Activité y86 :

A l'aide du simulateur Y86 (http://dept-info.labri.fr/ENSEIGNEMENT/archi/js-y86/index.html), on charge le fichier « cryptage.ys » suivant :

```
.pos 0

irmovl 8, %ecx
irmovl 0, %edi
boucle:
mrmovl image1(%edi), %eax
mrmovl image2(%edi), %ebx
xorl %eax, %ebx
rmmovl %ebx,image3(%edi)
iaddl 4, %edi
isubl 1, %ecx
jne boucle
halt

.pos 0x0040
image1:
.long 0x00ff0f00
.long 0x00ff0f00
.long 0x0ff00000
.long 0xf0ffff00
.long 0x00f0000f
.long 0x000f0f00
.long 0xf0000f00
.long 0x0f000f00

.pos 0x0070
image2:
.long 0xf00ff0f0
.long 0xf0f0f00f
.long 0x0f0f0f0f
.long 0xff000ff0
.long 0x0f00ff0f
.long 0x00f0f0f0
.long 0xff0ff00f
.long 0x0f0f0ff0

.pos 0x00a0
image3:
```

En 0x0040 (label Image1) l'image de départ à crypter est stockée sous forme d'entiers hexadécimaux d'une longueur de 4 octets (soit 32 bits). Chacun de ces nombres est composé de 8 chiffres hexadécimaux. On a choisi de représenter un pixel noir par « f » (1111 en binaire) et un pixel blanc par « 0 » (0000 en binaire). L'intérêt de ce choix est qu'après une opération XOR, on obtiendra forcément soit « 0 », soit « f ». *(Propriété à faire vérifier aux élèves ?)*

En 0x0070 (label Image2), on trouve l'image correspondant à la clef de cryptage : il s'agit d'une image où l'état des pixels a été choisi aléatoirement.

Les deux images sont les suivantes :

Image de départ :

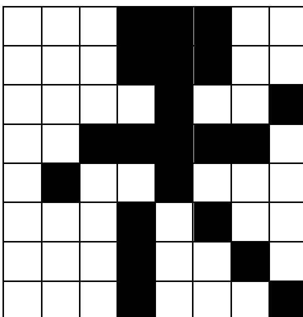
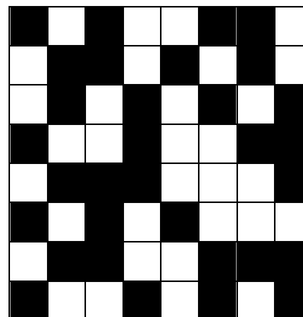


Image clef :



On peut les voir dans la colonne « Memory » du simulateur Y86 :

y86 Simulator

dept-info.labri.fr/ENSEIGNEMENT/archi/js-y86/index.html

Y86 Simulator | Assemble | Reset | Step | Start | Load | Examples | Wiki | GitHub

SOURCE CODE

```
1  .pos 0
2
3  irmovl 8, %ecx
4  irmovl 0, %edi
5  boucle:
6  mrmovl image1(%edi), %eax
7  mrmovl image2(%edi), %ebx
8  xorl %eax, %ebx
9  rmmovl %ebx,image3(%edi)
10 iaddl 4, %edi
11 isubl 1, %ecx
12 jne boucle
13 halt
14
15 .pos 0x0040
16 image1:
17 .long 0x00ff0f00
18 .long 0x00ff0f00
19 .long 0x0ff00000
20 .long 0xf0ffff00
21 .long 0x00f0000f
22 .long 0x000f0f00
23 .long 0xf0000f00
24 .long 0x0f000f00
25
26 .pos 0x0070
27 image2:
28 .long 0xf00ff0f0
29 .long 0xf0f0f00f
30 .long 0x0f0f0f0f
31 .long 0xff000ff0
32 .long 0x0f00ff0f
33 .long 0x00f0f0f0
34 .long 0xff0ff00f
35 .long 0x0f0f0ff0
36
37 .pos 0x00a0
38 image3:
39
```

OBJECT CODE

```
0x0000:              | .pos 0
                     |
0x0000: 30f108000000 | irmovl 8, %ecx
0x0006: 30f700000000 | irmovl 0, %edi
0x000c:              | boucle:
0x000c: 500740000000 | mrmovl image1(%edi), %eax
0x0012: 503770000000 | mrmovl image2(%edi), %ebx
0x0018: 6303         | xorl %eax, %ebx
0x001a: 4037a0000000 | rmmovl %ebx,image3(%edi)
0x0020: c0f704000000 | iaddl 4, %edi
0x0026: c1f101000000 | isubl 1, %ecx
0x002c: 740c000000   | jne boucle
0x0031: 00           | halt
                     |
0x0032:              | .pos 0x0040
0x0040:              | image1:
0x0040: 000fff00     | .long 0x00ff0f00
0x0044: 000fff00     | .long 0x00ff0f00
0x0048: 0000f00f     | .long 0x0ff00000
0x004c: 00fffff0     | .long 0xf0ffff00
0x0050: 0f00f000     | .long 0x00f0000f
0x0054: 000f0f00     | .long 0x000f0f00
0x0058: 000f00f0     | .long 0xf0000f00
0x005c: 000f000f     | .long 0x0f000f00
                     |
0x0060:              | .pos 0x0070
0x0070:              | image2:
```

REGISTERS		
%eax	0x00000000	0
%ecx	0x00000000	0
%edx	0x00000000	0
%ebx	0x00000000	0
%esp	0x00000000	0
%ebp	0x00000000	0
%esi	0x00000000	0
%edi	0x00000000	0

FLAGS

SF 0 ZF 0 OF 0

STATUS

STAT AOK
ERR
PC 0x0000

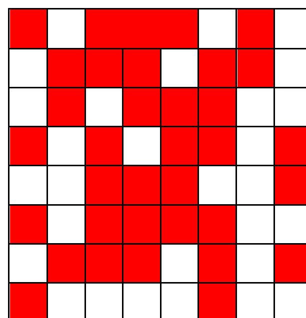
MEMORY

ADDR	VALUE
0020	c0f70400
0024	0000c1f1
0028	01000000
002c	740c0000
0030	00000000
0034	00000000
0038	00000000
003c	[illegible]
0040	000fff00
[illegible]	000fff00
[illegible]	0000f00f
[illegible]	00fffff0
[illegible]	0f00f000
[illegible]	000f0f00
[illegible]	000f00f0
[illegible]	000f000f
0060	00000000
0064	[illegible]
0068	00000000
006c	[illegible]
0070	f0f00ff0
[illegible]	0ff0f0f0
[illegible]	0f0f0f0f
[illegible]	f00f00ff
[illegible]	0fff000f
[illegible]	f0f0f000
[illegible]	0ff00fff
[illegible]	f00f0f0f
0090	00000000
0094	[illegible]
0098	00000000
009c	00000000
00a0	00000000
00a4	00000000
00a8	00000000
00ac	00000000
00b0	00000000
00b4	00000000
00b8	00000000

La différence entre l'aspect des images dans la dernière colonne et l'écriture des nombres dans le code source vient du fait que y86 fonctionne en « big endian », donc le nombre hexadécimal « 12345678 » sera stockée en mémoire sous la forme « 78563412 ». Il faudra y penser si on veut coder une autre image.

On fait fonctionner le programme pas à pas : bouton « Assemble », puis « Step ».

- Que se passe-t-il à chaque pas ?
- Lors de chaque instruction d'affectation, regardez la valeur de la case mémoire ou du registre concerné.
- Le programme charge dans les registres %eax et %ebx les nombres correspondant aux lignes successives des images 1 et 2. Quelle opération mathématique réalise-t-il entre ces deux registres ?
- A la fin de l'exécution du programme, que retrouve-t-on à l'adresse 0x00a0 (label Image3) ?
- Représenter ci-dessous l'image obtenue à la fin du programme :



- Modifier le code afin d'appliquer l'algorithme à l'image précédente. Attention à ne pas modifier l'image clef et à bien penser au fait que les nombres sont stockés en mémoire en « big endian ».
- Retrouve-t-on l'image de départ ?

<u>Prolongement :</u>

Faire un programme Python permettant de réaliser l'opération XOR entre les pixels d'une image à crypter et d'une image bruitée. Réaliser deux fois l'opération pour montrer qu'on retrouve l'image de départ :