

Cryptage d'une image par un programme assembleur.

Thématique :

Architecture

Pré-requis :

- en représentation des données : on supposera connue l'écriture d'un entier positif dans une base $b \geq 2$ notamment les bases 2 et 16.
- en langage et programmation : on supposera connues les constructions éléments (affectation, conditions et boucles).

Objectif :

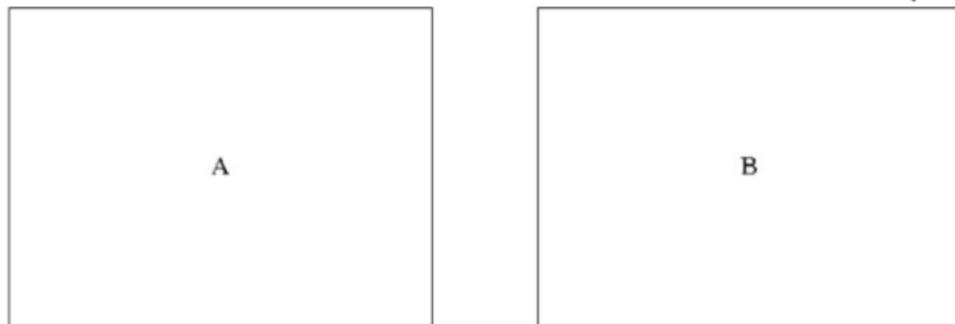
Cryptage d'une image en assembleur avec l'opérateur xor.

Les contenus du programme :

Événements clés de l'histoire de l'informatique	Situer dans le temps les principaux événements de l'histoire de l'informatique et leurs protagonistes.	Ces repères historiques seront construits au fur et à mesure de la présentation des concepts et techniques.
Modèle d'architecture séquentielle (von Neumann)	Distinguer les rôles et les caractéristiques des différents constituants d'une machine. Dérouler l'exécution d'une séquence d'instructions simples du type langage machine.	La présentation se limite aux concepts généraux. On distingue les architectures monoprocesseur et les architectures multiprocesseur. Des activités débranchées sont proposées. Les circuits combinatoires réalisent des fonctions booléennes.

<p>Valeurs booléennes : 0, 1.</p> <p>Opérateurs booléens : and, or, not.</p> <p>Expressions booléennes</p>	<p>Dresser la table d'une expression booléenne.</p>	<p>Le ou exclusif (xor) est évoqué.</p> <p>Quelques applications directes comme l'addition binaire sont présentées.</p> <p>L'attention des élèves est attirée sur le caractère séquentiel de certains opérateurs booléens.</p>
--	---	--

Activité d'introduction

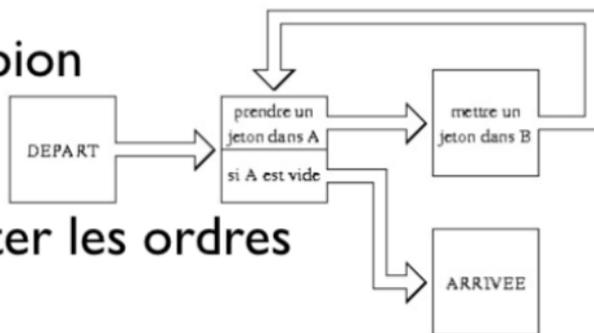


(+ 1 réserve)

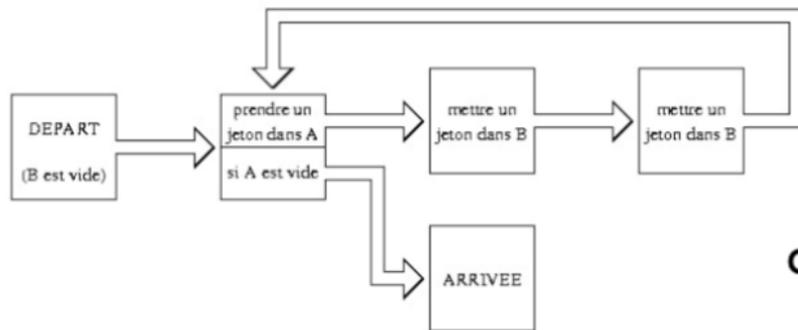
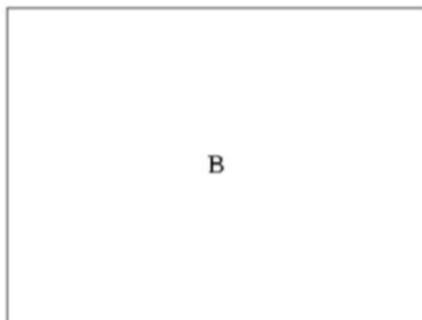
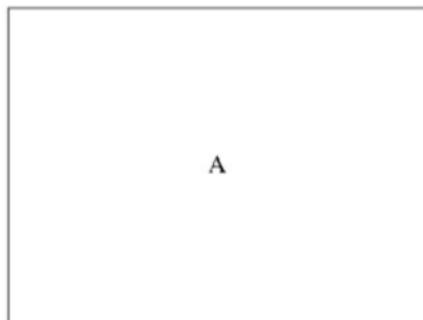
poser le pion

ici 

et exécuter les ordres

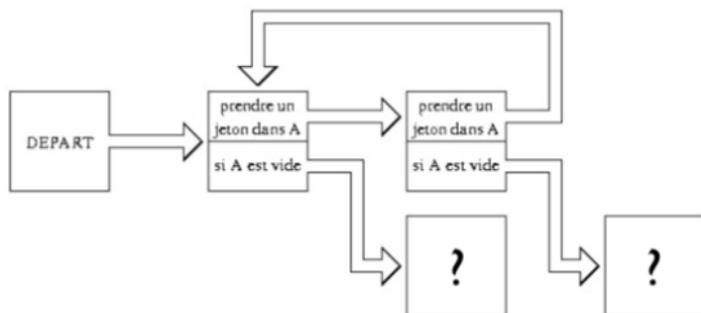
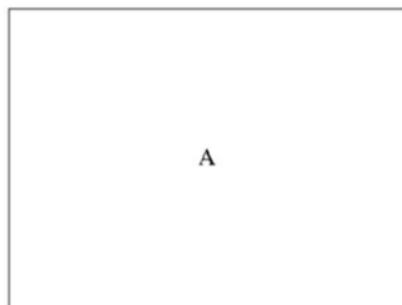


Quelle
opération
calcule cette
machine ?



**Quelle
opération
calcule cette
machine ?**

I seul registre :



Quelle
propriété
teste cette
machine ?

Le Cours

Il est organisé en trois parties :

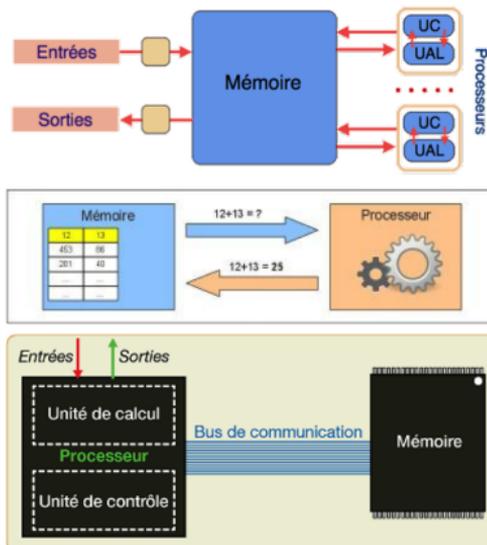
1. L'architecture des ordinateurs.
2. Le langage machine
3. Les portes logiques

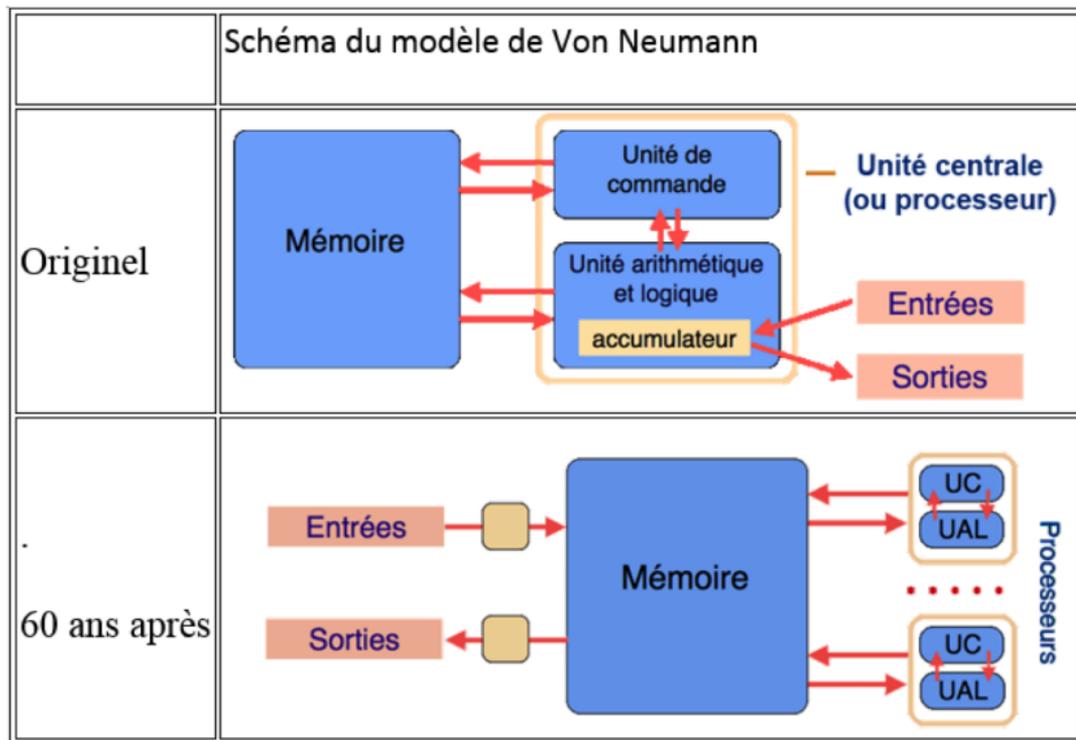
1. L'architecture des ordinateurs.

L'ordinateur minimal : Modèle de Von Neumann.

John Von Neumann est à l'origine d'un modèle de machine universelle de traitement programmé de l'information (1946). Cette architecture sert de base à la plupart des systèmes à microprocesseur actuel. Elle est composée des éléments suivants :

- une unité centrale
- une mémoire principale
- des interfaces d'entrées/sorties





2. Le langage machine

- Un ordinateur doit être capable d'exécuter un programme. Il faut donc un moyen d'indiquer au processeur la séquence des instructions qu'il doit exécuter.
- Pour échanger des données avec la mémoire, le processeur utilise deux procédés qui permettent l'un de transférer l'état d'un registre dans une case mémoire et l'autre de transférer l'état d'une case mémoire dans un registre.

- Les trois opérations utilisées sont :
 - load/store (Chargement/Rangement du registre accumulateur avec le contenu de l'adresse correspondante)
 - add/sub/..., and/or, ... (Addition du contenu de l'accumulateur avec le contenu de l'adresse correspondante)
 - Jump/boucle/... (Instruction de branchement, Notion de bloc, de code conditionnel, Boucles ...)

```
1
2  .pos 0
3      mrmovl N, %eax
4      xorl %ecx, %ecx
5  boucle:
6      iaddl 5, %ecx
7      isubl 1, %eax
8      jne boucle
9      halt
10
11  .align 4
12  N:  .long 10
```

3. Les portes booléennes

Au commencement était le transistor, puis nous créâmes les portes booléennes et, à la fin de la journée, les ordinateurs.

Un ordinateur :

- est construit autour d'un ensemble de circuits électroniques (ça passe ou ça ne passe pas) ;
- ces circuits s'appellent des portes booléennes ou parfois portes logiques ;
- il traite donc que des signaux assimilables à 0 ou 1.

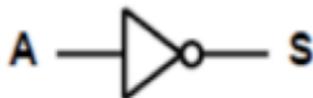
Les opérateurs booléens (George Boole, milieu du XIX^{ème} siècle) :

- partie des mathématiques qui s'intéresse aux opérations et fonctions sur les variables logiques ;
- utilisés dans la conception des circuits électroniques.

NEGATION

x	NOT x
F	V
V	F

x	$\neg x$
0	1
1	0



Négation des inégalités	Exemple
$\text{NOT } (x < y) = x \geq y$	$\text{NOT } (x < 0) = x \geq 0$
$\text{NOT } (x \leq y) = x > y$	$\text{NOT } (x \leq 1) = x > 1$
$\text{NOT } (x > y) = x \leq y$	$\text{NOT } (x > -2) = x \leq -2$
$\text{NOT } (x \geq y) = x < y$	$\text{NOT } (x \geq 2012) = x < 1012$

AND (ET) : cette fonction prend le ET logique

x	y	x AND y
F	F	F
F	V	F
V	F	F
V	V	V



x	y	x.y	Dans un ordinateur
0	0	0	
0	1	1	
1	0	0	
1	1	1	

OR (OU) : Cette fonction prend le OU logique

x	y	x OU y
F	F	F
F	V	V
V	F	V
V	V	F



x	y	x+y	Dans un ordinateur
0	0	0	
0	1	1	
1	0	1	
1	1	0	

XOR (OU EXCLUSIF) :

cette fonction prend le ou exclusif, elle n'est vrai que si une seule de ses entrées est vrai

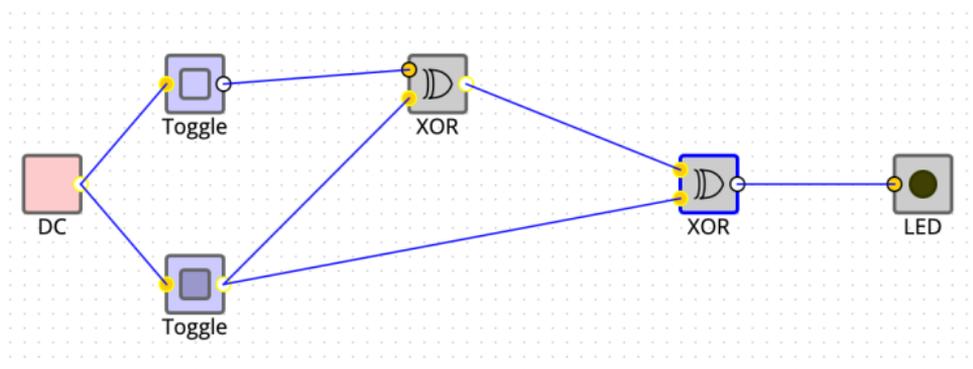
x	Y	x <u>XOR</u> y
F	F	F
F	V	V
V	F	V
V	V	F

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0



Exercice

Construire la table de vérité schématisée par le schéma ci-dessous :



Activité Y86

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

```
.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 0xf0fff000
.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 :

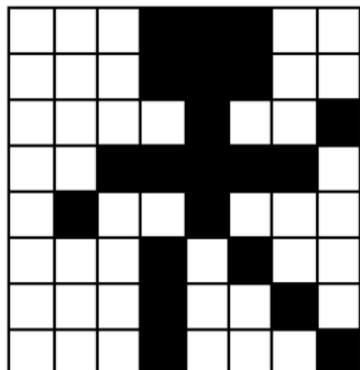
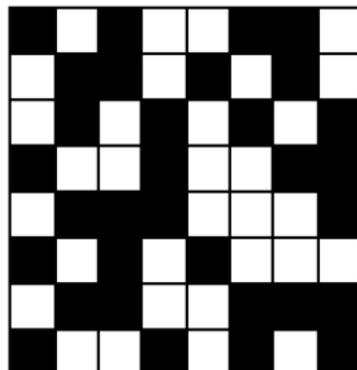


Image clef :



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

The screenshot displays the Y86 Simulator interface with the following sections:

- SOURCE CODE:**

```

1 .pos 0
2 irmovl 8, %ecx
3 irmovl 0, %edi
4 boucle:
5   mrmovl image1(%edi), %eax
6   mrmovl image2(%edi), %ebx
7   xorl %eax, %ebx
8   rmmovl %ebx, image3(%edi)
9   iaddl 4, %edi
10  jsuhl 1, %ecx
11  jne boucle
12  halt
13
14 .pos 0x0040
15 image1:
16 .long 0xb0ff0f00
17 .long 0xb0ff0f00
18 .long 0xb0ff0f00
19 .long 0xb0ff0f00
20 .long 0xb0ffff00
21 .long 0xb000000f
22 .long 0xb0000f00
23 .long 0xf0000f00
24 .long 0xbf000f00
25
26 .pos 0x0070
27 image2:
28 .long 0xf00ff0f0
29 .long 0xf00f0f0f
30 .long 0xf00f0f0f
31 .long 0xf0000f0f
32 .long 0xb0000f0f
33 .long 0xb0000f0f
34 .long 0xf00f0f0f
35 .long 0xf00f0f0f
36
37 .pos 0x00a0
38 image3:
39

```
- OBJECT CODE:**

```

0x0000: | .pos 0
0x0006: 30f700000000 | irmovl 8, %ecx
0x000c: 30f700000000 | irmovl 0, %edi
0x0010: | boucle:
0x001c: 500740000000 | mrmovl image1(%edi), %eax
0x0022: 503770000000 | mrmovl image2(%edi), %ebx
0x0028: 6303 | xorl %eax, %ebx
0x002e: 4037a0000000 | rmmovl %ebx, image3(%edi)
0x0034: c0f704000000 | iaddl 4, %edi
0x003a: c1f010000000 | jsuhl 1, %ecx
0x0040: 740c000000 | jne boucle
0x0046: 00 | halt
0x004c: |
0x0050: | .pos 0x0040
0x0054: 000ff000 | image1:
0x0058: 000ff000 | .long 0xb0ff0f00
0x005c: 000ff000 | .long 0xb0ff0f00
0x0060: 000ff000 | .long 0xb0ff0f00
0x0064: 000ff000 | .long 0xb0ff0f00
0x0068: 00ffff00 | .long 0xb0ffff00
0x006c: 0000000f | .long 0xb000000f
0x0070: 00000f00 | .long 0xb0000f00
0x0074: 00000f00 | .long 0xf0000f00
0x0078: 00000f00 | .long 0xbf000f00
0x007c: 00000f00 | .long 0xbf000f00
0x0080: |
0x0084: | .pos 0x0070
0x0088: | image2:
0x0094: 00000000 |
0x0098: 00000000 |
0x009c: 00000000 |
0x00a0: 00000000 |
0x00a4: 00000000 |
0x00a8: 00000000 |
0x00ac: 00000000 |
0x00b0: 00000000 |
0x00b4: 00000000 |
0x00b8: 00000000 |
0x00bc: 00000000 |
0x00c0: 00000000 |
0x00c4: 00000000 |
0x00c8: 00000000 |
0x00cc: 00000000 |
0x00d0: 00000000 |
0x00d4: 00000000 |
0x00d8: 00000000 |
0x00dc: 00000000 |
0x00e0: 00000000 |
0x00e4: 00000000 |
0x00e8: 00000000 |
0x00ec: 00000000 |
0x00f0: 00000000 |
0x00f4: 00000000 |
0x00f8: 00000000 |
0x00fc: 00000000 |

```
- MEMORY:**

ADDR	VALUE
0020	c0f70400
0024	0000c1f1
0028	01000000
002c	740c0000
0030	00000000
0034	00000000
0038	00000000
003c	00000000
0040	000ff000
0044	000ff000
0048	000ff000
004c	00ffff00
0050	0000000f
0054	00000f00
0058	00000f00
005c	00000f00
0060	00000f00
0064	00000f00
0068	00000f00
006c	00000f00
0070	00000f00
0074	00000f00
0078	00000f00
007c	00000f00
0080	00000f00
0084	00000f00
0088	00000f00
008c	00000f00
0090	00000000
0094	00000000
0098	00000000
009c	00000000
00a0	00000000
00a4	00000000
00a8	00000000
00ac	00000000
00b0	00000000
00b4	00000000
00b8	00000000
00bc	00000000
00c0	00000000
00c4	00000000
00c8	00000000
00cc	00000000
00d0	00000000
00d4	00000000
00d8	00000000
00dc	00000000
00e0	00000000
00e4	00000000
00e8	00000000
00ec	00000000
00f0	00000000
00f4	00000000
00f8	00000000
00fc	00000000
- REGISTERS:**

Register	Value
%eax	0x00000000
%ecx	0x00000000
%edx	0x00000000
%ebx	0x00000000
%esp	0x00000000
%ebp	0x00000000
%esi	0x00000000
%edi	0x00000000
- FLAGS:**

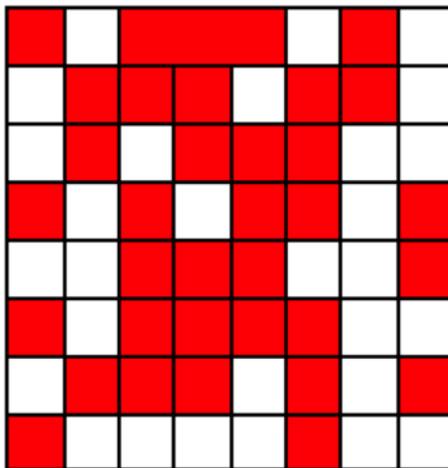
Flag	Value
SF	0
ZF	0
OF	0
STATUS	
ERR	0x0000
PC	0x0000

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 ?

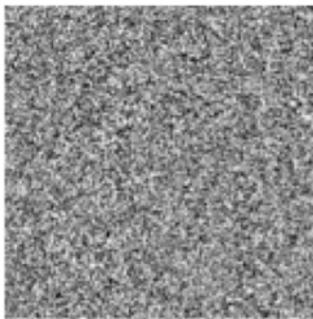
Prolongement

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 :

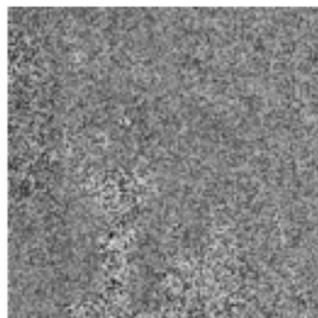
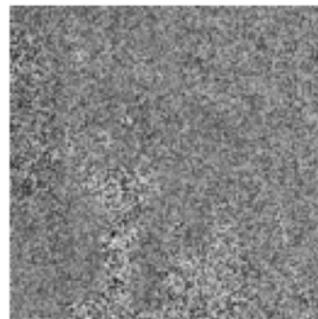
I



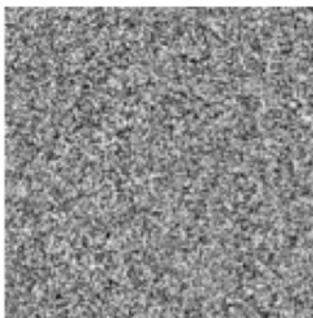
+



=



+



=

