L'Assembleur et l'architecture x86

Tuto écrit par Bushido [NTC]

http://xtx.free.fr

x86 est la dénomination d'une famille de microprocesseurs compatibles avec le jeu d'instructions de l'Intel 8086. Celle-ci couvre du 8086 (16 bits fabriqué par Intel en 1978) au Pentium Extreme Edition (microprocesseur multi cœur, ou multi-core) qui arrivent sur le marché. Ce qui va nous intéresser aujourd'hui c'est de découvrir de manière synthétique la structure du langage assembleur (le langage le plus proche du langage machine qui reste compréhensible pour l'homme) et ses communications avec le microprocesseur. On va aller directement à l'essentiel, c'est-à-dire tout ce qui pourra vous faire mieux maîtriser le debugging dans Olly, ou IDA par exemples (je ne parlerais pas de la programmation en ASM à proprement parler avec MASM par exemple ou un autre outil)...

Déjà pourquoi cracker des logiciels en assembleur? En fait contrairement à un langage de haut niveau (C++, Visual Basic, ...), il y a une correspondance entre le code assembleur et le langage machine, ainsi il est possible de traduire le code dans les deux sens sans perdre d'information. Voilà pourquoi l'analyse d'un programme en assembleur permet la modification exacte attendue; puisque que le processeur interprète un programme sous sa forme binaire (suite de 0 et de 1).

La pile de type LIFO :

Lien	Description
	Les principaux registres : Les données dont le processeur a besoin sont stockées dans ce que l'on appelle des registres (notés AX, BX, CX, DX, ...)
	fonctionnement du stack : Stack ou pile, c'est un espace mémoire ou l'on peut sauvegarder temporairement des données, étude du type LIFO (pile d'assiette ; -).

L'assembleur :

Lien	Description
	Les flags : Flag, ou fanion ou indicateur ou encore drapeau ; - p . Présentation des principaux bits caractérisant le FLAGS du registre EFLAGS.
	Le jeu d'instruction : Présentation des instructions assembleur qui vous seront utile pour bien appréhender le cracking, pour chaque instruction sera indiqué quel flag cité précédemment sera modifié.

Cracking "de base" :

Lien	Description
	Les sauts conditionnels : Description des 33 sauts conditionnels (nom, sur quel flag agit le saut, ...).
	Instruction fréquente : Vous pourrez y voir quelques techniques à connaitre pour optimiser la programmation en assembleur.

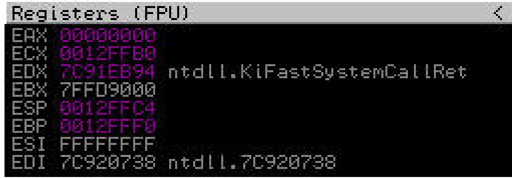
La pile de type LIFO

Les principaux registres :

Les données dont le processeur a besoin sont stockées dans ce que l'on appelle des registres (notés EAX, EBX, ECX, EDX, ...), ceux-ci sont nommés registres généraux (il existe aussi les registres de segment, et le registre d'état qu'on verra dans la rubrique des flags). Chacun de ces registres ont une taille de 32 bits et ont une utilité qui leur sont spécifique. Quand vous analysez un programme en assembleur (un crackme par exemple), le registre eax revient souvent sur le calcul avec des ADD ou sur des MOV par exemple, cela vient du fait que le registre eax est utilisé en tant qu'accumulateur (c'est-à-dire qu'il peut être utilisé pour tout calcul). Voilà une liste expliquant l'utilité de chaque registre :

Nom	Type	Description
EAX	accumulateur	tous calculs
EBX	base	adressage de tableaux (associé à DS), Registre auxiliaire de base
ECX	compteur	opérations de comptage (boucles, décalages...)
EDX	données	calculs (sauf ceux imposés sur AX)
EBP	pointeur	pointeur de base, associé à SS pour adressage indexé
ESP	pointeur	pointeur de pile
ESI	index source	index de tableau source
EDI	index destination	index tableau destination

Vous pouvez retrouver chacun de ces registres dans Olly, sur la fenêtre de droite, tout en haut :



Je vous ais sortie la description "brute" de chaque registre ce qui, pour un débutant, ne veut pas dire grand chose. C'est pour cela que je vais détailler les 4 premiers registres (EAX, EBX, ECX, EDX) parce que c'est ceux là qu'on va principalement rencontrer sous Olly.

Vous pouvez voir que pour chaque registre Olly affiche sa valeur correspondante à coté sous forme d'un nombre hexadécimal.

> Remarque :
>
> Faisons un petit calcul pour connaitre la taille en bit des registres : chaque registre est représenté par 8 chiffres en tout; en hexa 2 chiffres représentent 8 bits (0h = 0000 0000b, FFh = 1111 1111b), il y a en tout 4 blocs de 2 chiffres pour chaque registre; ce qui donne en binaire 4*8 (nombre de bloc * 8 bits) = 32. On retrouve bien la valeur de 32 bits pour chaque registre que j'ai cité plus haut).

Les registres AX, BX, CX et DX sont les registres les plus utilisés pour les calculs :

- Le registre **EAX** sert à effectuer des calculs arithmétiques ou à envoyer un paramètre à une interruption,
- Le registre **EBX** sert à effectuer des calculs arithmétiques ou bien des calculs sur les adresses,
- Le registre **ECX** sert généralement comme compteur dans des boucles,
- Le registre **EDX** sert à stocker des données destinées à des fonctions.

Si vous faîtes attention lors de vos prochaines séances de cracking vous pourrez constater par vous-même comment sont utilisés chacun de ses registres (stockage des serials, calcul dans des routines, ...).

Chose à savoir, on peut utiliser ces 4 registres par bloc d'un octet :

EAX (32 bits)		
	AX (16 bits)	
	AH (8 bits)	AL (8 bits)

Un petit exemple (en assembleur bien sur) :

Au départ EAX vaut 00112233, donc :

AX = 2233
AH = 22
AL = 33

Modifions la valeur de EAX :

MOV DWORD PTR DS:[ESI],EFCD26B0	; mets la valeur hexa EFCD26B0 dans ESI
MOV AL,BYTE PTR DS:[ESI]	; mets B0 dans AL » EAX devient 001122B0
INC ESI	; incrémente ESI
MOV AH,BYTE PTR DS:[ESI]	; mets 26 dans AH » EAX devient 001126B0

Comme vous le voyez, on peut modifier la valeur d'un registre soit directement en utilisant EAX, soit octet par octet en utilisant AX, AH, ou AL. Voilà, si vous avez du mal au niveau de la syntaxe, on verra ça plus loin dans le cours ; -) (ici pour les pressés).

Pour info, dans un registre 8 bits on peut représenter 256 valeurs différentes, dans un registre 16 bits on peut représenter 65536 valeurs différentes, dans un registre 32 bits on peut représenter 4 294 967 296 valeurs différentes.

Fonctionnement du stack :

La pile d'exécution (ou pile ou stack) est une zone particulière de la mémoire réservée à des stockages temporaires. Elle est utilisée quand on a plus suffisamment de registres pour sauvegarder une information, ce qui arrive très souvent vu le nombre de registres disponibles...

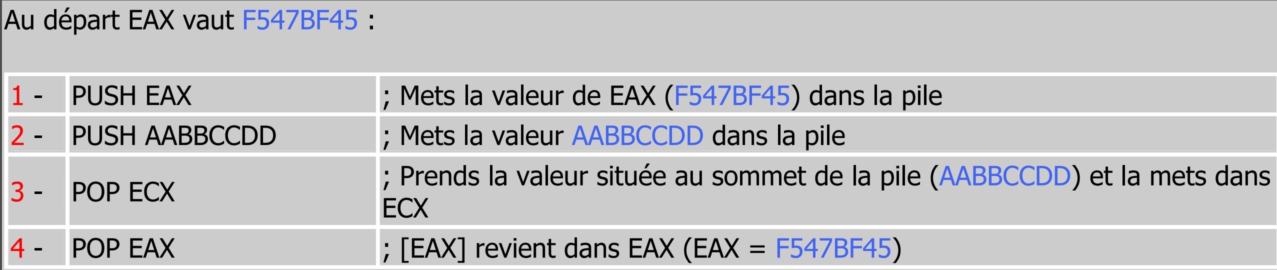
La pile de notre ordinateur est de type LIFO (principe "dernier entré, premier sortie"). Un exemple bien connu pour faire comprendre le concept est de se représenter la pile LIFO comme une pile d'assiette. Une assiette représente une valeur qu'on veut sauvegarder. On place une première assiette, puis une deuxième, une troisième, ... et ainsi de suite pour former une pile d'assiette. Si on veut récupérer une assiette spécifique on est obligé de dépiler les assiettes une à une jusqu'à atteindre l'assiette qu'on veut : en fait la première assiette déposée sur la pile sera la dernière qu'on pourra extraire par la suite.

En assembleur pour mettre une valeur sur la pile on utilise l'instruction PUSH, pour extraire la valeur qui est au sommet de la pile on utilise l'instruction POP. La pile fonctionne à l'aide du registre pointeur de pile ESP (du registre de segment) qui contient l'adresse offset sur 32 bits pointant vers la dernière valeur entière qui sera positionnée sur la pile, pour résumer en image ESP sauvegarde la position de l'assiette qui est au sommet de la pile pour savoir à quel "hauteur" se trouve la pile (pour éviter de casser la prochaine assiette qu'on empilera si vous voulez ; -).

Chose importante à comprendre : Lorsque qu'on fait un PUSH (sur 32 bits), l'opération d'empilage, dans un premier temps, **décrémente** le pointeur de pile de 4 (ESP) puis dans un deuxième temps copie la valeur à l'emplacement désigné par le pointeur de pile. A l'inverse, un POP transfère la valeur du sommet de la pile dans l'opérande de destination puis **incrémente** ESP.

En résumé, quand les assiettes s'empilent, l'offset diminue (en fait la pile d'assiette "pousserait" à partir du plafond) : En plongeant dans la pile, nous retrouvons des données de plus en plus anciennes, et des adresses de plus en plus **haute** (la pile "pousserait" à l'endroit si en plongeant dans la pile, nous retrouvons des données de plus en plus anciennes, et des adresses de plus en plus **basses**,c'est-à-dire si le registre ESP serait incrémenté).

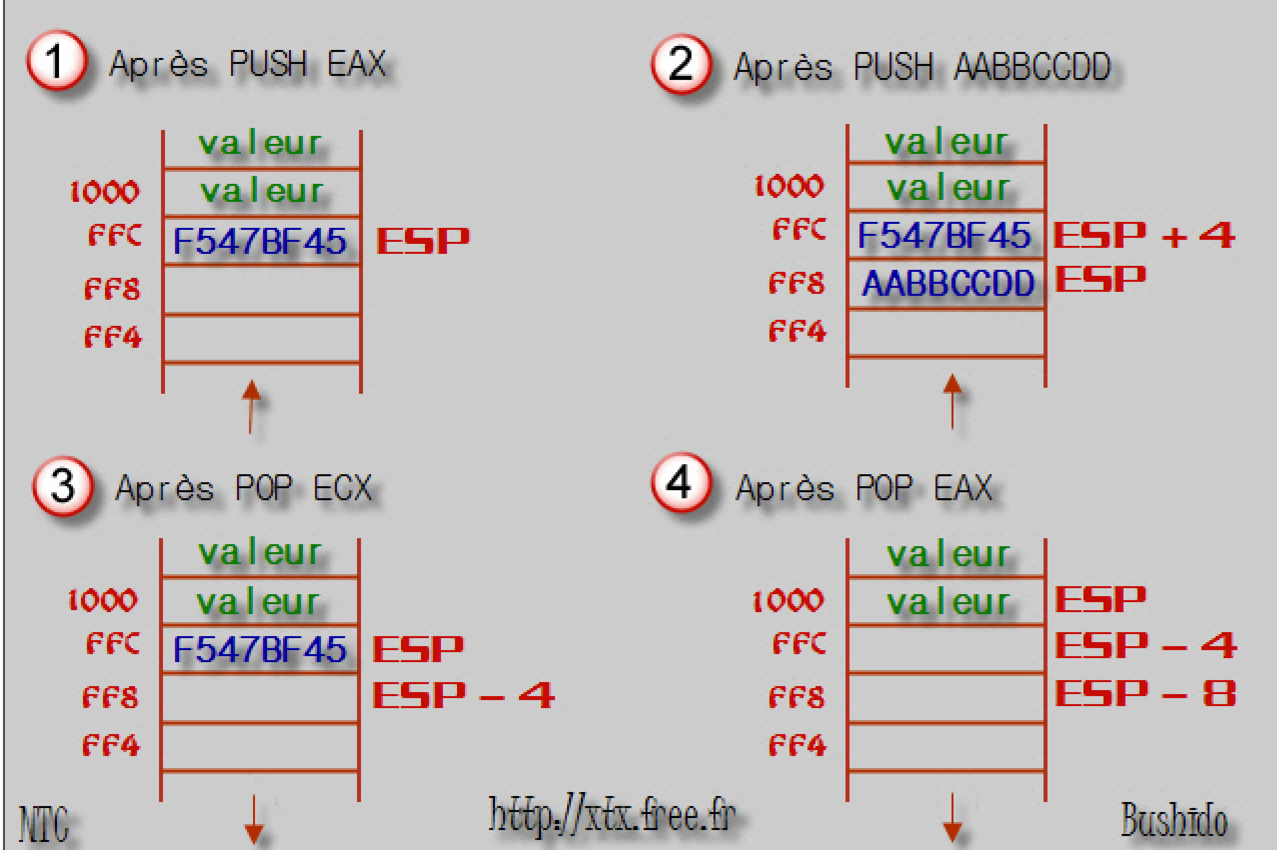
C'est vrai qu'à la première lecture, le concept de pile est difficile à appréhender, on va se faire un petit exemple en assembleur pour bien comprendre ce qui se passe au niveau de la pile quand on fait des PUSH et des POP :


Au départ EAX vaut F547BF45 :

1 -	PUSH EAX	; Mets la valeur de EAX (F547BF45) dans la pile
2 -	PUSH AABBCCDD	; Mets la valeur AABBCCDD dans la pile
3 -	POP ECX	; Prends la valeur située au sommet de la pile (AABBCCDD) et la mets dans ECX
4 -	POP EAX	; [EAX] revient dans EAX (EAX = F547BF45)

Donc là j'utilise des registres sur 32 bits; donc à chaque PUSH ESP va se décrémenter de 4. Si on utilise les registres sur 16 bits (AX, BX, ...) à chaque PUSH, ESP se décrémentera de 2.

J'ai représenté l'état de la pile **après** chacune des 4 instructions citées au dessus, avec la position de ESP (les ESP + X et ESP - X sont juste là pour vous aider) :



Pour votre culture personnel il existe d'autres types de pile d'exécution, comme la pile de type FIFO (principe "premier entré, premier sortie"). On pourrai imaginer une pile FIFO comme un tuyau qui se remplit par le haut et se vide par le bas (les premières valeurs enregistrées sur la pile sont les premières sorties). Pour faire l'analogie avec le type LIFO on peut se représenter l'utilisation de ces deux pile avec l'exemple d'un employé dans une superette : il fait du FIFO pour optimiser la gestion des ventes quand il range ses produits sur les rayons, même si les rayonnages sont plus prévue pour du LIFO. Je vous rassure c'est la dernière métaphore que je fais : -) ; en tout cas si vous avez compris cet exemple alors cela veut dire que vous avez bien pigé le concept ; -).

PUSHAD/POPAD :

PUSHAD et POPAD sont identiques à PUSH et POP mis à part que PUSHAD sauvegarde (mets sur la pile) tous les registres, l'instruction POPAD les restaurent tous. On rencontre PUSHAD et POPAD avec UPX par exemple : La compression d'un programme avec UPX est précédé d'un POPAD et est suivis d'un PUSHAD, cela pour sauvegarder tous les registres pour que la compression reste transparente et ne fasse pas perdre d'informations.

CALL / RET :

Vous connaissez les instructions CALL et RET ? Elles permettent d'utiliser un "sous-programme". Lorsque le microprocesseur rencontre un CALL il enregistre l'adresse de l'instruction qui suit le CALL dans la pile pour pouvoir y revenir après. Ensuite il saute à la sous-routine (désigné par le CALL par une adresse 16 bits, désignant la position du début de la procédure, ou bien du nom de la procédure) pour exécuter l'algorithme. La fin de la routine est désigné par l'instruction RET, lorsque le microprocesseur rencontre cette instruction celui-ci va récupérer l'adresse du CALL dans la pile (Le pointeur **IP** est alors chargé avec cette valeur) pour revenir juste après le CALL comme si rien ne s'était passé. Ceci implique que la pile doit être équilibrée au moment du RET pour retrouver l'adresse de retour du bon CALL au sommet de la pile.

On en a finis pour la pile, mais ce n'est pas fini; on passe aux autres sections :

Revenir au sommaire

L'assembleur

Les flags :

Tous les microprocesseurs disposent d'au moins un registre, interprétable non pas globalement mais bit par bit. Dans l'architecture IA, il s'agit du registre FLAGS (16 bits), étendu en EFLAGS depuis le 80386. Ce registre EFLAGS fait une taille de 32 bits, ce qui parait cohérent avec ce qu'on a vu pour l'instant(rappelez vous la partie sur les registres). Chaque bit du EFLAGS représente un état différent et se nomme un flag. Vous pouvez employer indifféremment les mots fanion, indicateur, voire drapeau. Mais le plus souvent, un flag particulier sera désigné par son abréviation. Par exemple le flag correspondant au 6ème bit du registre EFLAGS est appelé ZF pour Zero Flag.

Je vous rassure tout de suite, on va pas se taper les 32 bits un à un; on ne va considérer que certains bits du registre FLAGS, c'est-à-dire ceux qui vont nous servir dans Olly ; -) : On peut les voir dans la fenêtre de droite, sous les registres généraux :

Chacun de ses flags étant un bit ils ne peuvent prendre que deux valeurs : 1 ou 0. Suivant les instructions qu'exécutera le processeur chacun de ses flags passeront à 1 ou à 0 suivant certaines conditions. Par exemple, si le résultat d'un calcul vaut 0 (après un CMP par exemple), alors le registre ZF passera à 1 (Z dans Olly), si le résultat d'un calcul est différent de 0 alors ZF sera égal à 0. On peut ensuite exploiter la valeur de ses flags sur des sauts conditionnels par exemple (voir section sauts conditionnels).

Petite description des 8 flags qu'on peut voir dans Olly :

Nom	Abréviation	Etat	Description
Carry	CF (C dans Olly)	1	retenue lors de la dernière instruction arithmétique non signée (c'est-à-dire lorsque le résultat généré est trop grand pour l'opérande de destination).
Parity	PF (P dans Olly)	1	résultat d'une opération contient un nombre pair de bits 1
		0	résultat d'une opération contient un nombre impair de bits 1
Auxiliary	AF (A dans Olly)	1	lorsqu'il y a un débordement en BCD (binaire codé décimal) au-delà du 3ème bit ; - p
		0	0 sinon
Zero	ZF (Z dans Olly)	1	Résultat d'une opération est égal à 0
		0	Résultat d'une opération est différent de 0
Sign	SF (S dans Olly)	1	Résultat d'une instruction donne un nombre négatif (si bit de poids fort = 1)
		0	Résultat d'une instruction donne un nombre positif (si bit de poids fort = 0)
Trap	TF (T dans Olly)	1	déclenche un appel à une interruption après chaque instruction exécutée (utilisé pour le débuggage)
Direction	DF (D dans Olly)	1	décrémentation sur les opérations de chaînes
		0	incrémentation sur les opérations de chaînes
Overflow	OF (O dans Olly)	1	lorsqu'il y a un débordement (quand le signe change alors qu'il ne devrait pas)
		0	Sinon est à 0

A tout moment dans Olly on peut modifier l'un de ces flags pour modifier son état en double cliquant dessus. Je vous sens perplexe mais ne vous en faîtes pas, on va regarder comment sont modifiés les flags suivant les différentes instructions de l'assembleur avec des exemples dans cette deuxième partie: -). Sachez que les flags changent à la suite des opérations arithmétiques, des comparaisons et des opérations booléennes :

Le jeu d'instruction :

Les instructions booléennes et comparaisons :

Les opérations de ces instructions se font bit à bit. Et les opérandes et emplacements mémoires qu'elles utilisent sont en 8 ou 16 bits. Elles utilisent une opérande de destination et une opérande source selon ce schéma :

INSTRUCTION <destination>, <source>

Le résultat du calcul entre l'opérande source et l'opérande de destination sera placé dans l'opérande de destination, ceci s'applique pour la plupart des instructions de ce type. Pour faire simple, OR AX, BX est équivalant à AX = AX OR BX.

Voilà une description de chacune d'elle :

OR

L'instruction OR effectue un OU logique entre deux opérandes, voilà la table de vérité de OR avec l'exemple le plus simple d'une comparaison entre 2 bits :

1	2	»	OR
0	0	»	0
0	1	»	1
1	0	»	1
1	1	»	1

OR mets à zéro les deux flags Overflow et Carry.
OR modifie les flags Sign, Zero et Parity.

Imaginons l'instruction OR AL, AL. Suivant la valeur de AL voyez comment sont modifiés les flags Zero et Sign :

Valeur dans AL	»	Zero	Sign
> 0	»	0	0
= 0	»	1	0
< 0	»	0	1

Dès que le résultat est différent de zéro, ZF passe à zéro; SF passe à 1 quand le résultat du OR est négatif; remarquez que la valeur 0 désarme SF (mets SF à 0).

al > 0

OR al, al	; SF = 0, ZF = 0
OR al, 80h	; SF = 1 (80h = -128)

OR al, al permet d'extraire des informations au sujet de la valeur de al.

AND

L'instruction AND effectue un ET logique entre deux opérandes, voilà la table de vérité de AND avec l'exemple le plus simple d'une comparaison entre 2 bits :

1	2	»	AND
0	0	»	0
0	1	»	0
1	0	»	0
1	1	»	1

AND mets à zéro les deux flags Overflow et Carry.
AND modifie les flags Sign, Zero et Parity.

Imaginons l'instruction AND AL, AL. Suivant la valeur de AL les flags Zero et Sign sont modifiés de la même façon que OR.

al < 0	
AND al, al	; SF = 1, ZF = 0
OR al, 7Fh	; SF = 0 (7Fh = 127)

XOR

L'instruction XOR effectue un OU exclusif entre deux opérandes, elle est identique au OU LOGIQUE mise à part la dernière comparaison; voilà la table de vérité de XOR avec l'exemple le plus simple d'une comparaison entre 2 bits :

1	2	»	AND
0	0	»	0
0	1	»	1
1	0	»	1
1	1	»	0

XOR mets à zéro les deux flags Overflow et Carry.
XOR modifie les flags Sign, Zero et Parity.

Exemples avec AX = 10101111 et BX = 00010001 :	
XOR AX, BX	; AX = 10101111 OR 00010001 = 10111110
AND AX, BX	; AX = 10111110 AND 00010001 = 00010000
OR AX, AX	; AX = 00010000, SF = 0
Exemple pour le drapeau parity :	
mov AL, 10110101b	; 5 bits = parité impaire
xor AL, 0	; AL = 10110101b, PF passe à 0
mov AL, 11001100b	; 4 bits = parité paire
xor AL, 0	; AL = 11001100b, PF passe à 1
Pour obtenir la parité de A sans modifier A il suffit de faire :	
A XOR 0	; modifie PF suivant la valeur de A.

TEST

L'instruction TEST est identique à AND mais n'affecte pas le résultat à l'opérande de destination, elle permet, par exemple, de tester plusieurs bits à la fois :

On veut savoir si le bit 0 et 3 sont armés (=1) dans le registre AL :

Si AL = 00100101 :	
TEST AL, 00001001b	; retourne 00000001 (= AND 00100101, 00001001)
ZF passe à 0 car le résultat n'est pas égal à 0, les bits 0 et 3 de AL sont égales à 1.	

On veut savoir si le bit 4, 3 et 6 sont armés (=1) dans le registre AL :

Si AL = 11010011 :	
TEST AL, 00101100b	; retourne 00000000 (= AND11010011, 00101100)
ZF passe à 1 car le résultat est égal à 0, les bits 4, 3 et 6 de AL sont donc égale à 0.	

TEST mets à zéro les deux flags Overflow et Carry.
TEST modifie les flags Sign, Zero et Parity.

XOR est connu en cryptographie "de base", l'exemple le plus simple est de faire un XOR pour "crypter" A, puis de refaire un XOR pour "décrypter" A avec un B quelconque :

A XOR B	; A = A XOR B
A XOR B	; A = (A XOR B) XOR A = A

NOT

L'instruction NOT effectue un NON logique implicite sur une opérande et affecte le résultat dans la même opérande, la syntaxe est la suivante :

NOT <opérande>

Voici la table de vérité de NOT, elle ne fait qu'inverser chaque bit d'un nombre.

1	»	NOT
0	»	1
1	»	0

L'instruction NOT s'apparente à un complément à 1.
NOT modifie les drapeaux de façon approprié.

AX = 00001111b

NOT AX	; AX =11110000b
NOT AX	; AX = 00001111b

Le complément à 1 binaire se calcule facilement manuellement en soustrayant un nombre binaire (dont on veut connaitre le complément à 1) par 11111111 (pour un octet, i.e 8 bits) :

	11111111	11111111
	- 00001111	- 11110000
	-------------------	-------------------
	11110000	00001111

Je rappelle les opération d'addition sur une base binaire :
0 + 0 =0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 10 (1 placé en retenu)

Les instructions arithmétiques :

Les opérations de ces instructions se font bit à bit. Comme pour la plupart des instructions logiques les opérandes et emplacements mémoires que ces opérations utilisent une opérande de destination et une opérande source selon ce schéma :

INSTRUCTION <destination>, <source>

Le résultat du calcul entre l'opérande source et l'opérande de destination sera placé dans l'opérande de destination, ceci s'applique pour la plupart des instructions de ce type. Pour faire simple, ADD AX, BX est équivalant à AX = AX + BX.

ADD et SUB

Ces deux instructions s'apparentent à l'opération addition et soustraction. Elles s'utilisent exclusivement avec des nombres de même poids, on ne peut additionner CX avec BL par exemple car CX fais 8 bits et BL 16 bits.

AX = 148 (10010100)
BX = 69 (01000101)

ADD AX, BX	; AX =148 + 69 = 217
SUB AX, BX	; AX =217 - 69 = 148

ADD modifie les flags Zero, Overflow et Carry suivant le résultat.
SUB modifie les flags Zero et Sign suivant le résultat.

MUL et DIV

Ces deux instructions s'apparentent à l'opération multiplication et division. ATTENTION : Ces deux instructions n'utilisent qu'une seule opérande source, l'opérande de destination est toujours AX; le résultat du calcul est placé dans la destination (AX) :

MUL <opérande source>	»	AX = AX * opérande source
DIV <opérande source>	»	AX = AX / opérande source

MUL et DIV ne s'utilisent qu'avec des nombres non signés, c'est-à-dire qui ne sont pas négatif (16 bits en non signé: les nombres vont de 0 à 65535, 8 bits en non signé: les nombres vont de 0 à 255, ...). Le reste de la division est placé dans DX :

AX = 4
BX = 3
CX = 6

MUL BX	; AX = AX * BX = 4 * 3 = 12
MUL AX	; AX = AX * AX = 12² = 144
DIV CX	; AX = AX / CX = 144 / 6 = 24

MUL : OVERFLOW, CARRY = 0 si DX = extension de signe ZERO, SIGNE modifiés mais indéfinis
DIV : tous modifiés mais indéfini

IMUL et IDIV

Ces deux instructions s'apparentent à l'opération MUL et DIV mise à part qu'elles utilisent des nombres signés, c'est-à-dire qui peuvent être négatifs (16 bits en signé: les nombres vont de -32768 à 32767, 8 bits en signé: les nombres vont de -126 à 127, ...). Le reste de la division est placé dans DX :

AX = 20
BX = 3
CX = -4

IDIV BX	; AX = AX * BX = 20 / 3 = 6 (DX = 2)
IMUL CX	; AX = AX * CX = 6 * (-4) = -24

Remarque (détail pour DX) :

Division Euclidienne	» AX = quotient * AX + reste
IDIV BX	» 20 = 6*3 + 2

IMUL : OVERFLOW, CARRY = 0 si DX = extension de signe ZERO, SIGNE modifiés mais indéfinis
IDIV : tous modifiés mais indéfinis

NEG

NEG inverse le signe de l'opérande, comme MUL,DIV, IMUL et IDIV elle n'a besoin que d'une opérande de destination. ATTENTION : il ne faut pas confondre NEG avec NOT : NEG correspond au complément à 2 (NOT, je vous le rappelle au complément à 1).

AX =15 (= 00001111b)
BX = -36

NEG AX	; AX = -15
NEG BX	; BX = 36

Le complément à 2 binaire se calcule facilement manuellement en faisant d'abord le complément à 1 puis en ajoutant 1 :

	11111111	11110000
	- 00001111	+ 1
	------------------	------------------
	11110000	11110001

Pour votre culture perso, on remarque qu'en ajoutant le nombre et son complément à deux on obtient 0 :

00001111 +11110001= 0 (avec une retenue de 1...)

INC et DEC

INC et DEC permettent respectivement d'incrémenter et de décrémenter d'une unité un registre ou une adresse (INC et DEC ne peut être utilisé avec une valeur immédiate). INC et DEC n'ont besoin que d'une opérande de destination :

INC <destination>
DEC <destination>

AH = 12

INC AX	; AX = 12 + 1 = 13
DEC AX	; AX = 13 - 1 = 12

Remarque : INC est souvent utilisé pour analyser un serial caractère par caractère.

INC et DEC modifient les flags Sign, Zero et Overflow suivant le résultat. Carry n'est jamais modifiée.

MOV

MOV est une instruction que l'on rencontre très souvent en assembleur. Elle permet de copier le contenue de l'opérande source dans l'opérande de destination. La syntaxe reste classique :

MOV <destination>, <source>

Il est possible de sélectionner une certaine partie du registre de l'opérande source à copier dans la destination en précisant la taille que l'on veut :

type	capacité
BYTE / SBYTE	8 bits non signé / 8 bits signé
WORD / SWORD	16 bits non signé / 16 bits signé
DWORD / SDWORD	32 bits non signé / 32 bits signé
FWORD	48 bits
QWORD	64 bits
TBYTE	10 octet

Concrètement, vous verrez souvent sous olly du BYTE, WORD et DWORD. Petite précision, lorsque qu'un registre par exemple est entouré par des crochets cela veut dire qu'on sélectionne la donnée vers lequel il pointe :

MOV EAX, [EAX] » Transfère dans EAX la donnée qu'il pointe.

EAX = 00008632	
MOV DWORD PTR DS:[ESI], EFCD26B0	; mets la valeur hexa EFCD26B0 dans ESI
MOV AL,BYTE PTR DS:[ESI]	; mets B0 dans AL » EAX devient 000086B0
INC ESI	; incrémente ESI
MOV AL,BYTE PTR DS:[ESI]	; mets 26 dans AL » EAX devient 00008626
MOV AH,BYTE PTR DS:[ESI]	; mets 26 dans AH » EAX devient 00002626

MOV ne modifie aucun flag.

CMP

L'instruction CMP est identique à SUB mais n'affecte pas le résultat à l'opérande de destination, elle permet de vérifier si deux valeurs sont égales dans la plupart des cas, la syntaxe est la suivante :

CMP <destination>, <source>

L'instruction CMP modifie tous les drapeaux; c'est en analysant les flags que l'on pourra savoir si une expression est différente d'une autre par exemple :

AX = 10
BX = 12

CMP AX, BX	; effectue une soustraction "AX-BX"

Dans ce cas AX et BX sont différents, donc le résultat de la soustraction implicite sera différent de 0, ainsi le flag ZF sera mis à 0. Il suffira de mettre un JNZ par exemple qui saute suivant l'état du flag Z. De même on pourra faire des sauts suivant l'état des autres flags.

Evolution des flags ZF et CF avec l'instruction CMP g, d (gauche, droite); g et d étant non signés :

Hypothèse	»	Zero	Sign
Si g < d	»	0	1
Si g > d	»	0	0
Si g = d	»	1	0

Evolution des flags ZF et CF avec l'instruction CMP g, d (gauche, droite); g et d étant signés :

Hypothèse	»	flags
Si g < d	»	SF ≠ OF
Si g > d	»	SF = OF
Si g = d	»	ZF = 1

Je finis sur l'instruction CMP pour enchaîner sur les sauts conditionnels histoire de faire une belle transition ; -).

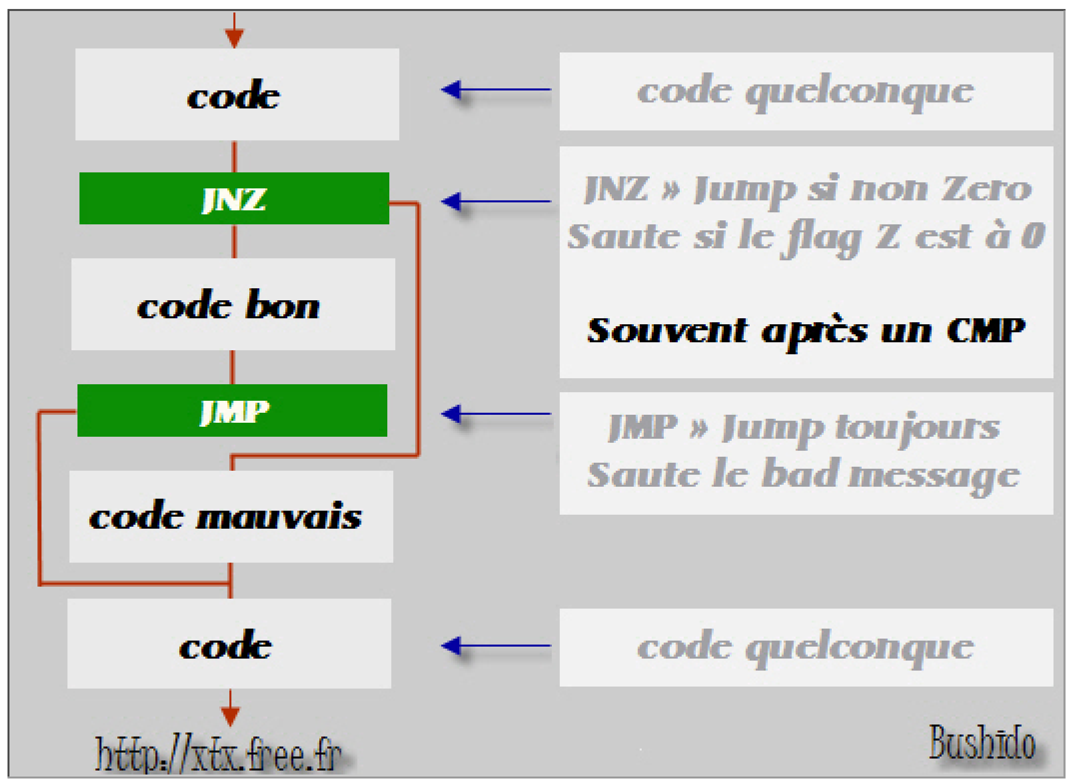
Revenir au sommaire

Cracking "de base"

Après avoir analysé le fonctionnement des flags et assimilé les quelques instructions de l'assembleur que je vous ais mis à disposition on va s'attaquer au cracking "de base", c'est-à-dire l'étude des sauts conditionnels. Cette partie n'a rien de très difficile en soi, elle est surtout là en temps qu'aide-mémoire sur le fonctionnement de chacun des sauts (je vais en lister 30 au total). Et pour finir je vous expliquerais quelques techniques à connaitre pour optimiser la programmation en assembleur (même si ce cours ne parle pas de la programmation en ASM à proprement parler). Allé c'est partis ; - p

Les sauts conditionnels :

Un saut conditionnel, comme son nom l'indique permet d'atteindre une certaine partie du code pour effectuer différentes instructions suivant une condition bien précise. Chaque saut est effectué par rapport à l'état d'un flag particulier, d'une égalité ou d'une comparaison : c'est la condition de saut. Voici le schéma typique d'un saut conditionnel, souvent utilisé pour les protections dans un log :



Saut fondés sur les drapeaux :

Nom	flag	Description
JZ	ZF = 1	Saute si un résultat est égal à 0
JNZ	ZF = 0	Saute si un résultat est différent de 0
JC	CF = 1	Saute s'il y a dépassement de capacité (sur des nombres non signés)
JNC	CF = 0	Saute s'il n'y a pas dépassement de capacité (sur des nombres non signés)
JO	OF = 1	Saute si le résultat a provoqué de débordement
JNO	OF = 0	Saute si le résultat n'a pas provoqué de débordement
JS	SF = 1	Saute si un résultat est un nombre signé négatif
JNS	SF = 0	Saute si un résultat est un nombre signé positif
JP	PF = 1	Saute si un résultat est un nombre de parité paire
JNP	PF = 0	Saute si un résultat est un nombre de parité impaire

Petite précision pour CF :

CMP ax, bx	; si ax < bx » CF = 1
JC <adresse>	; si ax < bx » le saut est pris

CMP exécute la soustraction virtuelle ax-bx, si ax est inférieur à bx alors le résultat du calcul ax-bx sera faux (nombres non signés avec JC), il y aura dépassement de capacité, CF passera à 1.

À noter que, suite à un ADD, si ZF=1, on a alors obligatoirement CF=1 aussi car si le résultat est nul, il y a forcément eu dépassement de capacité. Suite à un CMP, si ZF=1, la destination et la source qui ont été comparés doivent dans ce cas être identiques, on a alors obligatoirement CF=0 (voir CMP).

Saut fondés sur l'égalité :

Nom	Conditions	Description
JE	Egalité	Saute si un résultat est égal à 0 (équivalent à JZ)
JNE	Inégalité	Saute si un résultat est différent de 0 (équivalent à JNZ)
JCXZ	CX = 1	Saute si le registre [CX] contient la valeur zéro
JECXZ	ECX = 0	Saute si le registre [ECX] contient la valeur zéro

Saut fondés sur des comparaisons (d'entiers non signés) :

INSTRUCTION g, d (gauche, droite)

Nom	Désignation	flags	Description
JA	Above	ZF & CF = 0	Saute si g > d
JNA	Not Above	ZF & CF = 0	Saute si g n'est pas > d
JAE	Above Equal	CF = 0	Saute si g ≥ d
JNAE	Not Above Equal	CF = 1	Saute si g n'est pas ≥ d
JB	Below	CF = 1	Saute si g < d
JNB	Not Below	CF = 0	Saute si g n'est pas < d
JBE	Below Equal	ZF & CF = 0	Saute si g ≤ d
JNBE	Not Below Equal	ZF & CF = 0	Saute si g n'est pas ≤ d

MOV al, 7Fh	; al = 7Fh (7Fh ou +127)
CMP al, 80h	; al - 80h implicite (80h ou -128)
JA <adresse>	; 7Fh n'est pas > 80h (127 n'est pas > 128) » ne saute pas

Saut fondés sur des comparaisons (d'entiers signés) :

INSTRUCTION g, d (gauche, droite)

Nom	flags	Description
JG	ZF & SF = 0	Saute si g > d
JNG	SF ≠ OF & ZF = 1	Saute si g n'est pas > d
JGE	SF = OF	Saute si g ≥ d
JNGE	SF ≠ OF	Saute si g n'est pas ≥ d
JL	SF ≠ OF	Saute si g < d
JNL	SF = OF	Saute si g n'est pas < d
JLE	SF ≠ OF & ZF = 1	Saute si g ≤ d
JNLE	ZF & SF = 0	Saute si g n'est pas ≤ d

MOV al, 7Fh	; al = 7Fh (7Fh ou +127)
CMP al, 80h	; al - 80h implicite (80h ou -128)
JG <adresse>	; 7Fh > 80h (+127 > -128) » le saut est pris
MOV ax,5	; ax = 5
CMP ax, 6	
JL <adresse>	; si 5 < 6 » le saut est pris

Instructions fréquentes :

Modifier un flag simplement :

Technique pour modifier le statut d'un des flags Zero, Sign, Carry :

Peu importe la valeur de al :

AND al, 0	; ZF = 1
OR al, 1	; ZF = 0
OR al, 80h	; SF = 1
AND al, 7Fh	; SF = 0
STC	; CF = 1 (instruction dédié)
CLC	; CF = 0 (instruction dédié)

Remarque : toutes ces instructions modifient la destination. On peut utiliser XOR pour ne pas modifier la destination (XOR al, 0 par exemple).

Optimisation du code :

L'exécution de chaque instruction implique un nombre de cycle d'horloge plus ou moins important suivant l'instruction (A voir dans la documentation du microprocesseur / microcontrôleur). Plus le cycle est élevé plus le temps d'exécution de l'instruction est grande. On peut donc parler d'optimisation du code. En assembleur on peut obtenir un résultat identique avec des instructions très différentes, on privilégiera une instruction à une autre suivant la vitesse d'exécution du programme, exemples :

SUB CX, CX (simple accès à la mémoire) est plus rapide que MOV CX, 0 (double accès à la mémoire).
INC est plus rapide à l'exécution que ADD destination, 1.

Prenons un exemple simple :

Comment échanger deux variables a et b, sans variable intermédiaire ? Dans n'importe quel langage on fera comme ceci :

Soit a = 2 et b = 3 :

a = a + b	; a = 5
b = a - b	; b = 5 - 3 = 2
a = a - b	; a = 5 - 2 = 3
17 cycles	

Il est possible d'effectuer cette opération de différentes façons en assembleur. On peut classer ces différentes opérations du plus lent au plus rapide (il y en a sûrement d'autres, je n'en liste que 4) :

push a	; a va dans la pile
push b	; b va dans la pile
pop a	; a = b (pile) = 3
pop b	; b = a (pile) = 2
18 cycles	

mov eax, b	; eax = 3
add a, eax	; a = 2 + 3 = 5
mov eax, a	; eax = a = 5
sub eax, b	; eax = 5 - 3 = 2
mov b, eax	; b = eax = 2
mov eax, b	; eax = b = 2
sub a, eax	; a = a - eax = 5 - 2 = 3
17 cycles	

mov eax, a	; eax = 2
mov ecx, b	; ecx = 3
mov a, ecx	; a = 3
mov b, eax	; b = 2
15 cycles	

Astuces :

• Pour vérifier qu'une valeur en vaut une autre, plutôt que de faire un CMP, il vaut mieux faire un OR;
• Pour mettre un registre à zéro, plutôt que de faire un MOV (ex : MOV ax, 0) il suffis de faire un XOR sur lui même (ex : XOR ax, ax);
• Si vous voulez multiplier ou diviser la valeur d'un registre par un multiple de 2, utilisez plutôt un décalage de bits. Pour une multiplication, utilisez SHL (SHR pour une division) avec la puissance de 2. Exemple: pour multiplier AX par 8, il suffit de faire "SHL AX, 3" (car 8 = 2^3). Dès que possible, toujours utiliser les opérations sur les bits.

Liens :

Lien	Description
	Manuel de référence d'Intel complète sur l'assembleur et l'IA32, consultable en ligne (pdf).
	Site assez complet sur l'assembleur, claire et avec pas mal d'exemples. A voir.
	Section Assembleur de www.commentcamarche.net.
	Un très bon livre sur l'assembleur que je vous conseille.
	Plusieurs cours sur l'assembleur (developpez)
	Forum de developpez, section assembleur.

Revenir au sommaire

C'est fini !!!!! :

Voilà c'est (enfin) la fin, ouf !!! ; -). J'espère que ce cours sur l'assembleur vous aura été utile; en tout cas bravo si vous avez tout lu (et compris). Il est probable que j'ai fais quelques fautes dans ce cours, n'hésitez pas à venir gueuler sur le forum si c'est le cas : -) . Ce cours reste encore incomplet, mais je pense y faire des mises à jour dès que possible. Si vous n'avez pas compris certaines choses, ou si j'ai écris des choses incompréhensible (c'est possible, je le conçois ; -) n'hésitez pas à en parler sur le forum, je serais là pour vous répondre...

Petite dédicace à ma team NTC sans qui ce cours n'existerait sûrement pas, à tout ceux qui auront apprécié ce Tuto, et à mon clavier a qui j'ai fais subir les pires souffrances en écrivant ce tut ; -).

(`'·.¸(`'·.¸*¤*¤*¤*¸.·'´)¸.·'´)
«´¨`·..¤:BUSHIDO:¤..·´¨`»
(¸.·'´(¸.·'´*¤*¤*¤*`'·.¸)`'·.¸)

La connaissance est la seule chose qui s'accroît lorsqu'on la partage
[Sacha Boudjema]

Ce tutorial étant soumis au droit d'auteur, la reproduction partielle ou intégrale de ce tutorial est interdite sans autorisation de l'auteur. Merci à l'avance de respecter ce droit.