

INFO-F105 – Langages de programmation 1

Assembleur

1 Introduction

Un *langage assembleur* (*assembly language*) permet de représenter les instructions machines (du binaire) sous forme textuelle. On peut donc le voir comme un intermédiaire entre C++ et le langage machine utilisé dans les fichiers objets et exécutables.

Il est important de savoir qu’il existe plusieurs langages assembleurs dédiés à différentes architectures de processeur.

Pour ce cours, nous utiliserons exclusivement le **langage assembleur x86** de Intel^[1], qui est compatible nativement avec tous les processeurs Intel et AMD modernes mais pas avec les processeurs ARM (Apple Silicon compris).

1.1 Programmer en assembleur

Les fichiers d’assembleur ont généralement l’extension `.s`, `.S` ou `.asm`.

Pour compiler un code assembleur, il faut un outil qu’on appelle *assembleur* (*assembler*). Pour ce cours, nous utiliserons *NASM*^[2].

Vous pouvez utiliser l’IDE *SASM*^[3] pour compiler et déboguer votre code assembleur.

2 Memory layout

Le terme *mémoire* comprend toutes les zones où sont stockées des données :

- Disques durs
- RAM
- Registres

Le terme *mémoire principale* comprends à la fois la RAM et la *mémoire virtuelle* (*swap*)¹.

Il est essentiel de comprendre l’organisation de la mémoire principale et des registres à la fois pour programmer en assembleur et pour comprendre les fondements du langage C++.

¹ Voir cours de fonctionnement des ordinateurs

2.1 Segmentation

La mémoire principale est segmentée en plusieurs *segments*, qui ont chacun un rôle différent :

Segment	Contenu
<code>.text</code>	Code (instructions)
<code>.rodata</code>	Contantes globales (<i>read-only</i>)
<code>.data</code>	Variables globales initialisées
<code>.bss</code>	Variables globales non initialisées
Stack	Stack
Heap	Heap

Les lignes commençant par `section` délimitent les différentes sections du programme.

```
1 section .text
2 mov eax, 2
3 section .rodata
4 ptr1 db 5
5 section .data
6 ptr2 db 1
7 ptr3 dw 500
8 section .bss
9 ptr3 resb 1
```

Une fois compilé, votre code sera encodé en binaire et sera chargé dans la mémoire principale à l'exécution. Vous pouvez donc voir votre programme comme une portion de mémoire qui contient ces segments mis bout à bout.

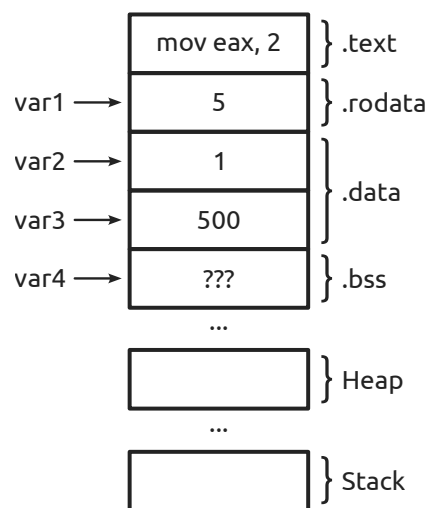


Figure 1: Exemple de segmentation de la mémoire

Les segments du stack et du heap sont définis à l'exécution par le système d'exploitation.

2.2 Registres

Les *registres* sont des petites zones mémoire très rapides situées à l'intérieur même du processeur et qui sont utilisées pour faire des opérations sur des données.

General-purpose

Il existe 6 registres qui peuvent être utilisés librement.

Registre	Nom anglais
eax	Accumulator
ebx	Base (pointer)
ecx	Counter
edx	Data
esi	Source Index
edi	Destination Index

Bien que ces registres soient généraux, ils ont tout de même des noms suggestifs.

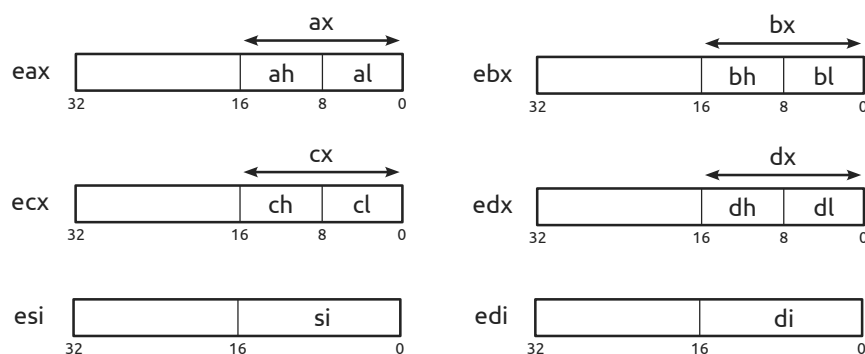


Figure 2: Registres généraux

Chacun de ces registres contient 32 bits au total mais peut être vu comme un registre plus petit en utilisant l'alias correspondant, comme on peut le voir sur la figure 2.

Program counter

Le *Program Counter* (PC), aussi appelé *Instruction Pointer* (IP), est un registre qui contient l'adresse de l'instruction en cours d'exécution.

Ce registre est automatiquement incrémenté à la fin de chaque instruction pour passer à la suivante, mais il est possible de changer l'exécution linéaire à l'aide d'un *jump* (cf. section 5).

Stack

La gestion du stack (cf. section 6.1) se fait à l'aide de deux autres registres :

- Stack pointer **esp**
- Base pointer **ebp** (\neq ebx)

Flags

Les *flags* (cf. section 5.1) sont enregistrés dans un registre dédié.

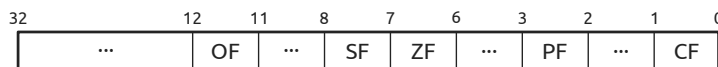


Figure 3: Flag register

3 Instructions

Les *instructions*^[1] sont des opérations très simples qui indiquent au processeur quoi faire. Elles sont composées d'un *opcode* et parfois d'une ou plusieurs *opérandes* (\approx paramètres).

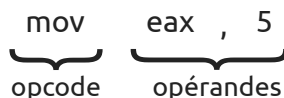


Figure 4: Exemple d'instruction

Rappel : les instructions vont dans la section `.text`

3.1 Compilation

Lors de la compilation, les instructions d'assembleur sont traduites en *langage machine*.

Par exemple, voici comment se traduit l'instruction de la figure 4 sur une architecture x86 :

```
b8 05 00 00 00
```

```
--
```

```
^-> Opcode pour l'instruction MOV sur le registre eax
```

```
^-> Opérande 5 sur 32 bits en little-endian
```

On remarque que l'opérande est encodée en *little-endian* et ses *bytes* (8 bits) sont donc inversés. C'est un choix d'architecture commun à tous les processeurs Intel et AMD, mais aussi à ceux de Apple et de nombreux autres processeurs ARM.

3.2 Modes d'adressage

Une instruction peut parfois supporter plusieurs types d'opérandes. Chaque variante de cette même instruction s'écrit de la même manière mais compilera avec un opcode binaire différent.

```
1 section .text
2 mov edx, 5          ; edx = 5
3 mov edx, eax        ; edx = eax
4 mov edx, [ebx]      ; edx = *ebx
5 mov dword[ebx], 5   ; *ebx = 0x00000005
6 mov [ebx], edx      ; *ebx = edx
```

3.3 Liste d'instructions

Vous trouverez une liste d'instructions utiles et leurs significations sur l'UV.

3.4 Exemple

Calculer la surface d'un rectangle de longueur $L = 555$ et de largeur $l = 333$

```
1 mov eax, 555 ; eax = 555
2 mov ebx, 333 ; ebx = 333
3 mul bx      ; dx:ax = ax * bx
4 shl edx, 16 ; edx <<= 16
5 or eax, edx ; eax /= edx
```

Prenez bien le temps de comprendre comment fonctionne le code ci-dessus.

4 Données

Il n'y a pas de variable à proprement parler en assembleur. Deux alternatives :

- *Immediate* est le mot anglais pour parler d'une valeur littérale.

```
mov eax, 5      ; 0x00000005
mov ebx, 0x100  ; 0x00000100
mov ecx, 0b1000 ; 0x00000008
```

Ces valeurs sont sur 32 bits puisque `mov` est utilisé sur des registres de 32 bits.

- Il est aussi possible de définir des pointeurs, comme nous allons le voir.

4.1 Définition

Il n'y a **pas** de type en assembleur : les registres et la mémoire contiennent uniquement des données binaires accessibles depuis leurs adresses (= pointeurs `void*`).

Il faut cependant toujours préciser la taille des données.

Nom anglais	Symbole	.bss	Taille
Byte	db	resb	8 bits
Word	dw	resw	16 bits
Dual word	dd	resd	32 bits
Quad word	dq	resq	64 bits

```
1 section .data                                1 section .bss
2 ptr1 db 0x12                                2 cptr1 resb 1 ; 1 byte
3 ptr2 dw 0x1234                                3 cptr2 resw 2 ; 2 words
4 ptr3 dd 0x12345678                            4 cptr3 resd 1 ; 1 dword
5 ptr4 dq 0x123456789abcdef0                    5 cptr4 resq 2 ; 2 qwords
```

4.2 Utilisation

L'opérateur `[]` permet de déréférencer le pointeur et donc d'accéder à la donnée stockée à cette adresse. Là aussi il faut préciser sa taille.

Symbole	Taille
<code>byte</code>	8 bits
<code>word</code>	16 bits
<code>dword</code>	32 bits
<code>qword</code>	64 bits

```
1 section .text
2 mov edx, dword[ptr1] ; edx = *ptr1
3 mov dword[ptr1], edx ; *ptr1 = edx
4 mov dword[ptr1], 5 ; *ptr1 = 5
```

Note : le compilateur peut dans certains cas déterminer lui-même la taille de la donnée, par exemple lorsqu'elle est utilisée avec un registre de taille définie.

```
5 mov edx, [ptr1] ; edx = *ptr1
6 mov [ptr1], edx ; *ptr1 = edx
```

4.3 Tableaux

Il est possible de définir plusieurs valeurs qui se suivent en mémoire (= array).

```
1 section .data
2 arr dw 1, 2, 3, 4, 5 ; arr[5] = { 1, 2, 3, 4, 5 }
3 section .text
4 mov dx, [arr+1*2] ; dx = arr[1]
5 mov [arr+1*2], dx ; arr[1] = dx
6 mov word[arr+1*2], 5 ; arr[1] = 5
```

La syntaxe `[arr+x*y]` permet d'accéder au $x^{\text{ème}}$ élément de y bytes du tableau `arr`.



Figure 5: Tableau

Il est commun de passer par des registres pour accéder aux éléments d'un tableau :

- `ebx` contient généralement l'adresse du tableau
- `esi` contient généralement l'indice dans le tableau

```
1 mov ebx, arr
2 mov esi, 1
3 mov dx, [ebx + esi]
```

5 Jumps

Un *jump* est une opération qui modifie le *Program Counter* (PC) et donc le fil d'exécution du programme. Cela peut être utile pour implémenter une condition ou une boucle.

5.1 Flags

Les *flags* sont des bits qui indiquent si l'opération précédente a provoqué certains événements et qui permettent de conditionner certaines instructions (cf. section 5.3).

Flag	Évènement
Carry (CF)	Overflow non signé
Overflow (OF)	Overflow signé
Parity (PF)	Résultat pair
Sign (SF)	Résultat négatif
Zero (ZF)	Résultat nul

Notez bien la différence entre *carry flag* (CF) et *overflow flag* (OF).


Non signé									Signé									
		1	1	1	1	1	1	1			0	1	0	0	0	0	0	0
+		0	0	0	0	0	0	0	1	+	0	1	0	0	0	0	0	0
=		0	0	0	0	0	0	0	0	=	1	0	0	0	0	0	0	0

Figure 6: Overflow non signé et signé sur 8 bits

Les flags sont mis à jour par certaines instructions, comme `add` ou `sub`.

5.2 Sauts inconditionnels

Un *label* est un pointeur (une adresse) vers une instruction.

L'instruction `jmp` permet de faire un saut inconditionnel vers un label dans le code.

```
1 main:
2     jmp label    ; pc = label
3     ; Code 1
4 label:
5     ; Code 2
```

Dans cet exemple, seul le code 2 sera exécuté car l'instruction `jmp` va modifier PC pour arriver au label `label` sans passer par le code 1.

5.3 Conditions

Il existe également toute une liste d'instructions de sauts conditionnels (`je`, `jz`, `je`, ...) que vous retrouverez avec les autres instructions d'assembleur sur l'UV.

Ces instructions sont souvent utilisées avec `cmp` ou `test` pour implémenter des conditions :

- `cmp` modifie les flags comme s'il s'agissait d'un `sub`
- `test` modifie les flags comme s'il s'agissait d'un `and`

	<i>; NASM</i>	<i>// C++</i>
1	<code>main:</code>	<code>if (eax > edx) {</code>
2	<code> cmp eax, edx</code>	<code> // Code 2</code>
3	<code> ja above</code>	<code>} else {</code>
4	<code> ; Code 1</code>	<code> // Code 1</code>
5	<code> jmp end</code>	<code>}</code>
6	<code>above:</code>	
7	<code> ; Code 2</code>	
8	<code>end:</code>	

Le saut inconditionnel `jmp` à la ligne 5 est nécessaire pour ne pas exécuter le code 2 une fois le code 1 terminé. Il a le même rôle qu'un `break` dans un `switch case`.

5.4 Boucles

Les boucles sont construites à l'aide d'un saut vers une instruction précédente.

	<i>; NASM</i>	<i>// C++</i>
1	<code>main:</code>	<code>for (size_t i = 5; i != 0; --i) {</code>
2	<code> mov ecx, 5</code>	<code> // Code</code>
3	<code>loop:</code>	<code>}</code>
4	<code> ; Code</code>	
5	<code> dec ecx</code>	
6	<code> jnz loop</code>	
7	<code>end:</code>	

Le compteur `ecx` est décrémenté ($5 \rightarrow 0$) de manière à pouvoir profiter de l'instruction `jnz`.

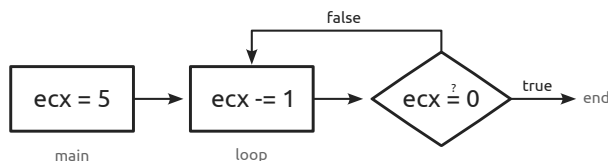


Figure 7: Diagramme de la boucle `loop`

Exercice : essayez de le faire dans l'autre sens : `ecx = 0 \rightarrow 5`

6 Fonctions

Il n'y a pas de fonction à proprement parler en assembleur. Il s'agit d'une notion conceptuelle construite à l'aide de jumps et du stack.

6.1 Stack

Le stack (cf. *Pointeurs & Tableaux*) peut être manipulé à l'aide de deux instructions :

```
1 push eax ; stack[++esp] = eax
2 pop eax  ; eax = stack[esp--]
```

Le *Stack Pointer* `esp` contient l'adresse de la dernière valeur ajoutée sur le stack.

On utilise deux instructions pour “appeler” et quitter une fonction :

- `call` enregistre l'adresse de retour sur le stack et saute à l'adresse indiquée
- `ret` saute à l'adresse de retour précédemment enregistrée sur le stack

Stack frame

Chaque fonction utilise une partie dédiée du stack qu'on appelle *stack frame*.

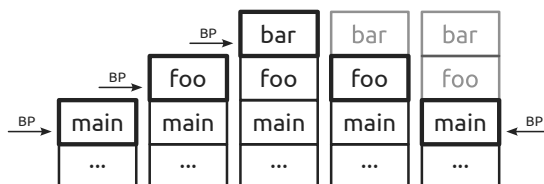


Figure 8: Pile de fonctions

Le *Base Pointer* (BP) `ebp` (\neq `ebx`) indique l'adresse du *stack frame* de la fonction actuelle.

	<i>; NASM</i>	<i>// C++</i>
1	<code>section .text</code>	
2	<code>global main</code>	<code>void main() {</code>
		<code>foo();</code>
2	<code>main:</code>	<code>}</code>
3	<code>call foo ; push pc+1</code>	
4	<code>; jmp foo</code>	
5	<code>bar:</code>	<code>void bar() {</code>
6	<code>ret ; pop edx</code>	<code>return;</code>
7	<code>; jmp edx</code>	<code>}</code>
8	<code>foo:</code>	
9	<code>call bar ; push pc+1</code>	<code>void foo() {</code>
10	<code>; jmp bar</code>	<code>bar();</code>
11	<code>ret ; pop edx</code>	<code>}</code>
12	<code>; jmp edx</code>	

Le symbole global `main` est reconnu par le *linker* comme symbole d'entrée du programme.

6.2 Variables locales

Lorsque c'est possible, il vaut mieux passer par des registres car c'est à la fois plus simple et plus performant. Mais il est parfois nécessaire de mettre des valeurs sur le stack.

L'instruction `enter` met à jour `ebp` et réserve de l'espace (en bytes) pour les variables locales, ce qui permet aux variables d'être accessibles à partir de `[ebp-x]` (cf. figure 9).

L'instruction `leave` restaure les valeurs initiales de `ebp` et `esp`.

	<i>; NASM</i>	<i>// C++</i>
1	<code>foo:</code>	
2	<code>enter 7,0</code>	<code>; push ebp</code>
3		<code>; mov ebp, esp</code>
4		<code>; sub esp, 7</code>
		<code>void foo() {</code>
5	<code>mov [ebp-1], 0x11</code>	<code>uint8_t a = 0x11;</code>
6	<code>mov [ebp-5], 0x11223344</code>	<code>uint32_t b = 0x11223344;</code>
7	<code>mov [ebp-7], 0x1122</code>	<code>uint16_t c = 0x1122;</code>
8	<code>leave</code>	<code>; mov esp, ebp</code>
9		<code>; pop ebp</code>
10	<code>ret</code>	<code>return c;</code>

Dans cet exemple, la fonction `foo` réserve 7 bytes pour les variables locales `a`, `b` et `c`.

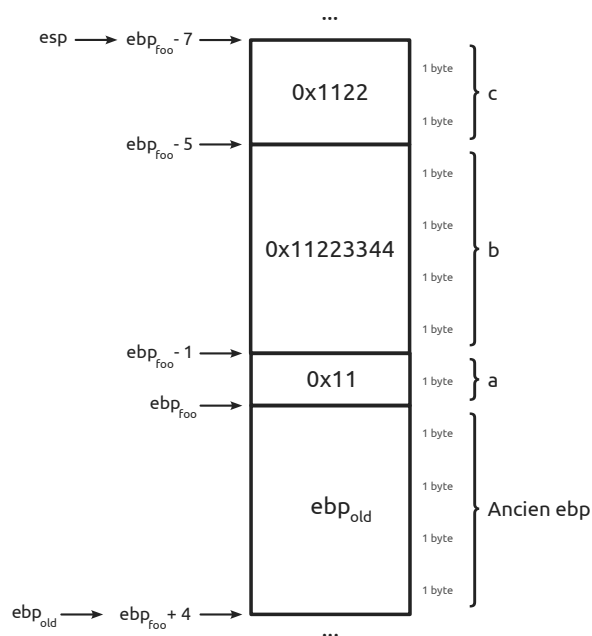


Figure 9: État du stack après la ligne 7

Il est important de noter que les adresses sur le stack vont **décroissant** de bas en haut.

Utilisez votre debugger pour voir l'évolution des registres et du stack à chaque instruction.

6.3 Paramètres & retour

Les paramètres sont **push** sur le stack juste avant le **call** et sont accessibles à partir de **ebp** :

- **[ebp+8]** est le premier paramètre de 4 bytes
- **[ebp+12]** est le second paramètre de 4 bytes

Note : les paramètres peuvent également être passés avec des registres lorsque c'est possible.

```

; NASM                                     // C++
1  main:
2      push 3          ; 4 bytes          int main() {
3      push 5          ; 4 bytes          int eax = foo(5, 3);
4      call foo        return 0;
5      add esp, 8      ; pop 8 bytes      }
6  end:
7      xor eax, eax    ; eax = 0
8      ret             ; return eax      int foo(int a, int b) {
9      foo:            int c = a + b;
10     enter 4,0        return c;
11     mov edx, [ebp+8] ; a
12     add edx, [ebp+12] ; b
13     mov [ebp-4], edx ; c
14     mov eax, [ebp-4]
15     leave
16     ret             ; return eax
```

Par convention, la valeur de retour est généralement mise dans **eax**.

Attention : il ne faut pas oublier de restaurer **esp** à la ligne 5 après l'appel de fonction.

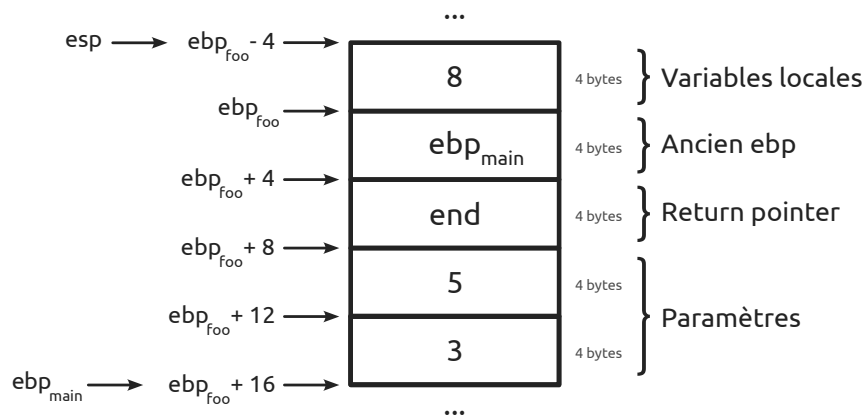


Figure 10: État du stack après la ligne 13

Références

- [1] “x86 and amd64 instruction reference.” Available: <https://www.felixcloutier.com/x86/>
- [2] “NASM.” Available: <https://www.nasm.us/>
- [3] “SASM.” Available: <https://dman95.github.io/SASM/english.html>

Exemple

Stocker et afficher les N premiers nombres de Fibonacci sur 16 bits.

```
1  %include "io.inc"
2  section .text
3  global main
4
5  main:
6      call fibo
7      call print
8  end:
9      xor eax, eax
10     ret
11
12 fibo:
13     movzx ecx, byte[N]
14     mov ax, 0
15     mov [arr], ax
16     mov dx, 1
17     mov [arr+2], dx
18 fibo_loop:
19     add ax, dx
20     xchg ax, dx
21     mov [arr+2*ecx], dx
22     dec ecx
23     jnz fibo_loop
24     ret
25
26 print:
27     movzx ecx, byte[N]
28     print_loop:
29         PRINT_UDEC 2, [arr+2*ecx]
30         NEWLINE
31         dec ecx
32         jnz print_loop
33         ret
34
35 section .rodata
36 N db 8
37
38 section .bss
39 arr resw 10
```