MICL2 - Lucky Summary

Sm!le42

1er juin 2021

Table des matières

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`. (Figure 1)
- Les registres `rdi`, `rsp` et `rbp` ont la même structure interne que `rsi`. (Figure 2)
- Les registres `r9` à `r15` ont la même structure interne que `r8`. (Figure 3)
- Les suffixes `b, w` et `d` signifient `byte`, `word` et `double word`.

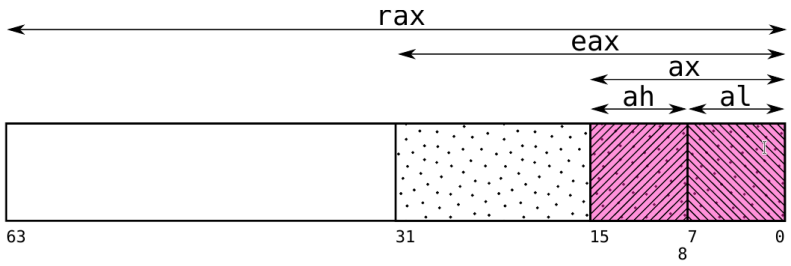

FIGURE 1 – Structure interne de rax

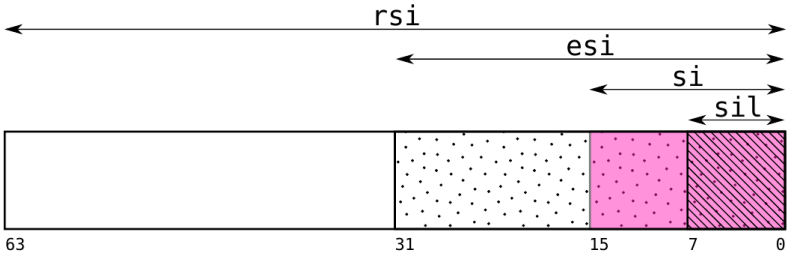

FIGURE 2 – Structure interne de rsi

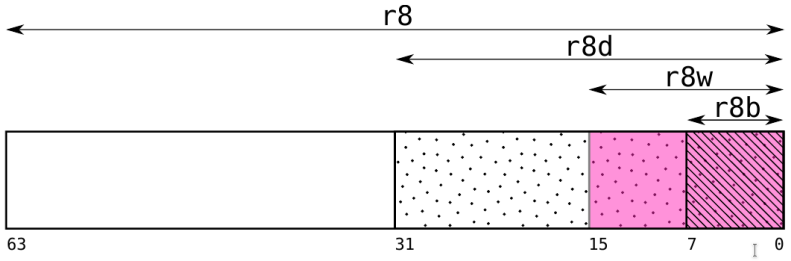

FIGURE 3 – Structure interne de r8

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 certain 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`.

```
;Exemple mov
mov rax, 1 ;Maintenant rax = 0x00_00_00_00_00_00_00_01

mov rax, 1111b ;Maintenant rax = 0x00_00_00_00_00_00_00_0F (le b signifie binaire)

mov rax, 0xFF_00_07_14_FA_21_42_77 ;Maintenant rax = 0xFF_00_07_14_FA_21_42_77

mov ah, 11110000b
mov al, 00001111b
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.

```
;Exemple not
mov al, 10011101b
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`).

```
;Exemple and, or, xor
mov al, 11100101b
mov ah, 10101010b
and al, ah ;Maintenant al = 10100000b, SF = 1, ZF = 0

mov al, 11100101b
mov ah, 01010101b
or al, ah ;Maintenant al = 11110101b, SF = 1, ZF = 0

mov dx, 1100010011100101b
mov si, 0011000001100010b
xor dx, si ;Maintenant dx = 1111010010000111b, SF = 1, ZF = 0

mov rax, 0xF0_F0_F0_F0_F0_F0_F0_F0
and rax, 0x80_00_00_00 ;Attention! Extension de signe
;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**.

```
;Exemple masque and
mov al, 11100101b ;On veut conserver les 4 bits de droite et mettre les 4 autres à 0
and al, 00001111b ;Le masque est 00001111b
;Maintenant al = 00000101b
```

2.1.8 Masque avec `or` (1 –> 1)

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

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

2.1.9 Masque avec `xor` (1 –> `not`)

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

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

2.2 Exercice 1

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

```
global main
section .text
main:
   mov al, 10011101b
   not al    ;al=01100010b, zf=0, sf=1

   mov al, 11100101b
   mov ah, 00101010b
   and al, ah    ;al=00100000b, zf=0, sf=0

   mov al, 11100101b
   mov ah, 00001010b
   and al, ah    ;al=00000000b, zf=1, sf=0

   mov al, 01100101b
   mov ah, 01010101b
   or al, ah    ;al=01110101b, zf=0, sf=0

   mov al, 11100101b
   mov ah, 01010101b
   or al, ah    ;al=11110101, zf=0, sf=1

   mov dx, 1100010011100101b
   mov si, 0011000001100010b
   xor dx, si    ;dx=1111010010000111b, zf=0, sf=1

   mov al, 11100101b
   mov ah, 11100101b
   xor al, ah    ;al=00000000b, zf=1, sf=0

end:
   mov rax, 60
   mor rdi, 0
   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.

b_7 →				0	0	0	0	1	1	1	1
b_6 →				0	0	1	1	0	0	1	1
b_5 →				0	1	0	1	0	1	0	1
Bits	b_4 ↓	b_3 ↓	b_2 ↓	b_1 ↓	Column → / Row ↓	0	1	2	3	4	5	6	7
	0	0	0	0	0	NUL	DLE	SP	0	@	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 4 – Table ASCII

```asm
global main
section .text
main:
   mov al, 'd'
   mov ah, al          ;Copie le contenu d'al dans ah
   and ah, 11011111b   ;Reset le bit 5 à 0 (car 'D' = 'd'-32 ou 68 = 100-32)
end:
   mov rax, 60
   mov rdi, 0
   syscall
```

2.4 Exercice 3

Recodez le programme de l'exercice précédant (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`).

```asm
global main
section .text
main:
   mov al, 'd'
   mov bx, al   ;Copie le contenu d'al dans bx
   btr bx, 5    ;Reset le bit 5 à 0 (car 'D' = 'd'-32 ou 68 = 100-32)
end:
   mov rax, 60
   mov rdi, 0
   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.

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

```asm
;Exemple comparaison cmp
mov rax, 4

cmp rax, 4 ;(4 - 4 == 0) donc ZF = 1 et SF = 0
cmp rax, 5 ;(4 - 5 == -1) donc ZF = 0 et SF = 1
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

```
global main
section .text
main:
   mov rax, 42
   cmp rax, 0
   jz endif:            ;Si rax = 0 alors saute à endif
      mov rbx, 1        ;Ici, rax != 0
endif:
   mov rax, 60
   mov rdi, 0
   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

```
global main
section .text
main:
   mov rax, 42
   bt rax, 0       ;Teste le bit de poid faible (pair ou impair)
   jc else         ;Si le bit de poid faible est à 1 alors saute à else
      mov r8, 0    ;Ici, rax est pair
      jmp endif    ;Saute à endif pour éviter le else
else:
   mov r8, 1       ;Ici, rax est impair
endif:
   mov rax, 60
   mov rdi, 0
   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

```
global main
section .text
   mov r14, 5
   mov r15, 10
   cmp r14, r15       ;Compare r14 à r15
   jnz else           ;Si r14 != r15 alors saute à else
      mov r14, 0      ;Ici, r14 == r15
      mov r15, 0
      jmp endif       ;Saute à endif pour éviter le else
else:
   mov r13, r14       ;Inverse r14 et r15 en utilisant une mémoire temporaire (r13)
```

```
    mov r14, r15
    mov r15, r13
endif:
    mov rax, 60
    mov rdi, 0
    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`

```
global main
section .text
    mov rax, 5
    mov rbx, 10
    cmp rax, rbx        ;Compare rax à rbx
    js else             ;Si s == 1 alors rax < rbx donc saute à else
        mov r8, rax     ;Ici, rax >= rbx
        mov r9, rbx
        jmp endif       ;Saute à endif pour éviter le else
else:
    mov r8, rbx
    mov r9, rax
endif:
    mov rax, 60
    mov rdi, 0
    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

```
global main
section .text
    mov rdi, 10
    bt rdi, 0           ;Vérifie le bit de poid faible de rdi
    jnc even            ;Si le bit de poid faible == 1 alors saute à even (pair)
        mov rsi, 0      ;Ici, rdi est impair
        jmp end         ;Saute à end pour éviter les autres conditions
even:
    bt rdi, 1           ;Vérifie le bit 1 (multiple de 2 et pas plus)
    jnc notTwo          ;Si pas multiple de 2, alors saute à notTwo
        mov rsi, 1      ;Ici, rdi est un multiple de 2
        jmp end         ;Saute à end pour éviter les autres conditions
notTwo:
    bt rdi, 2           ;Vérifie le bit 2 (multiple de 4 et pas plus)
    jnc notFour         ;Si pas multiple de 4 alors saute à notFour
        mov rsi, 2      ;Ici, rdi est un multiple de 4
        jmp end         ;Saute à end pour éviter la dernière condition
notFour:
    or rdi, 00000000b   ;Effectue un masque neutre pour récupérer les flags
    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
`.text`	Instructions exécutables du programme
`.data`	Variables globales explicitement initialisées
`.rodata`	Constantes globales explicitement initialisées
`.bss`	Variables globales implicitement initialisées à 0

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

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

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

Taille (octets)	Pseudo-instruction	Signification
1	`RESB`	Reserve Byte
2	`RESW`	Reserve Word
4	`RESD`	Reserve Doubleword
8	`RESQ`	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
```

```
   mov word [testW], 14    ;Il faut préciser que testW est un Word
   mov dword [testD], 21   ;Il faut préciser que testD est un Double Word
   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.

```
;Exemple little endian

section .data
   vw DW 0x0102
; ---> petites adresses ---> grandes adresses --->
;      vw     vw+1
; .../ 0x02 / 0x01 /...
   vq DQ 0x1122334455667788
; ---> petites adresses ---> grandes adresses --->
;      vq     vq+1   vq+2   vq+3   vq+4   vq+5   vq+6   vq+7
; .../ 0x88 / 0x77 / 0x66 / 0x55 / 0x44 / 0x33 / 0x22 / 0x11 /...
;
; Vue complète de la section .data :
; ---> petites adresses ---> grandes adresses --->
;      vw     vw+1   vq     vq+1   vq+2   vq+3   vq+4   vq+5   vq+6   vq+7
; .../ 0x02 / 0x01 / 0x88 / 0x77 / 0x66 / 0x55 / 0x44 / 0x33 / 0x22 / 0x11 /...
```

4.2 Exercice 1

Complétez les commentaires :

```
global main
section .data
   var1 DB 1
   var2 DB 2
   var3 DW 0x0304
   var4 DQ 0x000000008000FFFF
section .text
main:
   mov rax, var1      ;rax contient l'adresse de var1
   mov al, [var1]     ;al contient 00000001b
   mov ax, [var1]     ;ax contient 0000000000000001b
   mov al, [var3]     ;al contient 0x04
   mov ax, [var3]     ;ax contient 0x0304
   mov rax, -1        ;rax contient -1
   mov eax, [var4]    ;eax contient 0x8000FFFF
end:
   mov rax, 60
   mov rdi, 0
   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`.

```
global main
section .data
   nb DD 42         ;Déclaration de la variable nb sur 4 bytes
section .text
   mov rax, nb      ;On place l'adresse de nb dans rax
```

```
   mov rbx, [nb]    ;On place la valeur de nb dans rbx
end:
   mov rax, 60
   mov rdi, 0
   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`.

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

4.5 Exercice 4

Soient les déclarations suivantes :

```
section .data
   b0 DB 0
   b1 DB 0
   b2 DB 0
   b3 DB 0
section .rodata
   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).

```
gobal main
section .data
   b0 DB 0
   b1 DB 0
   b2 DB 0
   b3 DB 0
section .rodata
   nb DD 0x12345678
section .text
   mov al, nb            ;On stocke l'adresse de nb dans al (1 byte)
   mov byte [b0], al     ;On stocke le byte à l'adresse al dans b0
   mov byte [b1], al+1   ;On stocke le byte à l'adresse al+1 dans b1
   mov byte [b2], al+2   ;On stocke le byte à l'adresse al+2 dans b2
   mov byte [b3], al+3   ;On stocke le byte à l'adresse al+3 dans b3
end:
   mov rax, 60
   mov rdi, 0
   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

```
global main
section .data
   var1 DB 7         ;Déclaration var1 = 7
   var2 DB 14        ;Déclaration var2 = 14
section .text
   mov al, [var1]    ;Met la valeur de var1 dans le registre al (1 byte)
   mov ah, [var2]    ;Met la valeur de var2 dans le registre ah (1 byte)
   mov [var1], ah    ;Remplace var1 par la valeur de ah (var2)
   mov [var2], al    ;Remplace var2 par la valeur de al (var1)
end:
   mov rax, 60
   mov rdi, 0
   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.

```
global main
section .rodata
   var1 DB 7                  ;Déclaration de var1 = 7 (1 byte)
   var2 DB 14                 ;Déclaration de var2 = 14 (1 byte)
section .bss
   varMin RESB 1              ;Déclaration de varMin (1x1 byte)
section .text
   cmp var1, var2             ;Comparaison de var1 avec var2 (var1 - var2)
   js else                    ;Si SF = 1 alors var1 < var2 donc saute à else
      mov al, var2            ;Ici, var1 >= var2
      mov [varMin], al
      jmp endif               ;On saute à endif pour éviter le else
else:
   mov al, var1               ;Ici, var1 < var2
   mov [varMin], al
endif:
   mov rax, 60
   mov rdi, 0
   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