Chapitre 3

Format d'instruction et modes d'adressage

3.1 Le format d'instruction

Considérons une opération dyadique (par exemple l'addition) notée $\diamond$. Lorsqu'un processeur doit exécuter une telle opération, il doit connaître trois objets op_1, op_2, et op_3. L'action sur les trois objets peut être notée $op_1 \leftarrow op_2 \diamond op_3$.

Ces trois objets sont situés en mémoire, que ce soit la mémoire centrale ou la mémoire interne au processeur. On parle d'adressage parce que ces trois objets sont accédés par leur adresse. On peut distinguer quatre types de machines :

- machine à pile (sans adresses), tous les opérandes sont dans la pile, le résultat est lui aussi rangé dans la pile (pensez aux machines à calculer de Hewlett-Packard).
- machine à une adresse, $A \leftarrow A \diamond op_3$, ici l'accumulateur sert à la fois d'opérande et de destination du résultat.
- machine à deux adresses, $op_2 \leftarrow op_2 \diamond op_3$, ici un des deux opérandes donne aussi la destination du résultat.
- machine à trois adresses $op_1 \leftarrow op_2 \diamond op_3$, c'est le cas le plus général.

Par ailleurs une des instructions les plus fréquemment utilisée est la copie d'une donnée d'un endroit à un autre, et on peut la représenter sous la forme : $op_1 \leftarrow op_2$. Du fait que deux opérandes interviennent dans cette instruction, un choix de machine à deux adresses peut permettre une certaine *orthogonalité* dans la définition et le codage des instructions.

3.2 Modes d'adressage

Les *modes d'adressage* définissent le moyen d'accéder à un des opérandes mentionnés dans la section précédente. La terminologie n'est pas universellement admise. Nous utilisons ici la terminologie de Intel.

registre R, l'opérande est dans un registre (Exemple : `MOVL %EAX,%ECX`)

immédiat n, l'opérande est dans l'instruction (Exemple : `MOVL $0x0004FFFF,%EAX`)

direct $mem(n)$, l'adresse de l'opérande est dans l'instruction (Exemple : `MOVL 0x0004FFFF,%EAX`)

registre indirect $mem(R)$, l'adresse de l'opérande est dans un registre (Exemple : `MOVL (%EAX),%EBX`)

basé $mem(R + d)$ (Exemple : `MOVL %EAX,(%EBX+20)`)

indexé $mem(R + d)$ (Exemple : `MOVB TABLE(%ESI),%AL`)

indexé avec facteur d'échelle $mem(R * k + d)$ (Exemple : `ADDL %EAX,TABLE(%ESI*4)`)

basé indexé $mem(R_1 + R_2)$ (Exemple : `MOVL %EAX,(%EBX)(%ESI)`)

basé indexé avec facteur d'échelle $mem(R_1 * k + R_2)$ (Exemple : `MOVL %ECX,(%EDX*8)(%EAX)`)

basé indexé avec déplacement $mem(R_1 + R_2 + d)$ (Exemple : `MOVL %EDX,(%ESI)(%EBP+0FEEEE0h)`)

basé indexé avec facteur d'échelle et avec déplacement $mem(R_1 * k + R_2 + d)$
(Exemple : `MOVL %EAX,TABLE(%ESI*4)(%EBP+80h)`)

Pour que l'adressage soit parfaitement orthogonal, il faudrait pouvoir utiliser n'importe quel mode d'adressage pour chacun des opérandes. Cependant, parce que cela conduirait à des codes d'instructions très longs, ceci est rarement réalisé.

Adresse effective Dans l'adressage absolu, l'instruction contient l'adresse mémoire à laquelle on doit accéder. Pour les autres modes d'adressage en mémoire centrale un calcul peut être nécessaire : le résultat de ce calcul s'appelle l'*adresse effective* (en anglais : Effective Address, EA). On trouvera donc assez souvent l'acronyme **EA** soit dans le nom de certaines instructions, soit dans leur description.

3.3 Quelques exemples d'architecture

3.3.1 Le 8080 de Intel

A	F
B	C
D	E
H	L

SP
PC

Quelques instructions Ici, `r` et `s` désignent chacun un registre 8 bits parmi A, B, C, D, E, H, ou L ; `n` désigne une donnée sur 8 bits.

`MOV r, s`	$r \leftarrow s$
`MOV A, M`	$r \leftarrow mem(HL)$
`ADD r`	$A \leftarrow A + r$
`ADI n`	$A \leftarrow A + n$

3.3.2 Le Z80 de Zilog

A	F	A'	F'
B	C	B'	C'
D	E	D'	E'
H	L	H'	L'

IX
IY
SP
PC

Quelques instructions Ici, `r` et `s` désignent chacun un registre 8 bits parmi A, B, C, D, E, H, ou L ; `n` désigne une donnée sur 8 bits ; et `nn` désigne une donnée sur 16 bits.

`LD r, s`	$r \leftarrow s$
`LD r, (HL)`	$r \leftarrow mem(HL)$
`LD A, (BC)`	$A \leftarrow mem(BC)$
`LD A, (nn)`	$A \leftarrow mem(nn)$
`ADD A, r`	$A \leftarrow A + r$
`ADD A, n`	$A \leftarrow A + n$

3.3.3 Le 8086 de Intel

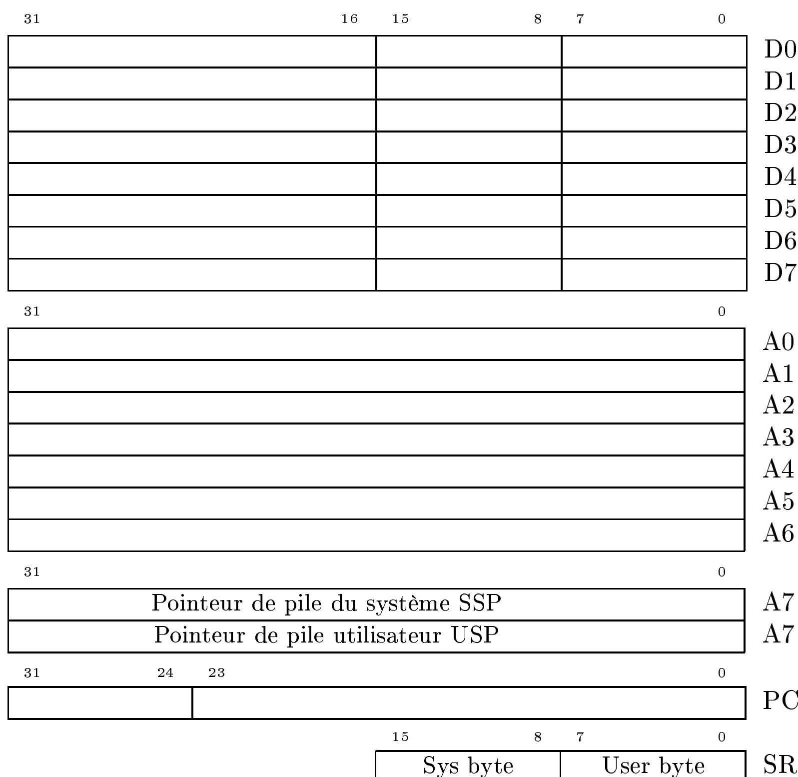
AH	AL
BH	BL
CH	CL
DH	DL
SP	
BP	
SI	
IP	
FH	FL
CS	
DS	
SS	
ES	

Quelques instructions Ici, `r` et `s` désignent chacun un registre 8, ou 16 bits parmi AH, BH, CH, DH, AL, BL, CL, DL, AX, BX, CX, ou DX, `n` désigne une donnée sur 8 bits ou 16 bits ; et `nn` désigne une donnée sur 16 bits.

`MOV r, s`	$r \leftarrow s$
`MOV r, (BX)`	$r \leftarrow mem(BX + 16 * DS)$
`MOV r, (nn)`	$r \leftarrow mem(nn + 16 * DS)$
`ADD r, s`	$r \leftarrow r + s$
`ADD r, n`	$r \leftarrow r + n$

3.3.4 Le 68000 de Motorola

3.3.4.1 Les registres du 68000



Dans le registre d'état, chaque bit est un indicateur :

- Bit 0 = Drapeau de retenue (*Carry flag*),
- Bit 1 = Drapeau de dépassement (*Overflow flag*),
- Bit 2 = Drapeau de zéro (*Zero flag*),
- Bit 3 = Drapeau de négativité (*Negative flag*),
- Bit 4 = Drapeau d'extension (*Extend flag*),
- Bits 8–10 = Masque d'interruption,
- Bit 13 = Indicateur du mode superviseur,
- Bit 15 = Indicateur du mode TRACE.

15							8	7							0
T	0	S	0	0	I_2	I_1	I_0	0	0	0	X	N	Z	V	C

Quelques instructions Ici, `r` et `s` désignent chacun un registre de données ou d'adresses ; `a` désigne un registre d'adresses ; `n` désigne une donnée sur 16 bits ; et `nn` désigne une donnée sur 32 bits.

`MOVE s, r`	$r \leftarrow s$
`MOVE (a), r`	$r \leftarrow mem(a)$
`MOVE nn, r`	$r \leftarrow mem(nn)$
`ADD s, r`	$r \leftarrow r + s$
`ADD #n, r`	$r \leftarrow r + n$

3.4 Le codage des instructions

3.4.1 Le codage du 8086

3.4.1.1 Opérande mémoire

Les valeurs combinées de `mod` sur 2 bits et `r/m` sur 3 bits, et éventuellement `dl` et `dh` tout deux sur 8 bits, permettent de désigner un opérande selon les règles suivantes.

- si `mod` vaut `00` alors `DISP` vaut 0, `dl` et `dh` sont absents,
- si `mod` vaut `01` alors `DISP` vaut `dl` étendu en mode signé sur 16 bits,
- si `mod` vaut `10` alors `DISP` vaut `dh :dl`,
- si `mod` vaut `11` alors `r/m` désigne un registre.

Dans les trois premiers cas, `r/m` désigne un emplacement mémoire, selon le tableau suivant :

`r/m`	Adresse effective
000	EA=BX+SI+DISP
001	EA=BX+DI+DISP
010	EA=BP+SI+DISP
011	EA=BP+DI+DISP
100	EA=SI+DISP
101	EA=DI+DISP
110	EA=BP+DISP
111	EA=BX+DISP

Ces trois champs apparaissent dans les instructions sous la forme suivante :

`mod` `r/m`	`dl`	`dh`

3.4.1.2 Opérande registre

Cette numérotation est utilisée soit dans la forme `reg` ou `r/m` lorsque `mod` vaut `11` ; de plus pour chacune des instructions un bit `w` indique s'il est nul que l'instruction porte sur des octets, et dans le cas contraire que l'instruction porte sur des mots de 16 bits.

16 bits (w=1)		8 bits (w=0)	
000	AX	000	AL
001	CX	001	CL
010	DX	010	DL
011	BX	011	BL
100	SP	100	AH
101	BP	101	CH
110	SI	110	DH
111	DI	111	BH

3.4.1.3 Codage du MOVE du 8086

Registre/mémoire depuis/vers registre

1 0 0 0 1 0 d w	mod reg r/m

Si `d` vaut 1 alors `reg`←`r/m`, sinon `r/m`←`reg`.
De registre à registre, cette instruction s'exécute en 2 cycles ; de mémoire à registre, en 8+EA cycles ; de registre à mémoire, en 9+EA cycles.

Opérande immédiat vers registre/mémoire

1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data low	data high

`r/m`←`data` en 10+EA cycles d'horloge.
Si `w` vaut 0, `data high` est absent.

Opérande immédiat vers registre

1 0 1 1 w reg	data	data

`reg`←`data` en 4 cycles d'horloge.
Si `w` vaut 0, `data high` est absent.

Opérande absolu vers accumulateur

1 0 1 0 0 0 0 w	al	ah

Si `w` vaut 0, `AL`←`smem(ah :al)`, sinon `AX`←`smem(ah :al+1) :smem(ah :al)` (en 10 cycles d'horloge).

Accumulateur vers opérande absolu

1 0 1 0 0 0 1 w	al	ah

Si `w` vaut 0, `smem(ah :al)`←`AL`, sinon `smem(ah :al) :smem(ah :al)`←`AX` (en 10 cycles d'horloge).

3.4.2 Quelques instructions du 80X86

`andl`	`src`, `dest`	effectue l'opération : `dest := dest AND src`.	`andl $1, %eax`
`cmpl`	`src`, `dest`	(CoMPare) effectue l'opération : `dest - src`. Le résultat de cette soustraction n'est pas envoyé dans `dest`, mais les indicateurs sont positionnés selon le résultat de cette opération.	`cmpl $0, val` `cmpl $255, -4(%ebp)`
`decl`	`dest`	(DECrement) effectue l'opération : `dest := dest - 1`.	`decl ptr` `decl %eax`

`enter` `size, level` — alloue de la place mémoire dans la pile d'exécution pour réaliser le passage des paramètres des langages de haut niveau. Le premier paramètre `size` indique combien d'octets doivent être alloués dans la pile pour la procédure entamée. Le second opérande indique le niveau d'imbrication (0 à 31) d'une procédure par rapport au programme source. En langage C, toutes les fonctions sont de niveau 0 (il n'y a pas de fonctions imbriquées).
Cette instruction réalise les opérations suivantes lorsque `level` est nul : `PUSHL %EBP ; MOVL %ESP,%EBP ; SUBL SIZE,%ESP`.

```
enter 12,0
```

`imull` `src, dest` — (Integer MULtiplication) effectue l'opération : `dest := dest` × `src`.

```
imull -4(%ebp), %eax
imull i, %eax
```

`incl` `dest` — (INCrement) effectue l'opération : `dest := dest + 1`.

```
incl (%eax)
incl %eax
incl i
```

`j`*`cond`* `dest` — (Jump ...) effectue un branchement à l'adresse désignée par `label` si la condition *cond* est vraie. Les différentes conditions sont une combinaison des lettres suivantes :

A	After	(CF==0) || (ZF==0)
B	Before	CF==1
C	Carry	CF==1
E	Equal	ZF==1
G	Greater	(SF==OF) && (ZF==0)
L	Lower	(SF !=OF)
N	Non	
O	Overflow	OF==1
P	Parity	PF==1
S	Sign	SF==1
Z	Zero	ZF==1

Les conditions valides sont JA, JAE, JB, JBE, JC, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JO, JP, JPE, JS, JZ, JNO, JNP, JNPE, JNS, JNZ.
La destination est donnée relativement au pointeur d'instruction.

```
je .L8
jne .L7
```

`jmp` `dest` — (JuMP) effectue un branchement inconditionnel à l'adresse désigné par `dest`.

```
jmp .L2
```

`leal` `src, dest` — (Load Effective Address) calcule l'adresse effective référencée par l'opérande `src` et la mémorise dans le registre `dest`

```
leal 0(,%eax,4), %ecx
leal -4(%ebp), %eax
```

`leave`		effectue l'opération inverse de ENTER. La zone mémoire allouée par ENTER est désalloué en ramenant %ESP au niveau de %EBP. Puis, l'ancienne valeur de %EBP qui avait été sauvegardée par ENTER dans la pile est restaurée : `MOVL %EBP,%ESP ; POPL %EBP`	`leave`
`movb`	`src`, `dest`	(MOVe) copie l'opérande d'un octet `src` dans l'opérande `dest`.	`movb $48, (%eax)`

`movl`	`src`, `dest`	(MOVe) copie l'opérande de 4 octets `src` dans l'opérande `dest`.	`movl $0, i`
			`movl %eax, %eax`
			`movl %eax, (%ecx,%edx)`
			`movl %esp, %ebp`
			`movl ptr, %eax`
			`movl $tableau, %edx`

`popl`	`dest`	restaure un mot dans `dest` en le dépilant.	`popl %ebp`
`pushl`	`src`	empile l'opérande dans la pile. Le pointeur de pile est d'abord décrémenté de 4, puis l'opérande est placé au sommet de la pile pointé par %ESP.	`push %ebp`
`ret`		(RETurn) effectue le retour d'une routine. L'adresse de la prochaine instruction à exécuter est dépilée.	`ret`
`testl`	`src`, `dest`	(TEST) effectue l'opération : `dest AND src` et positionne les drapeaux en conséquence, mais ne range pas le résultat quelque part.	`testl %eax, %eax`
`xorl`	`src`, `dest`	(eXclusive OR) effectue l'opération : `dest := dest XOR src`.	`xor %eax,%eax`

3.5 Exercices

Exercice 3.1 (Langage d'assemblage et adressage : le 8086) *Voici un source en langage C qui contient :*

- *la déclaration d'une variable globale entière* `i`*,*
- *la déclaration d'un tableau* `tableau` *de 256* `ints`*,*
- *la définition d'une procédure* `global_fill_tab`*,*

```
int i;
int tableau[256];

void global_fill_tab (void)
{
  i = 0;
  while (i < 256)
    {
    tableau[i] = i * i;
    i++;
    }
} /* global_fill_tab */
```

La procédure `global_fill_tab` *contient une boucle qui peut être graphiquement représentée comme sur la figure 3.1.*

Ce source une fois compilé génère du code machine. Le voici avec son interprétation en langage d'assemblage (ce qui est quand même plus lisible que le code machine en hexadécimal).

```
 1                    global_fill_tab:
 2   0000 55                  pushl   %ebp
 3   0001 89E5                movl    %esp, %ebp
 4   0003 C7050000            movl    $0, i
 5        00000000
 6        0000
 7   000d 8D7600              .p2align 2
 8                    .L3:
 9   0010 813D0000            cmpl    $255, i
10        0000FF00
11        0000
12   001a 7E04                jle     .L5
13   001c EB2E                jmp     .L4
14   001e 89F6                .p2align 2
15                    .L5:
16   0020 A1000000            movl    i, %eax
17        00
18   0025 89C0                movl    %eax, %eax
19   0027 8D0C8500            leal    0(,%eax,4), %ecx
20        000000
21   002e BA000000            movl    $tableau, %edx
22        00
23   0033 A1000000            movl    i, %eax
24        00
25   0038 0FAF0500            imull   i, %eax
26        000000
27   003f 890411              movl    %eax, (%ecx,%edx)
28   0042 FF050000            incl    i
29        0000
30   0048 EBC6                jmp     .L3
31   004a 89F6                .p2align 2
32                    .L4:
33   004c 5D                  popl    %ebp
34   004d C3                  ret
35                    .Lfe1:
36                            .comm   i,4,4
37                            .comm   tableau,1024,32
```

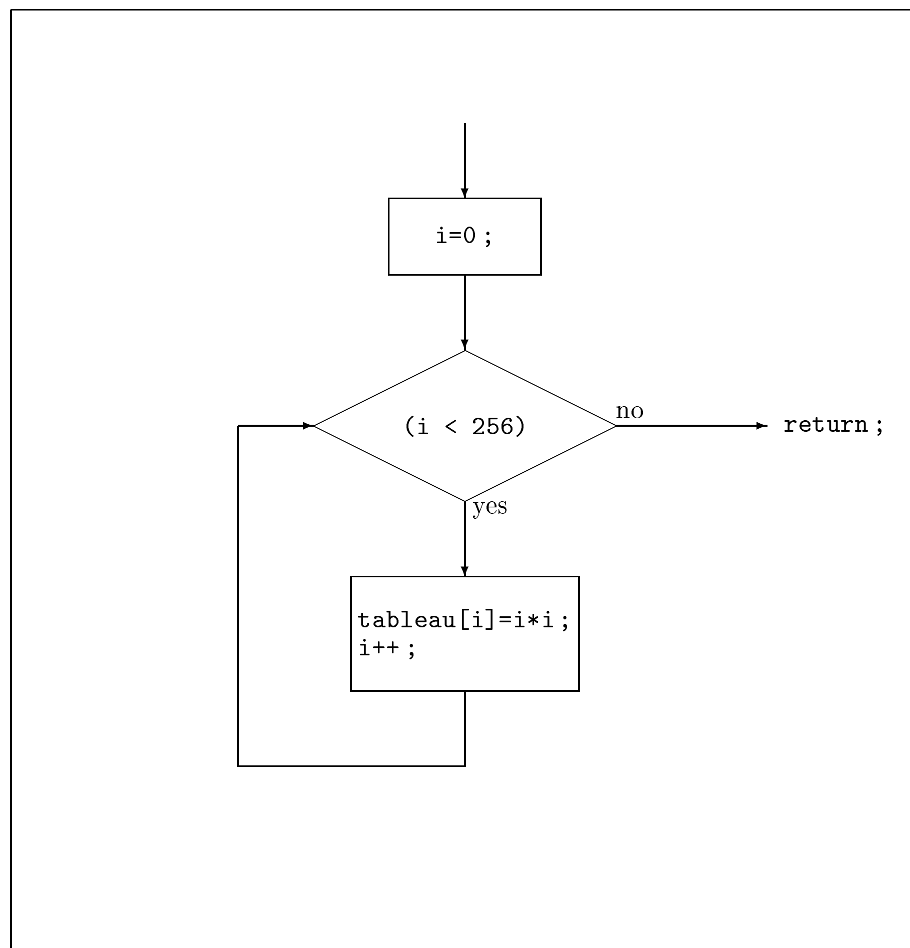


FIG. 3.1 – Organigramme de la procédure `global_fill_tab`

Question 3.1.1 *Redessinez l'organigramme de la procédure* `global_fill_tab` *en y incluant cette fois le texte en langage d'assemblage.*

Question 3.1.2 *A quoi correspondent sur l'organigramme les labels* `.L3` `.L4` *et* `.L5` *du listage ?*

Question 3.1.3 *Comment accède-t'on à la variable* `tableau[i]` *dans la procédure* `global_fill_tab` *? Sur quelles lignes est calculée l'adresse de cette variable ? A quelle ligne est effectivement réalisée l'affectation ?*

Question 3.1.4 *Combien d'instructions sont exécutées par une exécution de la procédure* `local_fill_tab` *? En supposant que chaque instruction soit exécutée en exactement 6 cycles d'horloge*[1]*, et une horloge à 500 MHz, combien de temps dure une exécution de la procédure* `global_fill_tab` *?*

On considère maintenant la procédure `local_fill_tab`*. La seule différence avec la procédure* `global_fill_tab` *est l'utilisation d'une variable locale j au lieu de la variable globale pour indexer et remplir le tableau* `tableau`*. Voilà le source en langage C :*

```
1   int tableau[256];
2
3   void local_fill_tab (void)
4   {
5     int j;
6
7     j = 0;
8     while (j < 256)
9       {
10      tableau[j] = j * j;
11      j++;
12      }
13  } /* local_fill_tab */
```

et voilà le résultat de la compilation :

[1] C'est une approximation relativement abusive, en fait c'est plutôt une moyenne. Les instructions ont des durées d'exécution très variable, pour le 8086 le minimum est de quatre cycles, les plus longues (multiplications et divisions) peuvent prendre jusqu'à 40 cycles !

```
1                    local_fill_tab:
2    0000 55                 pushl   %ebp
3    0001 89E5               movl    %esp, %ebp
4    0003 83EC04             subl    $4, %esp
5    0006 C745FC00           movl    $0, -4(%ebp)
6         000000
7    000d 8D7600             .p2align 2
8                    .L3:
9    0010 817DFCFF           cmpl    $255, -4(%ebp)
10        000000
11   0017 7E03               jle     .L5
12   0019 EB25               jmp     .L4
13   001b 90                 .p2align 2
14                   .L5:
15   001c 8B45FC             movl    -4(%ebp), %eax
16   001f 89C0               movl    %eax, %eax
17   0021 8D0C8500           leal    0(,%eax,4), %ecx
18        000000
19   0028 BA000000           movl    $tableau, %edx
20        00
21   002d 8B45FC             movl    -4(%ebp), %eax
22   0030 0FAF45FC           imull   -4(%ebp), %eax
23   0034 890411             movl    %eax, (%ecx,%edx)
24   0037 8D45FC             leal    -4(%ebp), %eax
25   003a FF00               incl    (%eax)
26   003c EBD2               jmp     .L3
27   003e 89F6               .p2align 2
28                   .L4:
29   0040 C9                 leave
30   0041 C3                 ret
31                   .Lfe1:
32                           .comm   tableau,1024,32
```

Question 3.1.5 *Quelles sont les différences dans ce résultat de compilation et celui de la procédure* ***global_fill_tab*** *?*

Exercice 3.2 *Voilà un listage en langage d'assemblage d'une fonction* `binprint` *:*

```
1                    binprint:
2    0000 55                 pushl   %ebp
3    0001 89E5               movl    %esp, %ebp
4    0003 C7050000           movl    $buffer+32, ptr
5         00002000
6         0000
7    000d A1000000           movl    ptr, %eax
8         00
9    0012 C60000             movb    $0, (%eax)
10   0015 FF0D0000           decl    ptr
11        0000
12   001b 833D0000           cmpl    $0, val
13        000000
14   0022 7514               jne     .L3
15   0024 A1000000           movl    ptr, %eax
16        00
17   0029 C60030             movb    $48, (%eax)
18   002c A1000000           movl    ptr, %eax
19        00
20   0031 89C0               movl    %eax, %eax
21   0033 EB4B               jmp     .L2
22   0035 8D7600             .p2align 2
23                   .L3:
24                           .p2align 2
25                   .L5:
26   0038 833D0000           cmpl    $0, val
27        000000
28   003f 7503               jne     .L7
29   0041 EB35               jmp     .L6
30   0043 90                 .p2align 2
31                   .L7:
32   0044 A1000000           movl    val, %eax
33        00
34   0049 83E001             andl    $1, %eax
35   004c 85C0               testl   %eax, %eax
36   004e 7410               je      .L8
37   0050 A1000000           movl    ptr, %eax
38        00
39   0055 C60031             movb    $49, (%eax)
40   0058 FF0D0000           decl    ptr
41        0000
42   005e EB0E               jmp     .L9
43                           .p2align 2
44                   .L8:
45   0060 A1000000           movl    ptr, %eax
46        00
47   0065 C60030             movb    $48, (%eax)
48   0068 FF0D0000           decl    ptr
49        0000
50                   .L9:
51   006e C13D0000           sarl    $1, val
52        000001
53   0075 EBC1               jmp     .L5
54   0077 90                 .p2align 2
55                   .L6:
56   0078 A1000000           movl    ptr, %eax
57        00
58   007d 40                 incl    %eax
59   007e 89C0               movl    %eax, %eax
60                   .L2:
61   0080 5D                 popl    %ebp
62   0081 C3                 ret
63                   .Lfe1:
64                           .comm   buffer,33,32
65                           .comm   ptr,4,4
66                           .comm   val,4,4
```

Question 3.2.1 *Dessiner l'organigramme de cette fonction.*

Question 3.2.2 *Cette fonction est analogue au programme que nous avons exécuté dans l'exercice 1.1 de la page 17. Etablir un isomorphisme entre ce listage et ce programme de la page 17 en répondant qux questions suivantes : A quoi correspondent le caractère* `'0'` *et le caractère* `'1'` *? A quoi correspondent* `mem[20]` *et* `mem[21]` *? A quelles lignes du programme de la page 17 correspondent les différentes étiquettes ? A quelles instructions correspondent les différentes lignes du programme de la page 17 ? A quelle ligne correspond l'identificateur* `binstr` *?*

Real computer scientists don't program in assembler. ~~They don't write in anything less portable than a number~~ two pencil.