Tutorial #8: De la lecture de l'assembleur

A présent il est temps pour vous d' apprendre à lire le langage assembleur, et surtout le comprendre. Sans cette étape primordiale, ne comptez pas comprendre un jour ce que vous faites: vous n'arriverez pas à reverser un programme sans suivre à la lettre le cours qui va avec, car vous n'aurez aucune idée de ce que vous êtes censés faire. Suivez attentivement ce descriptif (des bases nécessaires, il y a bien sur bien plus de choses à connaître que ce qui suit, mais ceci vous donnera déjà un bon aperçu) et n'hésitez pas à le relire si besoin pour bien vous en imprégner.

I. Pieces, bits et bytes:

- **BIT** - La plus petite partie de donnée existante. Le bit peut être 0 ou 1. En mettant un groupe de bits ensemble, on se retrouve avec le "systeme de nombres binaires".

  ex: 00000001 = 1 00000010 = 2 00000011 = 3 etc.

- **BYTE** - Un byte comprend 8 bits. Il peut avoir une valeur maximale de 255 (0-255). Pour rendre la compréhension du système binaire plus facile, on utilise le système "hexadécimal". C'est un système "sur base de 16", tandis que le binaire est un système "sur base de 2".

- **WORD** - Un word est simplement 2 bytes ensemble, soit 16 bits. Un Word peut avoir une valeur maximale de 0FFFFh (soit 65535d, ou h veut dire "en hexadécimal", et d "en décimal").

- **DOUBLE WORD** - Syntaxe DWORD: Un DWORD consiste en deux WORD ensemble, soit un maximum de 32 bits. Valeur maximale = 0FFFFFFFFh (ou encore 4294967295d).

- **KILOBYTE** - Un KILOBYTE n'est pas égal à 1000 bytes comme on pourrait le croire, il équivaut à 1024 (32*32) bytes.

- **MEGABYTE** - Un MEGABYTE n'est pas non plus egal à 1 million de bytes, mais (1024*1024) 1,048,578 bytes.

II. Les Registres:

Les registres sont les endroits spéciaux dans la mémoire de l'ordinateur ou l'on peut stocker des données. Imaginez un registre comme une petite boite dans laquelle on peut mettre quelque chose: un nom, un nombre, une phrase...

Sur les WinTel CPU moyens d'aujourd'hui, on peut avoir des registres de 932 bits (sans **registres flag**, cf plus loin). Ce sont les suivants:

EAX:	Extended Accumulator Register	/ Registre Accumulateur étendu
EBX:	Extended Base Register	/ Registre de Base étendu
ECX:	Extended Counter Register	/ Registre Compteur étendu
EDX:	Extended Data Register	/ Registre de Données étendu
ESI:	Extended Source Index	/ Index de Source étendu
EDI:	Extended Destination Index	/ Index de Destination étendu
EBP:	Extended Base Pointer	/ Pointeur de Base étendu
ESP:	Extended Stack Pointer	/ Pointeur de Pile étendu
EIP:	Extended Instruction Pointer	/ Pointeur d'Instruction étendu

Généralement la taille des registres est de 32 bits (= 4 bytes). Il peuvent contenir de l'information entre 0 et FFFFFFFFh (non-signés). Auparavant, la plupart des registres avaient un fonction définie comme leur nom l'indique, tel que ECX = Compteur, mais aujourd'hui on peut utiliser **presque** n'importe registre comme compteur. Je vous expliquerai les EAX, EBX, ECX, EDX, ESI et EDI lorsque j'aborderai les fonctions qui utilisent ces registres. Il nous reste donc EBP, ESP et EIP:

EBP: EBP a à voir avec la pile (= Stack en Anglais) et les stack frames. Rien de bien important lorsqu'on débute.

ESP: ESP pointe vers la pile d'un procédés en cours. La pile est l'endroit où l'on place des informations pour une utilisation ultérieure (voir les explications sur les instructions PUSH / POP).

EIP: EIP pointe toujours vers la prochaine instruction à exécuter.

Cependant il faut aussi savoir autre chose sur les registres: bien qu'ils aient une taille de 32 bits, certaines parties de ceux-ci (de taille 16 ou 8 bits) ne peuvent pas être adressées directement.

Registre 32 bit	Registre 16 bit	Registre 8 bit
EAX	AX	AH / AL
EBX	BX	BH / BL
ECX	CX	CH / CL
EDX	DX	DH / DL
ESI	SI	--------
EDI	DI	--------
EBP	BP	--------
ESP	SP	--------
EIP	IP	--------

Un registre est généralement utilisé comme suit:

|---------- **EAX**: 32bit (=1 DWORD =4BYTES) ----------|

|------- **AX**: 16bit (=1 WORD =2 BYTES) ----|

|- **AH**:8bit (=1 BYTE)- **|** - **AL**:8bit (=1 BYTE)-|

EAX est un registre de 32 bits, **AX** de 16 bits, **AL** et **AH** deux registres de 8 bits.
AX représente la partie "basse" de **EAX**.

AH est la partie "haute" de **AX** (H= High) et **AL** la basse de **AX** (L=Low).

Ainsi quand EAX=1234abcd ---> AX=abcd AL=cd et AH=ab.

[De manière générale, AX est utilisé lors des opérations arithmétiques, CX comme compteur et DX pour les données.]

Remarque: les registres 32 bits commencent tous par un E.

Au passage, 4 bytes = 1 DWORD, et 2 bytes = 1 WORD.

Cela nous permet de faire des distinctions quant à la taille:

- **I. Registre d'1 byte:** Comme leur nom l'indique, ce sont des registres d'une taille de 1 Byte. Cela ne veut pas dire que le registre entier de 32 bits est chargé de donnée! Au bout d'un moment, les espaces vides dans un registre sont simplement comblés avec des 0. Voici les registres d'un byte, soit 8 bits:
  - AL et AH
  - BL et BH
  - CL et CH
  - DL et DH
- **II. Registres WORD (word-size):** Font 1 WORD (= 2 bytes = 16 bits). Un registre de taille 1 WORD est constitué de registres de 2 bytes. La encore, on peut les séparer selon leur utilité:
  - 1. Registres d'utilité générale:

AX (taille de 1 WORD) = AH + AL -> Le '+' ne veut PAS dire qu'on les ajoute. AH et AL existent de manière indépendante, ce qui veut dire que si l'on change AH ou AL (ou les deux), AX va changer aussi!

-> 'accumulateur': Utilisé pour faire des opérations mathématiques, stocker des strings ('chaines' en français)...

BX -> 'base': Utilisé en conjonction avec la pile (stack).

CX -> 'counter' Compteur.

DX -> 'data': En général, c'est ici que le reste des opérations mathématiques est stocké.

DI -> 'destination index': Une chaine va être copiée dans **DI.**

SI -> 'source index': Une chaine va être copiée dans **SI.**

- 2. Registres d'Index:

BP -> 'base pointer': Pointe vers une position spécifiée sur la pile (cf plus loin) .
SP -> 'stack pointer': Pointe vers une position spécifiée sur la pile (cf plus loin) .

- 3. Registres de Segment:

CS -> 'code segment': Instructions qu'une application doit exécuter (cf plus loin).
DS -> 'data segment': Données dont l'application a besoin.
ES -> 'extra segment':
SS -> 'stack segment': Nous trouverons ici la pile (voir plus loin).

- 4. Spécial:

IP -> 'instruction pointer': Pointe vers la prochaine instruction (pas besoin pour le moment).

- **III. Registres DWORD (Doubleword-size):** Font 2 words = 4 bytes = 32 bits. **EAX, EBX, ECX, EDX, EDI**…

  Comme mentionné précédemment, si l'on trouve un 'E' devant un registre 16 bits, cela signifie que l'on a affaire a un registre 32 bits. Par conséquent, AX = 16-bits -> EAX = La version 32-bits d' EAX.

III. Les flags:

Les Flags sont des bits *uniques* qui indiquent le statut de quelque chose. Le registre Flag sur les CPU modernes de 32 bits est de 32 bits. Il existe 32 Flags differents, mais ne vous en faites pas. En reversing, on n'en a besoin majoritairement que de 3: le Z-Flag, le O-Flag et le C-Flag. On a besoin de connaitre et de comprendre le statut de ces flags pour savoir si un saut va etre éxécuté ou pas. Ce registre est en fait une collection de flags de 1 bit. Pensez à un flag comme à un drapeau, ou comme à un feu rouge: le vert veut dire "OK", et le rouge "PAS OK". Un flag ne peut etre que **1** ou **0**, c'est-à-dire **fixé** ou **non-fixé**.

- **Le Z-Flag**:
  - Le Z-Flag (zero flag) est de loin le plus utile pour cracker. Il est utilisé dans à peu pres 90% des cas. Il peut etre **fixé** (statut = 1) ou 'nettoyé' (statut = 0) par plusieurs opcodes lorsque la derniere instruction éxécutée a pour resultat 0. Vous allez vous demander pourquoi CMP (voir plus loin) peut fixer le Flag Zero, puisqu'il compare quelque chose - mais comment peut-on avoir 0 en résultat d'une comparaison? Voyez plus loin pour la réponse...

- **Le O-Flag**:
  - Le O-Flag (overflow flag) est utilisé dans a peu pres 4% des cas dans les tentatives de cracking. Il est fixé (= 1) lorsque la derniere opération a changé le bit le plus grand du registre qui a eu le résultat d'une opération. Par exemple: EAX contient la valeur 7FFFFFFF. Si l'on faisait une opération qui incrémente EAX de 1, le 0-Flag serait alors fixé car l'opération a changé le bit le plus haut de EAX (qui n'est pas fixé dans 7FFFFFFF, mais qui l'est dans 80000000 - utilisez calc.exe pour convertir de l'hexadecimal en binaire, ou encore ce petit outil

auquel j'ai ajouté un patch). L'autre cas ou le 0-Flag peut etre fixé, c'est quand la valeur du registre de destination n'est pas 0 avant l'instruction, ni après.

- **Le C-Flag**:
  - Le C-Flag (Carry flag) est utilisé dans 1% des cas. Il est fixé si l'on ajoute une valeur au registre de manière à ce qu'il soit plus grand que FFFFFFFF, ou si l'on soustrait une valeur, de manière que la valeur du registre soit plus petite que 0.

IV. Segments et Offsets:

Un segment est un partie de la mémoire ou les instructions (CS), data (DS), le stack (SS) ou bien juste un extra segment (ES) sont stockées. Chaque segment est divisé en offset. Dans les applications 32 bits (Windows 95/98/ME/2000), ces offsets sont numérotés de 00000000 a FFFFFFFF -- **65536** parties de mémoires, ainsi 65536 adresses mémoires par segment. Voici la notation standard pour les segments et offsets:

SEGMENT : OFFSET = Ensemble, ils pointent vers un endroit spécifique (adresse) de la mémoire.

Représentez-le vous comme ceci:

Un segment = page dans un livre : Un offset = une ligne specifique de cette page.

V. Le Stack (Pile):

Le Stack est une partie de la mémoire ou l'on peut stocker différentes choses pour les utiliser par la suite. Imaginez-le comme un coffre ou le dernier objet déposé sera le premier a en sortir. L'image la plus simple est celle d'un évier ou l'on fait la vaisselle, et ou se trouve une pile d'assiettes. L'évier est la pile, et les assiettes sont les adresses mémoire (indiquées par les "stack pointers" - pointeurs de pile) dans ce segment de la pile.

La règle inévitable: la dernière assiette déposée sera la première sortie... nous allons voir tout de suite que la commande **PUSH** sauvegarde le contenu d'un registre sur le haut de la pile, et a l'inverse, la commande **POP** attrape la dernière sauvegarde de contenu d'un registre pour la placer dans un registre spécifique.

VI. Les INSTRUCTIONS *(ordre alphabetique)*:

Notez tout d'abord en tant que rappel que toute les valeurs Mnemoniques en Assembleur (instructions) sont TOUJOURS en hexadécimal.

La plupart des instructions ont deux opérateurs (selon le modele "**add EAX, EBX**").

Certaines n'en ont qu'une ("**not EAX**") ...

et d'autres trois ("**IMUL EAX, EDX, 64**").

Quand on trouve une instruction de type "**DWORD PTR [XXX]**", alors la valeur du **DWORD** (= 4 bytes, je rappelle) à l'offset mémoire **[XXX]** est signifiée.

IMPORTANT: Les bytes sont toujours sauvegardés dans un ordre inverse en mémoire (ce qu'on appelle le format "LITTLE ENDIAN").

La plupart des instructions à deux opérateurs peuvent etre utilisées de la manière suivante (ici, example avec: ADD):

- add eax,ebx // Registre, Registre
- add eax,123 // Registre, Valeur
- add eax,dword ptr [404000] // Registre, Dword Pointeur [valeur]
- add eax,dword ptr [eax] // Registre, Dword Pointeur [registre]
- add eax,dword ptr [eax+00404000] // Registre, Dword Pointeur [registre+valeur]
- add dword ptr [404000],eax // Dword Pointeur [valeur], Registre
- add dword ptr [404000],123 // Dword Pointeur [valeur], Valeur
- add dword ptr [eax],eax // Dword Pointeur [registre], Registre
- add dword ptr [eax],123 // Dword Pointeur [registre], Valeur
- add dword ptr [eax+404000],eax // Dword Pointeur [registre+valeur], Registre
- add dword ptr [eax+404000],123 // Dword Pointeur [registre+valeur], Valeur

ADD (Addition)

Syntaxe: ADD destination, source

L' instruction ADD ajoute une valeur au registre ou a une adresse mémoire. Elle peut etre utilisée de 3 manieres différentes:

Elle peut **fixer** le Z-Flag, le O-Flag et le C-Flag (ainsi que d'autres dont on n'a pas besoin pour le cracking).

AND (And Logique)

Syntaxe: AND destination, source

L'instruction AND utilise un AND logique sur deux valeurs. Cette instruction réinitialise toujours le Z-Flag, le O-Flag et le C-Flag.

Pour mieux comprendre AND, regardez ces deux valeurs binaires:

1 0 0 1 0 1 0 1 1 0

0 1 0 1 0 0 1 1 0 1

Si on leur applique un AND, le Résultat est **0 0 0 1 0 0 0 1 0 0**

Quand les 1 sont les uns au-dessus des autres, le résultat de cette partie est 1, sinon le résultat est 0. On peut utiliser calc.exe (calculatrice Windows) pour calculer AND assez facilement. Ceci dit, il existe d'autres outils que vous trouverez dans la section de téléchargement des outils.

CALL (Call)

Syntaxe: CALL quelque chose

L' instruction CALL pousse la RVA (Relative Virtual Address = Adresse Virtuelle) de l'instruction qui suit le CALL vers la pile puis appelle une sub-procedure.

CALL peut être utilisé des façons suivantes:

- CALL 404000 // La plus commune: appelle une adresse.
- CALL EAX // Appelle Registre - si EAX etait 404000, ce serait la meme chose qu'au-dessus.
- CALL DWORD PTR [EAX] // Appelle l'adresse stockee dans [EAX].

- CALL DWORD PTR [EAX+5] // Appelle l'adresse qui est stockee a [EAX+5]

CDQ (Convertit DWord (4Byte) en QWord (8 Byte))

Syntaxe: CQD

CDQ est une instruction qui panique toujours un peu les débutants lorsqu'elle apparait pour la 1ere fois. Elle est utilisée devant des divisions et ne fait rien d'autre que de fixer les bytes de EDX a la valeur du plus haut bit de EAX:

ex: Si EAX <80000000, alors EDX sera 00000000.

Si EAX >= 80000000, alors EDX sera FFFFFFFF.

CMP (Compare)

Syntaxe: CMP dest, source

Le CMP compare 2 chaines et peut fixer les Flags C, O, Z si le résultat correspond:

- CMP EAX, EBX // compare 'eax' avec 'ebx', puis fixe le Z-Flag s'ils sont égaux.
- CMP EAX,[404000] // compare 'eax' avec le dword en 404000
- CMP [404000],EAX // compare 'eax' avec le dword en 404000

DEC (Decremente)

Syntaxe: DEC quelquechose

DEC est utilisé pour décrémenter une valeur (donc : valeur = valeur -1).

On l'utilise comme suit:

- dec eax // décrémente eax
- dec [eax] // décrémente le dword stocké dans [eax]
- dec [401000] // décrémente le dword stocké dans [401000]
- dec [eax+401000] // décrémente le dword stocké dans [eax+401000]

Cette instruction peut fixer les Flags O et Z si le résultat correspond.

DIV (Division)

Syntaxe: DIV diviseur

DIV s'utilise pour diviser EAX avec un diviseur (division non-signée). Le dividende est toujours EAX, le résultat stocké dans EAX, la valeur modulaire dans EDX.

Pour clarifier:

- mov eax,64 // EAX = 64h = 100d : on donne a EAX la valeur 64h
- mov ecx,9 // ECX = 9 : on met la valeur 9h dans EAX.
- div ecx // Divise EAX par ECX

Apres la division EAX = 100 / 9, donc 0B et ECX = 100 MOD 9 = 1

Cette instruction peut fixer les Flags C, O et Z si le résultat correspond.

IDIV (Integer Division)

Syntaxe: IDIV divisor

IDIV marche pareil que DIV, sauf que IDIV est une division signée.

Cette instruction peut fixer les Flags C, O et Z si le résultat correspond.

IMUL (Integer Multiplication)

Syntaxe: IMUL value

IMUL dest,value,value

IMUL dest,value

- IMUL multiplie soit EAX par une valeur (IMUL value)
- ou il multiplie deux valeurs et les envoie vers un registre de destination (IMUL dest, value, value)
- ou il multiplie un registre avec une valeur (IMUL dest, value).

Si le résultat de la multiplication est trop grand pour rentrer dans le registre de destination, IMUL peut fixer les Flags C, O et Z.

INC (Incremente)

Syntaxe: INC registre

INC est le contraire de l'instruction DEC: il augmente une valeur de 1.

Cette instruction peut fixer les Flags O et Z.

JUMPS

Ce sont les instructions les plus importantes, vous le savez. Voici les différents types de sauts ainsi que les conditions qui doivent être remplies pour qu'ils soient éxécutés.

NB - les sauts en rouge sont tres importants, ceux en jaune assez importants. On rencontre les autres beaucoup moins souvent.

JA	Jump if (unsigned) above	- CF=0 and ZF=0
JAE	Jump if (unsigned) above or equal	- CF=0
JB	Jump if (unsigned) below	- CF=1
JBE	Jump if (unsigned) below or equal	- CF=1 or ZF=1
JC	Jump if carry flag set	- CF=1
JCXZ	Jump if CX is 0	- CX=0
JE	Jump if equal	- ZF=1
JECXZ	Jump if ECX is 0	- ECX=0
JG	Jump if (signed) greater	- ZF=0 and SF=OF (SF = Sign Flag)
JGE	Jump if (signed) greater or equal	- SF=OF
JL	Jump if (signed) less	- SF != OF (!= is not)
JLE	Jump if (signed) less or equal	- ZF=1 and OF != OF
JMP	Jump	- Jumps always
JNA	Jump if (unsigned) not above	- CF=1 or ZF=1
JNAE	Jump if (unsigned) not above or equal	- CF=1
JNB	Jump if (unsigned) not below	- CF=0
JNBE	Jump if (unsigned) not below or equal	- CF=0 and ZF=0
JNC	Jump if carry flag not set	- CF=0
JNE	Jump if not equal	- ZF=0
JNG	Jump if (signed) not greater	- ZF=1 or SF!=OF
JNGE	Jump if (signed) not greater or equal	- SF!=OF
JNL	Jump if (signed) not less	- SF=OF
JNLE	Jump if (signed) not less or equal	- ZF=0 and SF=OF
JNO	Jump if overflow flag not set	- OF=0
JNP	Jump if parity flag not set	- PF=0
JNS	Jump if sign flag not set	- SF=0
JNZ	Jump if not zero	- ZF=0

JO	Jump if overflow flag is set	- OF=1
JP	Jump if parity flag set	- PF=1
JPE	Jump if parity is equal	- PF=1
JPO	Jump if parity is odd	- PF=0
JS	Jump if sign flag is set	- SF=1
JZ	Jump if zero	- ZF=1

LEA (Load Effective Address)

Syntaxe: LEA dest,src

LEA est assez similaire à MOV. Il n'est tres utilisé pour sa fonction originale, mais plutôt comme une multiplication:

- lea eax, dword ptr [4*ecx+ebx]

ce qui donne a EAX la valeur 4*ecx+ebxde .

MOV (Move)

Syntaxe: MOV dest,src

Cette instruction est très facile et très courante. MOV copie les valeurs de la source (src) vers la destination (dest), et la la source reste ce qu'elle était avant et n'est pas modifiée.

Il y a des variantes de MOV:

- **MOVS / MOVSB / MOVSW / MOVSD EDI, ESI**: Ces variantes copient le Byte / Word / Dword vers lesquels ESI pointe dans l'espace vers lequel EDI pointe.
- **MOVSX:** MOVSX étend les opérandes de Byte ou Word a la taille de Dword et garde le signe de la valeur.

- **MOVZX:** MOVZX étend les opérandes de Byte ou Word a la taille de Word ou Dword et remplit le reste de l'espace avec des 0.

MUL (Multiplication)

Syntaxe: MUL valeur

Cette instruction est la même que IMUL, sauf qu'elle multiplie en non-signé.

Cette instruction peut fixer les Flags O, Z et F.

NOP (No Operation)

Syntaxe: NOP

Cette instruction ne fait absolument **RIEN**. C'est pour cela qu'on l'utilise aussi souvent en reversing!

OR (Inclusive Logique Or)

Syntaxe: OR dest,src

Cette instruction connecte 2 valeurs en utilisant la logique inclusive OR.

Pour mieux comprendre OR, regardez ces deux valeurs binaires:

1 0 0 1 0 1 0 1 1 0

0 1 0 1 0 0 1 1 0 1

Si on leur applique un OR, ça donne **1 1 0 1 0 1 1 1 1 1**

Ce n'est que quand on a deux 0 l'un au-dessus de l'autre que le résultat est 0. Autrement, on a un bit de 1.

Cette instruction peut réinitialiser (mettre à 0) les Flags O, Z et C.

POP

Syntaxe: POP dest

POP charge la valeur du byte / Word / Dword ptr [esp] and la transfère vers la destination. Additionnellement, cette instruction incremente la pile par la valeur qui a ete POPée de la pile, de manière que le prochain POP prenne la valeur qui suit.

PUSH (pousse)

Syntaxe: PUSH operand

PUSH est l'opposé de POP. Il stocke une valeur sur la pile et la décrémente par la taille de l'opérande qui a été PUSHée, de manière que ESP pointe vers la valeur qui a ete PUSHée.

REP / REPE / REPZ / REPNE / REPNZ

Syntaxe: REP / REPE / REPZ / REPNE / REPNZ *ins*

Répète la chaîne d'instruction suivante: jusqu'a ce que CX soit = 0, ou jusqu'a la condition indiquée (ZF=1, ZF=1, ZF=0, ZF=0) soit remplie. La valeur ins doit être une chaîne d'opération telle CMPS, INS,

LODS, MOVS, OUTS, SCAS, or STOS.

RET (Return)

Syntaxe: RET

RET digit

RET ne fait que de retourner sur une partie du code ayant été déjà atteinte en utilisant une instruction CALL.

RET digit nettoie la pile avant de s'éxécuter (retourner).

SUB (Soustraction)

Syntaxe: SUB dest,src

SUB est le contraire de ADD. Il soustrait la valeur de la source a celle de la destination et stocke le resultat dans dans la destination.

Cette instruction peut fixer les Flags O, Z et F.

TEST

Syntaxe: TEST operand1, operand2

Cette instruction est dans 99% des cas utilisee comme "TEST EAX, EAX". Elle fait un AND Logique mais ne sauvegarde pas de valeur. It performs a Logical

AND(AND instruction) but does not save the values.

Cette instruction ne peut fixer que le Z-Flag quand **EAX = 0** ou bien le réinitialise quand EAX est différent de 0. Les Flags O et C sont toujours réinitialisés (mis à 0).

XOR (Logique Exclusive)

Syntaxe: XOR dest,src

Cette instruction connecte 2 valeurs en utilisant un OR de Logique Exclusive (souvenez-vous que OR utilise une logique INCLUSIVE).

Cette instruction réinitialise le O-Flag et le C-Flag et peut ne fixer que le Z-Flag.

Pour mieux comprendre, observez les binaires ci-dessous:

1 0 0 1 0 1 0 1 1 0

0 1 0 1 0 0 1 1 0 1

Si on leur applique un OR, on a alors **1 1 0 0 0 1 1 0 1 1**

Quand 2 bits l'un au-dessus de l'autre sont les mêmes, le résultat est 0. Autrement le résultat est 1.

Le plus souvent, on voit la forme "**XOR, EAX, EAX**", qui va remette EAX a 0 puisque quand on XOR une valeur avec elle-même le résultat est toujours 0.

VII. Operations Logiques:

Voici ce qui s'utilise le plus souvent dans une table de référence:

OPERATION	SRCE	DEST	RESULAT
AND	1	1	1
	1	0	0
	0	1	1
	0	0	0
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
XOR	1	1	0
	1	0	1
	0	1	1
	0	0	0
NOT	0	n/a	1
	1	n/a	0

Tutorial traduit de l'Anglais, adapté et complété par mes soins. Merci a l'auteur.