

MICL2 - Lucky Summary

Sm!le42

22 avril 2021

Table des matières

1	TD01	2
1.1	Théorie	2
1.1.1	Registres	2
1.1.2	nasm -f elf64 -F dwarf test.asm	2
1.1.3	ld -o test -e main test.o	2
1.1.4	./test	2
2	TD02	2
2.1	Théorie	2
2.1.1	Valeurs booléennes	2
2.1.2	Registre rflags	2
2.1.3	mov	3
2.1.4	not	3
2.1.5	and, or et xor	3
2.1.6	Masquage (and, or et xor)	3
2.1.7	Masque avec and (0 → 0)	4
2.1.8	Masque avec or (1 → 1)	4
2.1.9	Masque avec xor (1 → not)	4
2.2	Exercice 1	4
2.3	Exercice 2	5
2.4	Exercice 3	5
2.5	Exercice 4	5
3	TD03	6
3.1	Théorie	6
3.1.1	Comparaison (cmp)	6
3.2	Exercice 1	6
3.3	Exercice 2	6
3.4	Exercice 3	7
3.5	Exercice 4	7
3.6	Exercice 5	8
4	TD04	8
4.1	Théorie	8
4.1.1	Sections (.text, .data, .rodata, .bss)	8
4.1.2	Pseudo-instructions variables initialisées (DB, DW, DD, DQ)	8
4.1.3	Pseudo-instructions variables non initialisées (RESB, RESW, RESD, RESQ)	8
4.1.4	Accès à une variable	9
4.1.5	Taille de variable	9
4.1.6	Little endian (petit boutisme)	9
4.2	Exercice 1	9
4.3	Exercice 2	10
4.4	Exercice 3	10
4.5	Exercice 4	10
4.6	Exercice 5	11
4.7	Exercice 6	11
5	TD05	12
5.1	Théorie	12
5.1.1	Appels système (syscall)	12

1 TD01

1.1 Théorie

1.1.1 Registres

Les processeurs 64 bits de la famille x86 possèdent 16 registres d'utilité générale :

`rax`, `rbx`, `rcx`, `rdx`, `rsp`, `rbp`, `rsi`, `rdi`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14` et `r15`

- Les registres `rbx`, `rcx` et `rdx` ont la même structure interne que `rax`.
- Les registres `rdi`, `rsp` et `rbp` ont la même structure interne que `rsi`.
- Les registres `r9` à `r15` ont la même structure interne que `r8`.
- Les suffixes `b`, `w` et `d` signifient `byte`, `word` et `double word`.

1.1.2 `nasm -f elf64 -F dwarf test.asm`

`nasm` : Commande

`-f elf64` : Format du fichier objet

`-F dwarf` : Informations de débogage (facultatif)

`test.asm` : Fichier à compiler (code source)

Cette commande va générer un fichier objet nommé `"test.o"`.

1.1.3 `ld -o test -e main test.o`

`ld` : Commande

`-o test` : Fichier de destination `"test"`

`-e main` : Nom de la fonction (global `main`)

`test.o` : Fichier `test.o` à compiler

Cette commande va générer un fichier exécutable nommé `"test"`.

1.1.4 `./test`

Cette commande va exécuter le fichier `"test"`.

2 TD02

2.1 Théorie

2.1.1 Valeurs booléennes

Les valeurs booléennes sont au nombre de deux :

- Vrai = 1
- Faux = 0

2.1.2 Registre `rflags`

Le registre `rflags` est un registre de 64 bits dont certains sont des indicateurs (drapeaux ou flags).

Ceux que nous utiliserons ici sont les suivants :

- `CF` ou Carry Flag, bit 0 (Dernier bit testé)
- `ZF` ou Zero Flag, bit 6 (Vrai si 0 et Faux si pas 0)
- `SF` ou Sign Flag, bit 7 (Vrai si négatif et Faux si négatif)
- `OF` ou Overflow Flag, bit 11 (Vrai si overflow et Faux si pas overflow)

Remarque :

Attention au `ZF` qui peut porter à confusion :

- Si le nombre testé n'est **PAS ÉGAL** à 0, alors `ZF` = 0
- Si le nombre testé **EST ÉGAL** à 0, alors `ZF` = 1.

2.1.3 mov

L'instruction **mov** permet de placer une valeur à un emplacement de la mémoire. Elle est donc principalement utilisée pour changer les valeurs des registres ou des variables.

Attention : Cette instruction n'effectue **pas** une *extension de signe*, mais elle remplit les bits restants avec des 0.

```
1 ;Exemple mov
2 mov rax, 1 ;Maintenant rax = 0x00_00_00_00_00_00_01
3
4 mov rax, 1111b ;Maintenant rax = 0x00_00_00_00_00_00_0F (le b signifie binaire)
5
6 mov rax, 0xFF_00_07_14_FA_21_42_77 ;Maintenant rax = 0xFF_00_07_14_FA_21_42_77
7
8 mov ah, 11110000b
9 mov al, 00001111b
10 mov ah, al ;Maintenant ah = 00001111b
```

2.1.4 not

L'instruction **not** (également appelée complément à 1) a pour effet d'inverser tous les bits de son opérande.

Cette instruction n'a qu'un seul opérande qui joue le rôle de source et de destination, et qui doit être un registre ou une variable (8, 16, 32 ou 64 bits).

Le registre **rflags** n'est pas modifié.

```
1 ;Exemple not
2 mov al, 10011101b
3 not al ;Maintenant al = 01100010b
```

2.1.5 and, or et xor

Les instructions **and**, **or** et **xor** effectuent respectivement un et, un ou et un ou exclusif logiques *bit à bit* entre le source et la destination, et placent le **résultat dans la destination**.

Ces instructions ont deux opérandes : *La destination* à gauche et *la source* à droite. Ils peuvent être des registres ou des variables (8, 16, 32 ou 64 bits), mais ne **peuvent pas** être tous les deux des variables. La source peut être un immédiat de maximum 32 bits (il y a alors *extension de signe* si la destination est de 64 bits), ainsi, si les deux opérandes font 64 bits, il est impossible d'utiliser un immédiat comme source, mais il faudra passer par un registre ou une variable.

Le registre **rflags** est mis à jour, et les valeurs de **CF** et **OF** sont reset par défaut (mises à 0).

```
1 ;Exemple and, or, xor
2 mov al, 11100101b
3 mov ah, 10101010b
4 and al, ah ;Maintenant al = 10100000b, SF = 1, ZF = 0
5
6 mov al, 11100101b
7 mov ah, 01010101b
8 or al, ah ;Maintenant al = 11110101b, SF = 1, ZF = 0
9
10 mov dx, 1100010011100101b
11 mov si, 0011000001100010b
12 xor dx, si ;Maintenant dx = 1111010010000111b, SF = 1, ZF = 0
13
14 mov rax, 0xF0_F0_F0_F0_F0_F0_F0_F0
15 and rax, 0x80_00_00_00 ;Attention! Extension de signe
16 ;Maintenant rax = 0xF0_F0_F0_F0_80_00_00_00, SF = 1, ZF = 0
```

2.1.6 Masquage (and, or et xor)

Le masquage consiste à effectuer une *opération logique* afin de conserver certains bits d'un opérande et d'en modifier d'autres. (Ex : On veut modifier uniquement le 4ème et le 6ème bit)

2.1.7 Masque avec and (0 → 0)

Masque constitué de bits où **1** conserve et **0** reset.

```
1 ;Exemple masque and
2 mov al, 11100101b ;On veut conserver les 4 bits de droite et mettre les 4 autres à 0
3 and al, 00001111b ;Le masque est 00001111b
4 ;Maintenant al = 0000101b
```

2.1.8 Masque avec or (1 → 1)

Masque constitué de bits où **0** conserve et **1** set.

```
1 ;Exemple masque or
2 mov al, 11100101b ;On veut conserver les 4 bits de droite et mettre les 4 autres à 1
3 or al, 11110000b ;Le masque est 11110000b
4 ;Maintenant al = 11110101b
```

2.1.9 Masque avec xor (1 → not)

Masque constitué de bits où **0** conserve et **1** inverse.

```
1 ;Exemple masque xor
2 mov al, 11100101b ;On veut conserver les 4 bits de droite et inverser les autres
3 xor al, 11110000b ;Le masque est 11110000b
4 ;Maintenant al = 00010101b
```

2.2 Exercice 1

Quelles sont les valeurs des registres et des flags dans le code ci-dessous :

```
1 global main
2 section .text
3 main:
4     mov al, 10011101b
5     not al    ;al=01100010b, zf=0, sf=1
6
7     mov al, 11100101b
8     mov ah, 00101010b
9     and al, ah    ;al=00100000b, zf=0, sf=0
10
11    mov al, 11100101b
12    mov ah, 00001010b
13    and al, ah    ;al=00000000b, zf=1, sf=0
14
15    mov al, 01100101b
16    mov ah, 01010101b
17    or al, ah     ;al=01110101b, zf=0, sf=0
18
19    mov al, 11100101b
20    mov ah, 01010101b
21    or al, ah     ;al=11110101, zf=0, sf=1
22
23    mov dx, 1100010011100101b
24    mov si, 0011000001100010b
25    xor dx, si    ;dx=1111010010000111b, zf=0, sf=1
26
27    mov al, 11100101b
28    mov ah, 11100101b
29    xor al, ah    ;al=00000000b, zf=1, sf=0
30
31 end:
```

```

32  mov rax, 60
33  mor rdi, 0
34  syscall

```

2.3 Exercice 2

En utilisant la table ASCII, écrivez un code qui convertit un caractère minuscule en majuscule, à l'aide d'un masque.

<div><div>b₇ →</div><div>b₆ →</div><div>b₅ →</div><div>b₄ ↓</div><div>b₃ ↓</div><div>b₂ ↓</div><div>b₁ ↓</div><div>Bits</div></div>					<div><div>0 0 0 0 1 0 1 1</div><div>0 0 1 0 1 0 1 1</div><div>0 1 0 0 1 0 1 1</div><div>1 0 0 1 1 0 1 1</div></div>								
					Column →	0	1	2	3	4	5	6	7
					Row ↓	0	NUL	DLE	SP	@	P	`	p
0 0 0 1					1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0					2	STX	DC2	"	2	B	R	b	r
0 0 1 1					3	ETX	DC3	#	3	C	S	c	s
0 1 0 0					4	EOT	DC4	\$	4	D	T	d	t
0 1 0 1					5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0					6	ACK	SYN	&	6	F	V	f	v
0 1 1 1					7	BEL	ETB	'	7	G	W	g	w
1 0 0 0					8	BS	CAN	(8	H	X	h	x
1 0 0 1					9	HT	EM)	9	I	Y	i	y
1 0 1 0					10	LF	SUB	*	:	J	Z	j	z
1 0 1 1					11	VT	ESC	+	;	K	[k	{
1 1 0 0					12	FF	FC	,	<	L	\	l	
1 1 0 1					13	CR	GS	-	=	M]	m	}
1 1 1 0					14	SO	RS	.	>	N	^	n	~
1 1 1 1					15	SI	US	/	?	O	_	o	DEL

FIGURE 1 – Table ASCII

```

1  global main
2  section .text
3  main:
4      mov al, 'd'
5      mov ah, al           ;Copie le contenu d'al dans ah
6      and ah, 11011111b    ;Reset le bit 5 à 0 (car 'D' = 'd'-32 ou 68 = 100-32)
7  end:
8      mov rax, 60
9      mov rdi, 0
10     syscall

```

2.4 Exercice 3

Recodez le programme de l'exercice précédent (2) à l'aide d'une ou plusieurs instructions de manipulation de bits au lieu d'utiliser des masques. (Utilisez `bx` comme destination au lieu de `ah`).

```

1  global main
2  section .text
3  main:
4      mov al, 'd'
5      mov bx, al           ;Copie le contenu d'al dans bx
6      btr bx, 5            ;Reset le bit 5 à 0 (car 'D' = 'd'-32 ou 68 = 100-32)
7  end:
8      mov rax, 60
9      mov rdi, 0
10     syscall

```

2.5 Exercice 4

Écrivez un code qui, partant du contenu de `bl` dont on garantit qu'il s'agit d'un entier dans l'intervalle [0, 9], stocke dans `bh` le code ASCII du caractère représentant ce chiffre décimal.

```

1 global main
2 section .text
3 main:
4     mov bh, bl           ;Copie bl dans bh
5     or bh, 00110000b     ;Effectue un OR avec le masque pour Set les bits 5 et 6 à 1
6 end:
7     mov rax, 60
8     mov rdi, 0
9     syscall

```

3 TD03

3.1 Théorie

3.1.1 Comparaison (cmp)

Cette instruction compare l'opérande de gauche à celui de droite (8, 16, 32 ou 64 bits). Elle positionne les flags du registre `rflags` comme le ferait une **soustraction** de ceux-ci.

Attention : Les deux opérandes doivent avoir la **même taille**. Il peuvent être des registres ou des variables, mais ne **peuvent pas** être tous les deux des variables ! L'opérande de droite peut être un immédiat (max 32 bits).

Ainsi, si nous prenons cet exemple :

```

1 ;Exemple comparaison cmp
2 mov rax, 4
3
4 cmp rax, 4 ;(4 - 4 == 0) donc ZF = 1 et SF = 0
5 cmp rax, 5 ;(4 - 5 == -1) donc ZF = 0 et SF = 1
6 cmp rax, 2 ;(4 - 2 == 2) donc ZF = 0 et SF = 0

```

3.2 Exercice 1

Écrivez un code source complet qui :

1. Initialise `rax` à la valeur de votre choix
2. Met `rbx` à 1 si le contenu de `rax` est non nul

```

1 global main
2 section .text
3 main:
4     mov rax, 42
5     or rax, 00000000b     ;Effectue un masque neutre pour récupérer les flags
6     jz endif:             ;Si rax = 0 alors saute à endif
7     mov rbx, 1           ;Ici, rax != 0
8 endif:
9     mov rax, 60
10    mov rdi, 0
11    syscall

```

3.3 Exercice 2

Écrivez un code source complet qui :

1. Initialise `rax` à la valeur de votre choix
2. Met `r8` à 1 si le contenu de `rax` est impair
3. Met `r8` à 0 si le contenu de `rax` est pair

```

1 global main
2 section .text
3 main:
4     mov rax, 42

```

```

5      bt rax, 0      ;Teste le bit de poids faible (pair ou impair)
6      jc else        ;Si le bit de poids faible est à 1 alors saute à else
7          mov r8, 0    ;Ici, rax est pair
8          jmp endif    ;Saute à endif pour éviter le else
9  else:
10     mov r8, 1        ;Ici, rax est impair
11 endif:
12     mov rax, 60
13     mov rdi, 0
14     syscall

```

3.4 Exercice 3

Écrivez un code source complet qui :

1. Initialise **r14** et **r15** aux valeurs de votre choix
2. Assigne la valeur 0 aux registres **r14** et **r15** si leurs contenus sont égaux
3. Échange les contenus des registres **r14** et **r15** s'ils sont différents

```

1  global main
2  section .text
3      mov r14, 5
4      mov r15, 10
5      cmp r14, r15    ;Compare r14 à r15
6      jnz else        ;Si r14 != r15 alors saute à else
7          mov r14, 0    ;Ici, r14 == r15
8          mov r15, 0
9          jmp endif    ;Saute à endif pour éviter le else
10 else:
11     mov r13, r14      ;Inverse r14 et r15 en utilisant une mémoire temporaire (r13)
12     mov r14, r15
13     mov r15, r13
14 endif:
15     mov rax, 60
16     mov rdi, 0
17     syscall

```

3.5 Exercice 4

Écrivez un code source complet qui :

1. Initialise **rax** et **rbx** aux valeurs de votre choix
2. Copie dans **r8** le maximum des valeurs contenues dans **rax** et **rbx**
3. Copie dans **r9** le minimum des valeurs de **rax** et **rbx**

```

1  global main
2  section .text
3      mov rax, 5
4      mov rbx, 10
5      cmp rax, rbx    ;Compare rax à rbx
6      js else         ;Si s == 1 alors rax < rbx donc saute à else
7          mov r8, rax  ;Ici, rax >= rbx
8          mov r9, rbx
9          jmp endif    ;Saute à endif pour éviter le else
10 else:
11     mov r8, rbx
12     mov r9, rax
13 endif:
14     mov rax, 60
15     mov rdi, 0
16     syscall

```

3.6 Exercice 5

Écrivez un code source complet qui :

1. Initialise `rdi` à la valeur de votre choix
2. Met `rsi` à 0 si `rdi` est pair
3. Met `rsi` à 1 si `rdi` est un multiple de 2, sans être un multiple d'une plus grande puissance de 2
4. Met `rsi` à 2 si `rdi` est un multiple de 4, sans être un multiple d'une plus grande puissance de 2
5. Met `rsi` à 3 si `rdi` est un multiple de 8, sans être un multiple d'une plus grande puissance de 2

```
1  global main
2  section .text
3      mov rdi, 10
4      bt rdi, 0           ;Vérifie le bit de poids faible de rdi
5      jnc even           ;Si le bit de poids faible == 1 alors saute à even (pair)
6          mov rsi, 0      ;Ici, rdi est impair
7          jmp end        ;Saute à end pour éviter les autres conditions
8  even:
9      bt rdi, 1           ;Vérifie le bit 1 (multiple de 2 et pas plus)
10     jnc notTwo          ;Si pas multiple de 2, alors saute à notTwo
11         mov rsi, 1      ;Ici, rdi est un multiple de 2
12         jmp end        ;Saute à end pour éviter les autres conditions
13  notTwo:
14     bt rdi, 2           ;Vérifie le bit 2 (multiple de 4 et pas plus)
15     jnc notFour        ;Si pas multiple de 4 alors saute à notFour
16         mov rsi, 2      ;Ici, rdi est un multiple de 4
17         jmp end        ;Saute à end pour éviter la dernière condition
18  notFour:
19     or rdi, 00000000b   ;Effectue un masque neutre pour récupérer les flags
20     jz end              ;Si rdi == 0 alors saute à end car pas multiple de 8 ou plus
21         mov rsi, 3      ;Ici, rdi est un multiple de 8, 16, 32 ou 64
22  end:
23     mov rax, 60
24     mov rdi, 0
25     syscall
```

4 TD04

4.1 Théorie

4.1.1 Sections (.text, .data, .rodata, .bss)

Nom	Rôle
<code>.text</code>	Instructions exécutables du programme
<code>.data</code>	Variables globales explicitement initialisées
<code>.rodata</code>	Constantes globales explicitement initialisées
<code>.bss</code>	Variables globales implicitement initialisées à 0

4.1.2 Pseudo-instructions variables initialisées (DB, DW, DD, DQ)

Taille (octets)	Pseudo-instruction	Signification
1	<code>DB</code>	Define Byte
2	<code>DW</code>	Define Word
4	<code>DD</code>	Define Doubleword
8	<code>DQ</code>	Define Quadword

4.1.3 Pseudo-instructions variables non initialisées (RESB, RESW, RESD, RESQ)

Taille (octets)	Pseudo-instruction	Signification
1	<code>RESB</code>	Reserve Byte
2	<code>RESW</code>	Reserve Word
4	<code>RESD</code>	Reserve Doubleword
8	<code>RESQ</code>	Reserve Quadword

4.1.4 Accès à une variable

Une variable correspond à une adresse mémoire sur 8 octets. Si l'on veut pouvoir accéder à la valeur contenue dans la variable, il faudra placer l'adresse entre crochets.

```
1 ;Exemple variables
2
3 section .data
4     test DD 42          ;Entier sur 4 bytes
5 section .text
6     mov rax, test       ;On met l'adresse de test dans rax (8 bytes)
7     mov ebx, [test]     ;On met la valeur de test dans ebx (4 bytes car ebx)
8     mov ecx, [rax]      ;On met le contenu de rax dans ecx car [rax]=[test] (4 bytes)
```

4.1.5 Taille de variable

L'assembleur nasm ne retient pas la taille des variables. Lorsqu'on accède au contenu d'une variable, le nombre de *bytes* déréférencés à partir de l'adresse entre crochets est déduit de la taille du second opérande, s'il existe et s'il ne s'agit pas d'un immédiat. Dans le cas contraire, il faut renseigner la taille de la donnée à l'aide d'un des spécificateurs de taille (byte, word, dword, qword).

```
1 section .data          ;Déclaration des variables
2     testB DB -1
3     testW DW 23
4     testD DD -1
5     testQ DQ 130_761_944
6 section .text          ;Changeons la valeur des variables:
7     mov byte [testB], 7 ;Il faut préciser que testB est un Byte
8     mov word [testW], 14 ;Il faut préciser que testW est un Word
9     mov dword [testD], 21 ;Il faut préciser que testD est un Double Word
10    mov qword [testQ], 42 ;Il faut préciser que testQ est un Quad Word
```

4.1.6 Little endian (petit boutisme)

Contrairement au *big endian*, avec le *little endian*, le byte de rang le plus *petit* est stocké à l'adresse la plus *petite*.

L'architecture x86 adopte le little endian.

```
1 ;Exemple little endian
2
3 section .data
4     vw DW 0x0102
5     ; ---> petites adresses ---> grandes adresses --->
6     ;     vw      vw+1
7     ; .../ 0x02 / 0x01 /...
8     vq DQ 0x1122334455667788
9     ; ---> petites adresses ---> grandes adresses --->
10    ;     vq      vq+1  vq+2  vq+3  vq+4  vq+5  vq+6  vq+7
11    ; .../ 0x88 / 0x77 / 0x66 / 0x55 / 0x44 / 0x33 / 0x22 / 0x11 /...
12    ;
13    ; Vue complète de la section .data :
14    ; ---> petites adresses ---> grandes adresses --->
15    ;     vw      vw+1  vq      vq+1  vq+2  vq+3  vq+4  vq+5  vq+6  vq+7
16    ; .../ 0x02 / 0x01 / 0x88 / 0x77 / 0x66 / 0x55 / 0x44 / 0x33 / 0x22 / 0x11 /...
```

4.2 Exercice 1

Complétez les commentaires :

```
1 global main
2 section .data
3     var1 DB 1
```

```

4   var2 DB 2
5   var3 DW 0x0304
6   var4 DQ 0x000000008000FFFF
7   section .text
8   main:
9       mov rax, var1      ;rax contient l'adresse de var1
10      mov al, [var1]      ;al contient 00000001b
11      mov ax, [var1]      ;ax contient 0000000000000001b
12      mov al, [var3]      ;al contient 0x04
13      mov ax, [var3]      ;ax contient 0x0304
14      mov rax, -1         ;rax contient -1
15      mov eax, [var4]     ;eax contient 0x8000FFFF
16  end:
17      mov rax, 60
18      mov rdi, 0
19      syscall

```

4.3 Exercice 2

Écrivez un code source complet qui déclare une variable `nb` de taille 4 bytes. Il place ensuite l'adresse de cette variable dans `rax` et son contenu dans `rbx`.

```

1   global main
2   section .data
3       nb DD 42           ;Déclaration de la variable nb sur 4 bytes
4   section .text
5       mov rax, nb        ;On place l'adresse de nb dans rax
6       mov rbx, [nb]      ;On place la valeur de nb dans rbx
7   end:
8       mov rax, 60
9       mov rdi, 0
10      syscall

```

4.4 Exercice 3

Écrivez un code source complet qui déclare une variable sur 8 bytes implicitement initialisée à 0 puis lui assigne la valeur 42.

```

1   global main
2   section .bss
3       var RESQ 1         ;On déclare une variable "var" (1x8 bytes)
4   section .text
5       mov qword [var], 42 ;On assigne la valeur 42 à "var" en précisant la taille
6   end:
7       mov rax, 60
8       mov rdi, 0
9       syscall

```

4.5 Exercice 4

Soient les déclarations suivantes :

```

1   section .data
2       b0 DB 0
3       b1 DB 0
4       b2 DB 0
5       b3 DB 0
6   section .rodata
7       nb DD 0x12345678

```

Écrivez un code source complet qui stocke dans **b0** le byte de rang 0 de **nb**, dans **b1** celui de rang 1, dans **b2** celui de rang 2 et finalement dans **b3** celui de rang 3 (Utilisez un ou des registres intermédiaires).

```

1  global main
2  section .data
3      b0 DB 0
4      b1 DB 0
5      b2 DB 0
6      b3 DB 0
7  section .rodata
8      nb DD 0x12345678
9  section .text
10     mov al, nb                ;On stocke l'adresse de nb dans al (1 byte)
11     mov byte [b0], al        ;On stocke le byte à l'adresse al dans b0
12     mov byte [b1], al+1      ;On stocke le byte à l'adresse al+1 dans b1
13     mov byte [b2], al+2      ;On stocke le byte à l'adresse al+2 dans b2
14     mov byte [b3], al+3      ;On stocke le byte à l'adresse al+3 dans b3
15 end:
16     mov rax, 60
17     mov rdi, 0
18     syscall

```

4.6 Exercice 5

Écrivez un code source complet qui :

1. Déclare deux variables initialisées aux valeurs de votre choix
2. Échange les contenus de ces variables

```

1  global main
2  section .data
3      var1 DB 7                ;Déclaration var1 = 7
4      var2 DB 14               ;Déclaration var2 = 14
5  section .text
6      mov al, [var1]           ;Met la valeur de var1 dans le registre al (1 byte)
7      mov ah, [var2]           ;Met la valeur de var2 dans le registre ah (1 byte)
8      mov [var1], ah           ;Remplace var1 par la valeur de ah (var2)
9      mov [var2], al           ;Remplace var2 par la valeur de al (var1)
10 end:
11     mov rax, 60
12     mov rdi, 0
13     syscall

```

4.7 Exercice 6

Écrivez un code source complet qui déclare trois variables dont deux sont constantes et explicitement initialisées mais pas la troisième. Le contenu de cette dernière est calculé. Il est égal au minimum des deux autres.

```

1  global main
2  section .rodata
3      var1 DB 7                ;Déclaration de var1 = 7 (1 byte)
4      var2 DB 14               ;Déclaration de var2 = 14 (1 byte)
5  section .bss
6      varMin RESB 1            ;Déclaration de varMin (1x1 byte)
7  section .text
8      cmp var1, var2           ;Comparaison de var1 avec var2 (var1 - var2)
9      js else                  ;Si SF = 1 alors var1 < var2 donc saute à else
10     mov al, var2              ;Ici, var1 >= var2
11     mov [varMin], al
12     jmp endif                 ;On saute à endif pour éviter le else
13 else:
14     mov al, var1              ;Ici, var1 < var2
15     mov [varMin], al

```

```
16 endif:
17     mov rax, 60
18     mov rdi, 0
19     syscall
```

5 TD05

5.1 Théorie

5.1.1 Appels système (syscall)

Services offerts par le système d'exploitation pour effectuer diverses tâches. Chaque appel système est identifié par un numéro appelé *numéro de service*.

L'appel système se fait au travers de l'instruction `syscall`. Celle-ci a pour effet de basculer le CPU en mode privilégié et passer la main au service système demandé, identifié par son numéro. (Voir fichier `/usr/include/asm/unistd_64.h`)

Sous GNU/Linux 64 bits, un appel système en langage d'assemblage se fait en quatre étapes :

1. Placer le numéro du service désiré dans `rax`
2. Mettre les paramètres, s'il y en a, dans `rdi`, `rsi`, `rdx`, `rcx`, `r8` et `r9`
3. Appeler le système par l'instruction `syscall`
4. Consulter dans `rax` la valeur de retour, s'il y en a une, ou le statut d'erreur, si nécessaire ou utile

Remarque : Les registres `rcx`, `r11` et `rax` seront modifiés lors d'un `syscall`.

- `rcx` pour la sauvegarde de la valeur du registre `rip`
- `r11` pour la sauvegarde du registre `rflags`
- `rax` pour la valeur de retour de l'appel système