HE2B ESI

MICL : TD02 : Instructions logiques et de manipulation de bits

ABS – BEJ – DBO – HAL – NVS – SRE – YVO *

Année académique 2021 – 2022

Ce deuxième TD commence par la description du registre `rflags`. La découverte des instructions logiques `not`, `and`, `or` et `xor` suit. Les instructions de manipulation de bits `bt`, `btr`, `bts` et `btc` sont finalement étudiées.

1 Valeurs booléennes

Les valeurs booléennes [1] sont au nombre de deux : *Vrai* et *Faux*. Un seul bit suffit donc pour représenter une variable booléenne. Par convention, la valeur *Vrai* est codée par un bit à 1 tandis que *Faux* est représenté par un bit à 0.

2 Registre `rflags`

Le registre `rflags` [2] est un registre de 64 bits dont certains sont des indicateurs (drapeaux, *flags*). La Fig. 1 illustre le contenu de `eflags`, le registre des états des processeurs x86 32 bits. `rflags` en constitue une extension sur 64 bits. Les 32 nouveaux bits sont de poids 32 à 63. Ils sont tous réservés [3].

*Et aussi, lors des années passées : DHA – DWI – EGR – ELV – FPL – JDS – MBA – MCD – MHI – MWA.

1. https://fr.wikipedia.org/wiki/Bool%C3%A9en (consulté le 30 janvier 2020).
2. https://fr.wikipedia.org/wiki/RFLAGS (consulté le 30 janvier 2020).
3. Voir le manuel d'Intel [1, section 3.4.3.4, p. 3-18 Vol. 1] : *In 64-bit mode, EFLAGS is extended to 64 bits and called RFLAGS. The upper 32 bits of RFLAGS register is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS.*

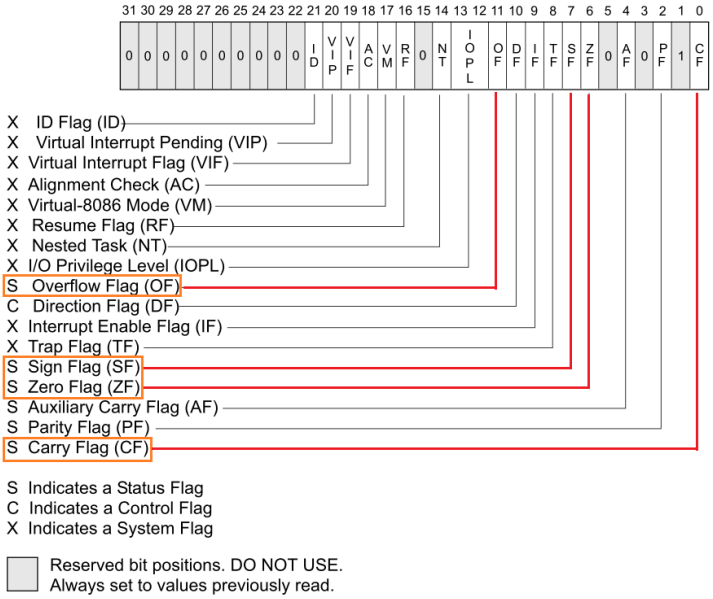

FIG. 1 – Le registre `eflags` (Illustration © Intel : *Intel© 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4* [a], p. 3-16 Vol. 1).

a. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf (consulté le 30 janvier 2020).

Les indicateurs fournissent des informations sur le déroulement du processus en cours, notamment sur le résultat de certaines opérations. Ils sont appelés « indicateurs d'état »(*status flags*) car ils permettent notamment de savoir comment se sont déroulées les opérations arithmétiques [4].

Nous ne voyons dans ce laboratoire que quatre indicateurs :

— *Carry flag* (`CF`) : indicateur de retenue [5], bit de rang 0 de `rflags` ;
— *Zero flag* (`ZF`) : indicateur de zéro [6], bit de rang 6 de `rflags` ;
— *Sign flag* (`SF`) : indicateur de signe [7], bit de rang 7 de `rflags` ;
— *Overflow flag* (`OF`) : indicateur de débordement [8], bit de rang 11 de `rflags`.

L'accès direct au contenu de `rflags` n'est pas possible. On accède indirectement à son contenu via des instructions particulières, telles celles de saut (TD03 et TD06). Pour cette raison, il n'est pas nécessaire de connaître la position précise d'un *flag* dans le

4. Les indicateurs d'état sont au nombre de six, mais seuls quatre d'entre eux sont vus dans ce labo. Outre les indicateurs d'état, il existe d'autres indicateurs, non vus lors des labos micro : les « indicateurs de contrôle »(*control flags*), associés aux instructions de manipulation de chaînes de caractères, et les « indicateurs système »(*system flags*), utilisés par le système d'exploitation.
5. https://fr.wikipedia.org/wiki/Indicateur_de_retenue (consulté le 30 janvier 2020).
6. https://en.wikipedia.org/wiki/Zero_flag (consulté le 30 janvier 2020).
7. https://en.wikipedia.org/wiki/Sign_flag (consulté le 30 janvier 2020).
8. https://fr.wikipedia.org/wiki/Indicateur_de_d%C3%A9bordement (consulté le 30 janvier 2020).

Instruction	Effet	Contraintes	*Flags* affectés
`not X`	Inverse tous les bits de `X`	`X` = registre ou variable de 8, 16, 32 ou 64 bits	Aucun

(a) Résumé.

```
      al   :  10011101b
not   al
     ----------------
      al   :  01100010b
```

(b) Exemple.

TABLE 1 – Instruction `not`.

registre `rflags`.

3 Instructions logiques

3.1 `not`

L'instruction `not`[9] n'a qu'un opérande qui joue le rôle de source et de destination et qui doit être un registre ou une variable (voir TD04 et TD07) de 8, 16, 32 ou 64 bits. `not` a pour effet d'inverser[10] tous les bits de son opérande. Cette opération est également appelée complément à 1[11].

La TABLE 1(a) donne un résumé de cette instruction, et la TABLE 1(b) en donne un exemple.

3.2 `and`, `or` et `xor`

Les instructions `and`[12], `or`[13] et `xor`[14] ont deux opérandes : la *destination*, à gauche de la virgule, et la *source*, à droite. Ils peuvent être des registres ou des variables de 8, 16, 32 ou 64 bits, mais pas tous les deux des emplacements mémoire. En outre, la source peut être un immédiat.

Ces instructions effectuent, respectivement, un *et*[15], un *ou*[16] et un *ou exclusif*[17] logiques *bit à bit* entre la source et la destination. Ils placent le résultat dans la destination,

9. https://www.felixcloutier.com/x86/not (consulté le 30 janvier 2020).
10. https://fr.wikipedia.org/wiki/Fonction_NON (consulté le 30 janvier 2020).
11. https://fr.wikipedia.org/wiki/Compl%C3%A9ment_%C3%A0_un (consulté le 30 janvier 2020).
12. https://www.felixcloutier.com/x86/and (consulté le 30 janvier 2020).
13. https://www.felixcloutier.com/x86/or (consulté le 30 janvier 2020).
14. https://www.felixcloutier.com/x86/xor (consulté le 30 janvier 2020).
15. https://fr.wikipedia.org/wiki/Fonction_ET (consulté le 30 janvier 2020).
16. https://fr.wikipedia.org/wiki/Fonction_OU (consulté le 30 janvier 2020).
17. https://fr.wikipedia.org/wiki/Fonction_OU_exclusif (consulté le 30 janvier 2020).

Instruction	Effet	Contraintes	*Flags* affectés
`and X, Y`	`X` ← `X` ET `Y` (bit à bit)	Registres ou variables de 8, 16, 32 ou 64 bits, pas deux variables, `Y` peut être un immédiat (8, 16 ou 32 bits)	`CF` ← 0 `OF` ← 0 `SF` ← bit de signe [a] du résultat `ZF` ← 1 si résultat nul, 0 sinon
`or X, Y`	`X` ← `X` OU `Y` (bit à bit)		
`xor X, Y`	`X` ← `X` XOR `Y` (bit à bit)		

(a) Résumé.

a. Le *bit de signe* d'un motif binaire est son *bit de rang le plus élevé*.

```
                al  :  11100101b
                ah  :  10101010b
        and al, ah  ____________
                al  :  10100000b
-------------------------------------
                al  :  11100101b
                ah  :  01010101b
         or al, ah  ____________
                al  :  11110101b
-------------------------------------
                dx  :  1100010011100101b
                si  :  0011000001100010b
        xor dx, si  ____________________
                dx  :  1111010010000111b
```

(b) Exemples.

TABLE 2 – Instructions `and`, `or` et `xor`.

sans modifier la source.

En plus, elles modifient le *sign flag* (`SF`), qui reçoit le bit de rang le plus élevé du résultat [18], et le *zero flag* (`ZF`), qui indique si le résultat est nul : 1 pour nul, 0 pour non nul. Le *carry flag* (`CF`) et le *overflow flag* (`OF`) sont mis à 0.

La TABLE 2(a) donne un résumé de ces instructions, et la TABLE 2(b) en donne des exemples d'utilisation.

Remarque Comme indiqué dans la TABLE 2(a), les immédiats s'étendent sur maximum 32 bits. Si on utilise un immédiat avec une destination dont la taille fait 64 bits, l'immédiat est étendu sur 64 bits par extension de signe [19] : son bit de signe, celui de rang 31, est recopié en bits 32 à 63.

Le code source suivant illustre ceci :

18. En représentation complément à deux, le bit de rang le plus élevé est un bit de signe : il vaut 0 pour un nombre positif, 1 pour un nombre négatif, d'où le nom de *sign flag*.

19. https://fr.wiktionary.org/wiki/extension_de_signe (consulté le 30 janvier 2020).

```asm
; 0_extension_signe.asm

global main

section .text
main:
    mov rax, 0xF0_F0_F0_F0_F0_F0_F0_F0
    and rax, 0x80_00_00_00  ; attention : extension de signe
            ; rax : 0xF0_F0_F0_F0_80_00_00_00

    mov rsi, 0xF0_F0_F0_F0_F0_F0_F0_F0
    mov rdi, 0xFF_FF_FF_FF_80_00_00_00
    and rsi, rdi            ; idem
            ; rsi : 0xF0_F0_F0_F0_80_00_00_00

    mov rbx, 0xF0_F0_F0_F0_F0_F0_F0_F0
    mov rcx, 0x80_00_00_00
    and rbx, rcx
            ; rbx : 0x00_00_00_00_80_00_00_00

    ; fin
    mov rax, 60
    mov rdi, 0
    syscall

```

Dès lors, si les deux opérandes font 64 bits, il est impossible d'utiliser un immédiat comme source, mais il faut utiliser un registre ou une variable.

3.3 Masquage

Le masquage [20] consiste à effectuer une opération logique afin de conserver certains bits d'un opérande, et d'en modifier d'autres. Nous effectuons des masquages à l'aide des instructions `and`, `or` et `xor`.

3.3.1 Masque avec `and`

L'instruction `and` permet de conserver certains bits d'un opérande et de mettre les autres à 0 [21].

20. https://en.wikipedia.org/wiki/Mask_(computing) (consulté le 30 janvier 2020).
21. https://en.wikipedia.org/wiki/Mask_(computing)#Masking_bits_to_0 (consulté le 30 janvier 2020).

Pour ce faire, on fait un `and` entre cet opérande et un second opérande appelé *masque*. Ce masque est constitué de bits à 1 aux positions qu'on désire conserver et de bits à 0 aux positions qu'on veut mettre à 0.

Par exemple, si on désire conserver les quatre bits de droite de `al` et mettre les autres à 0, on utilise le masque $00001111b$:

`al`	:	$11100101b$
`bl` (masque)	:	$00001111b$
`and al, bl`		
`al`	:	$00000101b$

3.3.2 Masque avec `or`

L'instruction `or` permet de conserver certains bits d'un opérande et de mettre les autres à 1 [22].

Pour ce faire, on fait un `or` entre cet opérande et le masque. Ce masque est constitué de bits à 0 aux positions qu'on désire conserver et de bits à 1 aux positions qu'on veut mettre à 1.

Par exemple, si on désire conserver les quatre bits de droite de `al` et mettre les autres à 1, on utilise le masque $11110000b$:

`al`	:	$11100101b$
`bl` (masque)	:	$11110000b$
`or al, bl`		
`al`	:	$11110101b$

3.3.3 Masque avec `xor`

L'instruction `xor` permet de conserver certains bits d'un opérande et d'inverser les autres [23].

Pour ce faire, on fait un `xor` entre cet opérande et le masque. Ce masque est constitué de bits à 0 aux positions qu'on désire conserver et de bits à 1 aux positions qu'on veut inverser.

Par exemple, si on désire conserver les quatre bits de droite de `al` et inverser les autres, on utilise le masque $11110000b$:

`al`	:	$11100101b$
`bl` (masque)	:	$11110000b$
`xor al, bl`		
`al`	:	$00010101b$

22. https://en.wikipedia.org/wiki/Mask_(computing)#Masking_bits_to_1 (consulté le 30 janvier 2020).
23. https://en.wikipedia.org/wiki/Mask_(computing)#Toggling_bit_values (consulté le 30 janvier 2020).

Instruction	Effet	Contraintes	*Flags* affectés
`bt X, Y`	Aucun	`X` = registre ou variable de 16, 32 ou 64 bits `Y` = registre de 16, 32 ou 64 bits ou immédiat sur 8 bits	`CF` ← valeur initiale du bit de rang `Y` de `X` `ZF` n'est pas modifié `OF` et `SF` sont indéfinis
`bts X, Y`	Le bit de rang `Y` de `X` est mis à 1		
`btr X, Y`	Le bit de rang `Y` de `X` est mis à 0		
`btc X, Y`	Le bit de rang `Y` de `X` est complémenté		

(a) Résumé.

```
          ax  :  0110100101011001b
      bt ax, 2
          ax  :  0110100101011001b
          CF  :  0
-------------------------------------
          ax  :  0010001001100110b
     bts ax, 0
          ax  :  0010001001100111b
          CF  :  0
-------------------------------------
          ax  :  0111011000011111b
    btr ax, 11
          ax  :  0111011000011111b
          CF  :  0
-------------------------------------
          ax  :  0101010010001001b
     btc ax, 3
          ax  :  0101010010000001b
          CF  :  1
```

(b) Exemples.

TABLE 3 – Instructions `bt`, `bts`, `btr` et `btc`.

4 Instructions de manipulation de bits : `bt`, `bts`, `btr`, `btc`

L'instruction `bt` [24] (*bit test*) teste *un* bit précis d'un motif binaire donné. Les instructions `bts` [25] (*bit test and set*), `btr` [26] (*bit test and reset*) et `btc` [27] (*bit test and complement*) testent également un bit *avant* de le mettre à 1, 0 ou de le complémenter, respectivement.

Le premier opérande de ces instructions est un registre ou une variable de 16, 32 ou 64 bits. Leur second opérande est un registre de 16, 32 ou 64 bits ou un immédiat sur 8

24. https://www.felixcloutier.com/x86/bt (consulté le 30 janvier 2020).
25. https://www.felixcloutier.com/x86/bts (consulté le 30 janvier 2020).
26. https://www.felixcloutier.com/x86/btr (consulté le 30 janvier 2020).
27. https://www.felixcloutier.com/x86/btc (consulté le 30 janvier 2020).

bits. Si le deuxième opérande est un registre, il doit être de même taille que le premier.

Ces quatre instructions copient dans le *carry flag* (`CF`) le bit du premier opérande dont le rang est fourni via le second opérande. C'est la partie *test*, à laquelle se limite `bt`. Ensuite, l'instruction :

— `bts` met ce bit du premier opérande à 1 ;
— `btr` met ce bit du premier opérande à 0 ;
— `btc` complémente ce bit du premier opérande.

Le *zero flag* (`ZF`) n'est pas modifié. L'*overflow flag* (`OF`) et le *sign flag* (`SF`) sont indéfinis.

La Table 3(a) donne un résumé de ces instructions. La Table 3(b) en donne des exemples d'utilisation.

5 Exercices

Pour réaliser les exercices qui suivent, vous ne pouvez utiliser que les instructions étudiées au long des TD01 et 02 : `mov`, `not`, `and`, `or`, `xor`, `bt`, `bts`, `btr`, `btc`, `syscall` et `nop`.

Ex. 1 Remplissez, *à la main*, c'est-à-dire sans employer l'ordinateur, les valeurs des registres et *flags* dans le code ci-dessous. Dans un deuxième temps, exécutez ce programme dans `KDbg` (le code source est fourni en annexe). Vérifiez que le contenu des registres, y compris `rflags`, est conforme à vos réponses.

```
1  ; 1_comprehension_log.asm
2
3  global main
4
5  section .text
6  main:
7      mov al, 10011101b
8      not al              ; al = ........, zf = ., sf = .
9
10     mov al, 11100101b
11     mov ah, 00101010b
12     and al, ah          ; al = ........, zf = ., sf = .
13
14     mov al, 11100101b
15     mov ah, 00001010b
16     and al, ah          ; al = ........, zf = ., sf = .
17
18     mov al, 01100101b
19     mov ah, 01010101b
20     or  al, ah          ; al = ........, zf = ., sf = .
21
```

```
22      mov al, 11100101b
23      mov ah, 01010101b
24      or  al, ah          ; al = ........, zf = ., sf = .
25
26      mov dx, 1100010011100101b
27      mov si, 0011000001100010b
28      xor dx, si          ; dx = ................
29                          ; zf = ., sf = .
30
31      mov al, 11100101b
32      mov ah, 11100101b
33      xor al, ah          ; al = ........, zf = ., sf = .
34
35      ; fin
36      mov rax, 60
37      mov rdi, 0
38      syscall
39
```

Notes sur KDbg Le contenu de `rflags` apparaît dans la section *Flags* de la vue (*View*) *Registers*. Il est possible de basculer entre une vue binaire et une vue plus explicite nommant les drapeaux levés en cliquant droit sur la ligne `eflags` et en sélectionnant *Binaire* ou *Défaut GDB*, respectivement, dans le menu surgissant.

Il est possible d'observer le contenu des registres comme `rax` dans la vue *Registers*. Cependant, si on n'est intéressé que par le contenu de `ah` et qu'on n'a pas envie de fouiller dans la représentation binaire de `rax`, il est possible de *surveiller* `ah`. Pour ce faire, il faut ouvrir la vue *Expressions surveillées* (*Watched Expressions*). Dans la zone d'édition, fournissez le nom du registre à guetter précédé du symbole `$` [28].

Attention cependant, les registres `r8b`, `r9b`, etc. jusque `r15b` sont inaccessibles via ces noms dans `KDbg`. Il faut utiliser les noms `r8l`, `r9l`, etc. jusque `r15l` pour accéder à leurs contenus. Donc, pour voir le contenu de `r8`, `r8d`, `r8w` et `r8b` dans `KDbg`, il faut surveiller les expressions `$r8`, `$r8d`, `$r8w` et `$r8l`, respectivement.

Il est par ailleurs possible de choisir le format d'affichage [29] des informations. Ainsi, pour examiner le contenu de `ah` en binaire, on doit fournir l'expression : `/t $ah` tandis que pour voir le contenu de ce même registre comme un caractère, il faut surveiller l'expression : `/c $ah`.

28. https://sourceware.org/gdb/current/onlinedocs/gdb/Registers.html#Registers (consulté le 30 janvier 2020).

29. https://sourceware.org/gdb/current/onlinedocs/gdb/Output-Formats.html#Output-Formats (consulté le 30 janvier 2020).

b_7 b_6 b_5					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 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

FIG. 2 – Table `ASCII` (Illustration Wikipedia [a]).

a. https://en.wikipedia.org/wiki/File:ASCII_Code_Chart-Quick_ref_card.png (consulté le 30 janvier 2020).

Ex. 2 En utilisant la table `ASCII` [30] présentée à la FIG. 2, remplissez les pointillés du code source suivant de manière à convertir le caractère minuscule en majuscule, à l'aide d'un masque (voir commentaires dans le code source).

```
1  ; 2_ascii_minVersMaj_partiel.asm
2
3  global main
4
5  section .text
6  main:
7      nop          ; ne fait rien
8
9      mov al, 'd' ; on charge une lettre minuscule dans al
10
11     ; ........       ; à compléter de sorte que ah contienne
12     ; ........       ; la même lettre que al mais en majuscule.
```

30. Attention !, la numérotation des bits qui y est utilisée commence à b_1 au lieu de b_0, contrairement à la convention standard actuelle.

```
13      ; ........      ; cela doit fonctionner sans modifier al,
14                      ; pour toutes les lettres. on suppose que
15                      ; le contenu de al est bien une lettre minuscule.
16
17      ; fin
18      mov rax, 60
19      mov rdi, 0
20      syscall
21
```

Ex. 3 Recodez le programme de l'exercice précédent à l'aide d'une ou plusieurs instructions de manipulation de bits au lieu d'utiliser des masques.

Ça ne compile pas ? Utilisez `bx` comme destination plutôt que `ah`.

Ex. 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.

Notions à retenir

Registre `rflags`, indicateurs d'état `CF`, `OF`, `SF` et `ZF`, instructions logiques `not`, `and`, `or` et `xor`, masquage, instructions de manipulation de bits `bt`, `bts`, `btr` et `btc`.

Références

[1] *Intel© 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, octobre 2017. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[2] Igor Zhirkov. *Low-Level Programming*. Apress, 2017. https://www.apress.com/gp/book/9781484224021.