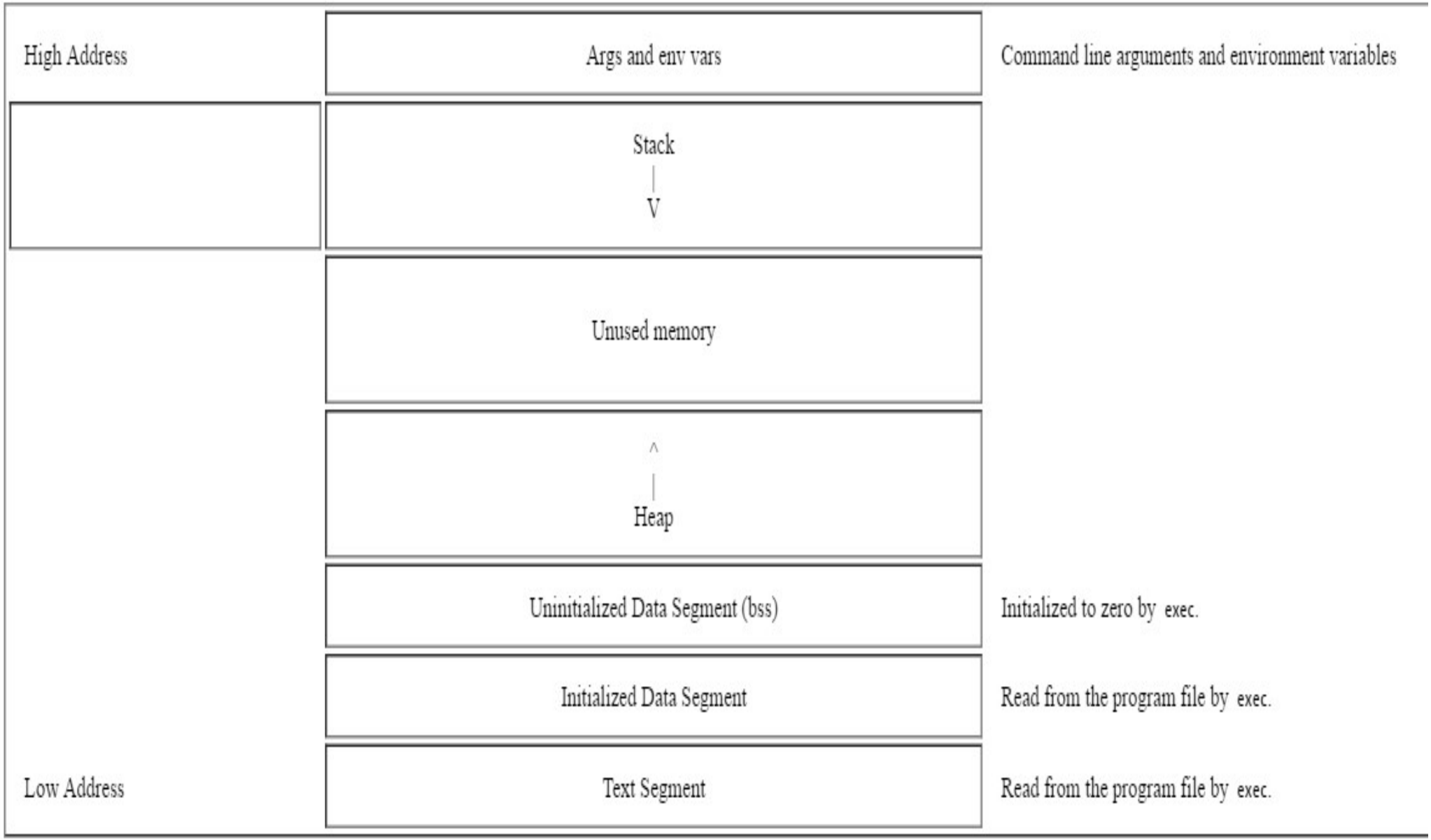
```
# define N 16

        .global _start

        .comm   BUFF    , N

_start: mov     $3      , %eax
        mov     $0      , %ebx
        mov     $BUFF   , %ecx
        mov     $N      , %edx
        int     $0x80

        mov     %eax    , %edx
        mov     $4      , %eax
        mov     $1      , %ebx
        mov     $BUFF   , %ecx
        int     $0x80

        mov     $1      , %eax
        mov     $0      , %ebx
        int     $0x80
```

Le contenu d'un fichier

Un fichier binaire contient les informations suivantes :

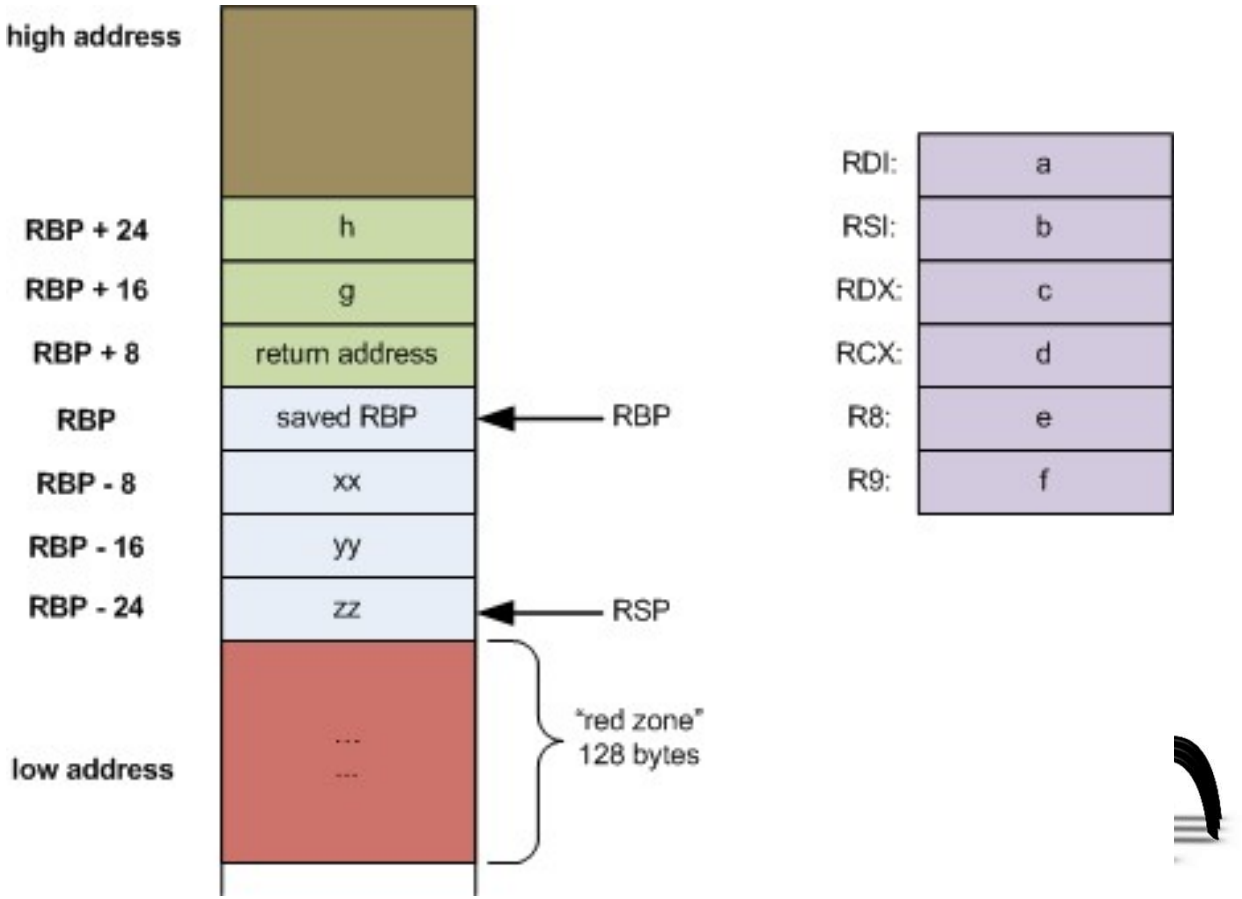
- Métadonnées
- Les différentes sections :
  - rodata : constantes
  - bss : données non initialisées
  - text : code
  - Etc.

L’espace mémoire d’une application



La pile

```
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}
```



L'enchainement des appels de fonctions

```c
int fonction1(int c, int d)
{
      int l_var3 = c+2;
      int l_var4 = d+2;
      fonction2(var3, Var4);

      return 0;
}

int fonction2(int e, int f)
{
      int l_var5 = e+2;
      int l_var6 = e+2;

      return 0;
}

int main(int argc, char *argv[])
{
      int l_var1 = 1;
      int l_var2 = 2;
      fonction1(var1, Var2);

      return 0;
}
```

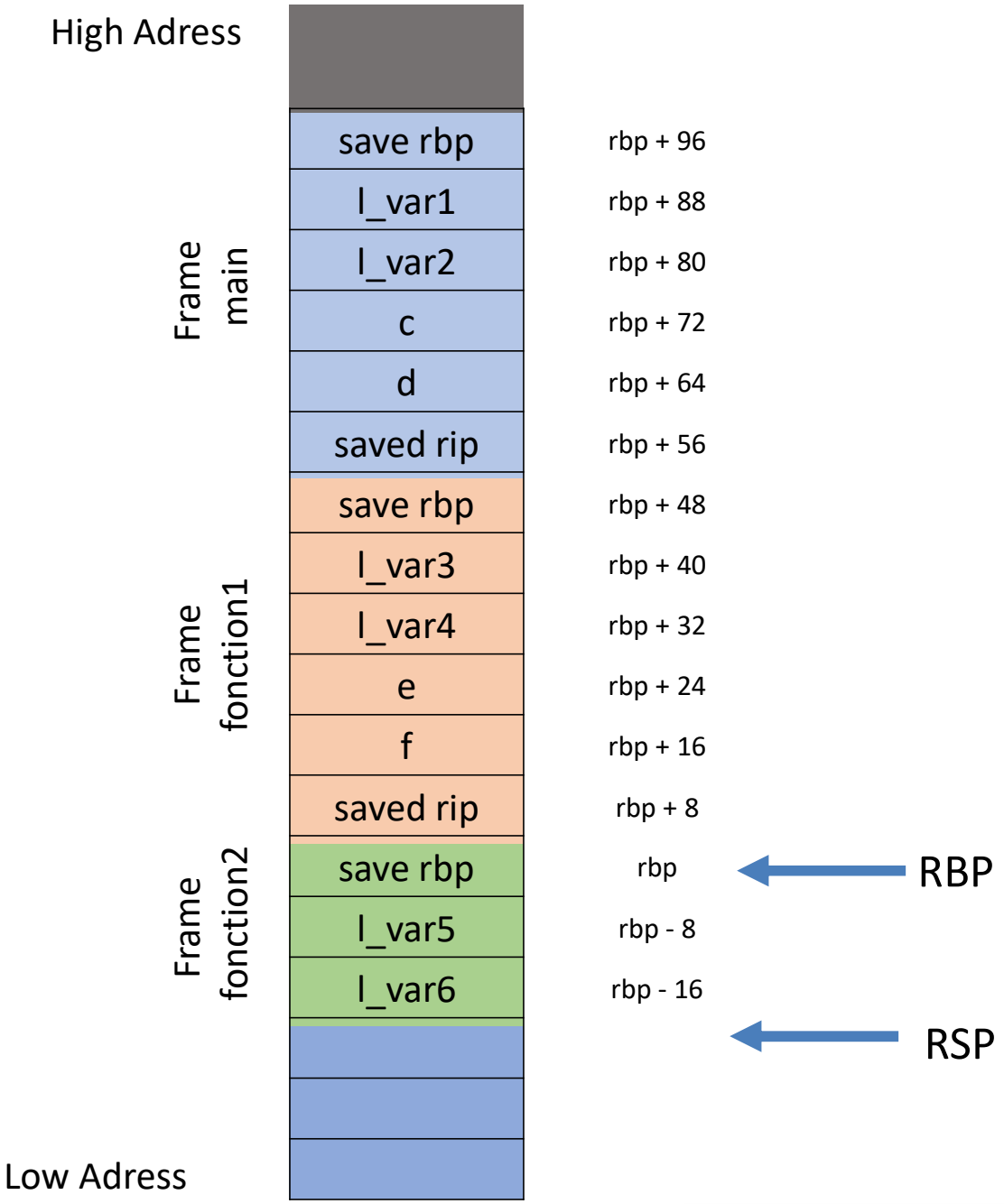
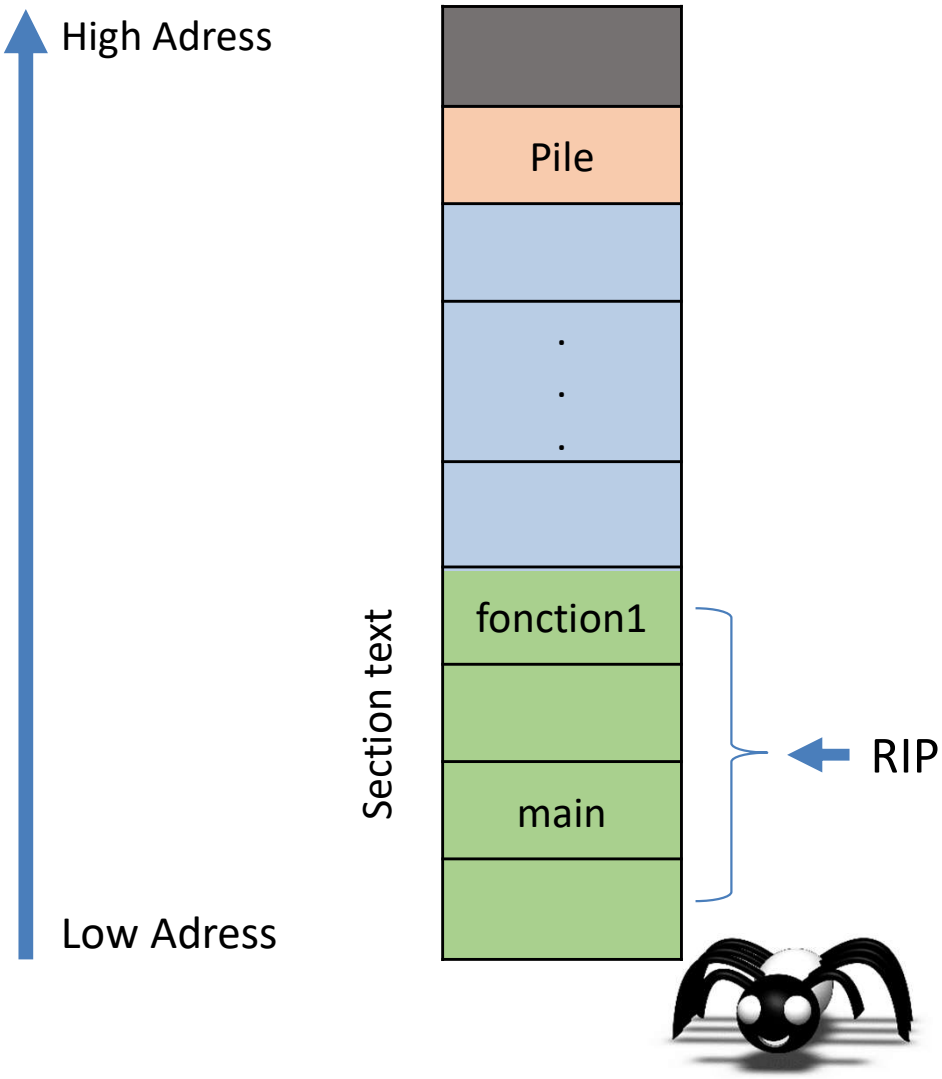


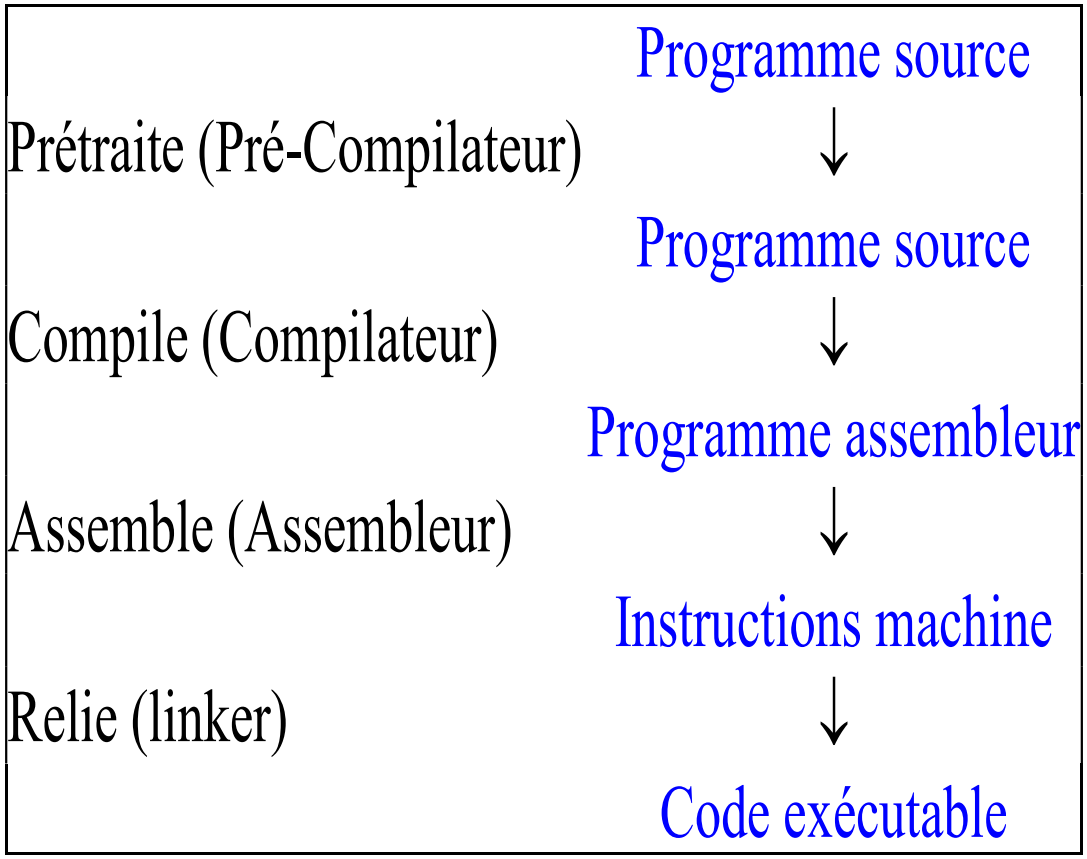
Schéma de la mémoire RAM

```
int fonction1(int c, int d)
{
     int l_var3 = c+2;
     int l_var4 = d+2;
     return l_var3  + l_var4;
}


int main(int argc, char *argv[])
{
     int l_var1 = 1;
     int l_var2 = 2;
     fonction1(var1, Var2);
     return 0;
}
```



Chaine de génération d'un binaire

Pour pouvoir développer des solutions complexes, nous utilisons des langages évolués comme le C, C++, java, Python, Pascal, etc.



L'OS

Dans l'embarqué, il est encore courant de compiler tout le logiciel en un seul fichier et de stocker en mémoire flash. En général, au démarrage seul un logiciel de base est installé et celui permet ensuite de gérer plusieurs logiciels en parallèle, etc.

Le code exécutable est dans ce cas généré dans un format spécifique compréhensible par le système d'exploitation. Celui-ci s'occupe ensuite de gérer le stockage (généralement en RAM) et la gestion des liens dynamiques. Il existe plusieurs types de format (out, COFF, ELF, etc..)

Le compilateur

C’est le compilateur qui gère la transformation du code en langage de haut niveau en langage assembleur.

Il identifie les meilleurs algorithmes pour optimiser les temps d’exécution. C’est lui qui est responsable de la création, de la gestion et de la définition du mécanisme de pile.

Il est totalement possible de faire des logiciels sans pile (stack). C’est cependant, une des solutions les plus performantes et donc mise en place par les compilateurs pour générer le code assembleur.

Dans notre TP, nous travaillerons avec le format ELF.

Les différents types de processeurs

Il existe différents types de processeurs :

- CISC : x86 :
- RISC : SPARC, POWER, PowerPC, MIPS et ARM.

X86 : PC 

ARM : téléphone portable 

- Exemple ASM ARM :

  ADD r0,r1,r2→r0=r1+r2 Addition

  ADC r0,r1,r2→r0=r1+r2+C Addition avec retenue

  SUB r0,r1,r2→r0=r1-r2 Soustraction

Les différentes syntaxes asm

Les différentes syntaxes asm

Il existe différents dialects :

- Yasm : the modular assembler
- Netwide asm : nasm
- Gas: assembleur GNU utilisé avec `gcc`
- Macro asm : assembleur Microsoft
- Turbo asm : assembleur Borland

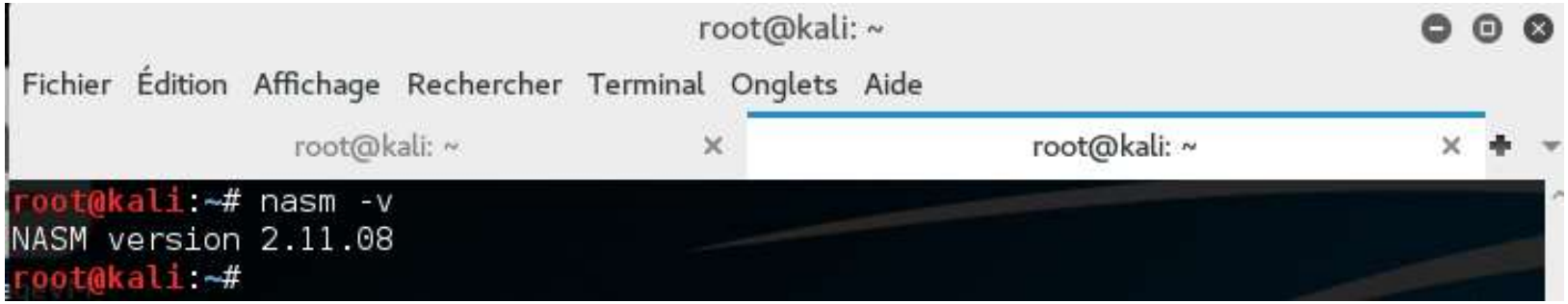
Environnement de développement

NASM

Il existe plusieurs syntaxe assembleur X86 t plusieurs outils pour générer du code assembleur pour architecture x86.

Dans ce cours nous utiliserons l'outil nasm et la syntaxe associée.

Nasm est déjà installé sur la machine kali associée au cours.



Syscall

Syscall

Les syscalls sont des routines offertes par l'OS.

Entre du 64bits et du 32bits, il y a des écarts de notation.

Exemple de d'appels système :

The following code snippet shows the use of the system call sys_exit –

```
mov     eax,1                   ; system call number (sys_exit)
int     0x80                    ; call kernel
```

The following code snippet shows the use of the system call sys_write –

```
mov     edx,4                   ; message length
mov     ecx,msg                 ; message to write
mov     ebx,1                   ; file descriptor (stdout)
mov     eax,4                   ; system call number (sys_write)
int     0x80                    ; call kernel
```

Tableau des syscalls

Tableaux des syscalls :

- Les conventions change en fonction de windows et linux.

The following table shows some of the system calls used in this tutorial –

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

Premier exemple

Premier exemple

Ex :

hello.asm

```
; ------------------------------------------------------------------------
; Writes "Hello, World" to the console using only system calls. Runs on 64-bit Linux only.
; To assemble and run:
;
;     nasm -felf64 hello.asm && ld hello.o && ./a.out
; ------------------------------------------------------------------------

          global    _start

          section   .text
_start:   mov       rax, 1                  ; system call for write
          mov       rdi, 1                  ; file handle 1 is stdout
          mov       rsi, message            ; address of string to output
          mov       rdx, 13                 ; number of bytes
          syscall                           ; invoke operating system to do the write
          mov       rax, 60                 ; system call for exit
          xor       rdi, rdi                ; exit code 0
          syscall                           ; invoke operating system to exit

          section   .data
message:  db        "Hello, World", 10      ; note the newline at the end
```

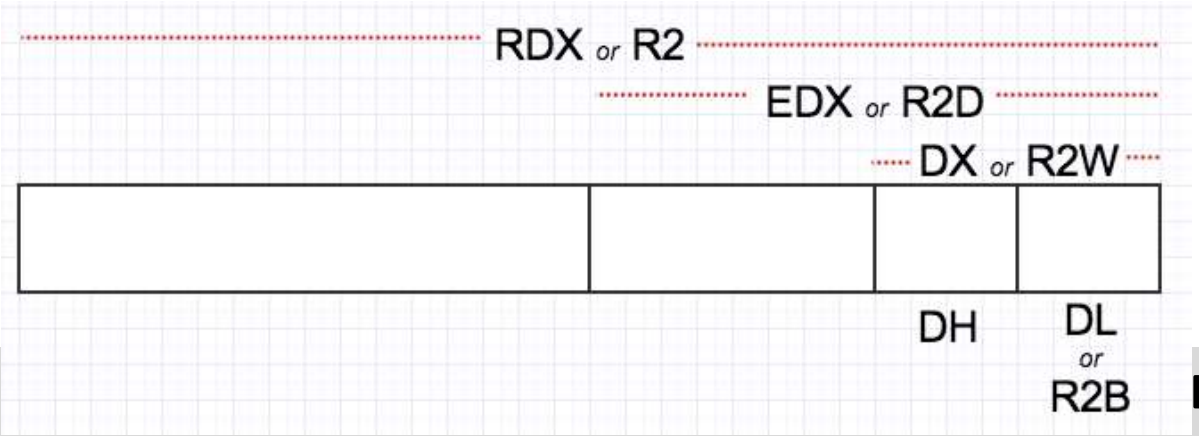
```
$ nasm -felf64 hello.asm && ld hello.o && ./a.out
Hello, World
```

Les regsitres

Registres

Le nom des registres :

Taille des registres	Nom des regsitres
64 bits	R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 RAX RCX RDX RBX RSP RBP RSI RDI
32 bits	R0D R1D R2D R3D R4D R5D R6D R7D R8D R9D R10D R11D R12D R13D R14D R15D EAX ECX EDX EBX ESP EBP ESI EDI
16 bits	R0W R1W R2W R3W R4W R5W R6W R7W R8W R9W R10W R11W R12W R13W R14W R15W AX CX DX BX SP BP SI DI
8 bits	R0B R1B R2B R3B R4B R5B R6B R7B R8B R9B R10B R11B R12B R13B R14B R15B AL CL DL BL SPL BPL SIL DIL
8bits de poids fort d'un mot de 16 bits	AH CH DH BH



Les instructions

Instructions

Il y a des centaines d'instructions en X86.

En voici quelques unes :

mov *x*, *y*	$x \leftarrow y$
and *x*, *y*	$x \leftarrow x$ and y
or *x*, *y*	$x \leftarrow x$ or y
xor *x*, *y*	$x \leftarrow x$ xor y
add *x*, *y*	$x \leftarrow x + y$
sub *x*, *y*	$x \leftarrow x - y$
inc *x*	$x \leftarrow x + 1$
dec *x*	$x \leftarrow x - 1$
syscall	Invoke an operating system routine
db	A pseudo-instruction that declares bytes that will be in memory when the program runs

Définir des données et réserver de l'espace

Définir des données

Définir des données :

```
db    0x55                ; just the byte 0x55
db    0x55,0x56,0x57      ; three bytes in succession
db    'a',0x55            ; character constants are OK
db    'hello',13,10,'$'   ; so are string constants
dw    0x1234              ; 0x34 0x12
dw    'a'                 ; 0x61 0x00 (it's just a number)
dw    'ab'                ; 0x61 0x62 (character constant)
dw    'abc'               ; 0x61 0x62 0x63 0x00 (string)
dd    0x12345678          ; 0x78 0x56 0x34 0x12
dd    1.234567e20         ; floating-point constant
dq    0x123456789abcdef0  ; eight byte constant
dq    1.234567e20         ; double-precision float
dt    1.234567e20         ; extended-precision float
```

Typiquement ces données sont dans la section .data

Réserver de l'espace mémoire

Réserver de l'espace mémoire :

```
buffer:         resb    64              ; reserve 64 bytes
wordvar:        resw    1               ; reserve a word
realarray:      resq    10              ; array of ten reals
```

Typiquement ces données sont dans la section .bss

Les constantes

Les constantes :

- “equ” n’est pas une instruction. C’est un label pour accéder à la donnée.

Opération d’adresse

Opération d'adressage

Les formes basiques d'adressage :

- [number]
- [reg]
- [reg + reg*scale] scale is 1, 2, 4, or 8 only
- [reg + number]
- [reg + reg*scale + number]

Vocabulaire :

- "number" est appelé le déplacement;
- Le registre à gauche est la base
- Le registre avec le facteur de multiplication est appelé l'index.

Opération d'adressage

Exemples :

```
[750]                ; displacement only
[rbp]                ; base register only
[rcx + rsi*4]        ; base + index * scale
[rbp + rdx]          ; scale is 1
[rbx - 8]            ; displacement is -8
[rax + rdi*8 + 500]  ; all four components
[rbx + counter]      ; uses the address of the variable 'counter' as the displacement
```

Comparaison et jump

Les instructions

Les instructions :

- “cmp” fait une comparaison
- “je” saute au label si la comparaison précédente était égale.
- “jne” saute au label si la comparaison précédente n’était pas égale (jump if not equal),
- “jl” (jump if less),
- “jnl” (jump if not less),
- “jg” (jump if greater),
- “jng” (jump if not greater),
- “jle” (jump if less or equal),
- “jnle” (jump if not less or equal),
- “jge” (jump if greater or equal),
- “jnge” (jump if not greater or equal), and many more.

Jump et comparaison

Exemple

```
        global    start
        section   .text
start:
        mov       rdx, output           ; rdx holds address of next byte to write
        mov       r8, 1                 ; initial line length
        mov       r9, 0                 ; number of stars written on line so far
line:
        mov       byte [rdx], '*'       ; write single star
        inc       rdx                   ; advance pointer to next cell to write
        inc       r9                    ; "count" number so far on line
        cmp       r9, r8                ; did we reach the number of stars for this line?
        jne       line                  ; not yet, keep writing on this line
lineDone:
        mov       byte [rdx], 10        ; write a new line char
        inc       rdx                   ; and move pointer to where next char goes
        inc       r8                    ; next line will be one char longer
        mov       r9, 0                 ; reset count of stars written on this line
        cmp       r8, maxlines          ; wait, did we already finish the last line?
        jng       line                  ; if not, begin writing this line
```

Jump et comparaison

Exemple :

```
$ nasm -fmacho64 triangle.asm && ld triangle.o && ./a.out
*
**
***
****
*****
******
*******
********
```

Utilisation de la pile

Utilisation de la pile

Les deux commandes pour utiliser la pile :

- push reg : copie le contenu de reg dans la pile à l'adresse contenue dans rsp
- pop reg : copie dans reg le contenu du dernier élément de la pile contenue à rsp – 8 (64 bits)

« Push reg » décremente rsp

« Pop reg » incémente rsp

Appel de fonction

La fonction call

Pour appeler des fonctions : call

hola.asm

```
; ------------------------------------------------------------------------------
; Writes "Hola, mundo" to the console using a C library. Runs on Linux.
;
;     nasm -felf64 hola.asm && gcc hola.o && ./a.out
; ------------------------------------------------------------------------------

          global    main
          extern    puts

          section   .text
main:                                   ; This is called by the C library startup code
          mov       rdi, message        ; First integer (or pointer) argument in rdi
          call      puts                ; puts(message)
          ret                           ; Return from main back into C library wrapper
message:
          db        "Hola, mundo", 0    ; Note strings must be terminated with 0 in C
```

```
$ nasm -felf64 hola.asm && gcc hola.o && ./a.out
Hola, mundo
```

Call convention

Call convention

Microsoft x64 calling convention

The Microsoft x64 calling convention is followed on Windows and pre-boot UEFI (for long mode on x86-64).

The first four arguments are placed onto the registers. That means RCX, RDX, R8, R9 for integer, struct or pointer arguments (in that order), and XMM0, XMM1, XMM2, XMM3 for floating point arguments. Additional arguments are pushed onto the stack (right to left).

Integer return values (similar to x86) are returned in RAX if 64 bits or less. Floating point return values are returned in XMM0. Parameters less than 64 bits long are not zero extended; the high bits are not zeroed.

The registers RAX, RCX, RDX, R8, R9, R10, R11 are considered volatile (caller-saved).

The registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15 are considered nonvolatile (callee-saved).

Call convention

System V AMD64 ABI

The calling convention of the System V AMD64 ABI is followed on Solaris, Linux, FreeBSD, macOS,[23] and is the de facto standard among Unix and Unix-like operating systems.

The first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, R9 (R10 is used as a static chain pointer in case of nested functions[25]:21), while XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and XMM7 are used for the first floating point arguments.

As in the Microsoft x64 calling convention, additional arguments are passed on the stack.[25]:22 Integer return values up to 64 bits in size are stored in RAX while values up to 128 bit are stored in RAX and RDX.

Floating-point return values are similarly stored in XMM0 and XMM1. The wider YMM and ZMM registers are used for passing and returning wider values in place of XMM when they exist.[25]:26,55

If the callee wishes to use registers RBX, RSP, RBP, and R12–R15, it must restore their original values before returning control to the caller. All other registers must be saved by the caller if it wishes to preserve their values.

Call convention

https://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_Calling_Conventions

Analyse statique du logiciel

E Code Assembleur

Dans cette partie, nous allons nous familiarisé avec le langage assembleur. Le but est de voir à quoi ressemble un code écrit en assembleur. Il existe plusieurs syntaxes pour le langage assembleur x86. Nous utiliserons celle de nasm.

Exercice 1 :

- Ecrire notre premier logiciel en X86 pour linux qui affiche Hello Word avec NASM.

Astuce : Une petite recherche sur internet…

Exercice 2 :

Analyser le fichier précèdent avec les outils readelf ou objdump :

- Q1 : Pour quel type de processeur est le binaire?
- Q2 : Pour quelle type d'OS ?
- Q3 : A quoi correspondent en ASCCI les premières valeurs du mot magique?
- Q4 : Quelle est l'adresse d'entrée du logiciel ?

E Exercice

Faire les challenges de Retro 1 à 5 sur https://syd-academy.secyourdev.com

Liens utiles

Nasm

Liens :

https://www.tutorialspoint.com/assembly_programming/assembly_system_calls.htm

https://cs.lmu.edu/~ray/notes/nasmtutorial/

https://www.lacl.fr/tan/asm

https://openclassrooms.com/fr/courses/2288321-apprenez-a-programmer-en-assembleur-x86/2288775-introduction-installation

https://www.supinfo.com/cours/1CPA/chapitres/09-assembleur-x86